Contributions to Confidentiality and Integrity Algorithms for 5G

Yang, Jing

2021

Document Version:
Publisher's PDF, also known as Version of record

Contributions to Confidentiality and Integrity Algorithms for 5G

Doctoral Dissertation

Jing Yang

LUND UNIVERSITY

Department of Electrical and Information Technology
Lund University, Lund, Sweden
November, 2021

Department of Electrical and Information Technology
Lund University
Box 118, SE-221 00 LUND
SWEDEN

This thesis is set in Computer Modern 10pt
with the LaTeX Documentation System

Series of licentiate and doctoral theses
ISSN 1654-790X144
ISBN: 978-91-8039-106-1 (print)
ISBN: 978-91-8039-105-4 (pdf)

Printed in Sweden by *Tryckeriet i E-huset*, Lund.
November 2021.

...to all the people who have inspired and encouraged me to become who I am...

Abstract

The confidentiality and integrity algorithms in cellular networks protect the transmission of user and signaling data over the air between users and the network, e.g., the base stations. There are three standardised cryptographic suites for confidentiality and integrity protection in 4G, which are based on the AES, SNOW 3G, and ZUC primitives, respectively. These primitives are used for providing a 128-bit security level and are usually implemented in hardware, e.g., using IP (intellectual property) cores, thus can be quite efficient.

When we come to 5G, the innovative network architecture and high-performance demands pose new challenges to security. For the confidentiality and integrity protection, there are some new requirements on the underlying cryptographic algorithms. Specifically, these algorithms should: 1) provide 256 bits of security to protect against attackers equipped with quantum computing capabilities; and 2) provide at least 20 Gbps (Gigabits per second) speed in pure software environments, which is the downlink peak data rate in 5G. The reason for considering software environments is that the encryption in 5G will likely be moved to the cloud and implemented in software.

Therefore, it is crucial to investigate existing algorithms in 4G, checking if they can satisfy the 5G requirements in terms of security and speed, and possibly propose new dedicated algorithms targeting these goals. This is the motivation of this thesis, which focuses on the confidentiality and integrity algorithms for 5G. The results can be summarised as follows.

- We investigate the security of SNOW 3G under 256-bit keys and propose two linear attacks against it with complexities 2^{172} and 2^{177}, respectively. These cryptanalysis results indicate that SNOW 3G cannot provide the full 256-bit security level.

- We design some spectral tools for linear cryptanalysis and apply these tools to investigate the security of ZUC-256, the 256-bit version of ZUC. We propose a distinguishing attack against ZUC-256 with complexity 2^{236}, which is 2^{20} faster than exhaustive key search.

- We design a new stream cipher called SNOW-V in response to the new requirements for 5G confidentiality and integrity protection, in terms of security and speed. SNOW-V can provide a 256-bit security level and achieve a speed as high as 58 Gbps in software based on our extensive evaluation. The cipher is currently under evaluation in ETSI SAGE (Security Algorithms Group of Experts) as a promising candidate for 5G confidentiality and integrity algorithms.

- We perform deeper cryptanalysis of SNOW-V to ensure that two common cryptanalysis techniques, guess-and-determine attacks and linear cryptanalysis, do not apply to SNOW-V faster than exhaustive key search.
- We introduce two minor modifications in SNOW-V and propose an extreme performance variant, called SNOW-Vi, in response to the feedback about SNOW-V that some use cases are not fully covered. SNOW-Vi covers more use cases, especially some platforms with less capabilities. The speeds in software are increased by 50% in average over SNOW-V and can be up to 92 Gbps.

Besides these works on 5G confidentiality and integrity algorithms, the thesis is also devoted to local pseudorandom generators (PRGs).

- We investigate the security of local PRGs and propose two attacks against some constructions instantiated on the P_5 predicate. The attacks improve existing results with a large gap and narrow down the secure parameter regime. We also extend the attacks to other local PRGs instantiated on general XOR-AND and XOR-MAJ predicates and provide some insight in the choice of safe parameters.

Popular Science Summary

When you make a phone call to someone, what you said will be transmitted from your mobile phone through the open air to a base station, and the base station forwards the messages to the one you want to talk to. I believe that you do not want your talk heard by any third person, no matter he is malicious or just curious. The weakest sub-link along the message transmission path lies over the air, as it is open to everyone and a malicious or curious person, let us call him an attacker, may be able to eavesdrop or even manipulate your talk. As a consequence, the messages should be specially protected over the air to ensure that the attacker will know nothing even if he captures the messages, and not be able to manipulate the messages without notice. Such a goal is achieved by encrypting your messages and adding special tags using some specific cryptographic algorithms, also called ciphers.

There are three standardised ciphers used in 4G for this purpose, named AES, SNOW 3G, and ZUC. All people (with mobile phones) around the world are using (at least one of) these three ciphers, though they might not notice it: these ciphers are typically put into the devices during the manufacturing phase and the encryption is performed automatically by the devices. AES, SNOW 3G, and ZUC provide sufficient security and speeds in 4G for protecting our message transmission.

How about the situation in 5G?

When we come to 5G, the innovate network architecture and high performance demands introduce higher requirements on these ciphers. Firstly, they should be more secure since attackers in the future can potentially have very strong capabilities and can possibly recover your messages, due to the development of quantum computing. Moreover, these ciphers should run much faster in some specific environments since we will be able to enjoy much higher speeds in 5G. Therefore, it is crucial to investigate whether the ciphers we are using today (in 4G) can satisfy these new requirements; and if necessary, design new ones particularly targeting these goals. This is exactly the primary motivation and content of this thesis.

What we did?

Analysing Existing Ciphers. We investigated the security of two of the three ciphers that we are using today in 4G, i.e., SNOW 3G and ZUC, to check if they can satisfy the new security requirements in 5G. We found some theoretical attacks against them, which

provided some reference information to the standardisation organisation (i.e., 3GPP) for choosing good candidates for 5G. But we do not have to worry about these attacks at this moment, because their costs are much beyond an attacker's practical capability today, and furthermore, the attacker will not be able to get enough data for launching such attacks.

Designing New Ciphers. We also designed new ciphers, called SNOW-V and SNOW-Vi as an extreme performance variant, particularly targeting the 5G requirements regarding security and speeds. They can be viewed as successors of SNOW 3G but with much stronger security and higher speeds. They have been submitted to the standardisation organisation for consideration of usage in 5G, and they are currently under evaluation.

Analysing Local Pseudorandom Generators. When using ciphers for encryption or more advanced applications, some random numbers are usually needed. You might think from the intuition that it is easy to generate a random number, but it is actually not so for an electronic device. There are special cryptographic constructions called pseudorandom generators (PRGs) for achieving this goal. We investigated the security of one particular type of PRGs that are very efficient in implementation, which are called local PRGs. We developed some analysis methods that can help choose secure parameters if these PRGs were to be used in practice.

Contribution Statement

This doctoral thesis concludes my work as a Ph.D. student, and is comprised of two main parts. The first part presents an overview of the research field and a brief summary of my work. The second part consists of six papers I worked on during my doctoral education, which are listed as below.

Paper I Jing Yang, Thomas Johansson, and Alexander Maximov. Vectorized linear approximations for attacks on SNOW 3G. IACR Transactions on Symmetric Cryptology, pages 249–271, 2019.

Paper II Jing Yang, Thomas Johansson, and Alexander Maximov. Spectral analysis of ZUC-256. IACR Transactions on Symmetric Cryptology, pages 266–288, 2020.

Paper III Patrik Ekdahl, Thomas Johansson, Alexander Maximov, and Jing Yang. A new SNOW stream cipher called SNOW-V. IACR Transactions on Symmetric Cryptology, pages 1–42, 2019.

Paper IV Jing Yang, Thomas Johansson, and Alexander Maximov. Improved guess-and-determine and distinguishing attacks on SNOW-V. IACR Transactions on Symmetric Cryptology, pages 54-83, 2021.

Paper V Patrik Ekdahl, Thomas Johansson, Alexander Maximov, and Jing Yang. SNOW-Vi: an extreme performance variant of SNOW-V for lower grade CPUs. In Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks, pages 261-272, 2021.

Paper VI Jing Yang, Qian Guo, Thomas Johansson, and Michael Lentmaier. Revisiting the Concrete Security of Goldreich's Pseudorandom Generator. IEEE Transactions on Information Theory, accepted, 2021.

In Paper III and Paper V, authors are listed in alphabetical order with equal contribution. The papers are slightly reformatted to fit within the overall thesis structure.

I was involved in writing, idea design, and implementation for each included paper above. The more detailed individual contributions are described as below.

- In Paper I, I explored the linear approximations together with the other authors, and did the implementations to collect large amounts of data to verify the biases. I was responsible for exploring how to launch distinguishing and correlation attacks based on these linear approximations. I took the main responsibility for writing the whole paper.

- In Paper II, I explored the linear approximations together with the other authors, and designed the spectral tools together with the third author. I did theoretical and experimental verification of some propositions, and was responsible for exploring how to launch a distinguishing attack based on the derived linear approximation. I took the main responsibility for writing the whole paper together with the third author.

- In Paper III, I, together with other authors, designed the cipher. I was solely responsible for evaluating the initialisation attacks (e.g., the maximum degree monomial (MDM) test and cube attacks based on division property) and the time-memory-data tradeoff attacks, and took charge of corresponding implementation and writing. I was also responsible for theoretical proof and its writing, and the reference implementations for the design.

- In Paper IV, I developed the ideas together with the other authors and was responsible for verifying them. I performed partial implementation and took the primary responsibility for writing the whole paper.

- In Paper V, I, together with other authors, designed the cipher. I chose the new LFSRs through implementation together with the third author. Again, I was solely responsible for evaluating the initialisation attacks (e.g., the MDM test and cube attacks based on division property) and time-memory-data tradeoff attacks, and took charge of corresponding implementation and writing. My cryptanalysis results and implementations have helped to choose good parameters for the designed cipher.

- In Paper VI, I, together with the other authors, designed the ideas. I took the primary responsibility to verify, polish, and extend the idea. I was responsible for the theoretical analysis and solely responsible for the implementation and evaluation of the idea. I took the primary responsibility for writing the whole paper.

Other Contributions

The following peer-reviewed papers have also been published during my doctoral education, but are not included in this thesis.

- Jing Yang and Thomas Johansson. An overview of cryptographic primitives for possible use in 5G and beyond. Science China Information Sciences, 63(12): 1-22, 2020.
- Qian Guo, Thomas Johansson, and Jing Yang. A novel CCA attack using decryption errors against LAC. International Conference on the Theory and Application of Cryptology and Information Security, Springer, Cham, 2019.

Acknowledgments

The first day I came to Lund: it was in late March, 2018; I remembered that day very clearly, as I was welcomed by such heavy snow and strong wind– in late Spring! Now after "many" impressive winters in the following almost four years, it is time to say goodbye to my Ph.D. life–in another winter. I have enjoyed the Ph.D. life here so much, though sometimes with challenges, frustration, loneliness, and self-doubt, and the thesis will not be possible without many people.

My deepest gratitude and respect first go to my main supervisor, Professor Thomas Johansson. Thank him for trusting and accepting me as his Ph.D. student even though I had no cryptography background at all at that time. He has such comprehensive knowledge and deep understanding in the research field and can always figure out the problems, which always turn out correct. He is always kind and patient in guiding me to gradually learn to do research and telling me the academic code of ethics in a good way. I like talking with him, not only about research but also random stuff. I once said during the second year that he was one of the best supervisors, and he joked that it was too early to say so. I think now it is time and I would still like to give the best compliments to him. I hope that we can have more collaborations in the future.

I also want to thank my co-supervisors, Paul Stankovski and Qian Guo, for providing me with much help and advice in research. Paul is so nice and talking with him always makes one feel comfortable. He is always willing to provide help whenever one has problems. Qian always has many good ideas and discussing with him can spark me a lot. He also provides me with much information and advice in research and career choices. Special thanks to my external collaborators, Alexander Maximov (Sasha) and Patrik Ekdahl. They are so experienced researchers with profound knowledge in a wide range of areas, and we have always collaborated very well. Sasha is kind of like my co-supervisor and I have learned so much from him. He is so intelligent, efficient, and generous to share ideas and provide help.

Many thanks to the security lab that has created such a friendly research environment and relationship. Everyone is so lovely, and I will miss you all. Thank Martin Hell for providing every Ph.D. student with much helpful information and gathering many nice activities. Thank Christian, Elena, Senyang, Linus, Erik, Jonathan, Alex, Pegah, Joakim, Martin, Syafiq, Hui, Vu, Rohon, and Denis: I have enjoyed the travels, lunches, fika, and talks with you so much. Jonathan and Erik –the most talkative guys in our lab–are the two most often visitors of my office sharing many fun (or "stupid") things to me. Thank Pegah: we have enjoyed the Ph.D. life together from the day I came here till now and I wish you all the best. Hui, one best friend in Lund, has accompanied me

while going through the hardest pandemic time and we have enjoyed so much fun and laughter. Thank Syafiq for his many jokes and book "The Danish Way of Parenting": now I have more advantages in raising kids:). Special thanks to Vu: he knows much about China and I have enjoyed the many nice talks with him about Chinese music, novels, and shows, etc. Thank Christoffer and Daniel, the second group of frequent visitors of my office, who have brought me much joy and fun. Thanks also go to the administrative and technical staff for their patient help throughout my years at EIT, especially Elisabeth Nordström and Erik Jonsson.

I also want to thank my Chinese friends in Lund: Hui, Xiaoqing, Xiaoya, Zhongguo, Qiuyan, and Haorui, who have shared so much fun and laughter with me. We have traveled and cooked together and there are always so many fun talks and jokes among us. It is so lucky to have their accompany in the four years, especially during the pandemic time. Thanks also go to Zehan and Yueyue, for organising the weekly Pingpong and badminton activities.

There are also many people outside Lund that I want to say thanks. My main supervisor in China, Professor Ji: I am so respectful and thankful to him for always encouraging and supporting me to pursue more advanced education. He has profound knowledge in communication security but always keeps modest and never stops learning more. He always views things in an open and long-sighted way. My co-supervisors, Professor Huang and Professor Yi, have provided me with much help in research and advice whenever I face choices. I feel so appreciated for what they have done for me. Thanks also go to Professor Duan and Professor Zheng, who have provided me much information and help in research and career choices. Special thanks to Siwei Sun from the Chinese Academy of Sciences, who also gave me much advice in research.

I also want to thank my colleagues and friends in China: Shuxin, Yajun, Shaoyu, Zheng Wan, Zheng Wu, Xinxin, Yawen, Tong, etc., who have helped me a lot and provided me with much timely information. I am also grateful for many administrative people who have helped me manage many things and care about my safety and life abroad in the four years.

I want to thank my warm family for giving me the love and courage to pursue my dream. I love you all. Deep thanks to my parents, who taught me to become a kind, positive and persistent person. Thank my sisters and brothers-in-law for encouraging me and taking care of our parents, enabling me to focus on my study. Special thanks to my husband, thank him for always respecting and supporting my choices, always cheering me up, and always being there whenever I needed.

Last but not least, I would like to thank the special person of my life, Dongmei Chen, who has guided me to become a kind, independent, strong, and positive girl. Without her, I would never have become who I am today. I love you!

Jing Yang

Lund, November 2021

List of Acronyms

3GPP 3rd Generation Partnership Project

AEAD Authenticated Encryption with Associated Data

AES Advanced Encryption Standard

ANF Algebraic Normal Form

BS Base Station

CBC Cipher Block Chaining

CCA Chosen-ciphertext Attack

CFB Cipher Feedback

CN Core Network

COA Ciphertext-only Attack

CPA Chosen-plaintext Attack

CTR Counter

DFT Discrete Fourier Transform

DLP Discrete Logarithm Problem

ECB Electronic Codebook

FFT Fast Fourier Transform

FHE Fully Homomorphic Encryption

FWHT Fast Walsh Hadamard Transform

GE Gate Equivalent

IFP Integer Factoring Problem

IoT Internet of Things

IV Initialisation Vector

KPA Known-plaintext Attack

LDPC Low-density Parity Check

LFSR Linear-feedback Shift Register

LLR Log-likelihood Ratio

MAC Message Authentication Code

MDM Maximum Degree Monomial

MILP Mixed-integer Programming Problem

MPC Multi-party Computation

NFSR Nonlinear Feedback Shift Register

OFB Output Feedback

PKC Public-key Cryptography

PQC Post-quantum Cryptography

PRG Pseudorandom Generator

RAN Radio Access Network

SAGE Security Algorithms Group of Experts

SEI Squared Euclidean Imbalance

SIMD Single Instruction Multiple Data

SPA Sum-product Algorithm

SPN Substitution-permutation Network

TLS Transport Layer Security

UE User Equipment

USIM Universal Subscriber Identity Module

WHT Walsh Hadmard Transform

Contents

Part I

Overview of Research Field

Chapter 1

Introduction

When people make a phone call or visit a web page with a mobile phone or any portable device through the cellular network, the network takes charge of finding the call recipient or the gateway the visit request should be forwarded to. From 1G involving to 4G (also commonly called LTE), the cellular network has played an essential role in the last three decades in people's communication and life, and will continue to serve indispensably in 5G and the future IoT (Internet of Things) era where everything is connected.

When getting whatever service from the cellular network, one will always want the data securely protected over each sub-link of the transmission. This is achieved by using cryptographic mechanisms and protocols. The weakest security point over the link lies in the air interface, as the radio channel is open to everyone, including the malicious adversaries. The secure data transmission over the air is guaranteed through the 3GPP (3rd Generation Partnership Project) standardised confidentiality and integrity protection, which gets data encrypted and authenticated, thus protecting it from eavesdropping and manipulation.

The confidentiality and integrity protection in cellular networks starts from 2G and has been improving in each generation to provide stronger security and better performance to accommodate the involvement of the network. Now when we come to the 5G era, the innovative network architecture and high performance demands pose new requirements on the confidentiality and integrity protection. This thesis is thus motivated and mainly focuses on the 5G confidentiality and integrity algorithms, and we hope it can help provide some insight in this topic.

1.1 Introduction to Cryptography

Cryptography is the study of designing schemes, called *cryptographic primitives* or *ciphers*, and protocols to secure data communication in the presence of adversaries. Security mechanisms that rely on cryptography are an integral part of almost any smart electronic device and communication system. It is safe to say that without cryptography, we would never be able to enjoy the cyber world as we do today.

Cryptography has a long history dating back to thousands of years ago, but only after

the 1980s with the advances in areas like mathematics, computer science, and electrical engineering, it became widely available to ordinary users. Modern cryptography mainly consists of *symmetric primitives*, *asymmetric primitives*, and *unkeyed primitives*. These primitives are used in different applications, and from them more advanced protocols can be constructed. For a comprehensive introduction to cryptography, we refer to [MVOV18, SS16, KL20].

1.1.1 Symmetric Cryptography

Symmetric cryptography works in a secret-key setting. In such a setting, the two communicating parties, say, Alice and Bob, share some secret information, called the *secret key*, in advance through a secure channel. Such a secret key is used to encrypt the message, which is usually called *plaintext*, using an *encryption* algorithm and the output is called *ciphertext*; and the other side uses the same secret key to decrypt the ciphertext using the corresponding *decryption* algorithm and recover the plaintext. In practice, the two communicating parties share a secret key, sometimes called the *master key*, and employ some key derivation functions to generate sub-keys to protect each communication session. The master key is typically 128-bit or 256-bit long and shared through a secure channel, e.g., physically or cryptographically protected.

The symmetric primitives can be mainly categorised into *stream ciphers, block ciphers* and *MACs* (*message authentication codes*). Figure 1.1 presents a simple illustration of how they work.

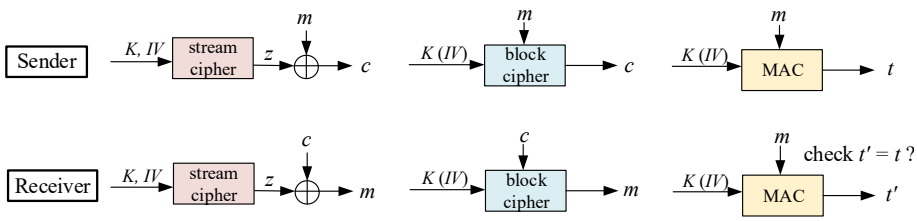

Figure 1.1: Symmetric primitives (K: secret key; IV: initialisation vector; m: plaintext; c: ciphertext; z: keystream; t and t': authentication tags).

1.1.1.1 Stream Ciphers

It is tricky to give an exact definition of a stream cipher, as there are some designs that perform like a stream cipher but can be categorised into other constructions according to their structures. Typically, in a stream cipher, a pseudorandom sequence, called *keystream*, is produced and then bit-wise XOR-ed with the plaintext to generate the ciphertext. The plaintext is not involved in the generation of the keystream sequence.

A stream cipher can be viewed as a loose instantiation of the *one-time pad* (OTP). An OTP is achieved by XOR-ing a completely random key with the plaintext, and the key should have at least the same length as the plaintext and never be used more than once. This is a strong requirement for practical implementation and restricts the usage of OTPs only in critical communications. A stream cipher combines a keystream

sequence with plaintext in a similar fashion as an OTP, but the keystream sequence is pseudorandom instead of completely random. As illustrated in Figure 1.1, a stream cipher takes a much shorter secret key, say 128 bits, and a public initialisation vector (IV) as inputs, and produces a pseudorandom keystream sequence which can be much longer. In practice, the length of a keystream sequence generated under one (key, IV) pair is usually limited to resist potential attacks. If more data needs encryption, new values are assigned to the IV to generate new keystream sequences. The security foundation of a stream cipher is that it is computationally infeasible to get the keystream sequence without knowing the value of the secret key, thus impossible to recover the plaintext.

Stream ciphers have the advantages of efficient implementations in hardware and software, zero error propagation, and low latency. Therefore, stream ciphers are widely used in applications that have unpredictable lengths of plaintext and require limited error propagation. For example, in mobile communications, data transmission can be initiated at any time and should be processed instantly; besides, an error happening over a bit position should not affect other bits. Thus the confidentiality and integrity algorithms used in cellular networks are mostly stream ciphers, e.g., SNOW 3G [3GP06] and ZUC [ETS11] used in 4G. This thesis mainly focuses on stream ciphers.

1.1.1.2 Block Ciphers

A block cipher iteratively applies a group of operations on the fixed-length groups of plaintext bits, called *blocks*, along with a secret key and possibly an IV to generate the ciphertext. This is the main difference to a stream cipher, i.e., the plaintext is involved in the state of the block cipher. Each application of the group of operations is called one *round*, and the number of rounds is specified. In each round, a *round key*, which is derived from the secret key according to specific *key schedule*, is introduced. Block ciphers are mainly built on two structures: the Feistel network and the SPN (substitution-permutation network).

In a Feistel cipher, the block of bits is split into two halves: a round function is applied to one half and the output of it is XOR-ed with the other half. The two halves are then swapped and used as the input of the next round. The round function consists of linear and non-linear operations to provide diffusion and confusion, respectively. The cipher DES (Data Encryption Standard) [SB88], once used as the federal encryption standard but later broken, is based on the Feistel structure.

In an SPN-based block cipher, each round is comprised of a *substitution layer* (S-layer) and a *permutation layer* (P-layer). The S-layer consists of several parallel S-boxes, which have the property that when one input bit is flipped, about half of the output bits will be flipped, correspondingly. Typically, the number of S-boxes is decided by dividing the block size by the size of the S-box. There are also some special constructions using a non-full S-layer with fewer S-boxes to reduce the number of AND gates, which is desired in applications like multi-party computation (MPC) and fully homomorphic encryption (FHE). For example, the block cipher LowMC [ARS+15] can adjust the number of S-boxes to accommodate different use cases. The P-layer permutes the block bits output from the S-layer and after that sends them to the S-layer in the next round. The P-layer has the property that the output bits of any S-box are distributed to as many S-boxes in the next round as possible. The most famous SPN cipher is AES (Advanced Encryption Standard) [DR99].

When using a block cipher to encrypt a variable-length message, the message should be partitioned into separate blocks of the same size, possibly with some paddings. A *mode of operation* is then chosen to specify how to process and combine each block. The commonly used modes are ECB (electronic codebook), CBC (cipher block chaining), CFB (cipher feedback), OFB (output feedback), and CTR (counter) modes. We refer to [MVOV18, SS16, KL20] for more details about how these modes work.

1.1.1.3 MAC

A message authentication code (MAC) is a short-length bit sequence used for message authentication: to confirm that the message comes from the stated sender and has not been manipulated during the transmission. The sender generates a MAC using a MAC scheme with a secret key, the plaintext, and possibly an IV as the inputs, and the receiver verifies it by comparing it with a local MAC generated using the same way. The receiver will accept the message only when the MACs are identical. The underlying cryptographic primitive for a MAC scheme can have various options, e.g., a block cipher or a hash function.

When the encryption and authentication of data are both required, the encryption is called *authenticated encryption* (AE), e.g., the confidentiality and integrity protection in cellular networks. The different combinations of encryption and MAC result in three main AE constructions: *Encrypt-then-MAC*, *Encrypt-and-MAC*, and *MAC-then-Encrypt*. If there are some associated data, e.g., a packet header, that is transmitted along with the ciphertext and must be authenticated as well, an *authenticated encryption with associated data* (AEAD) scheme is required. In 2013, the CAESAR competition [CAE] was announced to encourage new designs of AEAD schemes, and AEAD has now become a requirement in many applications.

1.1.2 Asymmetric Cryptography

The main advantage of symmetric cryptography is the high efficiency for processing large amounts of data. However, there are many modern communication scenarios that the communicating parities do not always share a secret key, or require more advanced cryptographic properties that symmetric primitives cannot provide, which motivates the application of asymmetric cryptography, or called public-key cryptography (PKC).

Different from a symmetric primitive using a shared secret key, two mathematically related keys are used in an asymmetric primitive: the *public key*, which is publicly known, and the *private key*, which can be never known by others except the owner. A sender can use a public key to encrypt data, and only the intended receiver holding the corresponding private key can decrypt it. Besides encryption, asymmetric primitives can be used to achieve more advanced cryptographic applications, e.g., digital signatures and non-repudiation protocols.

Asymmetric primitives are typically slower and more expensive than symmetric primitives as they are often built on hard problems, e.g., the most known *integer factoring problem* (IFP) and *discrete logarithm problem* (DLP). As a consequence, in many applications, a hybrid encryption mechanism involving both symmetric and asymmetric cryptography is adopted to ensure both efficiency and security. For example, an asymmetric primitive is used to encrypt a single-use symmetric key, and the key is used to

encrypt/decrypt the bulk of data with a symmetric primitive. TLS (Transport Layer Security) uses such a hybrid encryption mechanism. Another example is the 5G access: the UE uses the public key of the network to encrypt its identity information for the network side to have an initial verification of its legitimacy, and symmetric primitives are used to achieve subsequent mutual authentication, and confidentiality and integrity protection of data transmission.

Most of the asymmetric primitives we are using today can be broken by a large-scale quantum computer, as Shor's algorithm can solve the DLP and IFP in polynomial time [Sho94]. Post-quantum cryptography (PQC), built on mathematical problems resistant against quantum computing, has been widely studied and will be used in the foreseeable future. These quantum-safe algorithms can be mainly divided into five categories: *lattice-based*, *code-based*, *multi-variate*, *symmetric/hash-based*, and *isogeny-based*. In 2016, NIST initiated the PQC competition [NISb] to solicit, evaluate, and standardise quantum-resistant algorithms. After a three-round evaluation, 15 out of the initial 82 submissions entered the finalists or alternates (in 2020), which are regarded as promising PQC primitives. The 3GPP specification of 5G points to replace existing public key infrastructure with a quantum-safe one, which, however, will be a long-term evolution [3GP19].

1.1.3 Unkeyed Cryptography

There are some cryptographic primitives that do not require any secret key but still play essential roles in cryptographic mechanisms and protocols. The most used unkeyed primitives are *pseudorandom generators* (PRGs) and *hash functions*.

A PRG is a deterministic procedure that maps a shorter seed to a longer pseudorandom sequence that cannot be distinguished from random by any statistical test. This is a very generalised definition and any symmetric-key primitive can be viewed as an instantiation of it. PRGs have numerous applications in building cryptographic mechanisms or protocols, e.g., for generating a (pseudo)random number used as the IV in a symmetric-key primitive, as the challenge in a challenge-response authentication mechanism, or as a session key in a hybrid encryption scheme. If the seed and generated pseudorandom sequence are kept secret, a PRG in this setting is a symmetric-key primitive.

A hash function maps a message of arbitrary size to an output with a fixed size, which is usually called the *message digest*. If the hash function satisfies some cryptographic properties such as collision resistance, second preimage resistance, and preimage resistance, the message digest acts as a "fingerprint" of the message. Hash functions are commonly used in data storage, in which the message digest is used to index the corresponding data in a table. It can also be used to construct a MAC scheme or a signature.

1.2 Introduction to Cryptanalysis

Cryptanalysis is the study of analysing and evaluating cryptographic primitives. The cryptanalysis results are also called *attacks* and people who perform such analysis are called *attackers* or *cryptanalysts*. The strength of an attack is measured with *complexity* in terms of *time*, *memory*, and *data*. The time complexity is computed as the time needed to launch an attack, which is potentially the most important measure. It can be

expressed by the number of basic operations, where one basic operation may involve a reasonable number of bit operations or clock cycles. The memory complexity and data complexity denote the required amount of memory/storage and data, respectively. When a primitive is used in real-life applications, its implementation should be studied as well, as the *side-channel* attacks exploiting the information leakage of the implementation, e.g., the power consumption or running time, can recover some information about the secret key. We refer to [Jou09] for a comprehensive introduction to cryptanalysis.

The most straightforward attack is the *brute-force attack*, also known as *exhaustive key search*, in which the attacker tries all the possible values of the secret key. If the key has n bits, exhaustive key search will have complexity 2^n. Exhaustive key search is usually used as a benchmark of cryptanalysis results: if a cipher is claimed to provide an n-bit security level, there should not exist any attack with complexity below 2^n; otherwise, this cipher is considered as (at least theoretically) broken. Thus, when designing a new cipher, especially a symmetric one, one needs to visit every promising cryptanalysis technique and ensure that none of them applies to the cipher faster than exhaustive key search.

An attacker is usually assumed to have different capabilities and different attack goals, which are generalised as below.

1.2.1 Attack Goals

According to the different goals, attacks can be categorised into *key-recovery* attacks, *state-recovery* attacks, and *distinguishing* attacks, with a decreasing order in strength.

Key-recovery attacks aim to recover the secret key, which are the most powerful but typically difficult attacks. Exhaustive key search is a straightforward way of key recovery.

State-recovery attacks aim to recover the internal state of a cipher at a certain time instance. In a stream cipher, once the full state at a certain time instance is recovered, one can run the cipher forward and backward to get the whole keystream sequence under the specific key and IV. One may even recover the secret key if there is no special protection of it. In this case, a state-recovery attack is equivalent to a key-recovery attack.

Distinguishing attacks target to distinguish a cipher from random by identifying some non-randomness. Such non-randomness can be explored in various ways. Though not as powerful as key-recovery attacks, distinguishing attacks play an important role in cryptanalysis and the resistance against them has become a basic requirement for a cipher. In some cases, distinguishing properties can be used to recover some information of the secret key.

1.2.2 Attacking Assumptions

An attacker is usually assumed to have different capabilities when attacking a cipher, which correspond to the following different attacking scenarios.

Ciphertext-only attack (COA). An attacker only has access to the ciphertext, while not the plaintext. The attacker has the least capability in this attacking scenario, but the scenario is most relevant to the practical situation. For example, the transmitted (encrypted) messages can be eavesdropped in the open air in mobile communication.

Known-plaintext attack (KPA). An attacker is assumed to get a limited number of plaintext and ciphertext pairs. There are some practical scenarios connected to this attacking assumption. For example, some header messages in some protocols are publicly known and the attacker can get the corresponding plaintext and ciphertext. If the plaintext and ciphertext are known in the stream cipher setting, the keystream sequence is trivially known.

Chosen-plaintext attack (CPA). An attacker is able to choose plaintext and get the corresponding ciphertext by calling the encryption algorithm. A chosen-IV attack against stream ciphers can be relevant to this attacking scenario, where the attacker can choose different IV values to explore some possible weakness of the cipher.

Chosen-ciphertext attack (CCA). An attacker can choose arbitrary ciphertext and call the decryption algorithm to get the corresponding plaintext.

1.3 Security in Cellular Networks

Figure 1.2 presents a simplified illustration of the 4G system, which consists of UE (User Equipment), the radio access network (RAN), and the core network (CN). Other generations of cellular networks have the similar structure but possibly with different physical entities and interfaces.

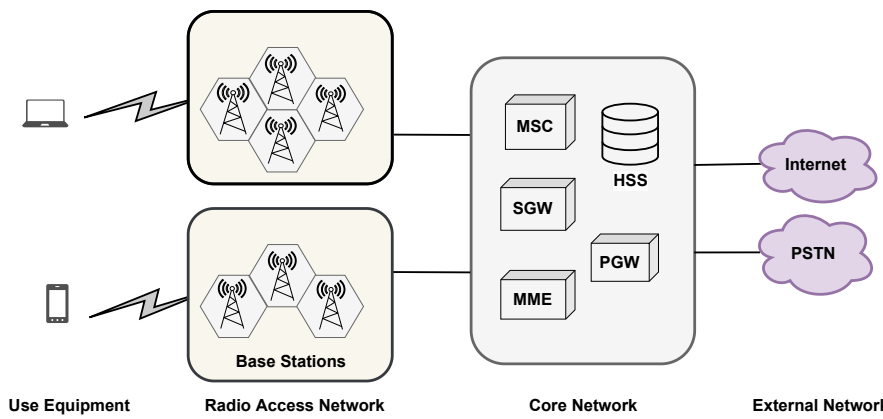

Figure 1.2: A simple illustration of the mobile network.

UE. The UE, which can be a mobile phone, a laptop, or any smart electronic device, has a globally unique identifier, commonly known as the IMSI (International Mobile Subscriber Identity). A secret key, called the *master key*, is shared between the UE and

the network. These user keying materials are usually assigned and burned to a physical card, commonly known as the USIM (Universal Subscriber Identity Module), in the manufacturing phase.

RAN. The RAN mainly consists of *base stations* (BSs), which are transceivers capable of sending/receiving and processing wireless signals. A BS consists of two modules: the RRU (Remote Radio Unit), which is typically placed close to the antennas and responsible for dealing with the radio-frequency signals; and the BBU (Base Band Unit), which is protected inside a special cabinet and responsible for central processing and control. The RAN connects to the UE through radio connection and to the CN through wired connection.

CN. The CN is the central part of a cellular network that provides services to the UE. These services can be cellular applications that are handled within the cellular network or other requests which are forwarded to external communication networks, e.g., the PSTN (Public Switched Telephone Network) and the Internet. The CN configures paths for the data transmission of these services between different RANs or different networks. There are a number of entities, e.g., HSS (Home Subscriber Server), MSC (Mobile Switch Center), SGW (Serving Gateway), PGW (Packet Data Network Gateway), and MME (Mobility Management Entity), residing in the CN and providing different network functionalities. For example, the HSS is basically a database that stores the keying materials of users, while the MSC is responsible for switching data packets.

The UE and the network use the master key to generate a hierarchy of sub-keys using the same key derivation function. At the network side, these sub-keys are generated in or distributed to different entities, e.g., the MME and BS. These sub-keys are used to protect data transmission over each sub-link. The secure data transmission over the wired part, e.g., between a RAN and the CN, or a CN with other networks, is achieved using IPsec (Internet Protocol Security) and/or TLS (Transport Layer Security). While over the air interface, the data transmission is secured using the *confidential and integrity protection*.

Confidentiality and Integrity Protection

The UE (or BS) encrypts the message and generates an authentication tag using a 3GPP standardised confidentiality and integrity algorithm, with a derived sub-key and an IV as inputs. The receiver, either the UE or BS, holding the same sub-key can correctly verify the integrity and decrypt the message, while any third party cannot achieve this even if he has eavesdropped the message over the air. The maximum length of plaintext encrypted under one (Key, IV) pair should not exceed 2^{64}. If there is more data to be encrypted, another IV value should be used.

The confidentiality and integrity protection in 2G was built around the stream cipher A5/1 and its variants [Qui04], the technical specifications of which were kept in secret until reverse-engineered in 1999. A number of severe weaknesses of A5/1 were identified which impeded its further adoption in 3G. The confidentiality and integrity protection in 3G is based on a block cipher KASUMI [ETS09] and a stream cipher SNOW 3G [3GP06]. SNOW 3G is kept in 4G, along with two new members: the block cipher AES (in CTR

Ciphers	Software Implementation (plaintext sizes)				Hardware Implementation		Attacks
	4096	2048	1024	256	Area	Throughput	
AES (256-bit)*	34.16 [EJMY19]	32.94	30.95	22.67	17232 GEs [UMHA16]	50.85 [UMHA16]	$2^{254.4}$ [BKR11]
SNOW 3G	8.89 [EJMY19]	8.50	7.81	5.38	18100 GEs [GCK13]	52.8 [GCK13]	2^{177} [YJM19]
ZUC	3.50 [AB15]	3.39	3.17	2.29	12500 GEs [LZM+16]	80 [LZM+16]	2^{236} [YJM20]

* The speeds of AES are for the 256-bit version, will be higher for AES-128.
Note: the implementations are under different platforms and resources.

Table 1.1: Some performance results of AES, SNOW 3G, and ZUC [YJ20] (columns 2-5 are the throughputs under different plaintext sizes; all throughputs are measured in Gbps and plaintext sizes are in bytes; hardware implementation area is measured in GE (Gate Equivalent)).

mode) [DR99] and another stream cipher called ZUC [ETS11]. All these algorithms except A5/1 use 128-bit keys.

These confidentiality and integrity algorithms are implemented in the UE and BSs, typically with special hardware support, e.g., using IP (intellectual property) cores, to provide high speeds. Table 1.1 presents some performance results of software/hardware implementations and best attacks achieved till now for these ciphers. One can see that the throughputs in hardware are typically very high, e.g., higher than 50 Gbps (Gigabits per second), while much lower in software. For example, the speeds for SNOW 3G and ZUC are both below 10 Gbps in software.

1.4 Security Challenges in 5G

5G is a highly heterogeneous network, with different access technologies coexisting and supporting various users requesting diversified services. There are three typical use cases in 5G with different features, specified as: 1) eMBB (enhanced mobile broadband) providing stable connections with very high data rates; 2) uRLLC (ultra Reliable Low Latency Communications) targeting low-latency transmissions of small payloads and ultra-high reliability; 3) mMTC (massive Machine Type Communications) supporting ultra-high device density and ultra-low energy consumption [Ser15]. To cope with these use cases with high requirements in different aspects, 5G has some fundamental changes in the network architecture and requires enhanced security compared to the legacy networks.

1.4.1 5G Architecture

The most significant evolution in 5G architecture lies in that it would be highly cloudified and virtualised. With the use of a bunch of virtualisation techniques and other enabling techniques such as SDN (software-defined networking) and NFV (network function virtualisation), different network functionalities can be virtualised as services in cloud [3GP21a]. Compared to the legacy cellular networks where a large variety of proprietary network entities and dedicated hardware appliances are deployed to pro-

vide different functionalities, the hardware in the cloudified system are general-purpose CPUs and memories, thus reducing the equipment and deployment cost and improving the flexibility for the management and evolution.

The access network can also be partly cloudified, and the resulting architecture is called *C-RAN* (Cloud/Centralized Radio Access Network). A C-RAN consists of distributed sites and a central cloud center, which is also viewed as the central BBU pool. The real-time functions, which mainly happen at the physical layer and lower MAC layer, e.g., access network scheduling, interference coordination, modulation, and coding, are still processed at the distributed sites with dedicated hardware support. In contrast, non-real-time functions in the upper layers with looser latency requirements, like intercell handover, cell selection/reselection, and user-plane encryption, can be moved to the cloud and implemented in software environments [Bro17]. Such a C-RAN architecture can reduce the cost of the deployment and management of the largely increased and densified base stations.

1.4.2 5G Security

A lesson has been learned from previous generations of cellular systems that security should be taken into account at the time of design, to avoid the "vulnerability-patching later" situation. The new architecture and performance requirements of 5G indicate new challenges to security, which require careful investigation and design. 3GPP has specified the security properties in 5G from different domains [3GP21b]. Generally, the security should be enhanced in several aspects: some new design and improvements in the security architecture and protocols are needed; security should be guaranteed at a higher level (e.g., 256-bit) due to the threat of quantum computers; the use of public-key cryptography may have to rely on new algorithms based on quantum-resistant problems; lightweight cryptographic algorithms and protocols will have to be analysed and adopted for resource-constrained IoT devices, etc [YJ20].

Confidentiality and Integrity Protection in 5G

3GPP has asked ETSI SAGE (Security Algorithms Group of Experts) to evaluate efficient confidentiality and integrity algorithms for 5G use [3GP19]. For the new algorithms, there are some desired properties driven from the requirements of 5G in terms of security level and speed.

1. The PDCP (Packet Data Convergence Protocol) layer, where the confidentiality and integrity protection is performed, is likely to be moved to the cloud and implemented in a software environment without specialised hardware support. Therefore, one expectation of the confidentiality and integrity algorithms is that they should provide a speed of **at least 20 Gbps in software**, which is the downlink peak data rate in 5G.

2. 3GPP is looking towards increasing the security level in 5G to 256 bits to resist against quantum computing [3GP19]. As a consequence, the confidentiality and integrity algorithms for 5G should be able to **provide security of 256 bits**.

There are two design routines for the confidentiality and integrity algorithms of 5G. One is to reuse existing algorithms in 4G, i.e., AES, SNOW 3G, and ZUC. The main advantage is that existing hardware circuits can be reused thus reducing upgrade cost, while the main drawback is that their performance in software might become a performance bottleneck for the cloudified system. As shown in Table 1.1, the algorithms cannot achieve sufficient speeds in software environments, except AES. Besides, their security under the 256-bit key length should be carefully investigated, as SNOW 3G and ZUC were only specified for the 128-bit key length. It might be risky if we just adopt current algorithms directly for the 256-bit security level without careful inspection. AES is highly likely to be kept in 5G, as it can achieve high performance in software due to the wide hardware support from mainstream CPUs, e.g., most Intel, AMD, and partly ARM processors, and support the 256-bit security level, which version has been widely used for a long time.

The other routine is to develop new algorithms with the performance and security requirements as the goals of the design. Such new algorithms would introduce update cost but can potentially have a longer service life in 5G and beyond, as the fact that the network in the future will be faster and more softwarised will not change. The old ciphers will be obsolete at some stage of the network evolution, if not in 5G, then highly likely in 6G. It is better to replace the algorithms earlier than later to make the system safer and more efficient.

1.5 Thesis Focus and Outline

This thesis mainly focuses on the confidentiality and integrity algorithms for 5G, which involves design and cryptanalysis of stream ciphers. Specifically, it targets to:

1. evaluate existing or new promising ciphers aiming for 5G confidentiality and integrity protection;

2. propose new ciphers for 5G confidentiality and integrity protection that satisfy 5G requirements in terms of security and speed.

Besides, we extend the topic a little bit, and

3. study the security of local pseudorandom generators,

which can be relevant to the confidentiality and integrity protection.

The thesis consists of two parts: the first part presents some background information and technical details which help readers to better understand the research field; and the second part contains six papers published during the Ph.D. education related to the research focus.

The outline of the first part is as follows. In Chapter 2, we review some technical preliminaries which are often used in design and cryptanalysis of a stream cipher. Chapter 3 and Chapter 4 present the popular constructions and cryptanalysis techniques of stream ciphers, respectively. Chapter 5 presents some basics and cryptanalysis techniques of local pseudorandom generators. Finally, we summarise the contributions of the thesis, draw the conclusions and discuss potential future work in Chapter 6.

Chapter 2

Technical Introduction

THIS chapter provides an overview of the concepts and tools that are commonly used in design and cryptanalysis of stream ciphers. Throughout the thesis, we use $\mathbb{F}_q$ or $GF(q)$ to denote a finite field of order q, and $\oplus$ to denote the bitwise XOR operation.

2.1 Complexity Theory Basics

Complexity theory is a central field of the theoretical foundations of computer science, concerned with the study of the *complexity of computational tasks*. By complexity, we are referring to the computational resources, which can be *time*, *space*, and *data*, required to solve a given task. We refer to the book [Gol08] for a comprehensive introduction to complexity theory. Based on the complexity, computational problems can be categorised into different *complexity classes.*

2.1.1 Complexity Classes

Complexity classes categorise problems based on how difficult they are to solve. The most often used measures are time complexity and circuit complexity.

Time Complexity Classes. The problems are categorised based on the number of steps taken by the algorithm to compute the problem. The most prominent complexity classes are `P` and `NP`, which include problems that can be solved and verified in polynomial time, respectively. Polynomial time means that the number of steps is upper bounded by a polynomial in the input size. Another very important complexity class is `NP-complete`, which include the hardest problems in `NP`.

Circuit Complexity Classes. An algorithm can be described using a *Boolean circuit*, which is a directed graph with vertices of three types: *input terminals*, *output terminals,* and *gates*. The gates denote Boolean operations, typically AND, OR, and unary NOT gates. When each gate has at most two incoming edges, the circuit is said to have *bounded fan-in*; otherwise, called *unbounded fan-in*. A Boolean circuit with n input

terminals and m output terminals defines a function from $\{0,1\}^n$ to $\{0,1\}^m$. For a given input value, the values of vertices are determined in a natural manner.

The two main measures of a circuit are the *size*, i.e., the number of gates, and the *depth*, i.e., the length of the longest directed path from an input to an output. These measures have a natural connection with time complexity, and problems can be correspondingly classified into different classes. The most common class is $\mathtt{P}/\mathrm{poly}$, including problems that can be solved by polynomial-size circuits. Two more interesting classes are $\mathtt{AC}$ and $\mathtt{NC}$. An $\mathtt{AC}$ construction has a circuit with polynomial size and polylogarithmic depth. $\mathtt{NC}$ is defined similarly to $\mathtt{AC}$, but can only have bounded fan-in. The special case $\mathtt{NC}^0$, in which each output depends on a constant number of input bits, is especially appealing as it can be computed very efficiently and may admit many advanced applications.

2.1.2 Complexity Evaluation

When we come to a more refined evaluation of the complexity, we can basically rely on three measures: asymptotic complexity, concrete complexity, and running-time complexity.

Asymptotic Complexity. Asymptotic complexity is usually used to identify the complexity of one class of problems when the input size n goes to infinity. Let $f(n)$ denote the number of operations required to run one task with an input size n, we use the following standard asymptotic notation.

- $f(n) = O(g(n))$ (resp., $f(n) = \Omega(g(n))$) if there exists a constant $C > 0$ such that $f(n) \leq C \cdot g(n)$ (resp., $f(n) \geq C \cdot g(n)$) for sufficiently large values of n.
- $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
- $f(n) = \tilde{O}(g(n))$ if $f(n) = O(\log^k(g(n))g(n))$ for a constant k.
- $f(n) = o(g(n))$ if for every constant $C > 0$, $f(n) < C \cdot g(n)$ for sufficiently large values of n.

Asymptotic complexity allows estimating the complexity of one class of problems when n goes large, which can help choose secure parameters when building new constructions. It is also the key to comparing different algorithms. However, asymptotic complexity typically cannot reflect the actual efforts required to compute a practical problem, and instead, the concrete complexity is considered.

Concrete Complexity. The concrete complexity measures the number of basic operations to finish the computation of a problem. However, it is tricky to define the basic operation, and one basic operation can consist of several or even a number of bit operations (or clock cycles). When comparing the complexities of two computing tasks, the efforts involved in each basic operation should be carefully considered.

Running-time Complexity. The running-time complexity measures the consumed time or clock cycles to compute a task. This measure is more relevant to the practice but

not particularly meaningful, since it heavily relies on the implementation techniques and hardware capabilities. However, the running time is still helpful to claim the complexity in some cases. For example, by specifying the running time for each basic operation in the concrete complexity, one can claim that the concrete complexity is reasonable.

2.2 Information Theory Basics

Let X be a discrete random variable that takes values from an alphabet $\mathcal{X}$, with a *distribution* determined by its probability mass function $P_X(x) = \mathbf{Pr}[X = x]$, where $\mathbf{Pr}[E]$ denotes the probability of occurrence of an event E. We can use *entropy* to measure the "uncertainty" of X given its distribution, which is defined as below:

$$H(X) = -\sum_{x \in \mathcal{X}} P_X(x) \log_2 P_X(x),$$

where $\log_2$ denotes the logarithm to the base 2 with the convention that $0 \cdot \log_2 \frac{0}{p} = 0$ and $p \cdot \log_2 \frac{p}{0} = \infty$ for $p > 0$.

If $P_X(x) = \frac{1}{|\mathcal{X}|}$ for all x values in $\mathcal{X}$, we say that X follows the *uniform random distribution* and achieves the maximum entropy in this case. Otherwise, we call the distribution of X *biased* and use the *bias* to evaluate how much it deviates from the uniform random distribution. There are many ways to define the bias, and below we give two examples for the binary and general cases.

Definition 1 (Bias)**.** If X is a binary variable, the bias ϵ of it can be expressed as

$$\epsilon = \mathbf{Pr}[X = 0] - \frac{1}{2}.$$

Thus, $\epsilon \in [-\frac{1}{2}, \frac{1}{2}]$, and $\mathbf{Pr}[X = 0] = \frac{1}{2} + \epsilon$, $\mathbf{Pr}[X = 1] = \frac{1}{2} - \epsilon$. To distinguish a binary distribution with bias ϵ from a uniform random distribution, that is to say, to distinguish a sequence in which elements are sampled from this given distribution from a uniform random sequence, the required number of samples, i.e., the length of the sequence, is in the order of $\frac{1}{\epsilon^2}$, up to some small constant factor [Mat93].

Lemma 1 (Piling-up Lemma [Mat93])**.** Let X_i be independent binary variables such that $\mathbf{Pr}[X_i = 0] = \frac{1}{2} + \epsilon_i$ for $i \in \{1, 2, \ldots, n\}$, then

$$\mathbf{Pr}[X_1 \oplus X_2 \oplus \cdots \oplus X_n = 0] = \frac{1}{2} + 2^{n-1} \prod_{i=1}^{n} \epsilon_i.$$

The piling-up lemma is widely used in cryptanalysis when computing the bias of the XOR sum of several independent variables.

Another parameter, called the log-likelihood ratio (LLR), of a binary variable X is also commonly used, which is defined as:

$$L_X = \log \frac{\mathbf{Pr}[X = 0]}{\mathbf{Pr}[X = 1]},$$

where log denotes the natural logarithmic function.

When X is non-binary, the *squared Euclidean imbalance* (SEI), is more commonly used to evaluate how far the distribution deviates from the uniform random distribution.

Definition 2 (Squared Euclidean Imbalance)**.** If X is a non-binary variable taking values from an alphabet $\mathcal{X}$, the squared Euclidean imbalance Δ of X is computed as

$$\Delta = |\mathcal{X}| \cdot \sum_{x \in \mathcal{X}} (P_X(x) - \frac{1}{|\mathcal{X}|})^2.$$

To distinguish a distribution of SEI Δ from a uniform random distribution, the required number of samples is in the order of $\frac{1}{\Delta}$, up to some small constant factor [Vau96, BJV04].

For two probability distributions, we can use *Kullback-Leibler (KL) divergence*, also called *relative entropy*, to express the distance between them [Cov99].

Definition 3 (Kullback-Leibler Divergence)**.** The Kullback-Leibler divergence between two probability distributions P_X and P_Y over the same alphabet is defined as

$$D(P_X||P_Y) = \sum_{x \in \mathcal{X}} P_X(x) \cdot \log \frac{P_X(x)}{P_Y(x)}.$$

The KL divergence is an important concept in cryptanalysis, as it measures the distance between two distributions and provides information about the number of samples that are needed to distinguish between them. The closer two distributions are, the smaller the KL divergence will be, and the more samples will be required. The distance is not in the strict sense, as it is not symmetric and does not satisfy the triangle inequality. A special case is when P_Y is the uniform random distribution, and one can evaluate how much the distribution P_X deviates from uniform random.

2.3 Hypothesis Testing

Hypothesis testing plays an important role in statistical analysis. One application of it is to distinguish between different distributions, which is widely used in cryptanalysis, especially in applications related to distinguishing attacks. We now describe the framework of the simplest form of hypothesis testing for distinguishing between two distributions, i.e., *binary hypothesis testing*.

Let P_0 and P_1 denote two different probability distributions defined over a same alphabet. Suppose $\mathbf{x}$ is a sequence of n sample symbols $\mathbf{x} = x_0, x_1, \ldots, x_{n-1}$, which are independently and identically distributed (*i.i.d.*) over a same distribution, either P_0 or P_1. From this sequence, we can define an *empirical distribution* or *sample distribution* P_X, whose entries are assigned with $\mathbf{Pr}[X = a] = N(a|\mathbf{x})/n$, where $N(a|\mathbf{x})$ denotes the number of occurrences of element a in the sequence $\mathbf{x}$. Now the task is to decide whether

P_X is following P_0 or P_1 using binary hypothesis testing. The problem can be modeled as below to decide between the *null hypothesis* H_0 and the *alternative hypothesis* H_1:

$$H_0 : P_X = P_0,$$
$$H_1 : P_X = P_1.$$

The decision can introduce two types of errors: *TYPE I* errors, where H_0 is rejected while it is true; and *TYPE II* errors, where H_0 is accepted while H_1 is true. Denote the probabilities of these two errors as α and β, respectively, then they can be expressed as:

$$\alpha = \mathbf{Pr}[\text{reject } H_0 | H_0 \text{ is true}],$$
$$\beta = \mathbf{Pr}[\text{accept } H_0 | H_1 \text{ is true}].$$

The Neyman-Pearson lemma provides the optimum decision in terms of minimising the error probabilities [Cov99].

Lemma 2 (Neyman-Person Lemma)**.** Let $X_0, X_1, \ldots, X_{n-1}$ be drawn *i.i.d.* according to a probability mass function P_X. Let P_0 and P_1 be two different distributions. Consider the decision problem corresponding to the two hypotheses $P_X = P_0$ *vs.* $P_X = P_1$. For $T \geq 0$, define a region

$$A_n(T) = \left\{ x_0, x_1, \ldots, x_{n-1} : \frac{P_0(x_0, x_1, \ldots, x_{n-1})}{P_1(x_0, x_1, \ldots, x_{n-1})} > T \right\}.$$

Let

$$\alpha^* = P_0^n(A_n^c(T)), \qquad \beta^* = P_1^n(A_n(T)),$$

be the error probabilities corresponding to the decision region $A_n(T)$, where $A_n^c(T)$ is the complement region. Let $B_n(T')$ be any other decision region given a different T' with associated error probabilities α and β. If $\alpha \leq \alpha^*$, then $\beta \geq \beta^*$.

The Neyman-Person lemma indicates that the optimum test for two hypotheses is of the form $\frac{P_0(x_0, x_1, \ldots, x_{n-1})}{P_1(x_0, x_1, \ldots, x_{n-1})} > T$, which is called the *likelihood ratio test* and can be rewritten in a log-likelihood ratio form as below:

$$L(x_0, x_1, \ldots, x_{n-1}) = \log \frac{P_0(x_0, x_1, \ldots, x_{n-1})}{P_1(x_0, x_1, \ldots, x_{n-1})}.$$

After some derivations [Cov99], the log-likelihood ratio can be expressed as:

$$L(x_0, x_1, \ldots, x_{n-1}) = nD(P_X || P_1) - nD(P_X || P_0). \tag{2.1}$$

Then the log-likelihood ratio test is derived as:

$$D(P_X || P_1) - D(P_X || P_0) > \frac{1}{n} \log T. \tag{2.2}$$

When T is set as 1, the test is simplified to check the difference between the KL divergences of the sample distribution to each of the two distributions, and decide that it follows the distribution with which the KL divergence is smaller.

Number of Required Samples

There are no universal expressions for α and β, however, the asymptotic expression of β can be linked to the KL divergence through the *Chernoff-Stein lemma* as shown below:

$$\lim_{n\to\infty} \frac{\log \beta}{n} = -D(P_0||P_1),$$

when α is fixed. Thus one can get:

$$\beta \approx 2^{-nD(P_0||P_1)}. \tag{2.3}$$

To restrict β at a desired level, the number of samples n should satisfy:

$$n = O(\frac{1}{D(P_0||P_1)}). \tag{2.4}$$

Equation (2.4) is usually used as the measure of the required number of samples to distinguish between two distributions. For more details of hypothesis testing, we refer to [BJV04, Cov99].

2.4 Coding Theory Basics

Coding theory is the study of codes, including the designs, properties, decoding techniques, and applications. Over the past decades, there has been substantial progress in coding theory, and codes are now widely used in many applications like *data compression*, *channel coding* (*error control*), *cryptographic coding*, and *data storage*. There are several different classes of codes, and linear codes are most widely used. In this section, we give a brief introduction to linear codes and popular decoding techniques, and refer to [RL09, LC01] for more details.

2.4.1 Linear Codes

Definition 4 (Linear Code)**.** An $[n, k]_q$ linear code $\mathcal{C}$ is a linear subspace of the vector space $\mathbb{F}_q^n$ with dimension k. The rate of the code is $\frac{k}{n}$.

When $q = 2$, the linear code is called a *binary code*. Linear codes are traditionally partitioned into block codes and convolutional codes [RL09].

A linear code is also characterised by the *generation matrix* and the *parity-check matrix*, defined as below.

Definition 5 (Generation Matrix)**.** The generation matrix $\mathbf{G}$ of an $[n, k]_q$ linear code $\mathcal{C}$ is a $k \times n$ matrix in $\mathbb{F}_q$ whose rows form a basis of $\mathcal{C}$, i.e.,

$$\mathcal{C} = \{\mathbf{c} : \mathbf{c} = \mathbf{uG}; \mathbf{u} \in \mathbb{F}_q^k\}.$$

Thus, the code $\mathcal{C}$ defines a linear mapping $f : \mathbb{F}_q^k \to \mathbb{F}_q^n$. The $\mathbf{u}$ vectors are usually referred to as *information symbol* vectors and the $\mathbf{c}$ vectors are called *codewords*. Code $\mathcal{C}$ has q^k codewords and any linear combination of codewords is again a codeword.

Definition 6 (Parity-check Matrix.)**.** The parity-check matrix $\mathbf{H}$ of an $[n,k]_q$ linear code $\mathcal{C}$ is an $(n-k)\times n$ matrix in $\mathbb{F}_q$ whose null space is $\mathcal{C}$, i.e.,

$$\mathcal{C}=\{\mathbf{c}\in\mathbb{F}_q^n : \mathbf{c}\mathbf{H}^T=\mathbf{0}\}.$$

Each column of $\mathbf{H}$ denotes one information symbol while each row denotes one parity check. The number of non-zero elements in a column indicates the number of parity checks that this information symbol is involved in, while the number of non-zero elements in a row indicates the number of information symbols involved in this check. These checks are all linear. If follows that $\mathbf{G}\mathbf{H}^T=\mathbf{0}$.

There are many different types of linear codes, and in recent years, the low-density parity check (LDPC) code, along with the iterative decoding algorithms for it, has received much study and abroad applications.

LDPC Codes. LDPC codes are one class of linear block codes which can provide rates extremely close to the Shannon capacity. It has been increasingly used in applications requiring reliable and highly efficient communication, e.g., for channel coding in 5G.

A binary (n,k) LDPC code is usually defined by the null space of the $(n-k)\times n$ binary parity-check matrix $\mathbf{H}$. The density of one's in $\mathbf{H}$ is sufficiently low to permit effective iterative decoding (will be elaborated in next subsection), thus getting the name.

An LDPC code can also be represented by a *Tanner graph*, which is a bipartite graph with nodes separated into two types and connected by edges. The two types of nodes are called *variable nodes* (VN) and *check nodes* (CN), which represent the information symbols and the parity checks, respectively. The VN j is connected to the CN i if $h_{ij}=1$, where h_{ij} denotes the entry of $\mathbf{H}$ in the i-th row and j-th column. The Tanner graph can help to describe iterative decoding, see next subsection.

2.4.2 Decoding Techniques

In a communication system, the information symbols are encoded into codewords and transmitted over a noisy channel; the receiver receives a noisy version of codewords and based on them recovers the transmitted information symbols, which is called the *decoding* process. The process may involve correcting some errors and the error-correcting capability is highly connected to how the code is constructed. There exist many different decoding techniques with respect to different criteria and are suitable for different codes.

2.4.2.1 Optimal Decoding

Some decoding approaches are regarded as *optimal*, by which we mean that the decoding error probabilities are minimised by using these decoding techniques. These optimal decoding approaches include minimum distance (MD) decoding, maximum likelihood (ML) decoding, and a posteriori probability (APP) decoding.

- **MD Decoding.** An MD decoder outputs the information symbol vector $\hat{\mathbf{u}}$ corresponding to the codeword $\hat{\mathbf{c}}$ that is closest to the received noisy codeword $\mathbf{r}$ in terms of the Hamming distance, i.e.,

$$\hat{\mathbf{c}}=\arg\max_{\mathbf{c}\in\mathcal{C}} d_H(\mathbf{c},\mathbf{r}).$$

- **ML Decoding.** An ML decoder outputs the information symbol vector $\hat{\mathbf{u}}$ corresponding to the codeword $\hat{\mathbf{c}}$ that maximises the likelihood function $p(\mathbf{r}|\mathbf{c})$, i.e.,

$$\hat{\mathbf{c}} = \arg \max_{\mathbf{c} \in \mathcal{C}} p(\mathbf{r}|\mathbf{c}).$$

  The most well-known application of ML decoding is the Viterbi algorithm [HH89].

- **APP Decoding.** An APP decoder outputs the posterior probabilities $p(u_i|\mathbf{r})$ for all symbols $u_i \in \mathbf{u}$, i.e.,

$$p(u_i = x|\mathbf{r}) = \sum_{\mathbf{u}:u_i=x} p(\mathbf{u}|\mathbf{r}), x \in \{0, 1\}.$$

  One can see that an APP decoder provides *soft outputs* (the posterior probabilities) of the output symbols instead of the *hard decisions* (i.e., 0 or 1). These soft outputs can be passed to other components for further processing. One well-known application of APP decoding is the BCJR (Bahl, Cocke, Jelinek, and Raviv) algorithm [McE96].

2.4.2.2 Sub-optimal Decoding

In practice, there are a large variety of *sub-optimal* decoding approaches that relax optimality and trade the performance against complexity. The most well-known example is the *iterative decoding*, which is widely used in many applications from different areas, e.g., for decoding LDPC codes.

In iterative decoding, the variable nodes and check nodes iteratively exchange information until reaching some stopping conditions. Each edge acts like a bus that conveys the information, which is typically probabilistic information, e.g., probabilities or LLRs. Each node, either a variable node or check node, acts like a local computing processor, which receives information from the connected nodes, updates new information, and sends the updated information back. The process is expected to converge after a certain number of iterations, and the information symbols are recovered according to their probabilities.

There are many variants of iterative decoding, and the *sum-product algorithm* (SPA), also called *belief propagation*, is a general one that provides near-optimal performance. Below we present the main steps of the SPA based on LLRs, and refer to [RL09, HOP96] for more details.

1. **Initialisation.** For every variable v, initialise its LLR value according to its a-priori LLR value $L_a(v)$, e.g., received channel LLR value, i.e., $L_v^{(0)} = L_a(v)$. For every edge connected to v, initialise the conveyed LLR value as $L_v^{(0)}$. After the initialisation, the decoder enters the iterations of exchanging information repeating steps 2 - 4.

2. **CN update.** In each iteration i, every check node c computes an outgoing LLR value over each of its edges e_k, based on the incoming LLR values updated in the

$(i-1)$-th iteration from every *other* edge e'_k connected to c, as below:

$$L_c^{(i)}(e_k) = 2\tanh^{-1}\left(\prod_{k'\neq k}\tanh\left(1/2L_v^{(i-1)}(e'_k)\right)\right), \tag{2.5}$$

where tanh() corresponds to the hyperbolic tangent function.

3. **VN update.** In each iteration i, every variable node v computes an outgoing LLR value over each of its edges e_j, based on the a-priori information and LLR values updated in the i-th iteration from every *other* edge e'_j connected to v, as below:

$$L_v^{(i)}(e_j) = L_a(v) + \sum_{j'\neq j} L_c^{(i)}(e_{j'}). \tag{2.6}$$

4. **Distribution update.** After each iteration, update the LLR value of every variable v using (2.6), but with every edge included, i.e.,

$$L_v^{(i)} = L_a(v) + \sum_{j} L_c^{(i)}(e_j). \tag{2.7}$$

An intermediate value for every variable v can be derived based on $L_v^{(i)}$: $v = 1$ if $L_v^{(i)} < 0$; otherwise, $v = 0$. The correctness of intermediate result can be easily verified by checking if $\mathbf{vH}^T = \mathbf{0}$. If correct, the algorithm immediately terminates; otherwise, it continues with a new iteration until reaching the maximum allowed iterations.

2.5 Fourier Transform

Fourier transform is a mathematical operation that transforms a function in a generalised time domain to the frequency domain. It is commonly used in signal processing, especially in applications that involve cumbersome computations in the original domain while the computation can be performed much faster in the frequency domain. For example, the convolution operation in the time domain only involves simple multiplication when processed in the frequency domain. Therefore, one can transform the data to the frequency domain, perform the operations there and finally transform it back using the inverse Fourier transform. There are many different forms of Fourier transform suitable for various applications, and below we give a brief introduction to the most commonly used ones in cryptanalysis: the discrete Fourier transform (DFT) and the Walsh Hadamard transform (WHT), which can be used for, e.g., analysing a Boolean function [PB06] and computing the bias of a composite noise [MJ05, LD16]. For more details about Fourier transform, we refer to the book [BB86].

2.5.1 Discrete Fourier Transform

DFT converts a finite-length sequence into a same-length sequence in the frequency domain, which is defined as below.

Definition 7 (Discrete Fourier Transform)**.** Suppose $\mathbf{x}$ is a length-N sequence $\mathbf{x} = x_0, x_1, \ldots, x_{N-1}$, its DFT is another length-N sequence of complex numbers $\mathbf{X} = X_0, X_1, \ldots, X_{N-1}$ each computed as follows:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn},\ k \in \{0, 1, \ldots, N-1\},$$

where $e^{-\frac{i2\pi}{N}}$ is an N-th primitive root of unity.

DFT is the most important discrete transform used in many practical applications, e.g., for processing the sampled sequences of signals. Directly computing DFT according to the definition requires complexity $O(N^2)$. However, there exist efficient implementations of DFT, called fast Fourier transform (FFT), that reduce the complexity from $O(N^2)$ to $O(N \log N)$ [BB86].

2.5.2 Walsh Hadmard Transform

Walsh Hadamard transform (WHT) is a special variant of DFT, which operates on a sequence of real numbers.

Definition 8 (Walsh Hadmard Transform)**.** Suppose $\mathbf{x}$ is a length-N sequence $\mathbf{x} = x_0, x_1, \ldots, x_{N-1}$, its WHT is another length-N sequence $\mathbf{X} = X_0, X_1, \ldots, X_{N-1}$, each element of which is computed as follows:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot (-1)^{k \cdot n},\ k \in \{0, 1, \ldots, N-1\},$$

where $k \cdot n$ denotes the bitwise dot product of the binary representations of k and n.

Each X_k has a real value. WHT can also be represented in a matrix way using Hadamard matrices. Similarly, there exist fast implementations of WHT, called fast WHT (FWHT), that speed up the computation and reduce the complexity from $O(N^2)$ to $O(N \log N)$ [LD16].

There are many properties of DFT and WHT like linearity, shifting in both domains, and scaling, that can be explored to help process data. One important property is Parseval's theorem, which applies to both DFT and WHT.

Definition 9 (Parseval's Theorem)**.** Suppose the length-N sequence $\mathbf{X}$ is the DFT or WHT of the length-N sequence $\mathbf{x}$, then the relation below holds:

$$\sum_{n=0}^{N-1} |x_n|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X_k|^2.$$

X_0 in the frequency domain in DFT or WHT is a special point, since its value equals to the sum of all elements in the original domain, i.e., $X_0 = \sum_{n=0}^{N-1} x_n$.

2.6 Boolean Functions and S-boxes

A Boolean function f in n variables is a map from $\mathbb{F}_{2^n}$ to $\mathbb{F}_2$, where n is called the *arity* of the function. When the output has more than one bit, say m, i.e., $f : \mathbb{F}_{2^n} \to \mathbb{F}_{2^m}$, the function is called a vectorial Boolean function, or an S-box. Boolean functions and S-boxes serve as fundamental concepts in cryptography, and their cryptographic properties are closely connected to the resistance of a cipher against different cryptographic attacks [CS17, CCH10, PB06]. Therefore, a deep understanding of Boolean functions and S-boxes helps to better design and evaluate a cipher.

2.6.1 Boolean Functions

2.6.1.1 Representations

A Boolean function can be represented in many ways, and the most commonly used are *algebraic normal form* (ANF), *truth table*, and *digital circuit* [O'D14].

ANF. A Boolean function f in n variables can be expressed as a polynomial as below:

$$f(x_1, x_2, \ldots, x_n) = a_0 \oplus a_1 x_1 \oplus a_2 x_2 \oplus \cdots \oplus a_{1,2,\ldots,n} x_1 x_2 \cdots x_n. \tag{2.8}$$

Each term is called a *monomial* and the a values are the corresponding monomial coefficients, which can be either 1 or 0. The maximum degree monomial (MDM) coefficient $a_{1,2,\ldots,n}$ serves as an important tool for analysing the initialisation process of a stream cipher, and we will provide more details in Section 4.3.

Truth Table. Another form to represent a Boolean function is using the truth table, which stores all the possible values of the inputs and the corresponding outputs. Based on the truth table, one can trivially recover the ANF.

Digital Circuit. A Boolean function can also be represented as a digital circuit, with the input terminals 0, 1 and the n variables $x_1, x_2, \ldots, x_n$.

2.6.1.2 Cryptographic Properties

The most crucial properties of Boolean functions that should be carefully considered when used in cryptographic primitives are *balancedness*, *algebraic degree*, *correlation immunity*, *algebraic immunity*, and *avalanche criterion*.

Balancedness. A Boolean function in n variables is said to be balanced if its output is uniformly distributed over $\{0, 1\}$, i.e., exactly 2^{n-1} input values produce an output of value one. Boolean functions used in cryptographic applications almost always need to be balanced.

Algebraic degree. The *algebraic degree* of f, denoted by $\deg(f)$ or just d, is the number of variables in the maximum monomial term whose coefficient is non-zero. When $\deg(f) \leq 1$, the function is called *affine* and a non-constant affine function is called *linear*.

When a Boolean function serves as a source of non-linearity in a cipher, it should have a sufficient high degree.

Correlation immunity [Sie84]. A Boolean function f in n variables is said to be correlation immune of order k, $1 \leq k \leq n$, if the output is statistically independent of any subset of k input variables. A balanced Boolean function with k-order correlation immunity is called a k-resilient function. High correlation immunity orders should be guaranteed; otherwise, a construction could be susceptible to correlation attacks. However, there is always a tradeoff between the correlation immunity order and algebraic degree of a Boolean function. Specifically, the degree d and correlation immunity order k satisfy the constraint below [Sie84]:

$$k + d \leq n. \tag{2.9}$$

Such a tradeoff is disappointing since a high algebraic degree and high correlation immunity order are both desired in most cases. Fortunately, the constraint can be eliminated by adding memory elements in a Boolean function [Rue85, MS92, Gol96b], such that the output depends not only on the current input but also on some previous inputs, which actually forms a finite state machine. Such an idea has almost become a design criterion of stream ciphers, and most modern stream ciphers are adopting it.

Algebraic Immunity [MPC04]. The algebraic immunity of a Boolean function f in n variables is defined as the lowest degree of the Boolean function $g : \mathbb{F}_2^n \to \mathbb{F}_2$ for which $fg = 0$ or $(f \oplus 1)g = 0$. The function g satisfying $fg = 0$ is called an *annihilator* of f. The algebraic immunity is closely connected to the resistance against algebraic attacks.

Avalanche Criterion [For88]. The avalanche criterion of a Boolean function says that complementing any one of the n input bits results in the output being changed for exactly half of the 2^{n-1} possible values of the remaining input bits. This is a very useful and important property, and almost all the Boolean functions in cryptographic applications satisfy this criterion.

WHT and Walsh transforms are frequently used to investigate a Boolean function, which are defined as below.

Definition 10 (WHT of a Boolean function.)**.** Suppose $f : \mathbb{F}_2^n \to \mathbb{F}_2$ is a Boolean function, the WHT of f is defined as

$$\hat{f}(w) = \sum_{x \in \mathbb{F}_2^n} f(x)(-1)^{x \cdot w},$$

where $x \cdot w$ denotes the inner product of x and w.

If WHT is applied to the sign function of f, i.e., $(-1)^f$, the result is the Walsh transform of f.

Definition 11 (Walsh transform of a Boolean function.)**.** Suppose $f : \mathbb{F}_2^n \to \mathbb{F}_2$ is a Boolean function, the Walsh transform of f is defined as

$$\mathcal{W}_f(w) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x) \oplus x \cdot w},$$

where $x \cdot w$ denotes the inner product of x and w.

Many properties of Boolean functions can be investigated through WHT or Walsh transform, and we refer to [CS17, CCH10, PB06] for more details.

2.6.2 S-boxes

A vectorial Boolean function, or S-box, $F : \mathbb{F}_{2^n} \to \mathbb{F}_{2^m}$, can be represented by a vector $(f_0, f_1, \ldots, f_{m-1})$, where each f_i $(0 \leq i \leq m-1)$ is a Boolean function and is called the *component* of the S-box. Typically, $m = n$ for S-boxes used in cryptographic primitives. The cryptographic properties of an S-box are closely related to its components. For example, an S-box is balanced if and only if its component functions are balanced; its algebraic degree is the maximal algebraic degree of its component functions.

The S-box is commonly used in symmetric ciphers, which provides one primary source of non-linearity. The design of an S-box is crucial for a cipher, which requires taking many aspects into consideration, e.g., most critically, the cryptographic properties. The implementation aspects should be taken into consideration as well. For example, the size of the S-box will affect the implementation cost. In AES and many large ciphers, the S-boxes are typically byte-based, while for lightweight ciphers, the sizes of S-boxes are usually smaller, e.g., 5-bit, 4-bit or even 3-bit. One efficient way to implement an S-box is to use the look-up table (LUT), which stores all the possible inputs and corresponding outputs. However, LUT usually requires more storage and is the operation leaking the most information; thus, a side-channel-resistant implementation prefers to use bit-sliced techniques, which use bit variables and express the function using single-bit logical operations [MM12].

2.7 Mixed-integer Linear Programming

A mixed-integer linear programming (MILP) problem is a mathematical optimisation problem that consists of one objective function and a group of constraints in a number of variables. The task is to find a solution satisfying these constraints that maximises the objective function (a minimisation objective can be rewritten as a maximisation objective). If no objective function is given, the problem turns into a feasibility problem. The variables can be discrete or non-discrete, while the objective function and the constraints, which can be equations or inequations, should all be linear, thus getting the name. A MILP problem can be expressed in the following standard formulation:

$$\begin{aligned} \text{maximise} \quad & \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \begin{Bmatrix} \geq \\ = \\ \leq \end{Bmatrix} \mathbf{b}, \\ & \mathbf{x}_{\min} \leq \mathbf{x} \leq \mathbf{x}_{\max}. \end{aligned}$$

The problem can be either *infeasible* when there is no solution; *unbounded* when no finite solution exists; or *feasible* in which case an optimal solution $\mathbf{x}^*$ maximising the objective

function can be derived and returned. There can be more than one optimal solution. In practice, many optimisation solvers like Gurobi [Gur] can be used to solve a MILP problem efficiently.

MILP has diversified applications in different areas, and in recent years, it has been widely used in cryptanalysis, e.g., to explore linear/differential/integral trails used for distinguishing attacks or cube attacks [SHW+14, XZBL16, FWG+16]. In such cryptanalysis, each operation in the cipher can be modeled as some constraints based on specific rules; with the help of an optimisation solver, one can solve the problem very efficiently thus be able to launch an attack. One basic but useful application is to find the minimum number of involved S-boxes [MWGP11] in an attack, and one can have a rough estimation of the nonlinearity and algebraic degree of a cipher.

2.8 Solving Equation Systems over Finite Fields

Soling a system of m equations in n variables over a finite field is a basic algorithmic problem with direct applications in cryptanalysis. In the symmetric regime, a stream cipher, block cipher, or hash function, can be expressed through a system of equations over a finite field with a solution that yields information about the secret key. For example, in algebraic attacks and guess-and-determine attacks, some equations corresponding to specific operations involving the key or state variables can be derived, and by solving the equation system, one can possibly recover the key or state variables. In the asymmetric regime, the security of some cryptographic primitives depends on the system solving problem, e.g., the multivariate public-key cryptosystems [DY09].

When $m > n$, the system is called *overdetermined*, and *underdetermined* when $m < n$. The equation system can be linear or non-linear, and there are different solving methods for them.

2.8.1 Solving Linear Systems

Given a matrix $\mathbf{A}$, its *rank* is the maximal number of linearly independent columns. The *null space*, also called *kernel*, is the set of all solutions of the equations $\mathbf{Ax} = \mathbf{0}$, the dimension of which is called *nullity*. For any matrix, the *rank-nullity theorem* always holds [Ala07].

Theorem 1 (rank-nullity theorem)**.** For an $m \times n$ matrix $\mathbf{A}$,

$$\text{rank}(\mathbf{A}) + \text{nullity}(\mathbf{A}) = n,$$

where rank() and nullity() return the rank and nullity of a matrix, respectively.

A linear system of m equations in n variables can be expressed as:

$$\mathbf{Ax} = \mathbf{b}, \tag{2.10}$$

where $\mathbf{A}$ is the $m \times n$ coefficient matrix, $\mathbf{x}$ is the variable vector and $\mathbf{b}$ is a constant vector of length n. The *rank-nullity theorem* can be used to determine the number of solutions (suppose the system is solvable): if $\text{rank}(\mathbf{A}) = n$, there is a unique solution; while if $\text{rank}(\mathbf{A}) = r < n$, there are $n - r$ independent solutions.

The most commonly used method to solve a linear system is *Gaussian elimination* requiring complexity $O(n^\omega)$, where $\omega \leq 2.376$ in theory. However, the (neglected) constant factor in this complexity is expected to be very large, and in practice, one should rather consider that ω is closer to 3 [Cou04b]. The Gaussian elimination can be similarly used for computing the determinant, the inverse, and the rank of a matrix. When the system is sparse, there are more efficient ways like Wiedemann algorithm [Wie86] with complexity $O(n^2)$. Iterative decoding is also widely used to solve large, sparse, linear systems due to its little space overhead.

2.8.2 Solving Non-linear Systems

The system becomes non-linear when the degrees of the (polynomial) equations are larger than one. The most well-known methods used to solve a non-linear system are *linearisation based algorithms* and *Gröbner basis algorithms*. Below we present a brief introduction to these algorithms and refer to [Alb10] for more details.

Linearisation based Algorithms. The basic idea of linearisation is to assign a new unknown variable for each of the monomials appearing in the system and solve the derived linear system using, e.g., Gaussian elimination [KS99]. If the degrees of the original equations are upper bounded by d, at most $\sum_{i=1}^{d} \binom{n}{i}$ ($\gg n$) new variables will be introduced. The complexity is then around $\binom{n}{d}^\omega$, where ω is the exponent of Gaussian elimination. When employing linearisation methods to solve a non-linear system, the system should be overdetermined, specifically, the number of equations is approximately the same as the number of monomials in the system.

There are a number of improving algorithms targeting to generate more linearly independent equations to improve the performance. The most publicised one is the XL (eXtended Linearisation) algorithm [CKPS00]. In the XL algorithm, one can generate many higher-degree polynomial equations by multiplying each equation with all possible monomials of some bounded degree, then linearise the expanded system and solve it. Other improved algorithms, e.g., XSL and MutantXL, exploiting special properties, e.g., sparsity, can perform better in certain cases.

Gröbner basis algorithms. Gröbner basis algorithms solve systems of equations through constructing Gröbner bases. During the construction of the Gröbner bases, variables are eliminated successively in some predefined order, and in the end, one univariate equation can be obtained and solved. Then one iteratively substitutes the values of the known variables backward to solve more variables, until all variables are known. The most well known Gröbner basis algorithms are like F_4 [Fau99] and F_5 [Fau02].

Since optimised methods are used to generate polynomials, the performance of Gröbner basis algorithms is at least as good as the linearisation methods. When the number of equations is close to the number of unknowns, Gröbner basis algorithms typically have more advantages.

2.9 Hardware and Software Implementations

In this section, we give a brief introduction to hardware and software implementations, and refer to [HP11, KHF10] for more details.

A program, say a cryptographic algorithm, can be implemented in a hardware environment or software environment. The running time of it can be expressed as:

$$\text{running time} = \text{CPU clock cycles for a program} \times \text{clock cycle time},$$

where the clock cycle time is the time duration of one clock cycle in the given CPU. For example, for a CPU working at 1 GHz (Gigahertz), one clock cycle time is $\frac{1}{1\ GHz} = 1$ ns (nanosecond). The performance of an implementation is largely influenced by the hardware capability and software organisation of the computing environment. The different application requirements have led to five different computing environments equipped with different capabilities, both in software architecture and hardware resources. These five computing environments are *embedded computers*, *personal mobile devices*, *desktop computers*, *servers*, and *clusters/warehouse-scale computers*.

A cryptographic algorithm can be implemented in different ways in these different computing environments. For example, in an embedded system or a PMD, the cryptographic algorithm is usually implemented in hardware, while in a server, it is likely to be implemented in software, especially in the future highly cloudified system. An implementation always needs to balance between performance and cost. When comparing two implementations, e.g., the performance gain of implementation Y by using some particular features over implementation X, the *speedup* is usually used, which is defined as below:

$$\text{speedup} = \frac{\text{running time of X}}{\text{running time of Y}}.$$

2.9.1 Hardware Implementations

A cryptographic algorithm can be implemented in hardware using special digital circuits, e.g., an IP core, to execute precisely this cryptographic algorithm. Such a core can be regarded as a *single-purpose* processor. High performance can be achieved because of the dedicated hardware support, but the design efficiency and flexibility are low. Therefore, some functionalities which are frequently used or have strict requirements in latency or performance can be implemented with special hardware. For example, the confidentiality and integrity algorithms are usually implemented in hardware in cellular networks, e.g., in the mobile devices and the CU part of a base station. The hardware implementations allow for timely encrypting large messages, which is very important for time-sensitive services, such as voice calls. There are several metrics that should be considered in a hardware implementation, e.g., *circuit size*, *throughput*, *latency* (corresponds to the time taken to obtain the output), and *power consumption* (the amount of power needed to use the circuit). An implementation needs to balance between these metrics according to different contexts and requirements.

The hardware efficiency is also relevant to *hardware acceleration*, where some dedicated intrinsic instructions for some operations are included on a processor. For example,

many CPUs have included the instructions for performing an AES round in very few clock cycles.

2.9.2 Software Implementations

In 5G and beyond, the computer network and mobile network are expected to be highly cloudified, in which the programs, or applications, are running over *general-purpose* processors. These general-purpose processors typically hold a large register file and general-purpose computing units without digital circuits for a dedicated program. Such a system allows for efficient and flexible design and update, which only require dealing with the programs.

For a software implementation, the *RAM consumption*, corresponding to the amount of data written to the memory, *code size*, and *throughput* should be considered. These metrics highly depend on the device's system architecture, e.g., the supported registers and intrinsic instruction sets.

SIMD (single instruction, multiple data) Instructions. SIMD is a type of parallel processing that performs the same operation on multiple (typically two to eight) items of data simultaneously. It enables a desktop and server processor to achieve significant performance speedup.

The SIMD instruction extensions started with the MMX (Multimedia Extensions) in 1996, followed by several SSE (Streaming SIMD Extensions) versions in the next decade, e.g., SSE (1999), SSE2 (2001), SSE3 (2004), and SSE4 (2007), and continue with AVX series, e.g., AVX (2010), AVX2 (2013), and AVX-512 (2013), and will go to more advanced instruction set supporting larger registers. MMX, SSE, AVX, and AVX-512 instructions can operate on 64-, 128-, 256-, and 512-bit registers, respectively. AVX includes preparations to extend to 1024 bits for future generations of architectures. Specific instructions can be applied in parallel on bytes, half words, words, double words, and even 128-bit registers, but should be in an aligned form [Int].

Cryptographic Instructions. Special cryptographic instruction sets are also commonly included in a CPU to support the implementation of commonly used cryptographic operations, e.g., the AES encryption round [Int]. These instructions can largely increase the performance of algorithms involving these operations, which motivates many new cryptographic designs built on these operations.

Parallelism. When implementing an algorithm in software, parallelism can be used to largely improve performance. Besides SIMD, another important type of parallelism is instruction-level parallelism (ILP), which allows instructions to be (partly) overlapped and executed simultaneously. The simplest and most common way to achieve ILP is to exploit parallelism among iterations of a loop, which is called *unrolling*. Such techniques work either by the compiler or through the hardware. Programmers should also consider optimising the code to allow for ILP.

Chapter 3

Design of Stream Ciphers

3.1 Introduction

In 2004, the European eSTREAM project was initiated to call for new stream ciphers as counterparts of the widely deployed block cipher AES. Two profiles of design were specified for the project [RB08]:

- Profile 1: stream ciphers for software applications with high throughput;
- Profile 2: stream ciphers for hardware applications with highly restricted resources.

The project has sparked many new designs of stream ciphers. After three rounds of evaluation and some public cryptanalysis results, the final output of the project is a portfolio of seven promising stream ciphers [RB08]. Actually, the two profiles of the design exactly correspond to the two use cases where a dedicated stream cipher might conceivably offer some advantages over block ciphers. Modern stream ciphers are typically designed intended for these two use cases.

A stream cipher consists of an *internal state*, which can be built on bits or larger alphabets (e.g., on bytes or 32-bit words). Software-efficient stream ciphers are usually built on larger alphabets to achieve high throughput, while hardware-restricted ones are typically bit-based, as many microprocessors in embedded devices are 8-bit. A stream cipher can be usually defined by the following three building-blocks:

- an initialisation procedure $\texttt{init}(K, IV)$ which loads the secret key K and initialisation vector IV to the internal state, after which the state is usually referred to as the *initial state*, and runs the cipher a number of rounds without producing any output;
- an update function $\texttt{update}(S)$ which updates the state S in each iteration;
- an output function $\texttt{output}(S)$ that produces one keystream word in each iteration after the initialisation.

The initialisation phase is used to thoroughly mix the key and IV, after which the output should behave random-like. During the initialisation, no output is produced: the

output is suppressed and usually fed back to some part of the state. The initialisation introduces extra time overhead, thus making a stream cipher less competitive for short-length plaintext. The number of initialisation rounds should be carefully designed to balance between security and efficiency.

During the key generation phase, one keystream word (or symbol) is generated in each iteration, which can be one bit, byte, 32-bit word, or word with an even larger size, depending on how the cipher is designed. The state is updated in each iteration as well. Based on how the state is updated, stream ciphers can be classified into: *synchronous*, if the state changes independently of the plaintext and ciphertext; and *self-synchronising*, if the state updates based on some previous ciphertext. Most stream ciphers are synchronous, and we mainly focus on this type.

3.2 Constructions of Stream Ciphers

There are several popular constructions of stream ciphers, which can be categorised into several types: LFSR (Linear Feedback Shift Register)-based, NFSR (Nonlinear Feedback Shift Register)-based, block cipher based, sponge-based, and other ad-hoc designs some of which are intended for special applications. We below present an overview of these constructions, and refer to [RB08, JHF20, MVOV18, SS16] for more details. Our thesis mainly focuses on the LFSR-based stream ciphers.

3.2.1 LFSR-based

An LFSR is a shift register whose input is a linear combination of its previous states. It has attractive properties required for a stream cipher, e.g., a long cycle length, good pseudorandomness, and high efficiency in hardware and software implementations. These properties make LFSRs the most popular components for building stream ciphers. Typically, an LFSR-based stream cipher consists of two parts: a linear part made of LFSRs serving as the source of pseudorandomness, and a non-linear part for disrupting the linearity.

3.2.1.1 Linear Part

The linear part can have one or several LFSRs defined over a finite field $\mathbb{F}_p$. Typically, $p = 2$ or $p = 2^m$, where m is usually 8, 16, and 32, to allow for efficient implementation. For example, SNOW 3G [3GP06] has one LFSR defined over $\mathbb{F}_{2^{32}}$ while E0 [LV04] used for Bluetooth communication has three LFSRs defined over $\mathbb{F}_2$. However, there can be some exceptions: for example, the LFSR in ZUC [ETS11] is defined over a prime field $\mathbb{F}_p$ where $p = 2^{31} - 1$. This enables ZUC to resist many classical cryptanalysis methods of stream ciphers, while on the other hand, thwarts its performance.

Figure 3.1 illustrates the structure of an LFSR defined over a finite field $\mathbb{F}_p$ with L cells. Each cell, called an *stage*, holds an value from $\mathbb{F}_p$. The contents of the L cells form the *state* of the LFSR. The c values, i.e., $c_1, c_2, \ldots, c_L \in \mathbb{F}_p$, are called *feedback coefficients*. The output sequence of the LFSR is uniquely determined by the *feedback polynomial* (or *connection polynomial*) $P(x)$, which is characterised by the feedback

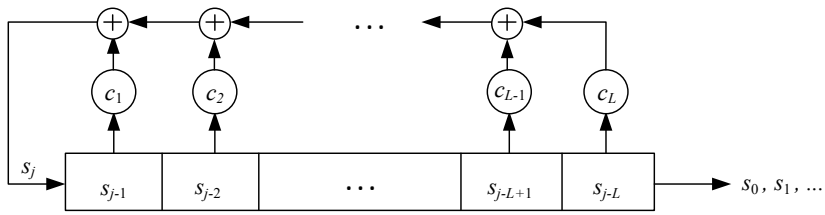

Figure 3.1: Structure of a linear feedback shift register.

coefficients as below:

$$P(x) = -c_L x^L - c_{L-1} x^{L-1} - \cdots - c_1 x + 1.$$

In each clocking, the lowest stage s_{j-L} exits the LFSR, the value in each other stage is right-shifted to the next one, and s_j is newly generated from previous stages as below:

$$s_j = c_1 s_{j-1} + c_2 s_{j-2} + \cdots + c_L s_{j-L},$$

for $j = L, L+1, \ldots$. Therefore, the LFSR implements the linear *recurrence relation* as below:

$$s_j = \sum_{i=1}^{L} c_i s_{j-i}, \text{for } j = L, L+1, \ldots,$$

with the first L symbols $s_0, s_1, \ldots, s_{L-1}$ forming the *initial state*.

If we view the state as a column vector of size L and use S_t to denote the state at clock t, i.e., $S_t = (s_t, s_{t+1}, \ldots, s_{t+L-1})^T$, the state at the next clock $t+1$ can be expressed as

$$S_{t+1} = M \cdot S_t,$$

where M denotes the $L \times L$ *transition matrix* over $\mathbb{F}_p$, with the form as below:

$$M = \begin{bmatrix} 0 & 1 & 0 & \ldots & 0 & 0 \\ 0 & 0 & 1 & \ldots & 0 & 0 \\ 0 & 0 & 0 & \ldots & 0 & 0 \\ & & & \ldots & & \\ 0 & 0 & 0 & \ldots & 0 & 1 \\ c_L & c_{L-1} & c_{L-2} & \ldots & c_2 & c_1 \end{bmatrix}.$$

If the initial state is denoted as S_0, which is typically the loading result of the key and IV, the state after t clocks can be expressed as $S_t = M^t \cdot S_0$. Therefore, if a state at a specific time clock is known, the initial state and possibly the secret key can be recovered.

The choice of an LFSR for building a stream cipher should consider several aspects. The most basic but essential criterion is that it should provide the maximum cycle

length to provide an excellent source of pseudorandomness. This requires the feedback polynomial to be primitive.

Definition 12. A polynomial $P(x)$ of degree L with coefficients in the finite field $\mathbb{F}_p$ is called *irreducible* if it cannot be written as the product of two polynomials of degrees smaller than L with coefficients in $\mathbb{F}_p$.

Definition 13. An irreducible polynomial $P(x)$ of degree L with coefficients in the finite field $\mathbb{F}_p$ is called *primitive* if the smallest positive integer n for which $P(x)$ divides $x^n - 1$ is $n = p^L - 1$.

The output sequence produced by an LFSR with a binary ($p = 2$) primitive feedback polynomial with degree L can achieve the maximum period $2^L - 1$. The Sagemath tool [sagc] can be used to help verify whether a polynomial is primitive using the built-in `is_primitive()` function.

The security and implementation aspects should be considered as well. For example, the number of feedback taps should be larger than three to resist against linear attacks, which has almost become a rule of thumb for stream cipher design. Besides, these feedback taps should be well chosen to allow for efficient implementation.

3.2.1.2 Non-Linear Combinations

LFSR itself is insufficient to build a stream cipher as it is a linear construction leading to easy cryptanalysis, e.g., using the Berlekamp-Massey algorithm [Ber15, Mas69]. Thus, non-linear components are required to disrupt such linearity. The main non-linear constructions include *clock-controlled generators*, *non-linear combination generators*, and *filter generators*.

- **Clock-controlled generators.** A clock-controlled generator introduces non-linearity by irregularly clocking the LFSRs. The clock can be controlled either by some external components, e.g., another LFSR, or by the combinations of several internal bits from multiple LFSRs. A typical example of this type is A5/1 [BSW00], which has three LFSRs each with one clocking bit; an LFSR is clocked if its clocking bit agrees with at least one clocking bit of the other two LFSRs.

- **Combination generators.** A non-linear combination generator, e.g., a non-linear Boolean function, is usually used when the linear part uses several LFSRs. These constituent LFSRs should be chosen to have primitive feedback polynomials for ensuring good statistical properties of their output sequences. The non-linear function should balance between multiple properties, e.g., the correlation immunity order, algebraic degree, and implementation efficiency. Memory elements are usually introduced into the function to guarantee the correlation immunity order and non-linearity. Vectorial Boolean functions can also be used as the combination generator to produce several output bits and increase throughput. The typical example of this type is the Bluetooth encryption algorithm E0 [LV04], which uses one non-linear combination generator with four bit memories to nonlinearly combine the outputs from four LFSRs.

- **Filter generators.** A filter generator applies a non-linear Boolean function or vectorial Boolean functions to several extracted stages from an LFSR and produces output. Similarly, the different properties of the function should be considered, and memory elements are usually included. This construction is the most popular design for LFSR-based stream ciphers, and many well-known stream ciphers are built on it, e.g., SNOW 3G [3GP06] and ZUC [ETS11].

3.2.2 NFSR-based

NFSR is becoming a new important component for building stream ciphers. It is the same as an LFSR except that the update function is non-linear. Introducing NFSRs in stream ciphers starts from the Grain cipher [HJM07] submitted to the eSTREAM project. Grain comprises one LFSR, one NFSR with output feeding to the LFSR, and one filter function extracting state bits from these two FSRs (feedback shift registers) to produce the output. Such a construction benefits from the high implementation efficiency of FSRs, and at the same time, introduces more security properties through the NFSR. On the other hand, the non-linear update is typically more complex and could be more expensive. Another famous NFSR-based stream cipher is Trivium [DC06], which uses three NFSRs feeding to each other and forms a circular construction.

NFSR-based stream ciphers introduce new cryptographic properties, and novel cryptanalysis techniques are required. Such construction is becoming popular in the design of lightweight stream ciphers. For example, NFSRs are used in several recent stream ciphers intended for lightweight applications, e.g., Sprout [AM15], Fruit [GHC16], and Plantlet [MAM16]. Though there are a number of cryptanalysis results of these ciphers, we can still expect the popular use of NFSRs in future lightweight stream ciphers.

3.2.3 Block Cipher based

A block cipher can act like a stream cipher when using some particular mode of operation. The most prominent mode for achieving this is the *counter mode* (CTR), in which the underlying block cipher takes the secret key and distinct counters as inputs and outputs a sequence of keystream blocks of the same length. The encryption (resp., decryption) is then the simple XOR operation between the keystream blocks and the plaintext (resp., ciphertext) blocks. The decryption uses the same algorithm to generate the keystream blocks; thus, no special decryption implementation is required. CTR mode allows for parallel processing of arbitrary lengths of plaintext, and is therefore widely used in applications where the length of the plaintext varies. AES is working in CTR mode for the confidentiality and integrity protection over the air in 4G.

There are some other designs of stream ciphers related to a block cipher, e.g., using the round function of a block cipher as one component of a stream cipher. For example, the stream ciphers AEGIS [WP13] and Rocca [SLN+21] build the round transforms with several parallel AES encryption rounds. Such ciphers can take full advantage of the AES instructions supported by mainstream CPUs, thus can be highly efficient in a software environment. On the other hand, the security of these ciphers heavily depends on AES, and once AES was broken, these ciphers can immediately become vulnerable.

Another popular construction of the round function is using a combination of addition, rotation, and XOR, and these ciphers are usually called ARX ciphers. The three

operations are relatively cheap and fast in a software implementation, and as a consequence, ARX-based ciphers are becoming popular [BP17]. Well-known examples are like Salsa20 [Ber08] and ChaCha [B+08].

3.2.4 Sponge-based

Since Keccak [BDPVA09] won the SHA-3 (Secure Hash Algorithm 3) competition, the sponge construction has become very popular for building many cryptographic primitives, e.g., hash functions, MACs, stream ciphers, and PRGs. A sponge construction typically has three components: the *state* of a certain size, a *permutation function* iteratively operating on the state, and a *padding function*. It takes in an arbitrary-length input, which is usually referred to as the "absorbing" phase, and produces an output of any desired length, which is called the "squeezing" phase.

In sponge-based stream ciphers, the key/IV and possibly additional associated data (AAD) are initially loaded to the state; then the permutation function is iteratively applied on the state for a certain number of rounds as initialisation. After that, the plaintext is divided into fixed-length blocks and padded based on specific rules, and the cipher takes one block in each iteration and XORs it with a part of the state to produce one ciphertext block. In some designs, the generated ciphertext block is in turn fed into the next iteration. After processing all the plaintext blocks, some part of the state is output as the MAC. Thus, sponge-based constructions can achieve AEAD very easily and become very popular choices for stream ciphers. Among the ten finalists of the NIST LWC (Lightweight Cryptography) competition, six are built on the sponge construction, i.e., ASCON, Elephant, ISAP, PHOTON-Beetle, Schwaemm and Esch, and Xoodyak [NISa]. Currently, 3GPP protocols do not support AEAD, but if 6G or the system beyond opens up for it, sponge-based ciphers can be good choices for the confidentiality and integrity protection over the air.

3.2.5 Other Constructions

Besides the above main constructions for stream ciphers, there are some ad-hoc designs intended for special applications. For example, the FLIP cipher [MJSC16], proposed in Eurocrypt 2016, is designed for use in FHE. It contains three main components: a *register* storing the key; a *bit permutation generator* parameterised by a public pseudorandom number generator used to permutate the secret key; and a *filtering function* extracting bits from the permutated key and producing the output. Such a construction is called *filter permutator*, and FLIP achieves a low constant noise for FHE thanks to it. However, FLIP was broken by a guess-and-determine attack in [DLR16].

In 2019, the authors proposed a new version of FLIP, called FiLIP [MCJS19b, MCJS19a], to improve the resistance against linear attacks. The filter function in FiLIP is carefully designed, taking different aspects into account. The authors proposed a number of FiLIP instances that use different Boolean functions as the filter functions, e.g., DSM (Direct Sums of Monomials) and XOR-MAJ (majority) predicates. These predicates are also relevant to the local pseudorandom generators, and we will present more details in Chapter 5. Our included Paper VI also focuses on investigating the security of local PRGs.

Chapter 4

Cryptanalysis of Stream Ciphers

When performing cryptanalysis of stream ciphers, the known-plaintext attack model is typically used, i.e., the plaintext, ciphertext, and correspondingly the keystream sequence, are assumed to be known. The goal of an attack can be to recover the secret key, retrieve the state, or distinguish the cipher from random based on the known keystream sequence. In this chapter, we present an overview of the most widely used cryptanalysis techniques of stream ciphers. All of these cryptanalysis techniques should be visited when designing a new stream cipher.

4.1 Exhaustive Key Search

Exhaustive key search is the most general and basic attack, as it requires no specific information of the cipher. In exhaustive key search, an attacker tries every possible value of the secret key and gets the correct one by checking if the tested key value can generate a keystream sequence identical to the observed one. The complexity of the attack is 2^n, where n is the size of the secret key. Exhaustive key search is usually regarded as the benchmark for other attacks: if one attack has complexity below 2^n, we claim that the attack is faster than exhaustive key search, and the cipher is regarded as (at least theoretically) broken. We mention that some cryptanalysis results with complexity above 2^n can still be helpful for understanding the security of a cipher.

For a stream cipher, the attacker can also guess its internal state instead of the secret key. However, as the state is typically at least twice larger than the secret key [HS05b, HS05a], simply guessing the state will result in much higher complexity. Instead, one can guess a part of the state and determine the rest through the publicly defined relations or employing more advanced techniques, e.g., using a decoding method. Actually, such ideas are used in guess-and-determine attacks (in Section 4.4) and correlation attacks (in Section 4.2.3), respectively.

4.2 Linear Cryptanalysis

4.2.1 Basics

Linear cryptanalysis was first introduced by Matsui in 1993 to attack DES in [Mat93]. Since then, it has become one of the most powerful cryptanalysis techniques against symmetric ciphers and should always be considered when designing a new cipher. The basic idea of linear cryptanalysis is to linearly approximate the non-linear operations in the cipher and further explore some linear relations. In a stream cipher, if these linear relations only involve keystream symbols, it can lead to a distinguishing attack indicating that the cipher is not purely random [HJB09]; while if the linear relations involve both the keystream symbols and initial states, it can result in a correlation attack which can recover the secret key [TIA18, ZXM15, Mei11, CJM02].

For a non-linear function $f : \mathbb{F}_{2^n} \to \mathbb{F}_{2^m}$ in a cipher, e.g., an S-box or a modulo addition, the linear approximation of f can be expressed as:

$$\beta \cdot f(x) = \alpha \cdot x, \tag{4.1}$$

where $x \in \mathbb{F}_{2^n}$ is the input and nonzero α, β are called *linear masks* [CHJ02]. We will give more details about linear masks soon. Such an expression is called a linear masking. The approximation in (4.1) does not always hold, and some biased *noise* will be introduced, which is expressed as:

$$e = \beta \cdot f(x) \oplus \alpha \cdot x. \tag{4.2}$$

The noise can be measured using *bias*, which determines the quality of the linear approximation and further influences the attack complexity. Thus, a cryptanalyst tries to explore "good" linear masks α and β such that the equation $\beta \cdot f(x) = \alpha \cdot x$ holds with a higher probability.

There are two ways of choosing the linear masks α, β and computing the bias, which result in *bitwise* (*binary*) [Mat93, JV03] and *multidimensional* [HCN19, BJV04] linear approximations, respectively.

Bitwise Linear Approximation. α and β are n-bit and m-bit non-zero binary vectors, respectively, and the $(\cdot)$ operation denotes the standard inner product. In this case, the linear approximation and the noise e is binary. The bias is evaluated as:

$$\begin{aligned} \epsilon &= \mathbf{Pr}[e = 0] - 0.5 \\ &= \frac{1}{2^n} \#\{x | x \in \mathbb{F}_{2^n}, \beta \cdot f(x) = \alpha \cdot x\} - 0.5. \end{aligned} \tag{4.3}$$

When n and m are not too large, the biases under different linear masks α, β can be exhaustively computed and stored in a $2^n \times 2^m$ table, e.g., the *linear approximation table* (LAT) [O'C94], denoted $LAT_f(\alpha, \beta)$. Each entry in the LAT is the number of equal parity checks corresponding to the specific (α, β) pair, i.e.,

$$LAT_f(\alpha, \beta) \stackrel{\text{def}}{=} \# \left\{ x | x \in \mathbb{F}_{2^n}, \bigoplus_{i=1}^{n} x[i] \cdot \alpha[i] = \bigoplus_{j=1}^{m} f(x)[j] \cdot \beta[j] \right\}. \tag{4.4}$$

In linear cryptanalysis, we are interested in those entries in the LAT that deviates far from 2^{n-1}. The bias corresponding to each (α, β) pair can be trivially computed from the LAT according to (4.3).

For a bitwise linear approximation with bias ϵ, the number of data samples required to distinguish e from random is in the order of $\frac{1}{\epsilon^2}$.

Multidimensional Linear Approximation. α and β are binary matrices of size $t \times n$ and $t \times m$, respectively, where $t \leq n, t \leq m$, which results in a t-dimension linear approximation. Each bit of the t-bit vector is computed as $\beta_i \cdot f(x) \oplus \alpha_i \cdot x$, where α_i and β_i are the i-th rows of α and β, respectively. The bias is measured in SEI according to the distribution D of the noise e, where each entry i is computed as below:

$$\text{for all } i \text{ in } \mathbb{F}_{2^t} : D[i] = \frac{1}{2^n} \#\{x | x \in \mathbb{F}_{2^n}, \beta \cdot f(x) \oplus \alpha \cdot x = i\}.$$

By trying out all the values of x, we can compute $e = \beta \cdot f(x) \oplus \alpha \cdot x$ and construct the distribution table. For a multidimensional linear approximation with SEI Δ, the number of data samples required to distinguish e from random is in the order of $\frac{1}{\Delta}$.

A multidimensional linear approximation typically provides more biased information if the matrix masks are well-chosen [HCN19]. However, it becomes difficult to exhaust all the possibilities of matrix masks when t is large, e.g., most likely $t = n = m$, and in many cases one should be satisfied with a linear masking with a decent bias.

The most commonly used linear approximations for modern stream ciphers are for S-boxes and modulo additions.

As we mentioned, the S-box is one of the most important components for providing non-linearity, and the linear approximation of it is very important. An m-to-m bit S-box is usually approximated as $\beta S(x) = \alpha x$ (we omit the notation $\cdot$ without ambiguity). As most S-boxes are byte-based or even smaller, an LAT is usually computed to store the biases under different linear masks. More complex forms of the linear approximation of S-boxes can be involved in many cases, for example, $\beta S(P \cdot x) = \alpha x$, where P is some permutation matrix. If α and β are nonzero, we call the S-box *active*. When more active S-boxes are involved in linear cryptanalysis, the bias will typically become smaller. Therefore, one basic rule in linear cryptanalysis is to find a linear approximation that involves as few S-boxes as possible. When designing a new cipher, designers usually have a rough estimation of how many S-boxes will be involved when performing linear cryptanalysis, thus having an initial insight into the achievable non-linear properties.

Another important non-linear operation in stream ciphers is the arithmetic modulo addition. The m-bit modulo addition is usually linearly approximated as $\gamma(x \boxplus y) = \alpha x \oplus \beta y$, where $\gamma \in \mathbb{F}_{2^m}$ is the output linear mask and $\alpha, \beta \in \mathbb{F}_{2^m}$ are the input linear masks. Such an approximation can hold with a large probability in many cases. For example, when γ, α, and β are the unit vectors with only the least significant bits being 1, $\gamma(x \boxplus y) = \alpha x \oplus \beta y$ always holds.

In practice, a linear approximation can be much more complex: with more variables, operations involved, and more complicated connections between them. In this case, one should consider the overall approximation instead of a locally optimal one. If we denote the overall non-linear function as F, F can be expressed as a composite of several

non-linear sub-functions representing different operations. If these sub-functions are independent, i.e., involving different independent input variables, the bias of F can be computed based on the distributions of sub-noises, e.g., using the piling-up lemma for bitwise approximations [Mat93]. However, in many cases, these sub-functions are correlated, e.g., they involve some same input variables or the output of one sub-function is the input of another sub-function. In these cases, the dependence between sub-functions should be carefully dealt with [NW06], e.g., using the *correlation theorem* in [Nyb01].

For an LFSR-based stream cipher, we can express the generated keystream sequence as $z = \mathtt{NF}(S)$, where S denotes the LFSR state and $\mathtt{NF}$ is the equivalent non-linear keystream generation function [CHJ02]. We can write $\mathtt{NF}$ as some linear function $\mathtt{LF}$ plus a biased noise e, i.e.,

$$z = \mathtt{NF}(S) = \mathtt{LF}(S) + e. \tag{4.5}$$

One can find many different linear functions ($\mathtt{LF}$) and correspondingly gets different noises (e). It is infeasible to try out all possibilities, and we usually have to be satisfied with a "good" one. We next show how the linear approximation is used in a distinguishing attack or correlation attack against a stream cipher.

4.2.2 Distinguishing Attacks

A distinguishing attack targets to distinguish a keystream sample sequence from random, i.e., determining if a given sequence is generated from this specific cipher or just random. A distinguishing attack is usually not as powerful as a key-recover attack, but the resistance against it is still used as one design criterion of stream ciphers.

The distribution of the original keystream sequence is often very close to the uniform distribution (otherwise, it is not a good cipher), and it is difficult to distinguish it from random directly. Instead, some well-chosen linear combinations of keystream symbols are considered, e.g., $x_t = \sum_{j \in I} c_j z_{t+j}$ for a time set I where c_j's are some constants. These symbols form a new sequence called *sample* sequence.

In (4.5), e is biased, while $\mathtt{LF}(S)$ is regarded as balanced. If the $\mathtt{LF}(S)$ part can be canceled, only the noise e remains and the left side involving only keystream symbols will become biased as well. Therefore, one key point of a distinguishing attack is to cancel the contribution from the linear part $\mathtt{LF}(S)$, which is achieved either using the feedback polynomial or a low-weight multiple of the feedback polynomial.

Definition 14 (The Low-Weight Polynomial Multiple (LWPM) Problem)**.** Given a binary polynomial $P(x) \in \mathbb{F}_2[x]$ of degree d_p, and two integers d and ω, find a multiple $K(x) = P(x)Q(x)$ of degree at most d and weight at most ω.

The expected number of such multiples is approximated by $\binom{d}{\omega-1} 2^{-d_P}$. There are many ways to find a multiple [Gol96a, LJ14, DLC07], which typically need to balance between time and memory.

Assume that the feedback polynomial or a multiple of it with k taps is denoted as $x^{t_1} + x^{t_2} + \cdots + x^{t_k} = 0$, then the LFSR states would always satisfy:

$$\bigoplus_{t_i \in T} S_{t+t_i} = 0, \ t \geq 0,$$

where $T = \{t_1, t_2, \ldots, t_k\}$. Thus, if we combine (4.5) at time instances in T, the LFSR contribution will be canceled (become zero), while only the keystream symbols and noise remain as below:

$$\bigoplus_{t_i \in T} z_{t+t_i} = \bigoplus_{t_i \in T} e_{t+t_i}.$$

Any time shift of the above expression still holds. Therefore, we can build new keystream samples of the form $x_t = \bigoplus_{t_i \in T} z_{t+t_i}$. Note that z_{t+t_i}'s do not necessarily denote the original keystream symbols, but could be some transform of them, e.g., linear combination, truncation, or concatenation. By collecting a large number of such samples, it is possible to distinguish the keystream sample sequence from random using tools like hypothesis testing.

Usually, the noise e in the single approximation is explored and computed, and the bias of a k-tuple noise $\bigoplus_{t_i \in T} e_{t+t_i}$ can be easily computed from e. For example, piling-up lemma can be applied if e is binary [Mat93], and Fourier transform can be used if e is multidimensional [MJ05]. Typically, a larger size of T will result in a smaller bias; thus, if the feedback polynomial has a high weight, one can turn to search for a low-weight multiple of it.

4.2.3 Correlation Attacks

A correlation attack explores the correlation between the LFSR states and the keystream symbols, which always exists. If the non-linear part involves M memory elements, the correlation can be explored between at most $M+1$ consecutive keystream bits and LFSR states [MS92, Gol96b]. A correlation attack is usually modeled as a decoding problem over $\mathbb{F}_{2^n}$ for $n \geq 1$ as shown in Figure 4.1.

The initial LFSR state denoted $\mathbf{u}$ of length l, $\mathbf{u} = (u_0, u_1, \ldots, u_{l-1})$, is modeled as the information symbol vector and the output sequence of the LFSR of length N is regarded as the codeword. We denote the codeword as $\mathbf{c}$, which is generated from $\mathbf{u}$ through $\mathbf{c} = \mathbf{uG}$, where $\mathbf{G}$ is some generation matrix. The keystream sample sequence $\mathbf{y} = (y_0, y_1, \ldots, y_{N-1})$ is a noisy version of the codeword received through a discrete memoryless channel. Note that $\mathbf{c}$ and $\mathbf{y}$ do not necessarily denote the original LFSR output and keystream sequence, respectively, but could be some transforms of them.

The noisy channel corresponds to the linear approximation noise e and the channel capacity can be computed as $C = n + \sum_{e_i \in \mathbb{F}_2^n} p(e_i) \cdot \log_2 p(e_i) \approx \frac{\Delta(e)}{2 \ln 2}$, where $p(e_i)$ is the probability of $e = e_i$ and $\Delta(e)$ is the SEI of e [ZXM15]. The attacker aims to recover the initial state according to the received keystream sample sequence, and this process is equivalent to decoding an $[N, l]$ linear code over $\mathbb{F}_{2^n}$. When the code rate is below the equivalent channel capacity, there exists some decoding method that can uniquely recover the information symbols according to coding theory.

A correlation attack consists of two phases: the *preprocessing phase*, during which many low-weight parity checks are explored offline; and *decoding phase*, during which some decoding techniques are used online to recover the information symbols based on the explored parity checks. The research of correlation attacks mainly focuses on these two phases, i.e., exploring more efficient ways for building low-weight parity checks and decoding.

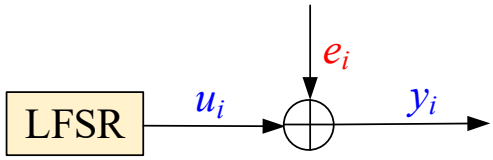

Figure 4.1: Model for a correlation attack.

1. **Preprocessing Phase.** If the feedback polynomial is low-weight, the parity checks can be derived directly according to it [MS89]. Thus it has been a rule of thumb that the weight of the feedback polynomial should not be too low. LFSRs used in modern ciphers typically have many feedback taps or are defined over higher-dimension fields. In this case, multiples of the feedback polynomials with low weights, usually 3, 4, or 5, are explored, and parity checks are constructed based on the multiples [CT00]. The number of required parity checks depends on the quality of the channel (i.e., the linear approximation) and the decoding method.

2. **Online Decoding Phase**. After collecting a large number of parity checks, some decoding algorithms are applied to recover the information symbols, i.e., the initial internal state, based on these parity checks. The decoding algorithms can be categorised into the *one-pass* and *iterative* methods. In the first type, some advanced decoding techniques, e.g., based on convolutional codes [JJ99b], turbo codes [JJ99a], and list decoding [MFI01], are used to recover the information symbols. In the iterative methods, the algorithm iteratively updates information of the information symbols and finally recovers them when the algorithm converges [MS89, CT00].

Modern software-efficient ciphers are usually constructed over a larger field, and the correlation attacks over extension fields appeared in recent years [ZXM15], which are much more complicated. One idea to simplify it is to transform the original $[N, l]$ code $\mathcal{C}_1$ into a simpler code $\mathcal{C}_2$ $[N', l']$ with $l' < l$, which will require less decoding complexity [CJS00]. This is achieved by finding k-tuples of column vectors in the generation matrix $\mathbf{G}$ such that the last $l - l'$ elements are all zero for some small k. If we find N' such combinations, the problem is converted into decoding an $[N', l']$ code. Finding such combinations can be achieved using algorithms solving the k-sum problem [Wag02].

4.3 Attacks on Initialisation

The initialisation of a stream cipher targets to thoroughly mix the secret key and IV. After the initialisation, the keystream output should behave random-like. Attacks on initialisation aim to find some non-randomness that usually indicates some weakness or an insufficient mixing effect, which in turn helps to check if the number of initialisation rounds has provided enough security margin. The non-randomness can appear in many different forms, e.g., some keystream bits are more likely to be one (or zero), the XOR

sum of some keystream bits is always zero given a subset of inputs. Since the IV is public, an attacker can arbitrarily choose the IV and hope to find some specific values that will introduce such non-randomness. Attacks under such a setting are usually called *chosen-IV attacks* [Saa06, EJT07].

Each keystream bit after initialisation (or a suppressed output bit in the initialisation) z can be expressed as a Boolean function of the secret key (say, n-bit), denoted $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$, and IV (say, m-bit), denoted $\boldsymbol{v} = (v_1, v_2, \ldots, v_m)$, as below:

$$\begin{aligned} z &= f(\boldsymbol{x}, \boldsymbol{v}) \\ &= a_0 + a_1 x_1 + \cdots + a_{n+1} v_1 + \cdots + a_M x_1 x_2 \cdots x_n v_1 v_2 \cdots v_m. \end{aligned} \tag{4.6}$$

For a stream cipher with a good mixing effect, the Boolean function will behave like a random function, and each monomial coefficient a_j is zero or one with a probability of 0.5. If one can get some information about the Boolean function, e.g., some coefficients are (more likely to be) zero (or one), it usually indicates that the mixing effect is not sufficient yet. However, such information is difficult to obtain since the Boolean function is highly complicated. Specifically, even when we view the IV as publicly known, the Boolean function has 2^{n-1} monomials in average and the degree can be as high as n [EJT07]. For such a Boolean function with a typical key size (e.g., 128 bits), we even do not have an efficient way to store it. Thus, the cryptanalysts (or attackers) usually consider a round-reduced initialisation and try to attack as many rounds as possible.

The most common attacks against the initialisation are usually defined over a *cube*. A cube is a subset of IV values in which some specific IV bits, let us call them *cube bits*, exhaust all possible values, while other bits are fixed with certain constant values. As a simple example, a cube of a 4-bit IV (v_1, v_2, v_3, v_4) can be $\{0001, 0011, 0101, 0111\}$, in which v_2 and v_3 are the cube bits exhausting all values, while v_1 and v_4 keep the constant values 0 and 1, respectively. One can see that there are many different cubes: the cube bits can vary, and the constant part can be fixed as different values as well. For example, an m-bit IV can have $\binom{m}{i}$ different cubes of size i if the constant part is set as zero, which is a typical setting. For certain cubes, some weakness in the output can be possibly explored, though such cubes are typically difficult to be identified. Below we show how the cubes are used in three most powerful attacks on the initialisation process.

4.3.1 Maximum Degree Monomial Tests

The maximum degree monomial (MDM) test is one cryptanalysis tool used to check the mixing effect of the initialisation. It can experimentally provide a lower bound on the required number of initialisation rounds by checking the MDM values (i.e., a_M in Equation (4.6)) of the Boolean functions of the suppressed outputs during the initialisation and keystream bits after initialisation [Sta10, Sta13].

Consider a modified initialisation that produces output in each initialisation round, and we can define a vectorial Boolean function to describe the entire initialisation procedure: $f : \{0,1\}^{n+m} \to \{0,1\}^l$, where l is the total number of the output bits. For example, if the cipher has r initialisation rounds and outputs w bits in each round, $l = r \cdot w$. If we use σ_i $(1 \le i \le l)$ to denote the MDM of each Boolean function f_i $(1 \le i \le l)$, the so-called *MDM signature* is defined as below [Sta10, Sta13]:

$$\sigma = \sigma_1 || \sigma_2 || \cdots || \sigma_l.$$

The MDM test is based on the observation that in the first few initialisation rounds, the mixing effect is not sufficient and a Boolean function of the output tends to have a low algebraic degree. The algebraic degree increases along with the initialisation rounds, and the high-degree monomials should appear with probabilities of 0.5 after sufficient rounds. The MDM test focuses on the MDM coefficient, as it is the last coefficient becoming random-like. Before that, the MDM coefficient keeps zero, e.g., $\sigma_1 = \sigma_2 = \cdots = \sigma_k = 0$ until some k, and the MDM test checks how long such a zero sequence is (i.e., how large k is) and if it is far below the number of initialisation rounds.

For an unknown Boolean function, if one had unlimited computational capabilities, he can try all the input values (of key and IV), get the outputs and correspondingly the truth table; from the truth table, he can recover the monomial coefficients. However, this is infeasible in practice, and instead, one considers cubes with much fewer cube bits that need to be exhausted and checks the MDM values over these cubes. Given a cube $\mathcal{C}$ of t cube bits, a new vectorial Boolean function $f_{\mathcal{C}} : \{0,1\}^t \to \{0,1\}^l$ can be defined and the MDM signature of it can be recovered through:

$$\sigma_{\mathcal{C}} = \bigoplus_{(\boldsymbol{x},\boldsymbol{v})\in\mathcal{C}} f_{\mathcal{C}}(\boldsymbol{x},\boldsymbol{v}).$$

That is, the variant MDM signature can be obtained by running the initialisation with the values in the cube as inputs and XOR-ing all the outputs. Based on the derived MDM signature, one can easily check after how many rounds the output behaves random-like and if it is reasonably below the number of initialisation rounds. As the test needs to exhaust every possible value of the cube bits, the attack is restricted to smaller cube sizes, e.g., smaller than 40.

For some certain cubes, the MDM test could result in longer zero sequences. However, there is no good way to explore such cubes, and in practice, one either chooses a cube randomly or uses a *greedy* algorithm to gradually increase the cube size [Sta10]. In the greedy method, one first finds the optimal cube of a small size that results in the longest zero-sequence, then in each greedy step adds one (or two, three) more bit that results in a longer zero-sequence compared to other bits to the cube. Obviously, it cannot achieve a globally optimal result.

Sometimes, secret key bits are used as cube bits as well, which results in a *nonrandomness detector* [Sta10]. This is adopted when designing a cipher to check how the key and IV are mixed. For more details of the MDM test, we refer to [Sta13].

4.3.2 Traditional Cube Attacks

Besides considering a round-reduced initialisation, one can also turn to recover a much simplified polynomial, called *superpoly*, instead of an original Boolean function. Based on the recovered superpoly, one can possibly get some information about the secret key. Such attacks are called cube attacks [DS09, Lat09].

A Boolean function can be rewritten as the following representation:

$$f(\boldsymbol{x},\boldsymbol{v}) = t_I \cdot p(\boldsymbol{x},\boldsymbol{v}) + q(\boldsymbol{x},\boldsymbol{v}),$$

where t_I is called the cube term, which is the product of the (symbolic) cube bits; $p(\boldsymbol{x},\boldsymbol{v})$ is called the superpoly, and $q(\boldsymbol{x},\boldsymbol{v})$ is the reminder polynomial, which misses at least one

variable in t_I. If we exhaust all possible values of t_I and add up the derived polynomials of $f(\boldsymbol{x}, \boldsymbol{v})$, the result will be exactly $p(\boldsymbol{x}, \boldsymbol{v})$. This is because $t_I = 1$ only when all the cube bits are one while $q(\boldsymbol{x}, \boldsymbol{v})$ is canceled as it appears an even number of times. For a well-chosen cube, the superpoly can be much simplified and possibly recovered.

Example. For a Boolean function $f(\boldsymbol{x}, \boldsymbol{v}) = v_1v_2x_1 + v_1v_2x_2 + v_2v_9x_2x_3x_5x_6x_9 + v_1v_3v_6v_9x_8x_9x_{10}$, if we set the cube term $t_I = v_1v_2$, the superpoly $p(\boldsymbol{x}, \boldsymbol{v}) = x_1 + x_2$, and $q(\boldsymbol{x}, \boldsymbol{v}) = v_2v_9x_2x_3x_5x_6x_9 + v_1v_3v_6v_9x_8x_9x_{10}$.

A cube attack targets to find "good" cubes such that the resulting superpolys are much simplified, specifically, linear and quadratic ones are preferred. By recovering these superpolys, some key information could be possibly retrieved. A cube attack mainly involves the following three steps.

1. **Superpoly recovery**: recover some low-degree superpolys $p(\boldsymbol{x}, \boldsymbol{v})$. In this step, some tools, e.g., linearity tests, can be used to ensure that these superpolys have low degrees [DS09].

2. **Superpoly value recovery**: recover the values of these superpolys. This is achieved by loading the values in the given cube to the cipher and XOR-ing all the outputs, i.e.,
$$\bigoplus_{\mathcal{C}_I} f(\boldsymbol{x}, \boldsymbol{v}) = p(\boldsymbol{x}, \boldsymbol{v}),$$
where $\mathcal{C}_I$ denotes the cube with variables in t_I as cube bits.

3. **Key recovery**: knowing the superpolys $p(\boldsymbol{x}, \boldsymbol{v})$ and their values, some key information can be recovered. For example, for one superpoly $p(\boldsymbol{x}, \boldsymbol{v}) = x_1 + x_2 = 1$, we will know $(x_1, x_2) = (0, 1)$ or $(1, 0)$.

The cube attack can have many variants, e.g., *dynamic cube attacks* [DS11], *cube testers* [ADMS09], and *conditional cube attacks* [HWX+17]. The critical point of a cube attack is to find "good" cubes such that the derived superpolys are much simplified. However, this is not easy in practice and is often achieved by many trial attempts. Besides, The size of a cube is restricted to a feasible range, e.g., smaller than 40 bits. Another disadvantage of the cube attack is that it uses the *black-box setting*, i.e., the cipher is dealt like a black box and its specific structure is not exploited at all. The cube attacks based on division property described [Tod15] in the next subsection provide a new and more efficient way [TIHM18], which has been a very hot topic in recent years.

4.3.3 Cube Attacks based on Division Property

Division property (DP) is a generalised integral property proposed by Todo at Eurocrypt 2015 [Tod15] and receives a wide range of research in the last few years. Before explaining the cube attacks based on division property, we first introduce some preliminaries.

Definition 15 (Bit Product Function)**.** Given $\boldsymbol{u}, \boldsymbol{x} \in \mathbb{F}_2^n$, the bit product function $\boldsymbol{x}^{\boldsymbol{u}}$ is defined as
$$\boldsymbol{x}^{\boldsymbol{u}} = \prod_{i=0}^{n-1} x[i]^{u[i]},$$

where $x[i]$ and $u[i]$ are the i-th bits of $\boldsymbol{x}$ and $\boldsymbol{u}$, respectively.

For example, $\boldsymbol{x} = 101, \boldsymbol{u} = 100$, $\boldsymbol{x}^{\boldsymbol{u}} = 1^1 \cdot 0^0 \cdot 1^0 = 1$. When $u[i] = 0$, the value of $x[i]^0$ will always be one and the true value of $x[i]$ is hidden; while when $u[i] = 1$, $x[i]^1 = x[i]$ and the true value of $x[i]$ is revealed. Therefore, the bit product function can be used to choose some bit positions for further investigation, e.g., to check if some non-randomness can be explored.

For two binary vectors $\boldsymbol{x}$, $\boldsymbol{y}$ of the same size, we denote $\boldsymbol{x} \succeq \boldsymbol{y}$ if $x_i \geq y_i$ at every position i. For example, $1101 \succeq 1001$.

Definition 16 (Bit-based Division Property [Tod15, TM16])**.** Given a multiset $\mathbb{X}$ whose elements take values in $\mathbb{F}_2^n$, its division property $\mathcal{D}_{\mathbb{K}}$, where $\mathbb{K} = \{\boldsymbol{k}_0, \boldsymbol{k}_1, ..., \boldsymbol{k}_t\}$ and each $\boldsymbol{k}_i$ is a length-n binary vector, gives the information about whether the XOR sum of the bit product function $\boldsymbol{x}^{\boldsymbol{u}}$ over $\mathbb{X}$ is 0 or *unknown*, for a vector $\boldsymbol{u}$:

$$\bigoplus_{\boldsymbol{x} \in \mathbb{X}} \boldsymbol{x}^{\boldsymbol{u}} = \begin{cases} unknown, & \boldsymbol{u} \succeq \boldsymbol{k} \text{ for some } \boldsymbol{k} \in \mathbb{K}, \\ 0, & otherwise. \end{cases}$$

It is not easy to characterise division property, as it is defined over a multiset $\mathbb{X}$. Basically, the division property divides the space of $\boldsymbol{u}$ into two subsets based on the XOR sum of the bit product function over $\mathbb{X}$: the *unknown-sum* subset, for which the XOR sum is unknown (i.e., could be either one or zero) which implies "randomness"; and the *zero-sum* subset, for which the XOR sum is known to be zero, which indeed indicates some kind of "non-randomness". When given the division property, one can immediately get the $\boldsymbol{u}$ vectors in the *zero-sum* subset and thus find some non-randomness.

We can understand as that in the initialisation phase of a stream cipher (or similarly, rounds of a block cipher), the size of the unknown-sum subset increases along with the initialisation rounds, until to some rounds that the whole space except the zero vector belongs to the unknown-subset. In this case, one cannot find any non-zero $\boldsymbol{u}$ vector that results in some trivial non-randomness. The number of initialisation rounds should be even larger to provide enough security margin.

One advantage of division property is that we can know how it propagates along with each round. Specifically, the propagation of division property for the three basic operations `XOR, AND`, and `COPY` can be modeled according to different rules, while all other complex operations, e.g., S-boxes and modulo additions, can be regarded as combinations of these basic operations [SWW17, ZR19, SWW19]. Thus, given a specific initial division property, we can learn how it involves and when the XOR sum becomes unknown for all non-zero $\boldsymbol{u}$ vectors, i.e., no non-randomness can be trivially found. A valid propagation trail from an initial division property to the output division property is called a *division trail*.

Furthermore, the propagation can be modeled as a MILP problem [XZBL16], where each basic operation is represented as one or a group of (in)equations. By setting different initial division properties, stopping conditions, and objective functions, the MILP problems are linked to different attacking scenarios. Some optimisation tools, e.g., Gurobi, can be used to efficiently solve a MILP problem, and one can correspondingly check if an attack exists based on the solutions.

Cube attacks based on division property. If $\boldsymbol{x}$ is a vector of n bit variables, say, $\boldsymbol{x} = (x_1, x_2 \ldots, x_n)$, the bit product function $\boldsymbol{x}^{\boldsymbol{u}}$ can denote a unique monomial term for each $\boldsymbol{u}$. For example, when $n = 4, \boldsymbol{u} = 1011$, the monomial term is $x_1x_3x_4$. By considering all $\boldsymbol{u}$ vectors, we can rewrite the ANF of a Boolean function in $\boldsymbol{x}$ as below:

$$f(\boldsymbol{x}) = \bigoplus_{\boldsymbol{u} \in \mathbb{F}_2^n} a_{\boldsymbol{u}} \cdot \boldsymbol{x}^{\boldsymbol{u}},$$

where $a_{\boldsymbol{u}}$ is the coefficient of the corresponding monomial term $\boldsymbol{x}^{\boldsymbol{u}}$.

The main steps of a cube attack based on division property follow a traditional cube attack, while with the main improvement in how the superpoly is recovered: the superpoly is recovered through recovering the values of all the monomial coefficients, thanks to the proposition below.

Proposition 1 ([TIHM18])**.** Let $f(\boldsymbol{x}, \boldsymbol{v})$ denote the polynomial of an output bit where $\boldsymbol{x}$ and $\boldsymbol{v}$ denote the n-bit secret key and m-bit IV variables, respectively. For a set of indices $I = \{i_1, i_2, \ldots, i_{|I|}\} \in \{1, 2, \ldots, m\}$, let $\mathcal{C}_I$ denote the cube in which $\{v_{i_1}, v_{i_2}, \ldots, v_{i_{|I|}}\}$ are taking all possible combinations of values. Let $\boldsymbol{k}_I$ be an m-dimensional bit vector such that $\boldsymbol{v}^{\boldsymbol{k}_I} = t_I = v_{i_1} v_{i_2} \cdots v_{i_{|I|}}$, i.e., $k_i = 1$ if $i \in I$ and $k_i = 0$ otherwise, and $\boldsymbol{e}_j$ be a unit vector of length n with one at the j-th position. If there is no division trail such that $(\boldsymbol{e}_j, \boldsymbol{k}_I) \xrightarrow{f} 1$, where $\xrightarrow{f}$ denotes the propagation of division property along f, the secret key bit x_j is not involved in the superpoly of the cube $\mathcal{C}_I$.

Thus, an attacker can define a cube and build a MILP problem for each secret key bit and get the information about whether this key bit is involved in the corresponding superpoly, based on the solutions of the MILP problem. After that, the superpoly can be recovered by trying out all possible combinations of the involved secret variables. Other steps are just the same as a traditional cube attack.

There are many new advances in cube attacks based on division property, e.g., estimating the degree of a superpoly [WHT+18], using three-subsets (i.e., one more subset containing the $\boldsymbol{u}$ vectors that result in an XOR sum of one) and its variants [TM16, HW19, WHG+18, HLM+21], and linking to other attacks [YT19, HJL+20]. In a recent paper [HSWW20], the authors give a new definition of division property which clarifies many open problems in this area. Compared to the traditional definition over a subset of inputs, the new one investigates the involvement of the polynomial itself, which is more accessible and understandable. Besides, they design accurate propagation rules for the division property, under which the resulting attack is a perfect detector, while some previous results may introduce false alarm errors.

4.4 Guess-and-determine Attacks

In a guess-and-determine (GnD) attack against a stream cipher, one guesses a part of the state variables and determines the remaining variables according to publicly known relations, e.g., the state update function, a recurrence relation derived from a multiple of the feedback polynomial, and the keystream generation function. Usually, several consecutive keystream words are required to be known, and the goal is to recover the full state that produce these given keystream words. If the required number of guesses

(in bits) is below the claimed security level, the attack is faster than exhaustive key search. A GnD attack can be generalised into three phases: *guessing*, *determining*, and *verification*.

1. **Guessing.** During the guessing phase, one decides a subset of the state variables for guessing. Such a subset should be able to determine the remaining variables and has a direct connection to the complexity; thus, one aims to find a subset of the minimum size. There are no systematic ways for achieving this, and ad-hoc approaches are explored for specific ciphers [HR02, FLZ+10, BP99]. Some heuristic path-searching approaches can be used to find a proper (but not always optimal) guessing subset [AE09, JLH20].

2. **Determining.** One assigns a value to the guessing subset and determines other variables according to the assigned value and the given keystream words, possibly in some well-organised order. If some conflicts appear in the middle, i.e., the assigned value invalidate some publicly known relations, one terminates the determining process and traces back to guess another value. The determining process may involve some tricky relations in which the solutions are unclear, e.g., an expression involving an unknown variable twice with non-linear operations in between. In this case, some efficient ways can be used to help find the solutions, e.g., using look-up tables and SAT solvers [ZK17].

3. **Verification.** After all the state variables corresponding to a specific time instance are fixed, i.e., either guessed or determined, one uses several additional keystream words to verify the correctness. This can be achieved by checking if the derived internal state can generate the keystream symbols identical to the observed ones. If the internal state is n-bit long, typically, at least n keystream bits are required to correctly recover the state.

If one loops over the guessing subset in a straightforward way, the complexity can be derived as 2^t, where t is the number of guesses in bits. In our included Paper IV [YJM21], we show that by carefully designing the order of the determining process, it is possible to terminate some guess-and-determine branches at an early stage through exploring potential conflicts. Thus, some efforts can be saved from going deeper and the complexity is reduced.

If the full state at a specific time instance is recovered, the attacker can run the cipher forward and backward to get the whole keystream sequence under this specific key and IV. If there is no special protection of the initialisation, e.g., using FP-(1) mode [HK18], the attacker can further recover the secret key.

The GnD attacks can be applied to other constructions as well. For example, we present two guess-and-determine-style attacks to the local pseudorandom generators in the included Paper VI [YGJL21], where we use Gaussian elimination and iterative decoding in the determining phase to recover the remaining variables.

4.5 Algebraic Attacks

An algebraic attack is a very powerful cryptanalysis method that applies potentially to a wide range of cryptosystems: stream ciphers, e.g., Toyocrypt [Cou02, Cou03], LILI-

128 [CM03], and E0 [AK03, Cou04a]; block ciphers, e.g., analysing AES [CP02]; and also some public cryptosystems, e.g., HFE [KS99]. The main idea behind an algebraic attack is to model a cryptosystem as a system of multivariate polynomial equations, and by solving this system using some methods reviewed in Section 2.8, one may recover the secret key or the whole state. An algebraic attack against a stream cipher can be generalised into three steps as below.

1. **Building an Initial System.** The first step is to construct an initial system of multivariate polynomial equations describing the relation between the secret key (or initial state) and the output. For example, for a stream cipher built over an LFSR with update function L filtered by a memoryless Boolean function f, which are both publicly known, we can get an equation at time instance t of the form below:
$$z_t = f(L^t(s_0, s_1, \ldots, s_{n-1})),$$
where $(s_0, s_1, \ldots, s_{n-1})$ is the initial LFSR state. When the filter function f has memory elements, several output bits should be considered in the above relation to build one equation [AK03, Cou04a]. If one collects m such equations, a multivariate polynomial system of m equations in n variables is constructed, and the task is to solve this system.

2. **Optimising the System.** If f has a low algebraic degree or can be approximated by a low-degree polynomial [Cou02], the initial system may be possibly solved efficiently if a reasonably large number of keystream bits are obtained. However, when f has a high algebraic degree, directly solving the initial system would involve high complexity. Thus, one should optimise the system to make it efficiently solvable. For example, one can find some well-chosen low-degree functions such that multiplying them with f results into low-degree multiples [CM03], i.e., finding low-degree functions g and h such that $fg = h$. Thus a new system of a lower degree is derived.

3. **Solving the Optimised System.** Once the system has been defined, the attacker will seek the most suitable methods, e.g., the methods mentioned in Section 2.8.2, to solve the system and get the solution.

The algebraic attack results in some new criteria for stream cipher design. For example, the filter function should involve many state bits, e.g., at least 32, and should not have a low degree or a low-degree multiple, and should not be too sparse [CM03].

4.6 Differential Attacks

Differential cryptanalysis is one of the most powerful cryptanalysis techniques for symmetric primitives, especially for block ciphers. It was proposed by Biham and Shamir and applied to DES in [BS91]. It investigates the propagation of differences inside the cipher and targets to find some specific output differences which occur with relatively higher probabilities.

Denote a primitive as F, suppose that two inputs X_1 and X_2 of the same length generate outputs Y_1 and Y_2, respectively, i.e., $Y_1 = F(X_1)$ and $Y_2 = F(X_2)$. The

input and output differences are then computed as $\Delta_I = X_1 \oplus X_2$ and $\Delta_O = Y_1 \oplus Y_2$, respectively. Such a (Δ_I, Δ_O) pair is called a *differential*, and we use $\Delta_I \to \Delta_O$ to denote it. The probability of the differential is denoted $P(\Delta_O|\Delta_I)$.

If F is an ideally randomizing cipher, the probability $P(\Delta_O|\Delta_I)$ will be 2^{-n} given any Δ_I, where n is the length of the output. In differential cryptanalysis, the attacker targets to find a specific Δ_I that will result in Δ_O with a probability distinguishably higher (or lower) than 2^{-n}. Thus, a differential attack is a CPA attack, in which the attacker can randomly select the input difference and examine the output difference. With a highly likely differential, the attacker can find some non-randomness of the output and can even recover some secret key bits.

Usually, one has to investigate how the difference propagates for each operation of the cipher, and such a propagation trail is called a *differential trail*. For a linear operation, the difference propagation can be predicted with probability one, while for a non-linear operation, probabilistic analysis is needed. The S-box is again the most interesting component. An S-box is called *active* if the input difference of it is nonzero; otherwise, it is *inactive*. A zero input difference of an S-box always gives a zero output difference. The attacker targets to carefully select the active S-boxes such that the overall differential holds with a higher probability. The MILP problem and corresponding optimisation tools can be used to help search for good differential trails [SHW+14].

There is no general framework for differential cryptanalysis of stream ciphers, and the approaches are mostly ad-hoc for specific stream ciphers. The differential characteristics can be defined either from the (key, IV) pair into the internal state, between the internal states, or from the internal state into the keystream [BD07]. The attacker can control the difference (Δ_K, Δ_{IV}) and observe the differences of the keystream sequences. The attacks can be related to weak-key or weak-IV attacks, where for some keys or IVs, the output differences deviate much from uniform random. For example, in the first version of ZUC, it is found that some special IV pairs can generate the same keystream sequence [WHN+12], which directly results in a new version of ZUC. The differential attacks are also applied to Toyocrypt, A5/1, and RC4 [BD07].

4.7 Time-memory-data Tradeoff Attacks

A time-memory-data tradeoff (TMD-TO) attack is a generic method of inverting a cipher by balancing time, memory, and data resources to make the overall complexity as low as possible. An underlying assumption is that these resources are sufficient for tradeoff. TMD-TO attacks can date back to the time-memory tradeoff attack applied to block ciphers given by Hellman in [Hel80]. This kind of attack can work without considering the specific structure of a cipher thus can be applied to any symmetric primitive, while on the other hand, may not be so powerful. One well-known application of the TMD-TO attack is to the A5/1 cipher [BSW00, Gol97].

A TMD-TO attack usually has the *preprocessing* and *real-time* two phases. During the preprocessing phase, one or several mapping tables from different secret keys (or internal states) to the outputs (e.g., keystream prefixes) are computed offline and stored with precomputation complexity P and memory M. Such a mapping defines a random function f which is easy to evaluate but hard to invert. During the real-time phase, an

attacker, who has captured D data, searches the data in the tables and expects to find a collision with time complexity T. When such a collision is found, the attacker can recover the internal state or even the secret key with a high probability. The key issue of TMD-TO attacks is to find a suitable trade-off between the sizes of the tables, the amount of required data, and the consumed time in the on-line phase.

In a TMD-TO attack against a stream cipher, the mapping function f can be defined from the internal state or secret key to the keystream prefix. The most well-known tradeoffs against a stream cipher are Babbage-Golic (BG) tradeoff [Bab95, Gol97] and Biryukov-Shamir (BS) tradeoff [BS00].

BG tradeoff. In the BG tradeoff, one first computes a table mapping M random initial states of size n bits to n-bit keystream prefixes. Denote the mapping function as f. The preprocessing time complexity P will be equal to M, i.e., $P = M$. During the online phase, one gets D keystream segments each of size n and searches these D segments in the table with complexity T, thus $T = D$. The D keystream segments can be generated from multiple key and IV pairs. Besides, an m-bit ($m > n$) keystream sequence under one specific key and IV pair can generate around $m-n+1$ n-bit keystream segments by considering each n consecutive keystream bits. When $TM = N$ where N is the number of possible values of the internal state, i.e., $N = 2^n$, it is expected to find one collision according to the birthday paradox, and the corresponding initial state is recovered. A typical point over the curve is $T = M = N^{\frac{1}{2}}$, which reduces the effective key length to the birthday bound $n/2$.

BS tradeoff. The BS tradeoff further involves the data into the tradeoff by combining the idea of BS tradeoff and Hellman's time-memory tradeoff for block ciphers [Hel80]. In the BS tradeoff, one constructs several tables, each using a distinct variant function f_i of the original function f. Each table contains m entries storing a (startpoint, endpoint) pairs: the startpoint is an n-bit random keystream prefix (assuming an n-bit internal state), and the endpoint is the result of iteratively applying f_i on the derived result t times. Assume the attacker captured D n-bit keystream segments, he can apply each f_i iteratively to each segment and compare the result with the m endpoints in the i-th table. The attack is successful if a match appears, which is likely to happen when $mt^2D = N$. Assume t/D tables are constructed, then $M = mt/D, P = N/D$. The attack time $T = t^2$, since for each of the D keystream segments, one has to iterate t/D functions t times. Thus the tradeoff curve is $TM^2D^2 = N^2, P = N/D, T \geq D^2$. This offers more trade-offs than the BG tradeoff, and covers the scenarios when less data may be obtained in practice. A typical point on this tradeoff curve is $T = M = N^{1/2}, D = N^{1/4}, P = N^{3/4}$.

The BG tradeoff and BS tradeoff consider the mapping function from an internal state to the keystream prefix, and the results yield the well-known design criterion of stream ciphers that the internal state length should be at least twice the length of the secret key. Modern stream ciphers typically adhere to this rule and have comparatively large internal state lengths. In 2018, the $FP(1)$-mode [HK18] for the initialisation was proposed which provides provable beyond-the-birthday-bound security of $\frac{2}{3}n$ against generic TMD-TO key recovery attacks. The basic idea is to include the secret key twice in the initialisation phase. Through this mode, even if the internal state at some time instance is recovered, it would still be difficult to recover the secret key. The cipher

LIZARD [HKM17] is the first instantiation of $FP(1)$-mode, and SNOW-V also adopts such a design.

A TMD-TO attack can also consider other mappings as well, e.g., from a (Key, IV) pair to the keystream prefix [HS05a, DK08]. Such tradeoff shows that if the IV is shorter than the key, the cipher is vulnerable to TMD-TO attacks, and it makes no sense to increase the size of the internal state of a stream cipher without increasing the size of the IV [DCLP05]. Besides, some improving techniques are frequently used in TMD-TO attacks, e.g., using *distinguished points* [BSW00] or *rainbow tables* [Oec03].

Chapter 5

Local Pseudorandom Generators

A pseudorandom generator (PRG) is a deterministic function that maps a short seed to a longer pseudorandom string, such that no statistical test can distinguish the string from an uniform random sequence. PRGs are widely used in cryptographic primitives and protocols, e.g., for generating (pseudo)random strings used as session keys in hybrid encryption systems or as challenges in challenge-response authentication mechanisms. The existence of PRGs in $\mathtt{NC}^0$ is much attractive, as it admits high efficiency and allows for advanced cryptographic applications. Such PRGs are called *local pseudorandom generators.*

5.1 Local Pseudorandom Generators

A local PRG is a pseudorandom generator in which each output bit only depends on a limited number of input bits. Such constructions are put into complexity class $\mathtt{NC}^0$, and can be implemented in parallel with a digital circuit with low depth, thus being highly efficient. The existence of local PRGs was established in [AIK06], which proved that many cryptographic tasks admit a local implementation. The most well-known construction is the Goldreich's pseudorandom generator, which is derived from a local one-way function [Gol00]. It is constructed as below.

Given a secret seed $\boldsymbol{x}$ of length n, $\boldsymbol{x} = (x_0, x_1, \ldots, x_{n-1})$, and a publicly known d-ary predicate $P : \{0,1\}^d \to \{0,1\}$, a Goldreich's PRG is defined as a function $f : \{0,1\}^n \to \{0,1\}^m$, where each of the m output bits is computed by applying P to an ordered subset of size d of the secret bits. Suppose the m d-tuple subsets are denoted as $x_{S_0}, x_{S_1}, \ldots, x_{S_{m-1}}$, the output sequence $\boldsymbol{y}$ is then computed as:

$$\begin{cases} y_0 = P(x_{S_0}) \\ y_1 = P(x_{S_1}) \\ y_2 = P(x_{S_2}) \\ \quad \ldots \\ y_{m-1} = P(x_{S_{m-1}}) \end{cases},$$

where y_i $(0 \leq i \leq m-1)$ is the i-th bit of $\boldsymbol{y}$. The integer d is called the *locality* of the PRG, which will decide the implementation complexity. A local PRG can be viewed as a *hypergraph* over n vertices and m ordered hyperedges each of cardinality d.

The length of the output m is an important parameter as it directly determines the efficiency and security of a local PRG. The safe stretches under different cryptanalysis techniques are widely investigated [CM01, MST03, AIK08, BQ09, OW14, App13]. The most aggressive and interesting setting is the polynomial stretch, i.e., $m = n^s$ for some constant $s > 1$, as it allows for many efficient or more advanced cryptographic applications [CDM+18], e.g., MPC-friendly primitives [MJSC16, ARS+15, GRR+16] and secure computation with constant computational overhead [ADI+17]. We say that a predicate P is s-pseudorandom if a local PRG instantiated over P with $m = n^s$ is likely to fool all efficiently computable tests [AL18]. However, such poly-stretch constructions result in overdetermined systems which may be susceptible to the solving methods described in Section 2.8. Thus, the choice of the predicate P is critical as well [ABR16].

The predicate P determines both the implementation efficiency and security. There are some properties that are important to be considered when choosing a predicate, which can be relevant to the properties of Boolean functions.

- **Resiliency.** A predicate is called k-resilient if it is balanced and has a k-order correlation immunity. High resiliency is one key requirement of P for achieving pseudorandomness. For example, it is shown in [FPV18, OW14] that resiliency larger than $2s-1$ yields s-pseudorandomness against attacks based on a large class of semidefinite programmings and statistical algorithms. In comparison, these attacks can break pseudorandomness when the resiliency is smaller than $2s-1$ [AL18].

- **Algebraic Degree.** The degree of P should not be one, otherwise Gaussian elimination can easily break the system even for $m = n$. Observed by [MST03], if the predicate P has an algebraic degree d, the resulting local PRG cannot be d-pseudorandom.

- **Rational Degree.** The rational degree of P is the smallest integer e for which there exist non-zero degree-e polynomials Q and R such that $PQ = R$. This is relevant to the algebraic attack where the attacker explores low-degree multiples of a polynomial. Thus, the rational degree characterises s-pseudorandomness against algebraic attacks [AL18].

- **Bit-fixing Degree.** A predicate P has r-bit fixing degree e if by fixing r input bits of P, the minimal $\mathbb{F}_2$ degree of a restriction is e. The bit-fixing degree was proposed in [AL18], and the authors show that the property of being s-pseudorandom against linear tests can be characterised by the resiliency and the bit fixing degree of the predicate.

With these properties taken into account, there are generally two types of predicates suggested and regarded as promising predicates for constructing local PRGs: the XOR-AND predicates [MST03, App16] and XOR-MAJ predicates [CM19, MCJS19a]. XOR-MAJ predicates are special instances of more generalised XOR-THR predicates.

Definition 17 (XOR-AND Predicate)**.** For positive integers k and q, an XOR_k-AND_q predicate is defined as below:

$$\text{XOR}_k\text{-AND}_q : x_1 + \cdots + x_k + x_{k+1}x_{k+2}\cdots x_{k+q}.$$

A concrete challenging parameter setting suggested in [App16] for XOR_k-AND_q predicates is $k = 2q$.

Definition 18 (XOR-THR Predicate)**.** For any positive integers k, d, and q such that $d \leq q + 1$, an XOR_k-$\text{THR}_{d,q}$ function is defined as below:

$$\text{XOR}_k\text{-THR}_{d,q} : x_1 + \cdots + x_k + \text{THR}_{d,q}(x_{k+1}, \ldots, x_{k+q}),$$

where $\text{THR}_{d,q}(x_{k+1}, \ldots, x_{k+q})$ is a threshold function whose value is one only when not less than d variables in the function are one; otherwise, the value is zero.

An XOR-MAJ predicate is a special form of the XOR-THR predicate, where the THR function is the majority function, i.e., the value will be one only when not less than half of the involved variables are one. Some well-chosen instances of XOR-MAJ predicates are used to build the new version of the FLIP cipher, called FiLIP [MCJS19a], which are intended for homomorphic encryption. It is mentioned in the paper that "no attack is known relatively to the functions XOR_k-$\text{THR}_{d,2d}$ or XOR_k-$\text{THR}_{d,2d-1}$ since $k \geq 2s$ and $d \geq s$".

A special predicate called P_5 has received much attention because of its simplicity and high efficiency. It is defined as below:

$$P_5(x_1, x_2, x_3, x_4, x_5) = x_1 \oplus x_2 \oplus x_3 \oplus x_4x_5.$$

P_5 can be regarded as a special case of the XOR-AND predicate of degree 2 or the XOR-THR predicate which has a threshold function $\text{THR}_{2,2}$. The local PRGs instantiated on P_5 achieve the best possible locality and have received much study. Our included Paper VI will investigate the concrete security of these local PRGs and more general ones instantiated on general XOR-AND and XOR-THR predicates with some suggested challenging parameters mentioned above.

5.2 Cryptanalysis of Local Pseudorandom Generators

The cryptanalysis of local PRGs can result in *key recovery* or *distinguishing attacks* indicating that the output sequence is not purely random. The cryptanalysis can be closely connected to solving multivariate systems of equations. The main cryptanalysis techniques can be categorised into *constraint satisfaction problem* (*CSP*)*-based analysis*, *myopics algorithms*, *algebraic attacks*, and *linear cryptanalysis*.

CSP-based Analysis. A local PRG system can be naturally viewed as a random constraint satisfaction problem in n variables, and each equation corresponds to one constraint. The inversion of the system can be naturally formulated as finding a "planted" solution for a CSP problem [OW14, App13]. Therefore, the ideas and results from solving CSPs can be adopted, e.g., using a SAT solver.

Myopic algorithms. Myopic algorithms are one kind of backtracking algorithm. In a myopic algorithm, an attacker in each step can read some output bits and assign a value to some input variables that are related to these read bits. The assigned values should be consistent with the already read output bits, i.e., all the read equations still hold. If some conflicts happen, the algorithm backtracks to some previous steps and assigns another value.

To resist myopic algorithms, the hypergraph should enjoy some expansion to avoid the "divide-and-conquer" situation. The expected success probability of a basic t-myopic algorithm is exponential low under sub-linear or linear stretches, while sub-exponential low under a polynomial stretch [CEMT09, ABR16].

Algebraic Attacks. The algebraic attacks against local PRGs are similar to the ones against stream ciphers. The system of polynomial equations corresponding to the outputs can be firstly manipulated and extended, e.g., by multiplying with low-degree polynomials, then solved by the classical methods described in Section 2.8.2, e.g., using linearisation or using Gröbner basis algorithms [App16, CDM$^+$18].

However, the situation becomes tricky for XOR-THR predicates, as the THR function cannot be processed in a straightforward way. That is one reason why XOR-THR predicates are considered to have more advantages over the XOR-AND predicates [MCJS19a].

Linear Cryptanalysis. In linear cryptanalysis, the non-linear terms are viewed as biased noises with a specific probability distribution. One can either cancel the noise by, for example, combining two equations sharing a same non-linear term, or exploring equations sharing the same linear part and take a majority vote over the noises to recover the value of the linear part with a high probability. In [CDM$^+$18], the authors introduce linear equations in local PRGs instantiated on P_5 through guessing the most often appeared variables in the quadratic terms, which results in a guess-and-determine-style attack. If many such linear equations are collected, one may recover the secret through solving a linear system [App16].

Linear tests can be viewed as one special type of linear cryptanalysis, in which the attacker considers linear combinations of multiple output bits and investigates the bias of these combinations. The result could be a distinguishing attack which indicates that the local PRG is not fully random. A predicate with either small resiliency or a small bit fixing degree can be susceptible to linear tests [AL18].

Sometimes, the recovered secret key is not fully correct, but the secret can still be recovered through some *self-correction* techniques, as long as the recovered secret is highly correlated with the true one [App16].

Chapter 6

Contributions and Future Work

This chapter summarises the contributions of the six included papers in the thesis, and presents some general conclusions and potential future work.

6.1 Contributions of the Thesis

This thesis has mainly covered research in the confidentiality and integrity protection algorithms for 5G, which primarily involves design and cryptanalysis of stream ciphers. Five papers strictly follow this topic, plus one more paper deviating a little bit to local pseudorandom generators, which could still be relevant to the main topic. These works provide some information to the community for better understanding the situation and encourage more research work to evaluate and design new algorithms for 5G and beyond. Figure 6.1 presents a general visualisation of the contributions of our papers.

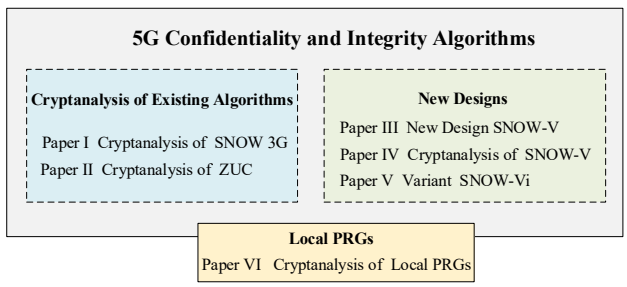

Figure 6.1: Contributions of the Thesis.

Specifically, in Paper I and II, we investigate the security of two confidentiality and integrity algorithms used in 4G: SNOW 3G and ZUC, and propose academic attacks against them which indicate that they cannot provide the full 256 bits of security. In Paper III, we design a new cipher called SNOW-V to satisfy the requirements of 5G in terms of speed and security, and perform deeper cryptanalysis of it in Paper IV. In Paper V, we introduce two minor modifications to SNOW-V and propose an extreme

performance variant called SNOW-Vi to cover more use cases, especially those where CPUs are less capable. In Paper VI, we deviate from the main research line a little bit and investigate the concrete security of local pseudorandom generators.

We elaborate the contributions in more detail for each paper below.

6.1.1 Paper I: Linear cryptanalysis of SNOW 3G [YJM19]

As 5G has new requirements in security on the confidentiality and integrity algorithms, it is crucial to investigate whether currently used ciphers in 4G satisfy these requirements. This motivates our work in Paper I, in which we investigate the security of SNOW 3G under 256-bit keys. Specifically, we perform linear cryptanalysis of it and propose two linear attacks with complexities much faster than exhaustive key search.

The basic idea is still to linearly approximate the non-linear FSM, but we try to explore a linear approximation in a higher dimension, which typically admits a higher bias. As the FSM has three 32-bit registers, we consider three consecutive keystream words and explore the correlation between these words and the LFSR states. We find a 24-bit linear approximation with bias measured in SEI of 2^{-40}, by concatenating the first bytes of the three consecutive keystream words. We experimentally verify this bias by constructing a large number of noise samples through running many SNOW 3G instances. With this linear approximation and its variants, we launch a distinguishing attack against SNOW 3G with complexity 2^{172} and a correlation attack with complexity 2^{177}.

These results indicate that if the key length of SNOW 3G was increased to 256 bits, there exist academic attacks much faster than exhaustive key search against it. In other words, SNOW 3G cannot satisfy the 5G requirement for the 256-bit security level. On the other hand, these attacks do not pose a direct threat to the cellular system in practice, as the attacks require very long keystream sequences (e.g., 2^{172}) while the maximum allowed keystream length corresponding to one (key, IV) pair is restricted, e.g., to 2^{64} in 4G. Nonetheless, our attacks become one motivating reason for SAGE to choose more efficient and secure candidates than SNOW 3G [saga].

6.1.2 Paper II: Spectral cryptanalysis of ZUC [YJM20]

Driven by the same motivation in Paper I, in Paper II, we perform linear cryptanalysis of another 4G standardised confidentiality and integrity algorithm, ZUC. One main contribution of the paper is the designed spectral tools that can have broad applications in linear cryptanalysis of symmetric cryptographic primitives. In the spectral analysis, we show how a linear masking in the original domain affects the spectral distribution for some commonly used operations, e.g., modulo addition, XOR, and S-box; and by aligning the points in the spectral domain, one can tweak the linear masks correspondingly in the time domain to admit a higher bias. Thus one can find more powerful maskings using a more efficient way. Another main contribution is our proposal of a new method for computing the bias of ZUC, which solves the main obstacle of linear cryptanalysis of ZUC introduced by the different fields in the linear and non-linear parts. We finally present a distinguishing attack against ZUC with complexity 2^{236}.

Though the improvement factor over exhaustive key search is not as significant as SNOW 3G, the results also indicate that ZUC cannot provide the full 256 bits of security.

Likewise, the attack does not pose an immediate threat to the usage of ZUC in 4G, due to the high data requirement in the attack while restricted lengths of keystream sequences in practice.

6.1.3 Paper III: A new design SNOW-V [EJMY19]

Considering the academic attacks in Paper I and Paper II against SNOW 3G and ZUC, respectively, plus their insufficient efficiency in software environments, in Paper III, we propose a new cipher called SNOW-V in response to the new requirements of 5G in terms of speed and security. We keep the general design rationale of the SNOW-family for SNOW-V, with a linear part consisting of two LFSRs feeding to each other and a non-linear part FSM. Both parts operate in larger sizes and are better aligned to allow for efficient implementations. In the FSM part, we use two AES encryption rounds to serve as the main source of non-linearity, whose implementation can take advantage of the intrinsic instructions, e.g., Intel AES-NI. An AEAD mode is also designed.

We further perform extensive cryptanalysis of SNOW-V in the paper and ensure that none of the classical cryptanalysis techniques applies to it faster than exhaustive key search. We also deeply evaluate the software and hardware implementations: the results show that SNOW-V can achieve speeds as high as 58 Gbps and 712 Gbps in a software and hardware environment, respectively.

SNOW-V is currently under evaluation in SAGE as a promising candidate for 5G confidentiality and integrity algorithms [saga, sagb]. Our paper has encouraged a number of research works devoted to investigating the security and performance of SNOW-V [JLH20, CBB20, HII+21, GZ21].

6.1.4 Paper IV: Cryptanalysis of SNOW-V [YJM21]

In Paper IV, we dig deeper into the security of SNOW-V and particularly investigate its resistance against guess-and-determine attacks and linear cryptanalysis.

We propose two GnD attacks against SNOW-V. We reformulate the way of computing the complexity of a GnD attack, and based on it design better guessing strategies. Specifically, we carefully design the order of the guessing and exploit as much side information as possible, to either reduce the required number of guesses or truncate as many guessing paths as possible to reduce the attack complexity. The complexities of the two GnD attacks are 2^{384} and 2^{378}, respectively.

In our linear cryptanalysis, we consider a simplified variant of SNOW-V, where the modulo additions are replaced with XOR operations. We explore various approximations to involve as few active S-boxes as possible. We finally build a 16-bit linear approximation by using a special linear matrix masking that can cancel 11 S-boxes out of 48. With this linear approximation, we launch a distinguishing attack of complexity 2^{303}.

The paper has helped us to better understand the security of SNOW-V. Specifically, the paper has sparked some ideas for further improving SNOW-V, which are incorporated into the variant SNOW-Vi in Paper V.

6.1.5 Paper V: An extreme performance variant SNOW-Vi [EMJY21]

In Paper V, we propose an extreme performance variant of SNOW-Vi to cover more use cases, especially those where CPUs are less capable. This is a response to some feedback from the community saying that some CPUs serving in base stations are relatively less powerful, and such use cases are not fully covered by SNOW-V.

We introduce two minor modifications in the LFSR part of SNOW-V to get SNOW-Vi, while keeping the FSM part unchanged. One modification is using two simpler but still safe fields for the LFSRs in SNOW-Vi, which reduces the update cost and allows for more efficient implementations. The second modification is moving one tap position from the FLSR to FSM to another location. The new position can provide better resistance against linear attacks, a better Key/IV mixing effect, and more efficient implementations. Some observations are from the cryptanalysis results of SNOW-V in Paper IV.

We perform extensive software evaluation on eight different platforms with different capabilities, which support different register sizes from 128 bits to 256 bits and different instruction sets from the old SSE4.2 to the latest AVX-512. The results show that the speeds of SNOW-Vi are increased by 50% over SNOW-V on all the eight different platforms and can be up to 92 Gbps. The security properties are kept and in some aspects even strengthened.

6.1.6 Paper VI: Cryptanalysis of local PRGs [YGJL21]

Our Paper VI deviates the main research line a little bit and targets the security of Goldreich's pseudorandom generators instantiated on the P_5 predicate. We exploit the low locality of P_5 and propose a guess-and-determine attack and a guess-and-decode attack using iterative decoding.

Both attacks have a guessing phase with similar guessing strategies. The strategies are based on the observation that some guessed variables can determine some other variables for free, and the designed guessing strategies target to introduce as many such "free" variables as possible. After guessing a certain number of variables, the guess-and-determine attack can recover the seed by solving a system of linear equations derived after guessing. We perform theoretical analysis and derive the asymptotic complexity of the attack, which matches well with our extensive experimental results. The results significantly improve the state-of-the-art algorithm proposed by Couteau et al. at ASIACRYPT 2018 [CDM+18] in terms of both asymptotic and concrete complexity.

In our guess-and-decode attack, instead of solving a linear system, we employ iterative decoding to solve the quadratic system and recover the secret, which further reduces the attack complexity. As classical iterative decoding applies to linear systems, we design new belief propagation techniques for non-linear parity checks. We experimentally verify these attacks and break the existing suggested challenge parameters with a large gap. We further propose some new challenge parameters to encourage more cryptanalysis.

We also extend the attacks to other local PRGs based on more general predicates, e.g., XOR-AND and XOR-MAJ predicates, by carefully designing the guessing strategies and belief propagation techniques.

Our results provide some insights in the possible secure parameters for local PRGs over promising predicates. Besides, our belief propagation techniques provide a new idea for solving a non-linear sparse system.

6.2 Conclusions and Future Work

6.2.1 Conclusions

Summarising the thesis work, we could have the following general conclusions.

- When designing a new cryptographic primitive, especially a symmetric one, designers should visit all the promising algorithmic attacks and ensure that none of them applies to the new design faster than exhaustive key search. Cryptanalysis results help to better understand the security of the design and may result in improvements, e.g., the cryptanalysis of SNOW-V results in one modification in SNOW-Vi. Besides, the design should balance between security, performance, and cost, which is though a well-known criterion. Specifically, the new designs in the future can focus more than before on the performance in software environments, considering that a communication network in the future will be more cloudified. However, hardware implementations, including the speed and cost, should still be considered for the massive IoT devices, which may have constrained resources.

- For LFSR-based stream ciphers, linear cryptanalysis is always one of the most powerful analysis techniques and should be primarily considered. Particularly, when the cipher is defined over larger fields, one can always try to explore higher-dimension linear approximations, which might provide more biased information. Besides, spectral tools like DFT and WHT can help find good linear maskings more efficiently and speed up bias computing. A guess-and-determine attack can be more powerful against a stream cipher if one carefully designs the guessing order and tries to truncate invalid guessing paths at an early stage of the guessing.

- For the confidentiality and integrity algorithms for 5G, SNOW 3G and ZUC might not be the best candidates due to the academic attacks against them and their insufficient efficiency in software environments. On the other hand, as we mentioned, these attacks are not immediate threats in practice. If SNOW 3G and ZUC were kept in 5G, more efficient implementations in software should be investigated to provide higher speeds. SNOW-V and its variant are promising candidates, while their security might need more investigation.

- When constructing local PRGs for efficient computation or advanced applications, the underlying predicates should have reasonably large localities as the belief propagation in our guess-and-decode attack applies to low-weight predicates with any form of non-linearity. XOR-MAJ predicates can be better candidates than XOR-AND predicates, as they have better cryptographic properties and resistance against our attacks. If in some extreme case a low-locality predicate, e.g., P_5, is used, the seed size should be large enough to guarantee security.

6.2.2 Future Work

Given the conclusions above, we think that there are several directions that can be further explored in the future.

Further investigations into SNOW-V and SNOW-Vi. Further investigations into SNOW-V and SNOW-Vi would be beneficial to provide more reference for the evaluation in SAGE and gain more confidence. Firstly, the security of these two ciphers should be further and extensively investigated and check if some modifications are needed based on the investigations. For example, in a recent pre-print paper [SJZ+21], Shi et al. propose correlation attacks against SNOW-V and SNOW-Vi with complexity around 2^{249}, slightly faster than exhaustive key search. This requires further investigations into the weaknesses that result in the attacks and making corresponding improvements. 3GPP has been made aware of these cryptanalysis results and asked SAGE to assess and determine any implications on SNOW-V [sagb]. Moreover, the performance of these two ciphers in software and hardware in different use cases and on a larger range of CPUs should be tested, which requires a full knowledge of the CPUs and circuits that are being used and will be used in the future. If we go a little bit further, the side-channel resistant implementations both in software and hardware should be looked into as well.

Investigating other ciphers for 5G and beyond. There are some other ciphers claiming for use in 5G and beyond, e.g., the Rocca cipher proposed in 2021 [SLN+21]. More focus can be devoted to AEAD ciphers based on, e.g., sponge constructions, as they can typically provide high security and performance. Currently, the cellular network (including 5G) does not support AEAD yet, but how about if 6G protocol opens up for it? It is crucial to investigate these constructions in advance to better understand their security and performance and help choose good candidates for future use. Besides the primitives for broadband communication, the lightweight cryptographic primitives targeting massive machine communication and low-latency communication also deserve more investigations from different aspects.

Spectral tools extended to other ciphers. Our vectorised approximation idea and spectral analysis tools can be possibly extended and applied to other constructions. I personally speculate that the spectral tools may be well applied to ARX-based ciphers since the transforms between different domains for the two main operations, XOR and modulo addition, are already investigated.

More investigations to local PRGs and relevant constructions. Our guess-and-determine and guess-and-decode attacks have mainly applied to P_5 and some general XOR-AND and XOR-MAJ predicates, while there are many new predicates or improved variants appearing as well, e.g., the variant of P_5 proposed in [LV17]. The attacks may have high complexity when directly applied to these predicates, as these predicates can be more complex. However, we believe that some advanced techniques can help, e.g., using a trellis or look-up tables. That will be one of our future work. Besides, we believe that our algorithmic attacks could give tighter theoretical bounds for some predicates, e.g., for XOR-AND predicates, which deserves further investigation. The peer reviewers also suggest investigating more in the theoretical complexity of the guess-and-decode attack, especially the complexity of iterative decoding for non-linear systems, for future work, and we think *density evolution* [CF02] might help. A similar idea using other decoding techniques can also be considered and applied to a broader range of instances, not only restricted to local PRGs.

References

[3GP06] 3GPP. Specification of the 3GPP confidentiality and integrity algorithms UEA2& UIA2, document 2: SNOW 3G specification, version 1.1, 2006.

[3GP19] 3GPP. TS 33.841 (V16.1.0): 3rd generation partnership project; technical specification group services and systems aspects; security aspects; study on the support of 256-bit algorithms for 5G. March 2019. `https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3422`.

[3GP21a] 3GPP. 3gpp. ts 23.501 (v17.1.1): 3rd generation partnership project; technical specification group services and system aspects; system architecture for the 5G system (5GS). June 2021. `https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3144`.

[3GP21b] 3GPP. TS 33.501 (V17.2.1): 3rd generation partnership project; technical specification group services and systems aspects; security architecture and procedures for 5G system. July 2021. `https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3169`.

[AB15] Roberto Avanzi and Billy Bob Brumley. Faster 128-EEA3 and 128-EIA3 software. In *Information Security*, pages 199–208. Springer, 2015.

[ABR16] Benny Applebaum, Andrej Bogdanov, and Alon Rosen. A dichotomy for local small-bias generators. *Journal of Cryptology*, 29(3):577–596, 2016.

[ADI+17] Benny Applebaum, Ivan Damgård, Yuval Ishai, Michael Nielsen, and Lior Zichron. Secure arithmetic computation with constant computational overhead. In *Annual International Cryptology Conference*, pages 223–254. Springer, 2017.

[ADMS09] Jean-Philippe Aumasson, Itai Dinur, Willi Meier, and Adi Shamir. Cube testers and key recovery attacks on reduced-round MD6 and Trivium. In *International Workshop on Fast Software Encryption*, pages 1–22. Springer, 2009.

[AE09] Hadi Ahmadi and Taraneh Eghlidos. Heuristic guess-and-determine attacks on stream ciphers. *IET Information Security*, 3(2):66–73, 2009.

[AIK06] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Cryptography in NC^0. *SIAM Journal on Computing*, 36(4):845–888, 2006.

[AIK08] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. On pseudorandom generators with linear stretch in NC^0. *Computational Complexity*, 17(1):38–69, 2008.

[AK03] Frederik Armknecht and Matthias Krause. Algebraic attacks on combiners with memory. In *Annual International Cryptology Conference*, pages 162–175. Springer, 2003.

[AL18] Benny Applebaum and Shachar Lovett. Algebraic attacks against random local functions and their countermeasures. *SIAM Journal on Computing*, 47(1):52–79, 2018.

[Ala07] Jesse Alama. The rank+ nullity theorem. *Formalized Mathematics*, 15(3):137–142, 2007.

[Alb10] Martin Albrecht. *Alorithmic algebraic techniques and their application to block cipher cryptanalysis*. PhD thesis, Citeseer, 2010.

[AM15] Frederik Armknecht and Vasily Mikhalev. On lightweight stream ciphers with shorter internal states. In *International Workshop on Fast Software Encryption*, pages 451–470. Springer, 2015.

[App13] Benny Applebaum. Pseudorandom generators with long stretch and low locality from random local one-way functions. *SIAM Journal on Computing*, 42(5):2008–2037, 2013.

[App16] Benny Applebaum. Cryptographic hardness of random local functions. *Computational complexity*, 25(3):667–722, 2016.

[ARS+15] Martin R Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 430–454. Springer, 2015.

[B+08] Daniel J Bernstein et al. ChaCha, a variant of Salsa20. In *Workshop record of SASC*, volume 8, pages 3–5, 2008.

[Bab95] SH Babbage. Improved "exhaustive search" attacks on stream ciphers. In *European Convention on Security and Detection, 1995.*, pages 161–166. IET, 1995.

[BB86] Ronald Newbold Bracewell and Ronald N Bracewell. *The Fourier transform and its applications*, volume 31999. McGraw-Hill New York, 1986.

[BD07] Eli Biham and Orr Dunkelman. Differential cryptanalysis of stream ciphers. Technical report, Computer Science Department, Technion, 2007.

[BDPVA09] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak sponge function family main document. *Submission to NIST (Round 2)*, 3(30):320–337, 2009.

[Ber08] Daniel J Bernstein. The Salsa20 family of stream ciphers. In *New stream cipher designs*, pages 84–97. Springer, 2008.

[Ber15] Elwyn R Berlekamp. *Algebraic coding theory (revised edition)*. World Scientific, 2015.

[BJV04] Thomas Baigneres, Pascal Junod, and Serge Vaudenay. How far can we go beyond linear cryptanalysis? In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 432–450. Springer, 2004.

[BKR11] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique cryptanalysis of the full AES. In *International conference on the theory and application of cryptology and information security*, pages 344–371. Springer, 2011.

[BP99] Daniel Bleichenbacher and Sarvar Patel. SOBER cryptanalysis. In *International Workshop on Fast Software Encryption*, pages 305–316. Springer, 1999.

[BP17] Alex Biryukov and Léo Paul Perrin. State of the art in lightweight symmetric cryptography. 2017. `https://orbilu.uni.lu/bitstream/10993/31319/1/SoK___lightweight_crypto.pdf`, Accessed: 2021-11-02.

[BQ09] Andrej Bogdanov and Youming Qiao. On the security of Goldreich's one-way function. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 392–405. Springer, 2009.

[Bro17] Gabriel Brown. Cloud-RAN, the next-generation mobile network architecture, Huawei white paper, 2017.

[BS91] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. *Journal of Cryptology*, 4(1):3–72, 1991.

[BS00] Alex Biryukov and Adi Shamir. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 1–13. Springer, 2000.

[BSW00] Alex Biryukov, Adi Shamir, and David Wagner. Real time cryptanalysis of A5/1 on a PC. In *International Workshop on Fast Software Encryption*, pages 1–18. Springer, 2000.

[CAE] CAESAR: Competition for authenticated encryption: Security, applicability, and robustness. `https://competitions.cr.yp.to/caesar.html`. Accessed: 2021-07-15.

[CBB20] Andrea Caforio, Fatih Balli, and Subhadeep Banik. Melting SNOW-V: improved lightweight architectures. *Journal of Cryptographic Engineering*, pages 1–21, 2020.

[CCH10] Claude Carlet, Yves Crama, and Peter L Hammer. Boolean functions for cryptography and error-correcting codes., 2010.

[CDM+18] Geoffroy Couteau, Aurélien Dupin, Pierrick Méaux, Mélissa Rossi, and Yann Rotella. On the concrete security of Goldreich's pseudorandom generator. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 96–124. Springer, 2018.

[CEMT09] James Cook, Omid Etesami, Rachel Miller, and Luca Trevisan. Goldreich's one-way function candidate and myopic backtracking algorithms. In *Theory of Cryptography Conference*, pages 521–538. Springer, 2009.

[CF02] Jinghu Chen and Marc PC Fossorier. Density evolution for two improved BP-based decoding algorithms of LDPC codes. *IEEE communications letters*, 6(5):208–210, 2002.

[CHJ02] Don Coppersmith, Shai Halevi, and Charanjit Jutla. Cryptanalysis of stream ciphers with linear masking. In *Annual International Cryptology Conference*, pages 515–532. Springer, 2002.

[CJM02] Philippe Chose, Antoine Joux, and Michel Mitton. Fast correlation attacks: An algorithmic point of view. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 209–221. Springer, 2002.

[CJS00] Vladimor V Chepyzhov, Thomas Johansson, and Ben Smeets. A simple algorithm for fast correlation attacks on stream ciphers. In *International Workshop on Fast Software Encryption*, pages 181–195. Springer, 2000.

[CKPS00] Nicolas Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 392–407. Springer, 2000.

[CM01] Mary Cryan and Peter Bro Miltersen. On pseudorandom generators in NC^0. In *International Symposium on Mathematical Foundations of Computer Science*, pages 272–284. Springer, 2001.

[CM03] Nicolas T Courtois and Willi Meier. Algebraic attacks on stream ciphers with linear feedback. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 345–359. Springer, 2003.

[CM19] Claude Carlet and Pierrick Méaux. Boolean functions for homomorphic-friendly stream ciphers. In *International Conference on Algebra, Codes and Cryptology*, pages 166–182. Springer, 2019.

[Cou02] Nicolas T Courtois. Higher order correlation attacks, XL algorithm and cryptanalysis of Toyocrypt. In *International Conference on Information Security and Cryptology*, pages 182–199. Springer, 2002.

[Cou03] Nicolas T Courtois. Fast algebraic attacks on stream ciphers with linear feedback. In *Annual International Cryptology Conference*, pages 176–194. Springer, 2003.

[Cou04a] Nicolas T Courtois. Algebraic attacks on combiners with memory and several outputs. In *International Conference on Information Security and Cryptology*, pages 3–20. Springer, 2004.

[Cou04b] Nicolas T Courtois. General principles of algebraic attacks and new design criteria for cipher components. In *International Conference on Advanced Encryption Standard*, pages 67–83. Springer, 2004.

[Cov99] Thomas M Cover. *Elements of information theory*. John Wiley & Sons, 1999.

[CP02] Nicolas T Courtois and Josef Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In *International conference on the theory and application of cryptology and information security*, pages 267–287. Springer, 2002.

[CS17] Thomas W Cusick and Pantelimon Stanica. *Cryptographic Boolean functions and applications*. Academic Press, 2017.

[CT00] Anne Canteaut and Michaël Trabbia. Improved fast correlation attacks using parity-check equations of weight 4 and 5. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 573–588. Springer, 2000.

[DC06] Christophe De Canniere. Trivium: A stream cipher construction inspired by block cipher design principles. In *International Conference on Information Security*, pages 171–186. Springer, 2006.

[DCLP05] Christophe De Canniere, Joseph Lano, and Bart Preneel. Comments on the rediscovery of time memory data tradeoffs. *Available as a link on the ECRYPT Call for Stream Cipher Primitives*, 3, 2005.

[DK08] Orr Dunkelman and Nathan Keller. Treatment of the initial value in time-memory-data tradeoff attacks on stream ciphers. *Information Processing Letters*, 107(5):133–137, 2008.

[DLC07] Frédéric Didier and Yann Laigle-Chapuy. Finding low-weight polynomial multiples using discrete logarithm. In *2007 IEEE International Symposium on Information Theory*, pages 1036–1040. IEEE, 2007.

[DLR16] Sébastien Duval, Virginie Lallemand, and Yann Rotella. Cryptanalysis of the FLIP family of stream ciphers. In *Annual International Cryptology Conference*, pages 457–475. Springer, 2016.

[DR99] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael. 1999.

[DS09] Itai Dinur and Adi Shamir. Cube attacks on tweakable black box polynomials. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 278–299. Springer, 2009.

[DS11] Itai Dinur and Adi Shamir. Breaking Grain-128 with dynamic cube attacks. In *International Workshop on Fast Software Encryption*, pages 167–187. Springer, 2011.

[DY09] Jintai Ding and Bo-Yin Yang. Multivariate public key cryptography. In *Post-quantum cryptography*, pages 193–241. Springer, 2009.

[EJMY19] Patrik Ekdahl, Thomas Johansson, Alexander Maximov, and Jing Yang. A new SNOW stream cipher called SNOW-V. *IACR Transactions on Symmetric Cryptology*, pages 1–42, 2019.

[EJT07] Håkan Englund, Thomas Johansson, and Meltem Sönmez Turan. A framework for chosen IV statistical analysis of stream ciphers. In *International Conference on Cryptology in India*, pages 268–281. Springer, 2007.

[EMJY21] Patrik Ekdahl, Alexander Maximov, Thomas Johansson, and Jing Yang. SNOW-Vi: an extreme performance variant of SNOW-V for lower grade CPUs. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 261–272, 2021.

[ETS09] ETSI/SAGE. Specification of the 3GPP confidentiality and integrity algorithms. document 2: KASUMI specification, 2009.

[ETS11] ETSI/SAGE. Specification of the 3GPP confidentiality and integrity algorithms 128-EEA3 & 128-EIA3. document 2: ZUC specification, 2011.

[Fau99] Jean-Charles Faugere. A new efficient algorithm for computing Gröbner bases (F4). *Journal of pure and applied algebra*, 139(1-3):61–88, 1999.

[Fau02] Jean Charles Faugere. A new efficient algorithm for computing Gröbner bases without reduction to zero (F5). In *Proceedings of the 2002 international symposium on Symbolic and algebraic computation*, pages 75–83, 2002.

[FLZ+10] Xiutao Feng, Jun Liu, Zhaocun Zhou, Chuankun Wu, and Dengguo Feng. A byte-based guess and determine attack on SOSEMANUK. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 146–157. Springer, 2010.

[For88] Réjane Forrié. The strict avalanche criterion: spectral properties of Boolean functions and an extended definition. In *Conference on the Theory and Application of Cryptography*, pages 450–468. Springer, 1988.

[FPV18] Vitaly Feldman, Will Perkins, and Santosh Vempala. On the complexity of random satisfiability problems with planted solutions. *SIAM Journal on Computing*, 47(4):1294–1338, 2018.

[FWG+16] Kai Fu, Meiqin Wang, Yinghua Guo, Siwei Sun, and Lei Hu. MILP-based automatic search algorithms for differential and linear trails for speck. In *International Conference on Fast Software Encryption*, pages 268–288. Springer, 2016.

[GCK13] Sourav Sen Gupta, Anupam Chattopadhyay, and Ayesha Khalid. Designing integrated accelerator for stream ciphers with structural similarities. *Cryptography and Communications*, 5(1):19–47, 2013.

[GHC16] Vahid Amin Ghafari, Honggang Hu, and Ying Chen. Fruit-v2: ultra-lightweight stream cipher with shorter internal state. *Int. Assoc. Cryptol. Res (IACR)*, 2016.

[Gol96a] Jovan Dj Golic. Computation of low-weight parity-check polynomials. *Electronics Letters*, 32(21):1981–1982, 1996.

[Gol96b] Jovan Dj Golić. Correlation properties of a general binary combiner with memory. *Journal of Cryptology*, 9(2):111–126, 1996.

[Gol97] Jovan Dj Golić. Cryptanalysis of alleged A5 stream cipher. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 239–255. Springer, 1997.

[Gol00] Oded Goldreich. Candidate one-way functions based on expander graphs. *IACR Cryptol. ePrint Arch.*, 2000:63, 2000.

[Gol08] Oded Goldreich. Computational complexity: a conceptual perspective. *ACM Sigact News*, 39(3):35–39, 2008.

[GRR+16] Lorenzo Grassi, Christian Rechberger, Dragos Rotaru, Peter Scholl, and Nigel P Smart. MPC-friendly symmetric key primitives. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 430–443, 2016.

[Gur] Gurobi. `https://www.gurobi.com/`. Accessed: 2021-10-13.

[GZ21] Xinxin Gong and Bin Zhang. Resistance of SNOW-V against fast correlation attacks. *IACR Transactions on Symmetric Cryptology*, pages 378–410, 2021.

[HCN19] Miia Hermelin, Joo Yeon Cho, and Kaisa Nyberg. Multidimensional linear cryptanalysis. *Journal of Cryptology*, 32(1):1–34, 2019.

[Hel80] Martin Hellman. A cryptanalytic time-memory trade-off. *IEEE transactions on Information Theory*, 26(4):401–406, 1980.

[HH89] Joachim Hagenauer and Peter Hoeher. A Viterbi algorithm with soft-decision outputs and its applications. In *1989 IEEE Global Telecommunications Conference and Exhibition'Communications Technology for the 1990s and Beyond'*, pages 1680–1686. IEEE, 1989.

[HII+21] Jin Hoki, Takanori Isobe, Ryoma Ito, Fukang Liu, and Kosei Sakamoto. Distinguishing and key recovery attacks on the reduced-round SNOW-V. *IACR Cryptol. ePrint Arch.*, 2021:546, 2021.

[HJB09] Martin Hell, Thomas Johansson, and Lennart Brynielsson. An overview of distinguishing attacks on stream ciphers. *Cryptography and Communications*, 1(1):71–94, 2009.

[HJL+20] Yonglin Hao, Lin Jiao, Chaoyun Li, Willi Meier, Yosuke Todo, and Qingju Wang. Links between division property and other cube attack variants. *IACR Transactions on Symmetric Cryptology*, pages 363–395, 2020.

[HJM07] Martin Hell, Thomas Johansson, and Willi Meier. Grain: a stream cipher for constrained environments. *International journal of wireless and mobile computing*, 2(1):86–93, 2007.

[HK18] Matthias Hamann and Matthias Krause. On stream ciphers with provable beyond-the-birthday-bound security against time-memory-data tradeoff attacks. *Cryptography and Communications*, 10(5):959–1012, 2018.

[HKM17] Matthias Hamann, Matthias Krause, and Willi Meier. LIZARD-a lightweight stream cipher for power-constrained devices. *IACR Transactions on Symmetric Cryptology*, 2017(1):45–79, 2017.

[HLM+21] Yonglin Hao, Gregor Leander, Willi Meier, Yosuke Todo, and Qingju Wang. Modeling for three-subset division property without unknown subset. *Journal of Cryptology*, 34(3):1–69, 2021.

[HOP96] Joachim Hagenauer, Elke Offer, and Lutz Papke. Iterative decoding of binary block and convolutional codes. *IEEE Transactions on Information Theory*, 42(2):429–445, 1996.

[HP11] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[HR02] Philip Hawkes and Gregory G Rose. Guess-and-determine attacks on SNOW. In *International Workshop on Selected Areas in Cryptography*, pages 37–46. Springer, 2002.

[HS05a] Jin Hong and Palash Sarkar. New applications of time memory data tradeoffs. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 353–372. Springer, 2005.

[HS05b] Jin Hong and Palash Sarkar. Rediscovery of time memory tradeoffs. *IACR Cryptol. ePrint Arch.*, 2005:90, 2005.

[HSWW20] Kai Hu, Siwei Sun, Meiqin Wang, and Qingju Wang. An algebraic formulation of the division property: Revisiting degree evaluations, cube attacks, and key-independent sums. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 446–476. Springer, 2020.

[HW19] Kai Hu and Meiqin Wang. Automatic search for a variant of division property using three subsets. In *Cryptographers' Track at the RSA Conference*, pages 412–432. Springer, 2019.

[HWX+17] Senyang Huang, Xiaoyun Wang, Guangwu Xu, Meiqin Wang, and Jingyuan Zhao. Conditional cube attack on reduced-round keccak sponge function. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 259–288. Springer, 2017.

[Int] Intel intrinsics guide. `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/`. Accessed: 2021-11-02.

[JHF20] Lin Jiao, Yonglin Hao, and Dengguo Feng. Stream cipher designs: a review. *Science China Information Sciences*, 63(3):1–25, 2020.

[JJ99a] Thomas Johansson and Fredrik Jönsson. Fast correlation attacks based on turbo code techniques. In *Annual International Cryptology Conference*, pages 181–197. Springer, 1999.

[JJ99b] Thomas Johansson and Fredrik Jönsson. Improved fast correlation attacks on stream ciphers via convolutional codes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 347–362. Springer, 1999.

[JLH20] Lin Jiao, Yongqiang Li, and Yonglin Hao. A guess-and-determine attack on SNOW-V stream cipher. *The Computer Journal*, 63(12):1789–1812, 2020.

[Jou09] Antoine Joux. *Algorithmic cryptanalysis*. Chapman and Hall/CRC, 2009.

[JV03] Pascal Junod and Serge Vaudenay. Optimal key ranking procedures in a statistical cryptanalysis. In *International Workshop on Fast Software Encryption*, pages 235–246. Springer, 2003.

[KHF10] Ryan Kastner, Anup Hosangadi, and Farzan Fallah. *Arithmetic optimization techniques for hardware and software design*. Cambridge University Press, 2010.

[KL20] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2020.

[KS99] Aviad Kipnis and Adi Shamir. Cryptanalysis of the HFE public key cryptosystem by relinearization. In *Annual International Cryptology Conference*, pages 19–30. Springer, 1999.

[Lat09] Joel Lathrop. *Cube attacks on cryptographic hash functions*. Rochester Institute of Technology, 2009.

[LC01] Shu Lin and Daniel J Costello. *Error control coding*. Prentice hall Scarborough, 2001.

[LD16] Yi Lu and Yvo Desmedt. Walsh transforms and cryptographic applications in bias computing. *Cryptography and Communications*, 8(3):435–453, 2016.

[LJ14] Carl Löndahl and Thomas Johansson. Improved algorithms for finding low-weight polynomial multiples in $\mathbb{F}_2[x]$ and some cryptographic applications. *Designs, codes and cryptography*, 73(2):625–640, 2014.

[LV04] Yi Lu and Serge Vaudenay. Faster correlation attack on Bluetooth keystream generator E0. In *Annual International Cryptology Conference*, pages 407–425. Springer, 2004.

[LV17] Alex Lombardi and Vinod Vaikuntanathan. Minimizing the complexity of Goldreich's pseudorandom generator. *IACR Cryptol. ePrint Arch.*, 2017:277, 2017.

[LZM+16] Zongbin Liu, Qinglong Zhang, Cunqing Ma, Changting Li, and Jiwu Jing. HPAZ: A high-throughput pipeline architecture of ZUC in hardware. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 269–272. IEEE, 2016.

[MAM16] Vasily Mikhalev, Frederik Armknecht, and Christian Müller. On ciphers that continuously access the non-volatile key. *IACR Transactions on Symmetric Cryptology*, pages 52–79, 2016.

[Mas69] James Massey. Shift-register synthesis and BCH decoding. *IEEE transactions on Information Theory*, 15(1):122–127, 1969.

[Mat93] Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 386–397. Springer, 1993.

[McE96] Robert J McEliece. On the BCJR trellis for linear block codes. *IEEE Transactions on Information Theory*, 42(4):1072–1092, 1996.

[MCJS19a] Pierrick Méaux, Claude Carlet, Anthony Journault, and François-Xavier Standaert. Improved filter permutators: Combining symmetric encryption design, Boolean functions, low complexity cryptography, and homomorphic encryption, for private delegation of computations. *IACR Cryptol. ePrint Arch.*, 2019:483, 2019.

[MCJS19b] Pierrick Méaux, Claude Carlet, Anthony Journault, and François-Xavier Standaert. Improved filter permutators for efficient FHE: better instances and implementations. In *International Conference on Cryptology in India*, pages 68–91. Springer, 2019.

[Mei11] Willi Meier. Fast correlation attacks: Methods and countermeasures. In *International Workshop on Fast Software Encryption*, pages 55–67. Springer, 2011.

[MFI01] Miodrag J Mihaljevi, Marc PC Fossorier, and Hideki Imai. Fast correlation attack algorithm with list decoding and an application. In *International Workshop on Fast Software Encryption*, pages 196–210. Springer, 2001.

[MJ05] Alexander Maximov and Thomas Johansson. Fast computation of large distributions and its cryptographic applications. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 313–332. Springer, 2005.

[MJSC16] Pierrick Méaux, Anthony Journault, François-Xavier Standaert, and Claude Carlet. Towards stream ciphers for efficient FHE with low-noise ciphertexts. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 311–343. Springer, 2016.

[MM12] Seiichi Matsuda and Shiho Moriai. Lightweight cryptography for the cloud: exploit the power of bitslice implementation. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 408–425. Springer, 2012.

[MPC04] Willi Meier, Enes Pasalic, and Claude Carlet. Algebraic attacks and decomposition of Boolean functions. In *International conference on the theory and applications of cryptographic techniques*, pages 474–491. Springer, 2004.

[MS89] Willi Meier and Othmar Staffelbach. Fast correlation attacks on certain stream ciphers. *Journal of cryptology*, 1(3):159–176, 1989.

[MS92] Willi Meier and Othmar Staffelbach. Correlation properties of combiners with memory in stream ciphers. *Journal of Cryptology*, 5(1):67–86, 1992.

[MST03] Elchanan Mossel, Amir Shpilka, and Luca Trevisan. On ϵ-biased generators in NC^0. In *Annual Symposium on Foundations of Computer Science*, volume 44, pages 136–145. Citeseer, 2003.

[MVOV18] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2018.

[MWGP11] Nicky Mouha, Qingju Wang, Dawu Gu, and Bart Preneel. Differential and linear cryptanalysis using mixed-integer linear programming. In *International Conference on Information Security and Cryptology*, pages 57–76. Springer, 2011.

[NISa] NIST lightweight cryptography competition. `https://csrc.nist.gov/projects/lightweight-cryptography/finalists`. Accessed: 2021-10-13.

[NISb] NIST post-quantum cryptography competition. `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization`. Accessed: 2021-07-15.

[NW06] Kaisa Nyberg and Johan Wallén. Improved linear distinguishers for SNOW 2.0. In *International Workshop on Fast Software Encryption*, pages 144–162. Springer, 2006.

[Nyb01] Kaisa Nyberg. Correlation theorems in cryptanalysis. *Discrete Applied Mathematics*, 111(1-2):177–188, 2001.

[O'C94] Luke O'Connor. Properties of linear approximation tables. In *International Workshop on Fast Software Encryption*, pages 131–136. Springer, 1994.

[O'D14] Ryan O'Donnell. *Analysis of Boolean functions*. Cambridge University Press, 2014.

[Oec03] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In *Annual International Cryptology Conference*, pages 617–630. Springer, 2003.

[OW14] Ryan ODonnell and David Witmer. Goldreich's PRG: evidence for near-optimal polynomial stretch. In *2014 IEEE 29th Conference on Computational Complexity (CCC)*, pages 1–12. IEEE, 2014.

[PB06] Bart Preneel and An BRAEKEN. Cryptographic properties of Boolean functions and S-boxes. *Departement elektrotechniek (ESAT)*, 2006.

[Qui04] Jeremy Quirke. Security in the GSM system. *AusMobile, May*, pages 1–26, 2004.

[RB08] Matthew Robshaw and Olivier Billet. *New stream cipher designs: the eSTREAM finalists*, volume 4986. Springer, 2008.

[RL09] William Ryan and Shu Lin. *Channel codes: classical and modern*. Cambridge university press, 2009.

[Rue85] Rainer A Rueppel. Correlation immunity and the summation generator. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 260–272. Springer, 1985.

[Saa06] Markku-Juhani O Saarinen. Chosen-IV statistical attacks on eSTREAM stream ciphers. *Proc. Stream Ciphers Revisited SASC*, 2006.

[saga] S3-211497: DRAFT Response LS on 256-bit algorithms based on SNOW 3G or SNOW V. `https://www.3gpp.org/ftp/tsg_sa/WG3_Security/TSGS3_103e/Inbox/Drafts`, Accessed: 2021-11-12.

[sagb] S3-213267: Response LS on 256-bit algorithms based on SNOW 3G or SNOW V. `https://www.3gpp.org/ftp/tsg_sa/WG3_Security/TSGS3_104e/Docs/S3-213267.zip`, Accessed: 2021-11-12.

[sagc] SageMath. `https://www.sagemath.org/`. Accessed: 2021-10-20.

[SB88] Miles E Smid and Dennis K Branstad. Data Encryption Standard: past and future. *Proceedings of the IEEE*, 76(5):550–559, 1988.

[Ser15] M Series. IMT Vision–Framework and overall objectives of the future development of IMT for 2020 and beyond. *Recommendation ITU*, 2083:0, 2015.

[Sho94] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. IEEE, 1994.

[SHW+14] Siwei Sun, Lei Hu, Peng Wang, Kexin Qiao, Xiaoshuang Ma, and Ling Song. Automatic security evaluation and (related-key) differential characteristic search: application to SIMON, PRESENT, LBlock, DES(L) and other bit-oriented block ciphers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 158–178. Springer, 2014.

[Sie84] Thomas Siegenthaler. Correlation-immunity of nonlinear combining functions for cryptographic applications. *IEEE Transactions on Information theory*, 30(5):776–780, 1984.

[SJZ+21] Zhen Shi, Chenhui Jin, Jiyan Zhang, Ting Cui, and Lin Ding. A correlation attack on full SNOW-V and SNOW-Vi. *Cryptology ePrint Archive*, 2021.

[SLN+21] Kosei Sakamoto, Fukang Liu, Yuto Nakano, Shinsaku Kiyomoto, and Takanori Isobe. Rocca: An efficient AES-based encryption scheme for beyond 5G. *IACR Transactions on Symmetric Cryptology*, pages 1–30, 2021.

[SS16] Nigel P Smart and Nigel P Smart. *Cryptography made simple*. Springer, 2016.

[Sta10] Paul Stankovski. Greedy distinguishers and nonrandomness detectors. In *International Conference on Cryptology in India*, pages 210–226. Springer, 2010.

[Sta13] Paul Stankovski. *Cryptanalysis of Selected Stream Ciphers*. Lund University, 2013.

[SWW17] Ling Sun, Wei Wang, and Meiqin Wang. Automatic search of bit-based division property for ARX ciphers and word-based division property. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 128–157. Springer, 2017.

[SWW19] Ling Sun, Wei Wang, and Meiqin Q Wang. MILP-aided bit-based division property for primitives with non-bit-permutation linear layers. *IET Information Security*, 14(1):12–20, 2019.

[TIA18] Yosuke Todo, Takanori Isobe, and Kazumaro Aoki. Fast correlation attack revisited–cryptanalysis on full Grain-128a, Grain-128, and Grain-v1 (from crypto2018). *IEICE Technical Report; IEICE Tech. Rep.*, 118(212):59–59, 2018.

[TIHM18] Yosuke Todo, Takanori Isobe, Yonglin Hao, and Willi Meier. Cube attacks on non-blackbox polynomials based on division property. *IEEE Transactions on Computers*, 67(12):1720–1736, 2018.

[TM16] Yosuke Todo and Masakatu Morii. Bit-based division property and application to SIMON family. In *International Conference on Fast Software Encryption*, pages 357–377. Springer, 2016.

[Tod15] Yosuke Todo. Structural evaluation by generalized integral property. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 287–314. Springer, 2015.

[UMHA16] Rei Ueno, Sumio Morioka, Naofumi Homma, and Takafumi Aoki. A high throughput/gate AES hardware architecture by compressing encryption and decryption datapaths. In *International conference on cryptographic hardware and embedded systems*, pages 538–558. Springer, 2016.

[Vau96] Serge Vaudenay. An experiment on DES statistical cryptanalysis. In *Proceedings of the 3rd ACM Conference on Computer and Communications Security*, pages 139–147, 1996.

[Wag02] David Wagner. A generalized birthday problem. In *Annual International Cryptology Conference*, pages 288–304. Springer, 2002.

[WHG+18] Senpeng Wang, Bin Hu, Jie Guan, Kai Zhang, and Tairong Shi. MILP method of searching integral distinguishers based on division property using three subsets. *IACR Cryptol. ePrint Arch.*, 2018:1186, 2018.

[WHN+12] Hongjun Wu, Tao Huang, Phuong Ha Nguyen, Huaxiong Wang, and San Ling. Differential attacks against stream cipher ZUC. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 262–277. Springer, 2012.

[WHT+18] Qingju Wang, Yonglin Hao, Yosuke Todo, Chaoyun Li, Takanori Isobe, and Willi Meier. Improved division property based cube attacks exploiting algebraic properties of superpoly. In *Annual International Cryptology Conference*, pages 275–305. Springer, 2018.

[Wie86] Douglas Wiedemann. Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory*, 32(1):54–62, 1986.

[WP13] Hongjun Wu and Bart Preneel. AEGIS: A fast authenticated encryption algorithm. In *International Conference on Selected Areas in Cryptography*, pages 185–201. Springer, 2013.

[XZBL16] Zejun Xiang, Wentao Zhang, Zhenzhen Bao, and Dongdai Lin. Applying MILP method to searching integral distinguishers based on division property for 6 lightweight block ciphers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 648–678. Springer, 2016.

[YGJL21] Jing Yang, Qian Guo, Thomas Johansson, and Michael Lentmaier. Revisiting the concrete security of Goldreich's pseudorandom generator. *IEEE Transactions on Information Theory*, accepted, 2021.

[YJ20] Jing Yang and Thomas Johansson. An overview of cryptographic primitives for possible use in 5G and beyond. *Science China Information Sciences*, 63(12):1–22, 2020.

[YJM19] Jing Yang, Thomas Johansson, and Alexander Maximov. Vectorized linear approximations for attacks on SNOW 3G. *IACR Transactions on Symmetric Cryptology*, pages 249–271, 2019.

[YJM20] Jing Yang, Thomas Johansson, and Alexander Maximov. Spectral analysis of ZUC-256. *IACR Transactions on Symmetric Cryptology*, pages 266–288, 2020.

[YJM21] Jing Yang, Thomas Johansson, and Alexander Maximov. Improved guess-and-determine and distinguishing attacks on SNOW-V. *IACR Transactions on Symmetric Cryptology*, pages 54–83, 2021.

[YT19] Chen-Dong Ye and Tian Tian. Revisit division property based cube attacks: key-recovery or distinguishing attacks? *IACR Transactions on Symmetric Cryptology*, pages 81–102, 2019.

[ZK17] Oleg Zaikin and Stepan Kochemazov. An improved SAT-based guess-and-determine attack on the alternating step generator. In *International Conference on Information Security*, pages 21–38. Springer, 2017.

[ZR19] Wenying Zhang and Vincent Rijmen. Division cryptanalysis of block ciphers with a binary diffusion layer. *IET Information Security*, 13(2):87–95, 2019.

[ZXM15] Bin Zhang, Chao Xu, and Willi Meier. Fast correlation attacks over extension fields, large-unit linear approximation and cryptanalysis of SNOW 2.0. In *Annual Cryptology Conference*, pages 643–662. Springer, 2015.

Part II

Included Papers

Paper I

Vectorized linear approximations for attacks on SNOW 3G

Abstract

SNOW 3G is a stream cipher designed in 2006 by ETSI/SAGE, serving in 3GPP as one of the standard algorithms for data confidentiality and integrity protection. It is also included in the 4G LTE standard. In this paper we derive vectorized linear approximations of the finite state machine in SNOW 3G. In particular, we show one 24-bit approximation with a bias around 2^{-37} and one byte-oriented approximation with a bias around 2^{-40}. We then use the approximations to launch attacks on SNOW 3G. The first approximation is used in a distinguishing attack resulting in an expected complexity of 2^{172} and the second one can be used in a standard fast correlation attack resulting in key recovery in an expected complexity of 2^{177}. If the key length in SNOW 3G would be increased to 256 bits, the results show that there are then academic attacks on such a version faster than the exhaustive key search.

Keywords: SNOW 3G, Stream Cipher, 5G Mobile System Security.

1 Introduction

SNOW 3G is a word-oriented stream cipher being used as the core of 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2 for UMTS and LTE networks [ETS06a]. It is one member of the SNOW family with two predecessors SNOW 1.0 [EJ00] and SNOW 2.0 [EJ02]. SNOW 1.0 was submitted to NESSIE project in 2000 by Ekdahl and Johansson but was refused due to some weakness. In 2002, the improved version SNOW 2.0 was published and later selected as an ISO standard in 2005. The SNOW ciphers consist of a linearly updated part through an LFSR (Linear Feedback Shift Register) and a non-linear part referred to as an FSM (Finite State Machine). They are all based on operations on 32-bit words, making them quite efficient in both software and hardware environments. SNOW 3G differs from SNOW 2.0 by introducing a third 32-bit register in the FSM and a second 32-bit S-box application to update that register. This presumably makes SNOW 3G a much harder target in an attack compared to SNOW 2.0.

Jing Yang, Thomas Johansson, and Alexander Maximov. Vectorized linear approximations for attacks on SNOW 3G. IACR Transactions on Symmetric Cryptology, pages 249–271, 2019.

Just as for other stream ciphers, the class of linear approximation attacks, like distinguishing attacks and correlation attacks, is the main threat to the SNOW ciphers. The basic idea for these attacks is to approximate nonlinear blocks used in the cipher with linear expressions and then derive a linear relationship between output values from different time instances. In a distinguishing attack, a cryptanalyst tries to derive some samples from the keystream and find evidence that such a sample sequence is not behaving like a truly random sequence using some statistical tools, e.g., hypothesis testing. When the linear relationship also involves symbols from the LFSR states, some correlation between the keystream and the LFSR states can be explored to recover the key, which is the foundation of a correlation attack.

Several distinguishing attacks and correlation attacks have been proposed on SNOW 1.0 and SNOW 2.0, where the basic idea is to approximate the FSM part. In [CHJ02], a distinguishing attack on SNOW 1.0 with complexity 2^{100} was proposed using linear masking to get a binary approximation with a bias $2^{-8.3}$, which became one reason for the rejection of SNOW 1.0 from the NESSIE project. In order to resist against this attack, the authors improved the design and proposed SNOW 2.0, which is, however, still vulnerable to some distinguishing attacks. A distinguishing attack based on a linear masking method with complexity 2^{225} and an improved version with complexity 2^{174} were proposed in [WBDC03] and [NW06], respectively. In [LLP08] and [ZXM15], correlation attacks on SNOW 2.0 were proposed with complexities $2^{204.38}$ and $2^{164.15}$. For SNOW 3G, however, no significant attack of this type has ever been published.

The cryptographic security of SNOW 3G has been studied in depth. As an algorithm appearing in a main standard, it has been thoroughly evaluated by the standardization consortium before its adoption. Some evaluation results can be found in [ETS06b]. There are several side-channel attacks and fault attacks, targeting specific implementations of the algorithm, such as the attacks given in [DC09] and [BHNS10]. There are also attacks targeting the initialization phase on versions of SNOW 3G with reduced number of initialization rounds [BPSZ10b, BPSZ10a]. At FSE 2006, Nyberg and Wallén [NW06] examined linear distinguishing attacks on SNOW 2.0, but devoted one section to SNOW 3G. The best linear approximation of the FSM they found had a bias of 2^{-274} and they also derived an upper bound on 2^{-96} for any binary linear approximation. Here the bias as given in the paper was recalculated, now expressed using *Squared Euclidean Imbalance*, as is commonly used for non-binary linear approximations. Note that [NW06] considered only binary approximations and the key to improvements is to use approximations over larger alphabets.

In this paper, we give one distinguishing attack and one correlation attack on SNOW 3G by finding efficient linear approximations of the nonlinear part of the FSM. We derive a 24-bit linear approximation[1] by masking and truncating three consecutive keystream words with the bias $2^{-37.37}$ and we further derive an 8-bit approximation from the 24-bit one with the bias $2^{-40.97}$. The 24-bit approximation is then employed to launch a distinguishing attack requiring a keystream length of around 2^{172}. This strongest and largest 24-bit approximation cannot be used in a correlation attack, but the derived 8-bit approximation, which is linear over $GF(2^8)$ can be used to give a correlation attack which

[1]The 24-bit noise distribution of the linear approximation is available at `https://portal.research.lu.se/portal/sv/publications/vectorized-linear-approximations-for-attacks-on-snow-3g(80dd21a7-5111-4af3-89b2-9a9661c040c2).html`.

has complexity around 2^{177}. This is to the best of our knowledge the first significant result on attacking the full SNOW 3G. In particular, if the key length in SNOW 3G would be increased to 256 bits, the results show that there are then academic attacks on such a version faster than the exhaustive key search.

The rest of this paper is organized as follows. We briefly describe the design and structure of SNOW 3G in Section 2 and then elaborate on the process of finding linear approximations of the FSM in Section 3. In Section 4, we give the experimental verification of the approximations by running the cipher to get a large number of samples. In Section 5, we give a distinguishing attack and a correlation attack, based on the vectorized linear approximations derived in Section 3, and in Section 6, we conclude the paper.

2 Description of SNOW 3G

In this section, we give a brief description of the SNOW 3G algorithm. We first note that a stream cipher like SNOW 3G takes as input a secret key K and a public value known as the IV(initial vector) value. For each such pair of key and IV, (K, IV), the algorithm produces an output sequence, usually called the *keystream*, denoted $z^{(t)}, t = 1, 2, \ldots$. In SNOW 3G, the key and IV are both 128-bit long and each keystream symbol is a 32-bit word, so we write $z^{(t)} \in GF(2^{32}), t = 1, 2, \ldots$. Furthermore, each pair should produce a unique keystream sequence, and the typical operation of such a stream cipher is to produce many different keystreams for many different public IV values, while using the same key.

The overall schematic of SNOW 3G algorithm is shown in Figure 1. Just as SNOW 1.0 and SNOW 2.0, it consists of a linear part, the LFSR, and a nonlinear part, the FSM. The FSM is used to break the linearity of the LFSR contribution. For more details on the design of SNOW 3G, we refer to the original design document [ETS06a].

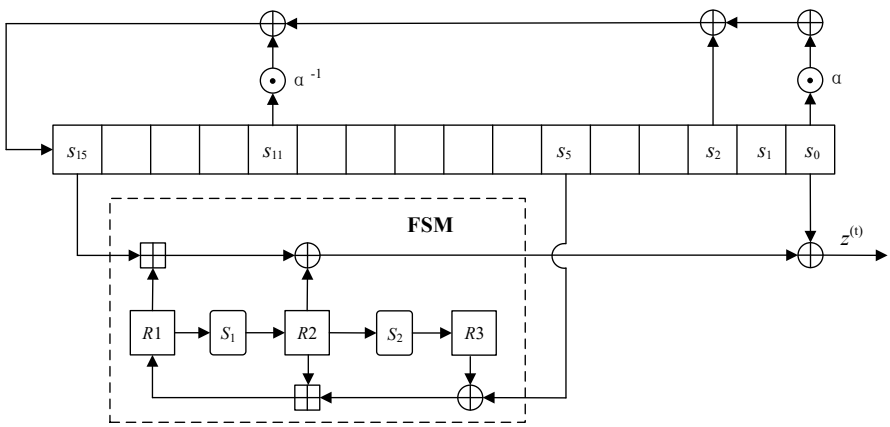

Figure 1: The keystream generation phase of the SNOW-3G stream cipher

The LFSR part consists of 16 cells denoted $(s_0, s_1, ..., s_{15})$ each containing 32 bits

thus having 512 bits in total. Every value in a cell is considered as an element from $GF(2^{32})$ and the LFSR sequence is defined by the generating polynomial

$$P(x) = \alpha x^{16} + x^{14} + \alpha^{-1}x^5 + 1 \in GF(2^{32})[x],$$

where α is a root of the polynomial $x^4 + \beta^{23}x^3 + \beta^{245}x^2 + \beta^{48}x + \beta^{239} \in GF(2^8)[x]$ and β is a root of $x^8 + x^7 + x^5 + x^3 + 1 \in GF(2)[x]$. If we denote the state at clock t as $(s_0^{(t)}, s_1^{(t)}, ..., s_{15}^{(t)})$, then at the next clock $t+1$, $s_i^{(t)}$ is shifted to $s_{i-1}^{(t+1)}$, i.e., $s_i^{(t)} = s_{i-1}^{(t+1)}$, for $1 \leq i \leq 15$, while $s_{15}^{(t+1)}$ is updated by:

$$s_{15}^{(t+1)} = \alpha^{-1}s_{11}^{(t)} \oplus s_2^{(t)} \oplus \alpha s_0^{(t)},$$

where $\oplus$ denotes a bitwise XOR operation. Note that α and α^{-1} are two constants in $GF(2^{32})$ and the update consequently includes two multiplications in this field.

For the FSM part, it has three internal 32-bit registers $R1$, $R2$ and $R3$, connected by some linear and nonlinear operations. The FSM takes two words from the LFSR part as the inputs, s_{15} and s_5, and outputs a 32-bit keystream word by xoring with s_0, giving the following formula for the generation of the keystream:

$$z^{(t)} = (R1^{(t)} \boxplus_{32} s_{15}^{(t)}) \oplus R2^{(t)} \oplus s_0^{(t)}.$$

Here $\boxplus_{32}$ denotes integer addition modulo 2^{32}. The registers in the FSM are then updated through the following steps:

$$\begin{aligned} R2^{(t+1)} &= S_1(R1^{(t)}), \\ R3^{(t+1)} &= S_2(R2^{(t)}), \\ R1^{(t+1)} &= R2^{(t)} \boxplus_{32} (R3^{(t)} \oplus s_5^{(t)}), \end{aligned}$$

where S_1, S_2 are substitution boxes (S-boxes) composed of four bytewise substitutions followed by the MixColumn operation of Rijndael. Below we give the details of how they are constructed by using little-endian style. The 32-bit registers in FSM could be expressed as four parallel bytes. Let $W = (w_0, w_1, w_2, w_3)$ be the input to the substitution boxes with w_0 being the least and w_3 the most significant byte. The operations of the two S-boxes are as follows.

S-Box S_1: S_1 is a 32-bit to 32-bit mapping operating on four bytes. Bytes are interpreted as elements of GF(2^8) defined by the polynomial $x^8 + x^4 + x^3 + x + 1$. The underlaying 8-bit S-box $S_R(x)$ is the Rijndael AES SBox [DR13]. In general, S_1 is described by

$$S_1(W) = L_1 \cdot S_R(W),$$

which can be expressed in more details as follows. Let $R = (r_0, r_1, r_2, r_3)$ be the four byte output through $R = S_1(W)$. Then

$$\begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \end{pmatrix} = \begin{pmatrix} x & x+1 & 1 & 1 \\ 1 & x & x+1 & 1 \\ 1 & 1 & x & x+1 \\ x+1 & 1 & 1 & x \end{pmatrix} \cdot \begin{pmatrix} S_R(w_0) \\ S_R(w_1) \\ S_R(w_2) \\ S_R(w_3) \end{pmatrix}. \quad (1)$$

S-Box S_2: S_2 is also a 32-bit to 32-bit mapping operating on four bytes. Bytes are again interpreted as elements of GF(2^8) but this time defined by the polynomial $y^8 + y^6 + y^5 + y^3 + 1$. The underlaying 8-bit SBox $S_Q(x)$ is another 8-to-8 bit S-box derived from the Dickson polynomials. In general, S_2 is described by

$$S_2(W) = L_2 \cdot S_Q(W),$$

and in more details $S_2(W)$ is:

$$\begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \end{pmatrix} = \begin{pmatrix} y & y+1 & 1 & 1 \\ 1 & y & y+1 & 1 \\ 1 & 1 & y & y+1 \\ y+1 & 1 & 1 & y \end{pmatrix} \cdot \begin{pmatrix} S_Q(w_0) \\ S_Q(w_1) \\ S_Q(w_2) \\ S_Q(w_3) \end{pmatrix}. \tag{2}$$

Like other stream ciphers, SNOW 3G has the initialization phase during which the cipher is clocked without producing output, to fully mix the key and IV into the LFSR state and the FSM registers. During the initialization phase, the key and the IV, each consisting of four 32-bit words, are first loaded into the LFSR state and the registers in the FSM are initialized to be zero. Then the cipher runs 32 times with the output from the FSM feeding back to the LFSR instead of giving a keystream output. After the initialization, the cipher enters the keystream mode, with the first output word from the FSM being discarded and the following FSM outputs form the keystream by xoring with s_0. Since the attacks in this paper only use the keystream mode, we do not give more details of the initilization mode, but refer to the design document [ETS06a].

3 Approximations of the FSM

A main class of attacks on stream ciphers are the so-called linear distinguishing attacks and (fast) correlation attacks. They both build on the idea of approximating some nonlinear operations as linear ones, thereby introducing some approximation noise. The most simple form utilizes binary approximations and has a strong connection to linear cryptanalysis of block ciphers. Recent work in cryptanalysis of stream ciphers have shown that approximations on larger alphabets can improve the attacks considerably, e.g., in [ZXM15] the authors used the terminology large-unit linear approximations.

For stream ciphers built around an LFSR, it makes sense to provide approximations that are linear with respect to some binary algebraic structure, such as $GF(2)^n$ or $GF(2^n)$. Since the LFSR part is linearly described in $GF(2)^n$ or possibly $GF(2^n)$, the main obstacle is to approximate the FSM. We will return in Section 5 to the case of how to use an approximation in attacks on the full cipher.

The FSM part in SNOW 3G takes inputs from $s_{15}^{(t)}, s_5^{(t)}, s_0^{(t)}$ and outputs $z^{(t)}$, with t varying. It also contains three unknown values in the registers $R1$, $R2$ and $R3$. As such, they need to be cancelled and a linear approximation of the FSM can thus be described as a linear expression including only $s_{15}^{(t)}, s_5^{(t)}, s_0^{(t)}$ and $z^{(t)}$ for different t values. Such an expression is a good approximation if the corresponding expression has a distribution that is biased. So in general, we are interested in finding an expression of the form

$$\bigoplus_{i \in I} (c_z^{(t+i)} z^{(t+i)} \oplus c_{15}^{(t+i)} s_{15}^{(t+i)} \oplus c_5^{(t+i)} s_5^{(t+i)} \oplus c_0^{(t+i)} s_0^{(t+i)}),$$

for some time set I, where operations are in $GF(2)^n$ (or $GF(2^n)$), and $c_z^{(t+i)}, c_{15}^{(t+i)}, c_5^{(t+i)}$, $c_0^{(t+i)}$ are now m-dimensional matrices and the inputs are considered as column vectors. In order to determine the quality of an approximation, we consider m-bit random variables $E^{(t)}$, defined as the above expression, i.e.,

$$E^{(t)} = \bigoplus_{i \in I} (c_z^{(t+i)} z^{(t+i)} \oplus c_{15}^{(t+i)} s_{15}^{(t+i)} \oplus c_5^{(t+i)} s_5^{(t+i)} \oplus c_0^{(t+i)} s_0^{(t+i)}).$$

Each such random variable has the same distribution, denoted D. The quality of the linear approximation is measured by the bias of the distribution, which can be defined in many ways. Using the SEI (Squared Euclidean Imbalance) as defined in [BJV04], the bias for the distribution D is computed as

$$\epsilon(D) = |D| \cdot \sum_{x=0}^{|D|-1} \left(D(x) - \frac{1}{|D|} \right)^2.$$

We note that when the bias is measured in SEI, the number of samples required to distinguish samples drawn from D from the uniform distribution is in the order of $1/\epsilon(D)$ [BJV04], [HG97]. We are now ready to investigate how to find expressions of the above form with a large bias.

3.1 A 24-bit linear approximation of the FSM

In this first approach, we are targeting an approximation with as large alphabet size as possible, in order to get a bias as large as possible. The novel parts consist in determining how to build the approximation and how to efficiently compute the bias when the alphabet size and the number of involved variables are large.

Let $\hat{R1}, \hat{R2}, \hat{R3}$ be the content of the FSM registers at time t. Then consider the following word-oriented expressions on three consecutive keystream words:

$$\begin{aligned} z^{(t-1)} &= (S_R^{-1}(L_1^{-1}\hat{R2}) \boxplus s_{15}^{(t-1)}) \oplus S_Q^{-1}(L_2^{-1}\hat{R3}) \oplus s_0^{(t-1)}, \\ z^{(t)} &= (\hat{R1} \boxplus s_{15}^{(t)}) \oplus \hat{R2} \oplus s_0^{(t)}, \\ L_1^{-1} z^{(t+1)} &= L_1^{-1}(\hat{R2} \boxplus (\hat{R3} \oplus s_5^{(t)}) \boxplus s_{15}^{(t+1)}) \oplus S_R(\hat{R1}) \oplus L_1^{-1} s_0^{(t+1)}. \end{aligned} \tag{3}$$

Let us introduce the following notation that applies to three byte-oriented 32-bit vectors:

$$\begin{pmatrix} A \\ B \\ C \end{pmatrix}_{[i,j,k]} = \begin{pmatrix} A[i] \\ B[j] \\ C[k] \end{pmatrix},$$

where i, j, k are corresponding bytes of A, B, and C, respectively. So $A[i]$ denotes the i-th byte of a 32-bit byte-oriented vector, for $i = 0, 1, 2$ or 3.

Now we consider a three-byte sampling of the following form:

$$\underbrace{\begin{pmatrix} z^{(t-1)} \\ z^{(t)} \\ L_1^{-1} z^{(t+1)} \end{pmatrix}_{[0,0,0]}}_{\text{Sampling in time } (t)} = \underbrace{\begin{pmatrix} (S_R^{-1}(L_1^{-1}\hat{R2}) \boxplus s_{15}^{(t-1)}) \oplus s_{15}^{(t-1)} \oplus S_Q^{-1}(L_2^{-1}\hat{R3}) \\ \hat{R2} \\ L_1^{-1}[(\hat{R2} \boxplus (\hat{R3} \oplus s_5^{(t)}) \boxplus s_{15}^{(t+1)}) \oplus s_5^{(t)} \oplus s_{15}^{(t+1)}] \end{pmatrix}_{[0,0,0]}}_{\text{24-bit Noise } N2}$$
$$\oplus \underbrace{\begin{pmatrix} 0 \\ (\hat{R1} \boxplus s_{15}^{(t)}) \oplus s_{15}^{(t)} \\ S_R(\hat{R1}) \end{pmatrix}_{[0,0,0]}}_{\text{24-bit Noise } N1} \oplus \underbrace{\begin{pmatrix} s_0^{(t-1)} \oplus s_{15}^{(t-1)} \\ s_{15}^{(t)} \oplus s_0^{(t)} \\ L_1^{-1}[s_0^{(t+1)} \oplus s_5^{(t)} \oplus s_{15}^{(t+1)}] \end{pmatrix}_{[0,0,0]}}_{\text{Contribution from the LFSR}}.$$

Basically, we want to achieve a linear approximation where the bias is as large as possible and the choice of multiplying $z^{(t+1)}$ with L_1^{-1} is chosen to have a really small influence of the noise related to the register $\hat{R1}$.

As already indicated in the above formula, our linear approximation is

$$\begin{pmatrix} z^{(t-1)} \\ z^{(t)} \\ L_1^{-1} z^{(t+1)} \end{pmatrix}_{[0,0,0]} = \begin{pmatrix} s_0^{(t-1)} \oplus s_{15}^{(t-1)} \\ s_{15}^{(t)} \oplus s_0^{(t)} \\ L_1^{-1}[s_0^{(t+1)} \oplus s_5^{(t)} \oplus s_{15}^{(t+1)}] \end{pmatrix}_{[0,0,0]} \oplus N_{tot}^{(t)},$$

where $N_{tot}^{(t)} = N1^{(t)} \oplus N2^{(t)}$ and

$$N1^{(t)} = \begin{pmatrix} 0 \\ (\hat{R1} \boxplus s_{15}^{(t)}) \oplus s_{15}^{(t)} \\ S_R(\hat{R1}) \end{pmatrix}_{[0,0,0]},$$

$$N2^{(t)} = \begin{pmatrix} (S_R^{-1}(L_1^{-1}\hat{R2}) \boxplus s_{15}^{(t-1)}) \oplus s_{15}^{(t-1)} \oplus S_Q^{-1}(L_2^{-1}\hat{R3}) \\ \hat{R2} \\ L_1^{-1}[(\hat{R2} \boxplus (\hat{R3} \oplus s_5^{(t)}) \boxplus s_{15}^{(t+1)}) \oplus s_5^{(t)} \oplus s_{15}^{(t+1)}] \end{pmatrix}_{[0,0,0]},$$

Since the distributions of $N_{tot}^{(t)}, N1^{(t)}, N2^{(t)}$ are independent of t, we simplify them by writing $N_{tot}, N1, N2$, respectively. Note also that $N1$ and $N2$ are independent.

3.1.1 Computation of the 24-bit noise distributions for $N1$ and $N2$

Computing the distribution of $N1$ is trivial, we simply run over all possible values of $\hat{R1}[0]$ and $s_{15}^{(t)}[0]$, which is of complexity $O(2^{16})$, where the notation $O(x)$ means that the number of simple operations is $c \cdot x$ for some small constant c.

Computation for $N2$ is more tricky and below we explain how we do that. We can

rewrite $N2$ as:

$$N2 = \begin{pmatrix} (S_R^{-1}(\underbrace{(L_1^{-1}\hat{R2})[0]}_{\text{Linear part } A}) \boxplus s_{15}^{(t-1)}[0]) \oplus s_{15}^{(t-1)}[0] \oplus S_Q^{-1}(\underbrace{(L_2^{-1}\hat{R3})[0]}_{\text{Linear part } B}) \\ \hat{R2}[0] \\ \underbrace{(L_1^{-1}((\hat{R2} \boxplus (\hat{R3} \oplus s_5^{(t)}) \boxplus s_{15}^{(t+1)}) \oplus s_5^{(t)} \oplus s_{15}^{(t+1)}))[0]}_{\text{Linear part } C} \end{pmatrix}.$$

The idea behind the computation technique is to compute 256 24-bit distributions of the triple bytes (A, B, C), conditioned on the value of the byte $\hat{R2}[0] \in [0, \ldots, 255]$. Let us denote those distributions as $D_{\hat{R2}[0]}(A, B, C)$. The distribution table of $N2$ is then constructed as follows: we initialize the distribution table $N2$ with all zeroes; then, for each combination of $\hat{R2}[0], s_{15}^{(t-1)}[0], A, B, C$ (in total 2^{40} choices) we do:

$$Pr\{N2 = \begin{pmatrix} (S_R^{-1}(A) \boxplus s_{15}^{(t-1)}[0]) \oplus s_{15}^{(t-1)}[0] \oplus S_Q^{-1}(B) \\ \hat{R2}[0] \\ C \end{pmatrix}\} += 2^{-16} \cdot D_{\hat{R2}[0]}(A, B, C),$$

where 2^{-16} is the normalization factor since for each (A, B, C) we also loop over $\hat{R2}[0]$ and $s_{15}^{(t-1)}[0]$ and, thus, there will be 256^2 distributions added to the table $N2$. Also note that we can actually compute distributions $D_{\hat{R2}[0]}(A, B, C)$ one by one for each value of $\hat{R2}[0]$ and add to the accumulating distribution of $N2$. This way, we do not need to store all of them in RAM simultaneously.

3.1.2 Computation of sub-noises $D_{\hat{R2}[0]}(A, B, C)$

What remains is to show how to compute $D_{\hat{R2}[0]}(A, B, C)$ for a given (fixed) byte value of $\hat{R2}[0]$. The expression for which we want to compute the distribution is as follows:

$$\begin{pmatrix} A \\ B \\ C \end{pmatrix} = \begin{pmatrix} (L_1^{-1}\hat{R2})[0] \\ (L_2^{-1}\hat{R3})[0] \\ (L_1^{-1}[(\hat{R2} \boxplus (\hat{R3} \oplus s_5^{(t)}) \boxplus s_{15}^{(t+1)}) \oplus s_5^{(t)} \oplus s_{15}^{(t+1)}])[0] \end{pmatrix}$$
$$= \begin{pmatrix} \bigoplus_{i=0}^{3} e_i \hat{R2}[i] \\ \bigoplus_{i=0}^{3} d_i \hat{R3}[i] \\ \bigoplus_{i=0}^{3} e_i[(\hat{R2}[i] \boxplus (\hat{R3}[i] \oplus s_5^{(t)}[i]) \boxplus s_{15}^{(t+1)}[i] \boxplus c_i) \oplus s_5^{(t)}[i] \oplus s_{15}^{(t+1)}[i]] \end{pmatrix},$$

where the coefficients are 8×8 Boolean matrices $e_i = L_1^{-1}[0, i]$ and $d_i = L_2^{-1}[0, i]$, for $i = 0, 1, 2, 3$. The new variables c_i are the input carry values that come from the arithmetical addition of the previous byte(s), i.e., bytes 0 to $i-1$, of the 3 terms: $\hat{R2}[i]$, $(\hat{R3}[i] \oplus s_5^{(t)}[i])$ and $s_{15}^{(t+1)}[i]$. Note that the range of these carry values is $0 \leq c_i \leq 2$ and the first one is $c_0 = 0$.

We will later explain how to deal with carry values c_i, but, at the moment, let us rewrite the above expression in a new form as follows. To simplify upcoming formulae,

let us define:

$$t_i = \hat{R2}[i] \boxplus (\hat{R3}[i] \oplus s_5^{(t)}[i]) \boxplus s_{15}^{(t+1)}[i],$$
$$u_i = s_5^{(t)}[i] \oplus s_{15}^{(t+1)}[i],$$

and then we have

$$\begin{pmatrix} A \\ B \\ C \end{pmatrix} = \begin{pmatrix} \bigoplus_{i=0}^{3} e_i \hat{R2}[i] \\ \bigoplus_{i=0}^{3} d_i \hat{R3}[i] \\ \bigoplus_{i=0}^{3} e_i[(t_i \boxplus c_i) \oplus u_i] \end{pmatrix} =$$

$$= \underbrace{\begin{pmatrix} e_0 \hat{R2}[0] \\ d_0 \hat{R3}[0] \\ e_0[(t_0 \boxplus c_0) \oplus u_0] \end{pmatrix}_{\to o_0}}_{E_{\hat{R2}[0],k=0,c_0=0,o_0=0..2}(A_0,B_0,C_0)} \oplus \underbrace{\begin{pmatrix} e_1 \hat{R2}[1] \\ d_1 \hat{R3}[1] \\ e_1[(t_1 \boxplus c_1) \oplus u_1] \end{pmatrix}_{\to o_1}}_{E_{\hat{R2}[0],k=1,c_1=0..2,o_1=0..2}(A_1,B_1,C_1)}$$
$$\oplus \underbrace{\begin{pmatrix} e_2 \hat{R2}[2] \\ d_2 \hat{R3}[2] \\ e_2[(t_2 \boxplus c_2) \oplus u_2] \end{pmatrix}_{\to o_2}}_{E_{\hat{R2}[0],k=2,c_2=0..2,o_2=0..2}(A_2,B_2,C_2)} \oplus \underbrace{\begin{pmatrix} e_3 \hat{R2}[3] \\ d_3 \hat{R3}[3] \\ e_3[(t_3 \boxplus c_3) \oplus u_3] \end{pmatrix}_{\to o_3}}_{E_{\hat{R2}[0],k=3,c_3=0..2,o_3=0..2}(A_3,B_3,C_3)},$$

where c_i are input carry values, and o_i are output carry values related to the sub-expressions $(t_i \boxplus c_i) \to (\text{8-bit resulting value}, \text{output carry } o_i)$.

Case with 4 parallel 8-bit adders $\boxplus_8$. In case we substitute 32-bit full adders $\boxplus$ with 4 parallel 8-bit adders $\boxplus_8$, all input and output carry values are all 0 and can be ignored. In this case, we have that the above distribution of (A, B, C) can be expressed as a XOR-convolution of 4 *independent* sub-distributions. This is due to each of the four sub-distributions are expressed using different byte variables, which are uniformly distributed and independent on each other.

Case with full 32-bit adders $\boxplus$. In this case the only dependency between the above 4 sub-distributions are the carry values that propagate from one sub-distribution to the next sub-distribution.

In order to compute the distribution of $D_{\hat{R2}[0]}(A, B, C)$ we will actually compute the number of combinations of the involved bytes for each resulting triple (A, B, C), then, in the end, the 24-bit vector of integer values will be normalized to actually get the distribution with probabilities. I.e., we will use a combinatorial approach in this section.

Let us define

$$\begin{pmatrix} A_k \\ B_k \\ C_k \end{pmatrix} = \begin{pmatrix} e_k \hat{R2}[k] \\ d_k \hat{R3}[k] \\ e_k[(t_k \boxplus c_k) \oplus u_k] \end{pmatrix}_{\to o_k}, \quad \text{for} \quad k = 0, 1, 2, 3,$$

and introduce intermediate E-vectors

$$E_{\hat{R2}[0],\ k=0,1,2,3,\ c_k=0,1,2,\ o_k=0,1,2}(A_k, B_k, C_k),$$

each of which is a 24-bit vector, i.e. of size 2^{24} for all choices of the three bytes (A_k, B_k, C_k), and each cell is an integer value (which can be large). In each index (A_k, B_k, C_k) the integer value corresponds to the number of combinations of the variables $\hat{R}2[k], \hat{R}3[k], s_5^{(t)}[k], s_{15}^{(t+1)}[k]$ involved in the 24-bit expression (A_k, B_k, C_k) for the k's sub-distribution, $k = 0, 1, 2, 3$, given the value of the input carry c_k, such that the resulting output carry in the sub-expression $t_k \boxplus c_k$ is o_k.

Then, the first 3 E-vectors for $k = 0$ and $o_0 = 0, 1, 2$,

$$E_{\hat{R}2[0],\ k=0,\ c_0=0,\ o_0=0,1,2}(A_0, B_0, C_0),$$

can be computed by trying all possible byte values of $\hat{R}3[0], s_5^{(t)}[0], s_{15}^{(t+1)}[0]$ in time $O(2^{24})$. Note that the value $\hat{R}2[0]$ is fixed and the input carry $c_0 = 0$. Thus, the first sub-distribution is only associated with 3 E-vectors, and we do not need to loop over the values of $\hat{R}2[0]$.

The next 3×3 E-vectors for $k = 1$ and $c_1, o_1 \in [0, 1, 2]$,

$$E_{\hat{R}2[0],\ k=1,\ c_1=0,1,2,\ o_1=0,1,2}(A_1, B_1, C_1),$$

are computed by trying all possible values of $c_1, \hat{R}2[1], \hat{R}3[1], s_5^{(t)}[1], s_{15}^{(t+1)}[1]$ in time $O(3 \cdot 2^{32})$. We continue this way to compute all E-vectors.

The next step is to use E-vectors in order to receive the combined 24-bit vector that contains the number of combinations resulting for each choice of the (A, B, C) triple.

Example. Let us first give a small example to demonstrate the technique. Assume we want to compute the 24-bit vector of combinations (A, B, C) when the carry value from the first sub-distribution to the second sub-distribution is $o_0 = 2$, the next carry value that propagates to the third sub-distribution is $o_1 = 0$, and the carry value that propagates to the last sub-distribution is $o_2 = 1$. Then, we should perform a XOR-convolution of E-vectors that are matching in their input/output carry values, i.e. $c_0 = 0, c_1 = o_0 = 2, c_2 = o_1 = 0, c_3 = o_2 = 1$. For a single choice $(A = a, B = b, C = c)$ we then compute:

$$\begin{aligned}
E_{\text{Example}}(A = a, B = b, C = c) = \sum_{o_3=0..2} \sum_{a_{0..2}, b_{0..2}, c_{0..2} \in [0..255]} & E_{\hat{R}2[0],k=0,c_0=0,o_0=2}(a_0, b_0, c_0) \\
\cdot & E_{\hat{R}2[0],k=1,c_1=2,o_1=0}(a_1, b_1, c_1) \cdot E_{\hat{R}2[0],k=2,c_2=0,o_2=1}(a_2, b_2, c_2) \\
\cdot & E_{\hat{R}2[0],k=3,c_3=1,o_3}(a \oplus a_0 \oplus a_1 \oplus a_2, b \oplus b_0 \oplus b_1 \oplus b_2, c \oplus c_0 \oplus c_1 \oplus c_2),
\end{aligned}$$

which implies that in order to compute the whole vector $E_{\text{Example}}(A, B, C)$ the complexity is $O(3 \cdot 2^{96})$, but the time for the above convolution can be reduced down through a series of XOR-convolutions: 4 forward and 1 inverse FWHTs, 3 point-wise vector multiplications, and 2 vector summations, i.e. $O((5 \cdot 24 + 3 + 2) \cdot 2^{24}) = O(125 \cdot 2^{24})$.

$$\begin{aligned}
E_{\text{Example}}(A, B, C) = & E_{\hat{R}2[0],k=0,c_0=0,o_0=2}(A_0, B_0, C_0) \\
& \times E_{\hat{R}2[0],k=1,c_1=2,o_1=0}(A_1, B_1, C_1) \\
& \times E_{\hat{R}2[0],k=2,c_2=0,o_2=1}(A_2, B_2, C_2) \\
& \times \sum_{o3=0}^{2} E_{\hat{R}2[0],k=3,c_3=1,o_3} \quad (A_3, B_3, C_3),
\end{aligned}$$

where $\times$ denotes a XOR-convolution over 24-bit E-vectors of integers, and $\sum$ is a point-wise arithmetical summation of the vectors. Since the last output carry o_3 is truncated in the 32-bit addition $\boxplus$, we then accumulate all 3 vectors corresponding to the value of o_3 into one vector all together, thus, the last output carry is ignored.

Final convolution. The idea is clear – we should try all possible variants of intermediate carry propagations, perform a series of XOR-convolutions of E-vectors that are matching in the input/output carries, and then accumulate the resulting vectors of combinations into one final vector, $E_{\hat{R}2[0]}(A, B, C)$, as follows:

$$\begin{aligned} E_{\hat{R}2[0]}(A,B,C) = \sum_{v_0=0}^{2}\sum_{v_1=0}^{2}\sum_{v_2=0}^{2}\sum_{v_3=0}^{2} & E_{\hat{R}2[0],\ k=0,\ c_0=0,\ o_0=v_0}(A_0,B_0,C_0) \\ & \times E_{\hat{R}2[0],\ k=1,\ c_1=v_0,\ o_1=v_1}(A_1,B_1,C_1) \\ & \times E_{\hat{R}2[0],\ k=2,\ c_2=v_1,\ o_2=v_2}(A_2,B_2,C_2) \\ & \times E_{\hat{R}2[0],\ k=3,\ c_3=v_2,\ o_3=v_3}(A_3,B_3,C_3). \end{aligned}$$

The resulting distribution is then

$$D_{\hat{R}2[0]}(A,B,C) = 2^{-120} \cdot E_{\hat{R}2[0]}(A,B,C),$$

where 2^{-120} is the normalization factor, i.e., it reflects the sum of all integer values in the final vector E, that can be verified by counting the expected total number of combinations: 24 bits of choices for $\hat{R}2$ ($\hat{R}2[0]$ is given), and 32 bits for each of $\hat{R}3, s_5^{(t)}, s_{15}^{(t+1)}$, thus we should have $2^{24+3\cdot 32} = 2^{120}$ combinations in total.

Optimizations and speed up. A single XOR-convolution can be done with Fast Walsh-Hadamard Transform (FWHT) having time complexity $O(N \log N)$ where, in our case, we have $N = 2^{24}$. Also note that FWHT is a linear transformation. So convolution in the time domain corresponds to point-wise multiplication in the frequency domain; and, summation in the time domain corresponds to summation in the frequency domain. Thus, there is no need to switch between the time and frequency domains for mixed summation and convolution operations, we can do most of the above in the frequency domain without switching.

We can speed up the computations even further as follows. Note that the E-vectors for $k = 1, 2, 3$ do not depend on the value of $\hat{R}2[0]$, thus, we can compute and combine through convolutions the upper 3 bytes only once, then use the resulting 3 vectors, corresponding to the input carry c_1, for further computations of $D_{\hat{R}2[0]}(A, B, C)$ for all values of $\hat{R}2[0]$.

Optimization ideas can be detailed by the following steps:

$$\forall v_2 \in [0..2] : T3_{v_2}(A,B,C) = \sum_{v_3=0}^{2} E_{\hat{R}2[0],\ k=3,\ c_3=v_2,\ o_3=v_3}(A_3,B_3,C_3)$$

$$\forall v_1 \in [0..2] : T2_{v_1}(A,B,C) = \sum_{v_2=0}^{2} T3_{v_2}(A,B,C) \times E_{\hat{R}2[0],\ k=2,\ c_2=v_1,\ o_2=v_2}(A_2,B_2,C_2)$$

$$\forall v_0 \in [0..2] : T1_{v_0}(A,B,C) = \sum_{v_1=0}^{2} T2_{v_1}(A,B,C) \times E_{\hat{R}2[0],\ k=1,\ c_1=v_0,\ o_1=v_1}(A_1,B_1,C_1)$$

$$E_{\hat{R}2[0]}(A,B,C) = \sum_{v_0=0}^{2} T1_{v_0}(A,B,C) \times E_{\hat{R}2[0],\ k=0,\ c_0=0,\ o_0=v_0}(A_0,B_0,C_0).$$

I.e., three vectors of $T1$ can be precomputed once and be used for all values of $\hat{R}2[0]$. Recall that in the original Section 3.1.2, without optimizations, the number of point-wise vector multiplications is $3^4 \cdot 3 = 243$ and the number of vector summations is $3^4 - 1 = 80$. With the proposed optimizations, the last step (having $T1$ vectors being precomputed) requires only 3 point-wise vector multiplications and 2 vector summations, as well as the amount of RAM needed is reduced a lot since all the above steps can be done as soon as intermediate E-vectors are ready. Also note, the 3 vectors of $T3$ can be ready "for free" if during the preparation of E-vectors for $k = 3$ we simply ignore the output carry o_3 by forcing it to be always 0.

Total time complexity for $N2$. Precomputation of 3 vectors of $T1$ has time complexity $O(3 \cdot 3 \cdot 2^{32}_{\text{to construct 21 } E \text{ vectors for } k=1,2,3} + 2^{24} \cdot (24 \cdot (3+9+9)_{\text{FWHTs on 21 } E\text{-vectors}} + 18_{\times} + 12_{+})) \approx O(2^{35.47})$. In order to compute one $D_{\hat{R}2[0]}(A,B,C)$ we therefore need to compute three E-vectors for $k = 0, c_0 = 0, o_0 = 0, 1, 2$, in complexity $O(2^{24})$, and perform 4 24-bit FWHTs: 3 for $E_{\hat{R}2[0],k=0,c_0=0,o_0=0,1,2}$ and 1 inverse FWHT in the end. We also need to make 3 point-wise vector multiplications and 2 vector summations. Thus, the additional time complexity per $\hat{R}2[0]$ value is therefore $O(2^{24}_{\text{to construct } E \text{ for } k=0} + 2^{24} \cdot (24 \cdot 4_{\text{FWHTs}} + 3_{\times} + 2_{+})) \approx O(2^{30.67})$. Accumulating the above, the total time complexity to compute the distribution of $N2$ can be estimated as $O(2^{35.47} + 2^8 \cdot 2^{30.67} + 2^{40}) \approx O(2^{40.53})$.

3.1.3 Computation results and bias values

We have implemented the above computation method, computed the distribution for $N1$ and $N2$, and then the 24-bit total noise distribution for $N_{tot} = N1 \oplus N2$ for the proposed approximation. Note that the approximation takes into account the full 32-bit adders $\boxplus$. We have received the following results regarding the corresponding biases: $\epsilon(N1) > 1$, $\epsilon(N2) \approx 2^{-29.391880}$, and

$$\epsilon(N_{tot}) \approx 2^{-37.37}, \quad \epsilon(2 \times N_{tot}) \approx 2^{-80.21}, \quad \epsilon(3 \times N_{tot}) \approx 2^{-121.66}, \quad \epsilon(4 \times N_{tot}) \approx 2^{-162.76}.$$

Here $\epsilon(i \times N_{tot})$ is the notion for the bias of the resulting distribution when summing i independent random variables distributed as N_{tot} using bitwise XOR.

3.2 An 8-bit approximation

As will be clear later, the 24-bit approximation cannot be used in a straight-forward manner in a fast correlation attack. For such a case, one would need an approximation that can be completely described over a finite field.

From the 24-bit noise distribution that we computed in the previous subsection, we can further derive an 8-bit approximation with operations in the Rijndael field $GF(2^8)$, with the noise now denoted N'_{tot}. The approximation has the following form,

$$\begin{aligned}\Lambda z^{(t-1)}[0] \oplus z^{(t)}[0] \oplus \Gamma(L_1^{-1} z^{(t+1)})[0] &= N'_{tot} \\ \oplus (\Lambda(s_0^{(t-1)} \oplus s_{15}^{(t-1)}) \oplus s_0^{(t)} \oplus s_{15}^{(t)} \oplus \Gamma L_1^{-1}[s_0^{(t+1)} \oplus s_5^{(t)} \oplus s_{15}^{(t+1)}])[0], \quad (4)\end{aligned}$$

where Γ, Λ are some nonzero constants in $GF(2^8)$. For each possible choice of $\Gamma, \Lambda \in GF(2^8)$ we compute the 8-bit distribution of N'_{tot} directly from the given 24-bit distribution of N_{tot} and then we compute the corresponding bias,

$$N'_{tot} = \Lambda N_{tot}[0] \oplus N_{tot}[1] \oplus \Gamma N_{tot}[2].$$

Searching through all choices of Γ and Λ would normally imply the computational complexity $O(2^{48})$, but we can reduce it down to $O(\sim 2^{40})$ by the following technique. In the loop for Γ, we first precompute the joint 16-bit distribution of $(N_{tot}[0], N_{tot}[1] \oplus \Gamma N_{tot}[2])$ with complexity $O(2^{24})$, then we loop for Λ and use the precomputed joint distribution to derive the 8-bit distribution of N'_{tot}. The best choice for constants appears to be $\Gamma =$ `0x9c` and $\Lambda =$ `0x08` (alternatively, $\Lambda =$ `0x18` also gives the best approximation) and the resulting bias of N'_{tot} is:

$$\begin{aligned}&\epsilon(N'_{tot}) \approx 2^{-40.97}, \quad \epsilon(2 \times N'_{tot}) \approx 2^{-81.94}, \\ &\epsilon(3 \times N'_{tot}) \approx 2^{-122.91}, \quad \epsilon(4 \times N'_{tot}) \approx 2^{-163.88}.\end{aligned}$$

4 Experimental verification

In this section, we experimentally verify the correctness of the bias derived in the previous section. The experimental verification can be done by running the cipher and collecting a large number of samples. For distributions of smaller sizes one can fully verify them by experimentally determining the exact distribution, i.e., every probability value in the distribution is correct; while for larger distributions, this might not be computationally possible. Instead, we can use them in a hypothesis testing and in this way demonstrate that it can be used in a distinguisher. This will be the case for the 24-bit approximation from Section 3.

We consider deciding the sample distribution between the uniform distribution and the noise distribution derived in Section 3 by hypothesis testing. We will follow the hypothesis testing approach as formulated in information theory, see [CT12]. It is centered around the divergence (or relative entropy, or Kullback Leibler distance), denoted $D(P_X||P_Y)$, between two distributions P_X and P_Y over the same alphabet and defined as $D(P_X||P_Y) = \sum_i P(X=i) \log \frac{P(X=i)}{P(Y=i)}$. The relative entropy is used to measure the distance between two distributions: the closer the distributions are, the smaller $D(P_X||P_Y)$ would be. If the distributions are the same, $D(P_X||P_Y) = 0$.

Furthermore, if we have a sequence of n sample symbols $\mathbf{x} = x_0, x_1, \ldots, x_{n-1}$ from the same alphabet $\mathcal{A}$ then we can count the number of occurrences of each symbol $a \in \mathcal{A}$, denoted $N(a|\mathbf{x})$ and forming the *type* (or empirical distribution; or sample distribution) by assigning $P(X = a) = N(a|\mathbf{x})/n$.

Let us denote the uniform distribution as P_U, the noise distribution derived in Section 3 as P_N. Assume we collect n samples $\mathbf{x}$ from an unknown distribution P_X. Then the hypothesis testing can be modeled as below with two hypotheses:

$$\begin{cases} H_0 : P_X = P_N, \\ H_1 : P_X = P_U. \end{cases} \tag{5}$$

In our case we are considering 24-bit distributions, so $|\mathcal{A}| = 2^{24}$ and the sample distribution is denoted P_{X^n}, with n as the length of the sample symbols.

We use the Neyman-Pearson lemma to make the optimum decision for the hypothesis testing, according to the distances from P_{X^n} to P_U and P_N, respectively. The decision problem is a maximum-likelihood test and the log-likelihood ratio can be written as

$$L = nD(P_{X^n}||P_U) - nD(P_{X^n}||P_N).$$

Then we define the decision rule as below

$$P_X = \begin{cases} P_N, & \text{if } D(P_{X^n}||P_U) > D(P_{X^n}||P_N), \\ P_U, & \text{if } D(P_{X^n}||P_U) < D(P_{X^n}||P_N). \end{cases} \tag{6}$$

With the hypothesis-testing problem defined above and P_N being the 24-bit noise distribution from the previous section, we build a distinguisher to decide the underlying sample distribution. We run 64 parallel SNOW 3G instances with random initial states, each clocking 2^{40} times and collect the targeted samples. At each clock t, we combine $(z^{(t-1)} \oplus s_0^{(t-1)} \oplus s_{15}^{(t-1)})[0], (z^{(t)} \oplus s_{15}^{(t)} \oplus s_0^{(t)})[0], (L_1^{-1}[z^{(t+1)} \oplus s_0^{(t+1)} \oplus s_5^{(t)} \oplus s_{15}^{(t+1)}])[0]$, which is exactly the xor-sum of the keystream and LFSR part in (3), to make a 24-bit integer and increase the occurrence of the corresponding entry in the distribution table. We also collect the least three significant bytes of $z^{(t)}$ and concatenate them into a 24-bit variable, which is regarded as a comparison sample drawn from a uniform distribution.

After this process, we get the tables of occurrences of all possible 24-bit values and their corresponding probabilities. Then these sample distributions are tested by the decision rule given in (6) to get the answer to which distribution they follow, by calculating the distances to the uniform distribution P_U and noise distribution P_N, respectively. There are two types of errors to the correctness of the distinguisher: TYPE I errors, the errors of guessing a noise distribution as random; and TYPE II errors, the errors of falsely guessing a uniform distribution as the biased one.

Figure 2 shows the distances of one sample sequence to a uniform distribution and the noise distribution and their differences under different lengths of samples. We can see from the first subfigure that with an increase in the length of samples, the distances to the uniform and noise distribution are both decreasing. This means that the sample distribution is approaching both the random distribution and the noise distribution, but it is hard to tell to which one the sample distribution is closest to, just from the first subfigure. Instead, we can get the answer from the second subfigure, showing the

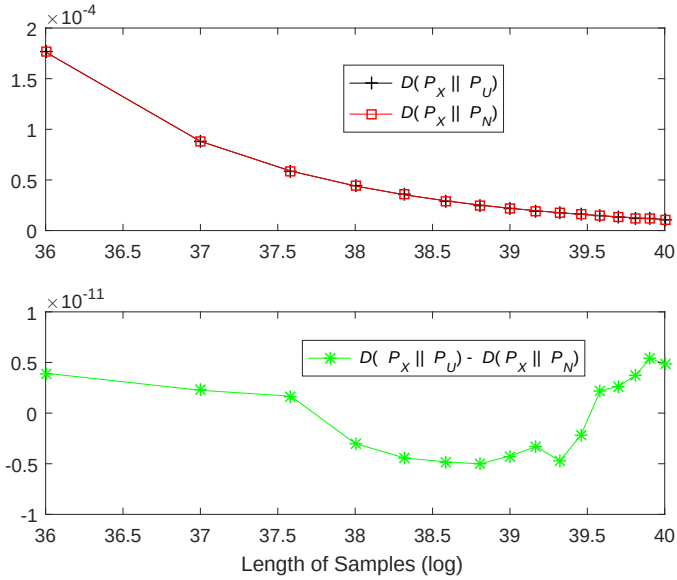

Figure 2: Distances to the uniform distribution and noise distribution

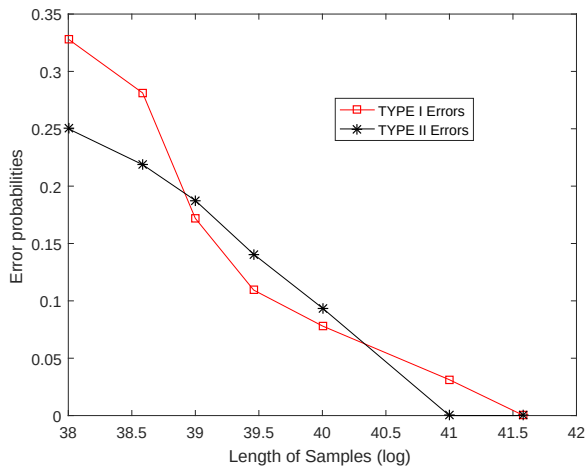

Figure 3: Error probabilities under different lengths of samples

difference of the divergence of the two distances, i.e., $D(P_X||P_U) - D(P_X||P_N)$. We can see that while fluctuating around 0 in the beginning, the difference becomes stable and positive after length $2^{39.58}$, indicating $D(P_X||P_U) > D(P_X||P_N)$ and that the sample distribution is closer to the noise distribution. The difference at length 2^{40} is $0.5 * 10^{-11}$, i.e., around $2^{-37.54}$ and we can expect that it will converge to around $2^{-37.37}$. This is

so because with the increase in keystream samples, $D(P_X||P_N)$ will converge to 0 and $D(P_X||P_U)$ would get close to the bias we derived [BJV04].

Figure 3 then shows the TYPE I and II error probabilities for the distinguisher under different lengths of samples. We run 64 parallel SNOW 3G instances with random initial states, each clocking 2^{40} times, and record the distribution table of the samples for each instance. Whenever another 2^{38} samples are collected, we make a hypothesis testing of the obtained distribution and record the type I and type II errors. After the collection has finished, we make larger samples by combining distribution from some different instances, e.g., samples of length 2^{41} from two 2^{40} instances, and further record the errors under the hypothesis testing. From the result in Figure 3, we could see that while fluctuating, the error probabilities are becoming smaller with an increase in the amount of samples, which indicates that the guesses are becoming more accurate. From length 2^{40}, we can distinguish the samples with large success probabilities, while at the length $2^{41.5}$, there are no errors in our 21 sample sequences. The result matches well with the bias obtained in Section 3 and the conclusion that $O(1/\epsilon)$ samples are needed to distinguish the distribution from random when the bias is ϵ.

5 Attacks based on the new vectorized linear approximations

We are now ready to use our vectorized linear approximations of the FSM to launch attacks. We recall that the approximation on three bytes we derived in Section 3 is of the form below,

$$\begin{aligned}\left(z^{(t-1)}[0], z^{(t)}[0], (L_1^{-1}z^{(t+1)})[0]\right) &= (n_0, n_1, n_2) \\ \oplus \left((s_{15}^{(t-1)} \oplus s_0^{(t-1)})[0], (s_{15}^{(t)} \oplus s_0^{(t)})[0], (L_1^{-1}s_5^{(t)} \oplus L_1^{-1}s_{15}^{(t+1)} \oplus L_1^{-1}s_0^{(t+1)})[0]\right), \quad (7)\end{aligned}$$

where (n_0, n_1, n_2) denotes the noise in the 24-bit linear approximation. We have computed the bias of this noise to be about 2^{-37} in Section 3 and experimentally verified it in Section 4. We now consider how to launch a distinguishing attack with this 24-bit approximation in the next subsection. Fast correlation attacks will be considered in Section 5.2.

5.1 A distinguishing attack

In a distinguishing attack we build an algorithm that takes a sequence as input and determines with a small error probability whether the sequence stems from the considered keystream generator, or if it is a truly random sequence. A potential application would be a case when only two messages $\mathbf{m}, \mathbf{m}'$ are possible, and from the ciphertext $\mathbf{c}$ only, we would like to determine which message was sent. One then forms a candidate keystream by computing $\mathbf{c} \oplus \mathbf{m}$ and inputs this to the distinguisher. If the distinguisher finds that this candidate keystream is likely to have been generated from the keystream generator, it is likely that the sent message was $\mathbf{m}$.

The basic idea for finding a distinguishing attack in our scenario is to completely remove the contribution from the LFSR part, leaving only a linear function of known

output symbols as a sample from a noisy distribution. After collecting enough samples, one can distinguish the considered keystream from a truly random sequence.

Considering the relationship in (7), we would like to cancel the LFSR contribution

$$S^{(t)} = \left((s_{15}^{(t-1)} \oplus s_0^{(t-1)})[0], (s_{15}^{(t)} \oplus s_0^{(t)})[0], (L_1^{-1} s_5^{(t)} \oplus L_1^{-1} s_{15}^{(t+1)} \oplus L_1^{-1} s_0^{(t+1)})[0] \right).$$

Since $s_i^{(j)} = s_0^{(i+j)}$ and for simplicity, we write $s_0^{(t)}$ simply as $s^{(t)}$. It is easy to verify the following theorem.

Theorem 1. If one can find t_1, t_2, t_3 such that $s^{(0)} \oplus s^{(t_1)} \oplus s^{(t_2)} \oplus s^{(t_3)} = 0$ then

$$S^{(t)} \oplus S^{(t+t_1)} \oplus S^{(t+t_2)} \oplus S^{(t+t_3)} = (0, 0, 0).$$

Proof. Since $s^{(0)} \oplus s^{(t_1)} \oplus s^{(t_2)} \oplus s^{(t_3)} = 0$, with any time shift the equation would still hold, i.e., $s^{(t)} \oplus s^{(t+t_1)} \oplus s^{(t+t_2)} \oplus s^{(t+t_3)} = 0$. For the other terms in $S^{(t)}$, the xor-sum from the values at $0, t_1, t_2, t_3$ would also be 0. Let us take the term $s_{15}^{(t-1)}$ for example: since $s_{15}^{(t-1)} = s_0^{(t+14)}$, then $\bigoplus_{i=0}^{3} s_{15}^{(t+t_i-1)} = \bigoplus_{i=0}^{3} s_0^{(t+t_i+14)} = 0$. Then we have $\bigoplus_{i=0}^{3} S^{(t+t_i)} = (0, 0, 0)$. □

Assuming that we have found t_1, t_2, t_3 satisfying Theorem 1, we can create samples from a biased distribution by computing samples $x^{(t)}$ as

$$x^{(t)} = \sum_{i=0}^{3} \left(z^{(t+t_i-1)}[0], z^{(t+t_i)}[0], (L_1^{-1} z^{(t+t_i+1)})[0] \right),$$

where $t_0 = 0$. The samples $x^{(t)}$ are then drawn from a noisy distribution, which is the distribution of the sum of 4 noise variables like N_{tot}. This was previously computed to have a bias of $\epsilon(4 \times N_{tot}) = 2^{-163}$ and hence it requires in the order of 2^{163} keystream symbols to distinguish the samples from a uniform distribution. It should be noted here that we assume that the noise variables at the four time instances are independent, thus resulting in the total bias as $\epsilon(4 \times N_{tot}) = 2^{-163}$. The bias is actually larger since there is a dependence between the LFSR states from the four time instances, due to their sum being zero. But we have a too high complexity in computing the bias for such a case, so we regard the noise variables as independent.

The remaining problem here is to examine the computational complexity of finding t_1, t_2, t_3 satisfying $s^{(0)} \oplus s^{(t_1)} \oplus s^{(t_2)} \oplus s^{(t_3)} = 0$. The sequence $s^{(t)}$ is generated using the feedback polynomial $P(x) = \alpha x^{16} + x^{14} + \alpha^{-1} x^5 + 1 \in GF(2^{32})[x]$. We are thus seeking a weight 4 multiple $K(x)$ of the feedback polynomial where all coefficients are set to one. We may first argue about the expected degree q of such a polynomial. Let us consider all $t \leq q$, then we can create $\binom{q}{3}$ different combinations of $s^{(0)} \oplus s^{(t_1)} \oplus s^{(t_2)} \oplus s^{(t_3)}$ expressed in the initial state. Since there are 2^{512} possible such combinations, we can expect that we need to go to a degree such that roughly $q^3/6 \approx 2^{512}$, resulting in $q \approx 2^{172}$.

Finally, we need an efficient way to find a weight 4 multiple. Here we use a slight generalization of the algorithm proposed by Löndahl and Johansson in [LJ14]. The algorithm solves the problem with computational complexity of only around 2^d, where

$d = \log q$, and similar storage. The algorithm uses the idea of duplicating the desired multiple to many instances and then finding one of them with very large probability. The solution to problem associated to the SNOW 3G case can be described as follows:

Assume that $K(x)$ is the weight 4 multiple of the lowest degree and assume that its degree is around 2^d as expected. Algorithm 1 considers and creates all weight 4 multiples up to degree 2^{d+b} where b is a small integer, but will only find those that include two monomials x^{i_1} and x^{i_2} such that $\phi(x^{i_1} + x^{i_2} \bmod P(x)) = 0$, where $\phi()$ means the d least significant bits in the representation of the polynomial.

Algorithm 1 Finding a multiple of $P(x)$ with weight 4 and all nonzero coefficients one

Input Polynomial $P(x)$, a small integer b

Output A polynomial multiple $K(x) = P(x)Q(x)$ of weight 4 and expected degree 2^d with nonzero coefficients set to be one

1. From $P(x)$, create all residues $x^{i_1} \bmod P(x)$, for $0 \leq i_1 < 2^{d+b}$ and put $(x^{i_1} \bmod P(x), i_1)$ in a list $\mathcal{L}_1$. Sort $\mathcal{L}_1$ according to the residue value of each entry.

2. Create all residues $x^{i_1} + x^{i_2} \bmod P(x)$ such that $\phi(x^{i_1} + x^{i_2} \bmod P(x)) = 0$, for $0 \leq i_1 < i_2 < 2^{d+b}$ and put in a list $\mathcal{L}_2$. Here $\phi()$ means the d least significant bits. This is done by merging the sorted list $\mathcal{L}_1$ by itself and keeping only residues $\phi(x^{i_1} + x^{i_2} \bmod P(x)) = 0$. The list $\mathcal{L}_2$ is sorted according to the residue value.

3. In the final step we merge the sorted list $\mathcal{L}_2$ with itself to create a list $\mathcal{L}$, keeping only residues $x^{i_1} + x^{i_2} + x^{i_3} + x^{i_4} = 0 \bmod P(x)$.

As $K(x)$ is of weight 4, any polynomial $x^{i_1}K(x)$ is also of weight 4 and since we consider all weight 4 multiples up to degree 2^{d+b} we will consider $2^{d+b} - 2^d$ such weight 4 polynomials, i.e. about $2^d(2^b - 1)$ duplicates of $K(x)$. As the probability for a single weight 4 polynomial to have the condition $\phi(x^{i_1} + x^{i_2} \bmod P(x)) = 0$ can be approximated to be around 2^{-d}, there will be a large probability that at least one duplicate of $x^{i_1}K(x)$ will survive in Step 2 in Algorithm 1 and will be included in the output. Further details and experimental verification can be found in [LJ14].

Regarding complexity, we note that the tables are all of size around 2^d. Creation of $\mathcal{L}_1$ costs roughly 2^d and creation of $\mathcal{L}_2$ costs about the same as we are only accessing entries in $\mathcal{L}_1$ with the same d least significant bits. For a sufficiently large b, one iteration of Algorithm 1 (inner loop) succeds with a high probability.

To conclude, we have described a distinguishing attack on SNOW 3G for which we need a keystream length of around 2^{172} and similar complexity. It uses a precomputation step of complexity around 2^{172} and required memory is of the same size.

5.2 A fast correlation attack

A fast correlation attack is a key recovery attack, which is a much stronger attack than a distinguishing attack. It tries to recover the key by exploring the correlation between the keystream and the output of the LFSR states, which always exists for nonlinear functions[Sie84]. It is commonly modeled as a decoding problem in $GF(2)^n$ or $GF(2^n)$, with the observed keystream samples $\mathbf{y} = (y_0, y_1, ..., y_{N-1})$ being the noisy version of the LFSR sequence $\mathbf{u} = (u_0, u_1, ..., u_{N-1})$ through a discrete memoryless channel (DMC) with non-uniform noise $\mathbf{e} = (e_0, e_1, ..., e_{N-1})$, i.e., $y_i = u_i + e_i$ for

$1 \leq i \leq N-1$. It should be noted here that $\mathbf{u}$ and $\mathbf{z}$ might not be the exact output from the LFSR and the keystream of the considered keystream generator, but can be some linear combinations of them. The LFSR sequence $\mathbf{u} = (u_0, u_1, ..., u_{N-1})$ is regarded as a codeword generated from the information words $(u_0, u_1, ..., u_{l-1})$, which is the initial state of LFSR by $\mathbf{u} = (u_0, u_1, ..., u_{l-1})G$, where $G = (\mathbf{g}_0, \mathbf{g}_1, ..., \mathbf{g}_{N-1})$ with each element $\mathbf{g}_i$ being a column vector of length l. Then the correlation attack is converted into decoding an $[N, l]$ linear code with N being the code length and l the length of the information word. The main problem is to find methods of efficient decoding as a straight-forward approach of running through all codewords to find the closest one to the received vector is requiring a huge computational complexity. A common approach is to find parity checks that can be combined to form parity checks in a lower-dimensional code, which is then more efficiently decoded.

The correlation attacks always have two stages: a preprocessing stage, during which as many parity check equations of the $[N, l]$ code as possible are generated and processed; and an online decoding stage, during which the decoding is executed according to these generated parity check equations. Research on correlation attacks have been mainly focused on these two aspects: finding parity check equations with low weights [Pen96], [CT00], typically 3, 4 or 5, and exploring more effecient decoding techniques to solve the decoding process [JJ99b], [JJ99a].

The framework of a fast correlation attack is defined over a finite field. It is usually the binary case but a larger alphabet is possible as long as the operations are in a finite field. Our strongest (24-bit) approximation does not fulfill this condition and it cannot be used in a straight-forward manner. Instead, we rely on the 8-bit approximation from (4), which is an approximation over $GF(2^8)$. Below we first give a correlation attack using the method in [ZXM15] with the byte-based linear approximation in (4). Specifically, the attack employs the k-tree algorithm in [Wag02] to generate parity check equations during the preprocessing stage and the Fast Walsh Transform (FWT) technique to accelerate the decoding process during the decoding stage.

5.2.1 A fast correlation attack based on an approximation in $GF(2^8)$

We consider the two sequences,

$$y^{(t)} = \Lambda z^{(t-1)}[0] \oplus z^{(t)}[0] \oplus \Gamma(L_1^{-1} z^{(t+1)})[0]$$

and

$$u^{(t)} = (\Lambda(s_0^{(t-1)} \oplus s_{15}^{(t-1)}) \oplus s_0^{(t)} \oplus s_{15}^{(t)} \oplus \Gamma L_1^{-1}[s_0^{(t+1)} \oplus s_5^{(t)} \oplus s_{15}^{(t+1)}])[0],$$

both defined in $GF(2^8)$ and we know from before that $y^{(t)} = u^{(t)} + e^{(t)}$, where $e^{(t)}$ has the same distribution as N'_{tot}. Note that $u^{(t)}$ is a sequence defined over $GF(2^8)$, while the $s^{(t)}$ sequence is defined over an extension field of $GF(2^8)$. It can be verified that an LFSR with state $(s_0, s_1, ..., s_{15}) \in GF(2^{32})^{16}$ can also be described through a length 64 LFSR over $GF(2^8)$ with state $(u_0, u_1, ...u_{62}, u_{63}) \in GF(2^8)^{64}$, i.e., $u^{(t)}$ can indeed be described as the sequence from a length 64 LFSR over $GF(2^8)$. Below we give the 8-bit correlation attack based on such an LFSR.

Preprocessing Stage: Generating the Parity Check Equations

Since decoding an $[N, l]$ linear code involves a large complexity when l is large, the method in [CJS00] can be used to convert the $[N, l]$ code $\mathcal{C}_1$ into a simpler one $[N', l']$ denoted $\mathcal{C}_2$ with $l' < l$. The key point is to find a k-tuple of column vectors $(\mathbf{g}_{i_1}, \mathbf{g}_{i_2}, ..., \mathbf{g}_{i_k})$ from G satisfying $\mathbf{g}_{i_1} \oplus \mathbf{g}_{i_2} \oplus ... \oplus \mathbf{g}_{i_k} = (c_0, c_1, ..., c_{l'-1}, 0, ..., 0)^T$, i.e., the xor-sums of the last $l - l'$ elements are all zero. Then for such a tuple, the following equation holds, which is a parity check for $u_1, ..., u_{l'}$,

$$\bigoplus_{j=1}^{k} u_{i_j} = (u_0, u_1, ...u_{l-1}) \bigoplus_{j=1}^{k} \mathbf{g}_{i_j} = c_0 u_0 \oplus c_1 u_1 \oplus ... \oplus c_{l'-1} u_{l'-1}.$$

Correspondingly,

$$\bigoplus_{j=1}^{k} y_{i_j} = \bigoplus_{j=1}^{k} (u_{i_j} \oplus e_{i_j}) = c_0 u_0 \oplus c_1 u_1 \oplus ... \oplus c_{l'-1} u_{l'-1} \oplus \bigoplus_{j=1}^{k} e_{i_j}.$$

If we denote $Y_i = \bigoplus_{j=1}^{k} y_{i_j}$, $U_i = \bigoplus_{j=1}^{k} u_{i_j}$ and $E_i = \bigoplus_{j=1}^{k} e_{i_j}$, we have $Y_i = U_i \oplus E_i$. If we collect N' such parity checks, we can construct a new $[N', l']$ code, with $\mathbf{U} = (U_0, U_1, ..., U_{N'-1})$ being the output of a converted LFSR with l' states and $\mathbf{Y} = (Y_0, Y_1, ..., Y_{N'-1})$ being the noisy version of $\mathbf{U}$ through a more noisy channel with noise $\mathbf{E} = (E_0, E_1, ..., E_{N'-1})$. Since $l' < l$, the decoding complexity is reduced. Then the remaining work is to solve the decoding problem efficiently, which we will describe in detail in the processing stage.

As for the complexity for the preprocessing stage, with the k-tree algorithm in [Wag02] employed to find such parity check equations, the time/space complexities are $O(k2^{n(l-l')/(1+\log k)})$ and the sizes of lists are $O(2^{n(l-l')/(1+\log k)})$, with n being the size of the finite field, $n = 8$ in our case. Note that $\rho^{1+\log k}$ such tuples could be found with ρ times as much work as finding a single solution, i.e., $O(\rho k 2^{n(l-l')/(1+\log k)})$ for time and space and $O(\rho 2^{n(l-l')/(1+\log k)})$ for the size of each list, as long as $\rho \leq 2^{n(l-l')/(\log k(1+\log k)}$.

Processing Stage: Decoding the code

We now move to the process of decoding the $[N', l']$ code, following the method in [ZXM15] and using the FWT to accelerate the decoding process. The main idea is to make a distinguisher defined as $I(\hat{\mathbf{u}}) = c_{i_0}(u'_0 \oplus u_0) \oplus ... \oplus c_{i_{l'-1}}(u'_{l'-1} \oplus u_{l'-1}) \oplus E_i = Y_i \oplus c_{i_0} u'_0 \oplus ... \oplus c_{i_{l'-1}} u'_{l'-1}$ for a guess $\hat{\mathbf{u}} = (u'_0, u'_1, ..., u'_{l'-1})$ of the first l' LFSR states, where i denotes the i-th tuple. $I(\hat{\mathbf{u}})$ would be biased for the correct guess since only the noise term E_i remains.

The next step is to check the balancedness of $I(\hat{\mathbf{u}})$ for every guessed $\hat{\mathbf{u}}$ to find the correct key. Firstly, the correlations $c(\langle a, I \rangle)$ of the Boolean function $\langle a, I \rangle$, i.e., the inner product of a and I where $a \in GF(2)^n$, is obtained and then the SEI of $I(\hat{\mathbf{u}})$ can be derived by $\Delta(\hat{\mathbf{u}}) = \sum_{a \in GF(2^m)} c^2(\langle a, I \rangle)$ according to [NH07]. Then we can verify whether $I(\hat{\mathbf{u}})$ is biased or not and further recover the key. To get the correlations $c(\langle a, I \rangle)$ efficiently, the method in [LV04] could be used. Firstly, the vectorial Boolean

function I can be divided into n linearly independent Boolean functions $I_1, ..., I_n$, each expressed as $I_j = \langle w_j, \hat{\mathbf{u}} \rangle \oplus \langle v_j, Y_i \rangle$ where $w_j \in GF(2)^{nl'}, v_j \in GF(2)^n$ are two binary coefficient vectors. Then FWT can be used to compute the correlation of each I_i. It is stated in [ZXM15] that the total correlation can be further derived by the Piling-up Lemma. We refer to [ZXM15] for a more detailed description of this process and we use the complexity formulas from it.

For SNOW 3G, we use the 8-bit linear approximation which has a bias of $\epsilon(N'_{tot}) \approx 2^{-40.970689}$. We can first rewrite the LFSR sequence symbols as linear functions of 64 initial state bytes, with a new and more complex generating polynomial. Then use the preprocessing stage described before to generate the parity check equations with parameters $l = 64$, $k = 4$. The SEI of $k = 4$ folded noise variables is $\epsilon(4 \times N'_{tot}) \approx 2^{-163.88}$. We then tested different choices for l' and found that under $l' = 20$ the total complexity is the lowest. The number of parity check equations m_k required in this case is $m_k = 2^{171.67}$. The time/space complexity of preprocessing is $\rho k 2^{n(l-l')/(1+\log k)} = 2^{176.56}$ and the required length of the keystream is $2^{176.56}$. With complexity of $n(m_k + l' n 2^{l'n}) + 2^{n+l'n} = 2^{174.75}$, $20 \cdot 8 = 160$ bits of the LFSR initial states could be recovered. Therefore, the time/memory/data/pre-computation complexities are all upper bounded by $2^{176.56}$.

5.2.2 Potential Correlation Attack using a 16-bit approximation

In Section 3, we got the 24-bit and 8-bit linear approximations with biases 2^{-37} and 2^{-41}, respectively. We would obviously like to use the 24-bit approximation to launch a correlation attack. But as explained before, the 24-bit approximation is not defined over a finite field and cannot be used directly in a fast correlation attack. Now we report some findings on building a 16-bit approximation by an experimental method based on the 8-bit approximation derived before. Specifically, we concatenate two consecutive 8-bit symbols to build one 16-bit symbol, i.e., $(y^{(t)}, y^{(t+1)})$. The theoretical distribution for such a pair of bytes is too computationally consuming to compute, but since we know the bias for a single byte we can get an bound on the bias.

We run a large amount of SNOW 3G instances in parallel with random initial states and collect $(y^{(t)}, y^{(t+1)})$ by (4) at each clock, obtaining 2^{53} 16-bit samples in total. We then record the occurrence of each entry in the distribution table, and got the bias $2^{-36.8293}$. However, the bias here is not very accurate and more samples are needed for full confidence. Even so, we could still get a general estimation of the bias. After we collected these samples, we used the method in Section 4 to distinguish the 16-bit and 8-bit samples between the uniform distribution and the respective 16-bit and 8-bit distributions we derived.

Table 1 shows the TYPE I errors and probabilities for the two cases under different lengths of samples. We can see directly from the table that the error probability for 16-bit distinguishing is much smaller than the 8-bit one, indicating the bias of the 16-bit approximation is larger than the latter. Considering the error probabilities to be 0 for 16-bit and 8-bit distinguishing are after lengths 2^{42} and 2^{46}, respectively, we could make a general estimation that the bias for the 16-bit approximation is around 2^{-38}.

Now we briefly explain how to use this 16-bit approximation in a correlation attack. First we point out that any output from the LFSR at clock t, $u^{(t)}$, can be derived from the initial states $(u_0, u_1, ..., u_{63})$ by $u^{(t)} = \bigoplus_{i=0}^{63} c_i^{(t)} u_i$, where $c_i^{(t)} \in GF(2^8)$. Then we

Length	2^{40}	2^{41}	2^{42}	2^{43}	2^{44}	2^{45}	2^{46}
Samples	8192	4096	2048	1024	512	256	128
16-bit	626(0.076)	94(0.023)	3(0.001)	0	0	0	0
8-bit	2895(0.353)	1260(0.308)	445(0.217)	148(0.144)	40(0.078)	7(0.027)	0

Table 1: Errors and error probabilities (in the brackets) under different lengths of samples

have,

$$y^{(t)} = u^{(t)} + e^{(t)} = \bigoplus_{i=0}^{63} c_i^{(t)} u_i \oplus e^{(t)}. \tag{8}$$

At the next clock $t+1$, the value in the $(i+1)$-th cell is shifted to the i-th cell for $0 \le i \le 62$ and only the 63-rd cell is updated, which can be expressed as

$$u^{(t+1)} = \bigoplus_{i=0}^{62} c_i^{(t)} u_{i+1} \oplus c_{63}^{(t)} u'_{63},$$

where u'_{63} is the new value for the 63-rd cell updated by $u'_{63} = \bigoplus_{i=0}^{62} \gamma_i u_i$, where γ_i's are the feedback coefficients of the LFSR in $GF(2^8)$. Then $y^{(t+1)}$ can be expressed as,

$$y^{(t+1)} = u^{(t+1)} + e^{(t+1)} = \bigoplus_{i=0}^{62} c_i^{(t)} u_{i+1} \oplus c_{63}^{(t)} \bigoplus_{i=0}^{62} \gamma_i u_i \oplus e^{(t+1)}. \tag{9}$$

Assume by the k-tree algorithm, we have found a k-tuple combination, say $k = 4$, with (t_1, t_2, t_3, t_4) which maps the xor-sum of the output to the first l' 8-bit symbols in the LFSR, i.e.,

$$\bigoplus_{j=1}^{4} y^{(t_j)} = \bigoplus_{j=1}^{4} (\bigoplus_{i=0}^{63} c_i^{(t_j)} u_i \oplus e^{(t_j)}) = \bigoplus_{i=0}^{l'-1} c_i u_i \oplus E,$$

i.e., $c_i^{(t1)} \oplus c_i^{(t2)} \oplus c_i^{(t3)} \oplus c_i^{(t4)} = c_i$ for $0 \le i \le l'-1$, while $c_i^{(t1)} \oplus c_i^{(t2)} \oplus c_i^{(t3)} \oplus c_i^{(t4)} = 0$ for $l' \le i \le 63$, where $E = \bigoplus_{j=1}^{4} e^{(t_j)}$. We aim to build another part of the 16-bit symbol by getting $\bigoplus_{i=1}^{4} y^{(t_i+1)}$ from (9). We get

$$\bigoplus_{j=1}^{4} y^{(t_j+1)} = \bigoplus_{j=1}^{4} (u^{(t_j+1)} \oplus e^{(t_j+1)}) = \bigoplus_{j=1}^{4} (\bigoplus_{i=0}^{62} c_i^{(t_i)} u_{i+1} \oplus c_{63}^{(t_i)} \bigoplus_{i=0}^{62} \gamma_i u_i \oplus e^{(t_i+1)}).$$

Since $c_i^{(t1)} \oplus c_i^{(t2)} \oplus c_i^{(t3)} \oplus c_i^{(t4)} = 0$ and $c_{63}^{(t1)} \gamma_i \oplus c_{63}^{(t2)} \gamma_i \oplus c_{63}^{(t3)} \gamma_i \oplus c_{63}^{(t4)} \gamma_i = 0$ for $l' \le i \le 63$, we could get

$$\bigoplus_{j=1}^{4} y^{(t_j+1)} = \bigoplus_{j=1}^{4} (\bigoplus_{i=0}^{l'-1} c_i^{(t_i)} u_{i+1} \oplus c_{63}^{(t_i)} \bigoplus_{i=0}^{l'-1} \gamma_i u_i \oplus e^{(t_i+1)}).$$

We can see $\bigoplus_{j=1}^{4} y^{(t_j+1)}$ can be derived from $(u_0, u_1, ...u_{l'})$ and we express it as

$$\bigoplus_{j=1}^{4} y^{(t_j+1)} = \bigoplus_{i=0}^{l'} c_i' u_i \oplus E',$$

with c_i' being some new coefficients and $E' = \bigoplus_{j=1}^{4} e^{(t_i+1)}$. Then the 16-bit approximation can be expressed as:

$$(\bigoplus_{j=1}^{4} y^{(t_j)}, \bigoplus_{j=1}^{4} y^{(t_j+1)}) = (\bigoplus_{i=0}^{l'-1} c_i u_i, \bigoplus_{i=0}^{l'} c_i' u_i) \oplus (E, E').$$

Here (E, E') can be regarded as following the 16-bit distribution we obtained in the beginning of this subsection. We could see, compared to the 8-bit correlation attack where the output is mapped to the first l' states, here the 16-bit symbol is mapped to the first $l'+1$ states, i.e., one more state is involved. If we find N' tuples like this, we could build a new $[N', l'+1]$ code with the 16-bit approximation as the noisy channel. Then we proceed to the decoding stage to recover the $l'+1$ states.

Next, let us check the complexity. For the preprocessing phase, we could still use the k-tree method to find parity check equations, but through a different DMC with a smaller noise. Using the method before, we could get the best result in terms of complexity is now under $l' = 19$. Now $2^{159.72}$ parity check equations are required with time/space complexity of $2^{175.24}$ and the required length of keystream sample is $2^{175.24}$. For the decoding process, the complexity is $2^{176.06}$. From the result, we can find that while the complexity is still upper bounded by the same order around 2^{176}, the required number of parity check equations reduces from $2^{171.67}$ to $2^{159.72}$. These complexity results are based on only experimental result and are not with full confidence.

6 Conclusion

In this paper, we propose a distinguishing attack and a correlation attack on SNOW 3G using new linear approximations over larger alphabets. We first derive a 24-bit and an 8-bit linear approximation of the FSM and verify them by an experimental test using hypothesis testing. Then we used the derived approximations to launch a distinguishing attack and correlation attack. For the distinguishing attack, we find a weight 4 multiple of the generating polynomial to cancel out the contribution from the LFSR and distinguish the corresponding keystream sample sequence with complexity around 2^{172}. For the correlation attack, we use the 8-bit approximation to recover 160 bits of the initial state with complexity around 2^{177}. As far as we know, these are the first distinguishing and correlation attacks on SNOW 3G. If the key length in SNOW 3G would be increased to 256 bits, the results show that there are then academic attacks on such a version faster than the exhaustive key search.

A possible way to improve the results and achieve a higher bias would be to consider even larger alphabets in the approximations, but this would require much more complex simulation tasks to find and verify biases of different choices of approximation. Another interesting question would be to launch distinguishing attacks which are based directly

on the weight 4 recurrence relation for the LFSR, which has nonzero coefficients that are not all one. If so, it would remove the requirement of having a very long keystream and the attacks could be applied on a large set of short keystreams generated with different IVs.

Acknowledgements

We would like to thank all reviewers for providing valuable comments to the manuscript, and the Ericsson Research Data Center team for their help with compute resources that we used to collect 2^{53} noise samples.

This work was in part financially supported by the Swedish Foundation for Strategic Research, grant RIT17-0005. The author Jing Yang is also supported by the scholarship from the National Digital Switching System Engineering and Technological Research Center, China.

References

[BHNS10] Billy Bob Brumley, Risto M Hakala, Kaisa Nyberg, and Sampo Sovio. Consecutive S-box lookups: A timing attack on SNOW 3G. In *International Conference on Information and Communications Security*, pages 171–185. Springer, 2010.

[BJV04] Thomas Baigneres, Pascal Junod, and Serge Vaudenay. How far can we go beyond linear cryptanalysis? In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 432–450. Springer, 2004.

[BPSZ10a] Alex Biryukov, Deike Priemuth-Schmid, and Bin Zhang. Differential resynchronization attacks on reduced round SNOW 3G. In *International Conference on E-Business and Telecommunications*, pages 147–157. Springer, 2010.

[BPSZ10b] Alex Biryukov, Deike Priemuth-Schmid, and Bin Zhang. Multiset collision attacks on reduced-round SNOW 3G and SNOW 3G. In *International Conference on Applied Cryptography and Network Security*, pages 139–153. Springer, 2010.

[CHJ02] Don Coppersmith, Shai Halevi, and Charanjit Jutla. Cryptanalysis of stream ciphers with linear masking. In *Annual International Cryptology Conference*, pages 515–532. Springer, 2002.

[CJS00] Vladimor V Chepyzhov, Thomas Johansson, and Ben Smeets. A simple algorithm for fast correlation attacks on stream ciphers. In *International Workshop on Fast Software Encryption*, pages 181–195. Springer, 2000.

[CT00] Anne Canteaut and Michaël Trabbia. Improved fast correlation attacks using parity-check equations of weight 4 and 5. In *International Conference on*

the Theory and Applications of Cryptographic Techniques, pages 573–588. Springer, 2000.

[CT12] Thomas M Cover and Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[DC09] Blandine Debraize and Irene Marquez Corbella. Fault analysis of the stream cipher SNOW 3G. In *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 103–110. IEEE, 2009.

[DR13] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[EJ00] Patrik Ekdahl and Thomas Johansson. SNOW – A new stream cipher. In *Proceedings of First Open NESSIE Workshop, KU-Leuven*, pages 167–168, 2000.

[EJ02] Patrik Ekdahl and Thomas Johansson. A new version of the stream cipher SNOW. In *International Workshop on Selected Areas in Cryptography*, pages 47–61. Springer, 2002.

[ETS06a] ETSI/SAGE. Specification of the 3GPP confidentiality and integrity algorithms UEA2& UIA2, document 2: SNOW 3G specification, version 1.1, 2006.

[ETS06b] ETSI/SAGE. Specification of the 3GPP confidentiality and integrity algorithms UEA2& UIA2, document 5: Design and evaluation report, version 1.1, 2006.

[HG97] Helena Handschuh and Henri Gilbert. χ^2 cryptanalysis of the SEAL encryption algorithm. In *International Workshop on Fast Software Encryption*, pages 1–12. Springer, 1997.

[JJ99a] Thomas Johansson and Fredrik Jönsson. Fast correlation attacks based on turbo code techniques. In *Annual International Cryptology Conference*, pages 181–197. Springer, 1999.

[JJ99b] Thomas Johansson and Fredrik Jönsson. Improved fast correlation attacks on stream ciphers via convolutional codes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 347–362. Springer, 1999.

[LJ14] Carl Löndahl and Thomas Johansson. Improved algorithms for finding low-weight polynomial multiples in $F_2[x]$ and some cryptographic applications. *Designs, codes and cryptography*, 73(2):625–640, 2014.

[LLP08] Jung-Keun Lee, Dong Hoon Lee, and Sangwoo Park. Cryptanalysis of Sosemanuk and SNOW 2.0 using linear masks. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 524–538. Springer, 2008.

[LV04] Yi Lu and Serge Vaudenay. Faster correlation attack on Bluetooth keystream generator E0. In *Annual International Cryptology Conference*, pages 407–425. Springer, 2004.

[NH07] Kaisa Nyberg and Miia Hermelin. Multidimensional Walsh transform and a characterization of bent functions. In *2007 IEEE Information Theory Workshop on Information Theory for Wireless Networks*, pages 1–4. IEEE, 2007.

[NW06] Kaisa Nyberg and Johan Wallén. Improved linear distinguishers for SNOW 2.0. In *International Workshop on Fast Software Encryption*, pages 144–162. Springer, 2006.

[Pen96] Walter T Penzhorn. Correlation attacks on stream ciphers: Computing low-weight parity checks based on error-correcting codes. In *International Workshop on Fast Software Encryption*, pages 159–172. Springer, 1996.

[Sie84] Thomas Siegenthaler. Correlation-immunity of nonlinear combining functions for cryptographic applications (corresp.). *IEEE Transactions on Information theory*, 30(5):776–780, 1984.

[Wag02] David Wagner. A generalized birthday problem. In *Annual International Cryptology Conference*, pages 288–304. Springer, 2002.

[WBDC03] Dai Watanabe, Alex Biryukov, and Christophe De Canniere. A distinguishing attack of SNOW 2.0 with linear masking method. In *International Workshop on Selected Areas in Cryptography*, pages 222–233. Springer, 2003.

[ZXM15] Bin Zhang, Chao Xu, and Willi Meier. Fast correlation attacks over extension fields, large-unit linear approximation and cryptanalysis of SNOW 2.0. In *Annual Cryptology Conference*, pages 643–662. Springer, 2015.

Paper II

Spectral analysis of ZUC-256

Abstract

In this paper we develop a number of generic techniques and algorithms in spectral analysis of large linear approximations for use in cryptanalysis. We apply the developed tools for cryptanalysis of ZUC-256 and give a distinguishing attack with complexity around 2^{236}. Although the attack is only 2^{20} times faster than exhaustive key search, the result indicates that ZUC-256 does not provide a source with full 256-bit entropy in the generated keystream, which would be expected from a 256-bit key. To the best of our knowledge, this is the first known academic attack on full ZUC-256 with a computational complexity that is below exhaustive key search.

Keywords: ZUC-256, Stream Cipher, 5G Mobile System Security.

1 Introduction

ZUC is the stream cipher being used as the core of 3GPP Confidentiality and Integrity Algorithms UEA3 & UIA3 for LTE networks [ETS11a]. It was initially proposed in 2010 as the candidate of UEA3 & UIA3 for use in China. After external and public evaluation and two ZUC workshops, respectively in 2010 and 2011, it was ultimately accepted by 3GPP SA3 as a new inclusion in the LTE standards with a 128-bit security level, i.e., the secret key is 128-bit long.

Like most stream ciphers, ZUC has a linear part, which is an LFSR, and a non-linear part, called the F function, to disrupt the linearity of the LFSR contribution. The design is different from common stream ciphers which are often defined over binary fields $GF(2)$ or extension fields of $GF(2)$, the LFSR in ZUC is defined over a prime field $GF(p)$ with $p = 2^{31} - 1$ while the registers in F are defined over $GF(2^{32})$. There is a bit-reorganization (BR) layer between the LFSR and F serving as a connection layer to extract bits from the LFSR and push them into F. Thus standard cryptanalysis against common stream ciphers can not be directly applied to ZUC and till now, there is no efficient cryptanalysis of ZUC with an attack faster than exhaustive key search.

After ZUC was announced, there were a number of research work conducted to evaluate the cipher [ETS11b], [STL10], [WHN+12]. A weakness in the initialization phase was found in [STL10], [WHN+12] and this directly resulted in an improved version. After

Jing Yang, Thomas Johansson, and Alexander Maximov. Spectral analysis of ZUC-256. IACR Transactions on Symmetric Cryptology, pages 266–288, 2020.

the adoption as the UEA3 & UIA3 standard, there were additional work in cryptanalysis of ZUC. A guess-and-determine attack on ZUC was proposed in [GDL13] based on half-words, i.e. 16-bit blocks, by splitting the registers in the LFSR and FSM into high and low 16 bits, where some carry bits are introduced due to the splitting. It requires 6 keystream words and the complexity is $O(2^{392})$, which is, however, higher than exhaustive key search. In [ZFL11], a differential trail covering 24 rounds of the initialization stage is given, but this does not pose a threat since ZUC has 32 initialization rounds. [LMVH15] also shows that weak inputs do not exist in ZUC when it is initialized with 32 rounds. These results indicate that ZUC is resistant against common attacks.

In January 2018, ZUC-256 was announced as the 256-bit version of ZUC [Tea18] in order to satisfy the 256-bit security level requirement of 5G from 3GPP [3GP18]. Compared to ZUC-128, the structure of ZUC-256 remains the same, while only the initialization and message authentication code generation phases are improved to match with the 256-bit security level. Subsequently, in July 2018, a workshop on ZUC-256 was held and some general cryptanalyses were presented, but no obvious weaknesses of ZUC-256 were found. To conclude, till now, there are no efficient cryptanalysis techniques succeeding to reduce the claimed security levels of ZUC (128-bit or 256-bit).

In this paper, we propose a distinguishing attack on ZUC-256 with computational complexity around 2^{236}, by linearly approximating the non-linear part F and the different finite fields between the LFSR and F. The important techniques we employ to find a good linear approximation and compute the bias are called *spectral tools* here for cryptanalysis, using e.g., the Walsh Hadamard Transform(WHT) and the Discrete Fourier Transform (DFT). The spectral tools for cryptanalysis are widely used in linear cryptanalysis to, for example, efficiently compute the distribution or the bias of a linear approximation, since there exist fast algorithms for WHT and DFT which can reduce the computational complexity from $O(N^2)$ to $O(N \log N)$ [MJ05], [LD16]. It is also widely used to investigate the properties of Boolean functions and S-boxes, which can be considered as vectorial Boolean functions, like correlation, autocorrelation, propagation characteristics and value distributions [NH07], [HN12]. We explore the use of WHT and DFT and find new results about efficiently computing the bias or correlations. Importantly, we show how a permutation or a linear masking in the time domain would affect the spectrum points in the frequency domain for widely used operations and components, such as $\boxplus, \oplus$, and S-boxes. Based on that, we give a number of further results on how to choose linear maskings in the time domain by considering the behavior of noise variables in the frequency domain such that a decent approximation with a large bias can be found.

We employ the new findings in spectral analysis of ZUC-256 and use them to develop a distinguishing attack. Even though the distinguishing attack is not a very strong one, it indicates that ZUC-256 can not achieve the full 256-bit security level under this case.

The rest of this paper is organized as follows. We first give the general design and structure of ZUC-256 in Section 2 and then the spectral analysis techniques are given in Section 3. After that, we in Section 4 give a distinguishing attack on ZUC-256 using the spectral tools. Specifically, we first derive a linear approximation in Section 4.1; and then we show how to efficiently derive the bias of the approximation in Section 4.2 $\sim$ Section 4.4 by using the spectral analysis and a technique called "bit-slicing technique"; and lastly we give the distinguishing attack based on the derived approximation. In

Section 5, we conclude the paper.

2 Description of ZUC-256

In this section we give a brief description of the ZUC-256 algorithm. Basically, the structure of ZUC-256 is exactly the same as that of ZUC-128, except that the length of the secret key K is changed to be 256-bit long and the loading process of the key and IV is modified accordingly [Tea18]. ZUC-256 takes a 256-bit secret key K and a 128-bit initial vector IV as input and produces a sequence that is usually called *keystream*. In this paper, we use $Z^{(t)}$ to denote the generated keystream block at a time instance t for $t = 1, 2, \ldots$. In ZUC-256, each keystream block is a 32-bit word, so we write $Z^{(t)} \in GF(2^{32})$, $t = 1, 2, \ldots$. Furthermore, each (K, IV) pair should produce a unique keystream sequence, and in practice K is usually fixed and the IV value varies to generate many different keystream sequences.

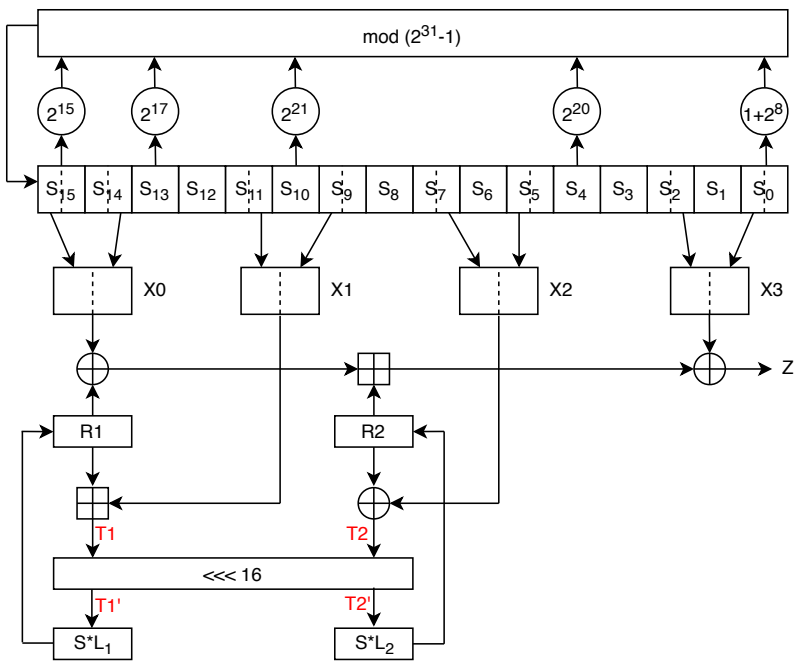

Figure 1: The keystream generation phase of the ZUC-256 stream cipher

The overall schematic of the ZUC-256 algorithm is shown in Figure 1. It consists of three layers: the top layer is a linear feedback shift register (LFSR) of 16 stages; the bottom layer is a nonlinear block which is called F function; while the middle layer, called bit-reorganization (BR) layer, is a connection layer between the LFSR and F. Now we would give some details of the three layers, and for more details we refer to the original design document [ETS11a].

The LFSR Layer

The LFSR part consists of 16 cells denoted $(s_0, s_1, ..., s_{15})$ each holding 31 bits and giving 496 bits in total. Every value in the cells is an element from the finite field $GF(p)$, where $p = 2^{31} - 1$ and it can be written in a binary representation as

$$x = x_0 + x_1 2 + \ldots + x_{30} 2^{30},$$

where $x_i \in \{0, 1\}$ for $0 \leq i \leq 30$. Then $2^k \cdot x \mod p$ is computed as $x \lll_{31} k$, where $\lll_{31} k$ is the 31-bit left circular shift by k steps. This makes the implementation quite efficient. One can see that the LFSR in ZUC is operating over a prime field instead of $GF(2)$ or $GF(2^n)$ as most stream ciphers do. This makes it insusceptible to common linear cryptanalysis. The feedback polynomial of the LFSR is given by:

$$P(x) = -x^{16} + 2^{15}x^{15} + 2^{17}x^{13} + 2^{21}x^{10} + 2^{20}x^4 + (1 + 2^8) \equiv 0 \mod p.$$

$P(x)$ is a primitive polynomial over $GF(p)$ and this ensures that the LFSR sequence is an m-sequence with period $p^{16} - 1 \approx 2^{496}$. If we denote the LFSR state at clock t as $(s_0^{(t)}, s_1^{(t)}, ..., s_{15}^{(t)})$, then at the next clock $t+1$, s_i is shifted to s_{i-1}, i.e., $s_i^{(t)} = s_{i-1}^{(t+1)}$, for $1 \leq i \leq 15$, while $s_{15}^{(t+1)}$ is updated by:

$$s_{15}^{(t+1)} = 2^{15}s_{15}^{(t)} + 2^{17}s_{13}^{(t)} + 2^{21}s_{10}^{(t)} + 2^{20}s_4^{(t)} + (1 + 2^8)s_0^{(t)} \mod p.$$

If $s_{15}^{(t+1)} = 0$, then set $s_{15}^{(t+1)} = p$ (i.e., the representation of element 0 is the binary representation of p).

The BR Layer

The BR layer is the connection layer between the LFSR and F. It extracts 128 bits from the LFSR and forms four 32-bit words $X0, X1, X2, X3$ with the first three being fed to F and the last one XOR-ed with the output of F to finally generate the keystream symbol. For a cell s_i in the LFSR, the low and high 16 bits are extracted as:

$$s_{iL} = s_i[0...15] \quad \text{and} \quad s_{i_H} = s_i[15...30].$$

Then $X0, X1, X2, X3$ are constructed as follows:

$$X0 = s_{15H}||s_{14L}, \qquad X1 = s_{11L}||s_{9H}, \qquad X2 = s_{7L}||s_{5H}, \qquad X3 = s_{2L}||s_{0H},$$

where $h||l$ denotes the concatenation of two 16-bit integers h and l into a 32-bit one, with l being the least significant bits and h being the most significant bits of the result. Then $X1, X2$ will be sent into F to update the registers there.

The Non-linear Layer F

The nonlinear layer F has two internal 32-bit registers $R1$ and $R2$ being updated through linear and nonlinear operations. It is a compression function taking $X0, X1, X2$

as the input and producing one 32-bit word which would be used to generate the keystream symbol as below:

$$Z^{(t)} = ((R1^{(t)} \oplus X0) \boxplus_{32} R2^{(t)}) \oplus X3.$$

Then F is updated by:

$$\begin{aligned} T1 &= R1^{(t)} \boxplus_{32} X1, \\ T2 &= R2^{(t)} \oplus X2, \\ R1^{(t+1)} &= S(L_1(T1_L||T2_H)), \\ R2^{(t+1)} &= S(L_2(T2_L||T1_H)). \end{aligned}$$

Here $S = (S0, S1, S0, S1)$ is a 32×32 S-box composed of four juxtaposed S-boxes, where S_0 and S_1 are two different 8-to-8-bit S-boxes. L_1, L_2 are two 32×32 linear transforms which are defined as follows:

$$\begin{aligned} L_1(X) &= X \oplus (X \lll_{32} 2) \oplus (X \lll_{32} 10) \oplus (X \lll_{32} 18) \oplus (X \lll_{32} 24), \\ L_2(X) &= X \oplus (X \lll_{32} 8) \oplus (X \lll_{32} 14) \oplus (X \lll_{32} 22) \oplus (X \lll_{32} 30). \end{aligned}$$

Just like other stream ciphers, ZUC-256 uses an initialization phase before generating a keystream sequence, to fully mix the secret key and IV. During the initialization phase, the key and IV are loaded into the LFSR registers and the cipher runs 32 iterations with the output from the F function being fed back to the LFSR instead of producing keystream symbols. After the initialization, the cipher enters the keystream mode, with the first output word from F being discarded and the following outputs forming the keystream symbols by XOR-ing with $X3$. Since the attack in this paper only uses the keystream mode, we do not give the details of the initialization mode, but refer to the design document for the details [ETS11a], [Tea18].

3 Spectral tools for cryptanalysis

In multidimensional linear cryptanalysis one often has to deal with large distributions, and be able to find good approximations with large biases that can further be used in an attack. In this section, we give several techniques in spectral analysis which help to efficiently explore a good linear approximation and compute its bias. We will later use most of the presented techniques in cryptanalysis of ZUC-256.

Notations. Let $X^{(1)}, X^{(2)}, \ldots, X^{(t)}$ be t independent random variables taking values from an alphabet of n-bit integers, such that the total size of the alphabet is

$$N = 2^n.$$

For a random variable X, let the sequence of $X_k, k = 0, 1, \ldots, N-1$ represent the distribution table of X, i.e., $X_k = Pr\{X = k\}$, or a sequence of occurrence values in the time domain, e.g. $X_k =$ the number of occurrences of $X = k$. If such a sequence of numbers would be normalized by dividing each entry by the total number of occurrences, we would talk about an empirical distribution or a *type* [CT12].

We will denote by $\mathcal{W}(X)$ the N-point Walsh-Hadamard Transform (WHT) and by $\mathcal{F}(X)$ the N-point Discrete Fourier Transform (DFT). Individual values of the transforms will be addressed by $\mathcal{W}(X)_k$ and $\mathcal{F}(X)_k$, for $k = 0, 1, \ldots, N-1$. We will denote by $\hat{X}_k$ the spectrum value of a point k, i.e., $\hat{X}_k = \mathcal{W}(X)_k$ or $\hat{X}_k = \mathcal{F}(X)_k$, depending on the context. The values $\hat{X}_k$, for $k = 0, 1, \ldots, N-1$, in the frequency domain constitute the *spectrum* of X.

When considering Boolean operations, such as $k \cdot M$, where k is an n-bit integer (or an index) and M is an $n \times n$ Boolean matrix, it should be understood as that the integer k is 1-to-1 mapped to a Boolean vector of length n containing the corresponding bits of the integer k in its binary representation. Then a Boolean multiplication is performed modulo 2, and the resulting Boolean vector can thus be 1-to-1 mapped back to an n-bit integer.

WHT and DFT. The DFT is defined as:

$$\hat{X}_k = \mathcal{F}(X)_k = \sum_{j=0}^{N-1} X_j \cdot e^{-\frac{i2\pi}{N}kj}, \quad \text{for } k = 0, 1, ..., N-1,$$

where $\omega_0 = e^{-i2\pi/N}$ is a primitive N-th root of unity. Every point value $\mathcal{F}(X)_k$ is a complex number with the real part Re() and imaginary part Im(), i.e., $\hat{X}_k = \mathrm{Re}(\hat{X}_k) + i \cdot \mathrm{Im}(\hat{X}_k)$. WHT is a special variant of DFT and it is defined as

$$\hat{X}_k = \mathcal{W}(X)_k = \sum_{j=0}^{N-1} X_j \cdot (-1)^{k \cdot j},$$

where $k \cdot j$ now denotes the bitwise dot product of the binary representation of the n-bit indices k and j. I.e., one can rewrite the dot product in the vectorial binary form:

$$k \cdot t = (k_0, k_1, \ldots, k_{n-1}) \cdot (j_0, j_1, \ldots, j_{n-1})^T \mod 2,$$

where k_i, j_i are the i-th bits of the binary representation of k and j, for $i = 0, 1, \ldots, n-1$. Every $\mathcal{W}(X)_k$ has only the real part and it is an integer.

The squared magnitude at a point k is derived by $|\hat{X}_k|^2 = \mathrm{Re}(\hat{X}_k)^2 + \mathrm{Im}(\hat{X}_k)^2$. The point $k = 0$ in the spectrum represents the sum of all values in the time domain for both WHT and DFT cases, i.e.,

$$|\hat{X}_0| = \sum_{j=0}^{N-1} X_j. \tag{1}$$

There are many well-known fast algorithms computing DFT or WHT in time $O(N \log N)$ and this makes the spectral transform widely used in cryptanalysis and in many other areas as well.

Convolutions. A typical operation in linear multidimensional cryptanalysis is to compute the distribution of a noise variable which is the sum ($\oplus$ or $\boxplus$) of other noise variables (referred to as sub-noise variables). While computing the distribution directly in the time domain might be complicated, the complexity could be largely reduced when using DFT and WHT [MJ05] through:

$$\begin{aligned}(X^{(1)} \boxplus X^{(2)} \boxplus \ldots \boxplus X^{(t)})_k &= \mathcal{F}^{-1}(\mathcal{F}(X^{(1)}) \cdot \mathcal{F}(X^{(2)}) \cdot \ldots \cdot \mathcal{F}(X^{(t)}))_k, \\ (X^{(1)} \oplus X^{(2)} \oplus \ldots \oplus X^{(t)})_k &= \mathcal{W}^{-1}(\mathcal{W}(X^{(1)}) \cdot \mathcal{W}(X^{(2)}) \cdot \ldots \cdot \mathcal{W}(X^{(t)}))_k,\end{aligned} \tag{2}$$

where $\cdot$ is the point-wise multiplication of two spectrum vectors. In particular, the overall complexity is now $O(t \cdot N \log N)$.

3.1 Precision problems and the bias in the frequency domain

The bias of a multidimensional noise variable X is often expressed in the time domain as the *Squared Euclidean Imbalance (SEI)*, which is also called the *capacity* in some papers [HN12], defined in [BJV04] as follows:

$$\epsilon(X) = N \sum_{i=0}^{N-1} (X_i/f - 1/N)^2, \tag{3}$$

where $f = \sum_{i=0}^{N-1} X_i$ is the normalization factor, used in case when the distribution table of X is not normalized. For example, the table in the time domain for X stores the number of occurrences of each entry. If the distribution table of X is already normalized then $f = 1$, as expected for the sum of all probabilities.

It is known that to distinguish a noise distribution X with the above bias $\epsilon(X)$ from random using a hypothesis testing, one needs to collect $O(1/\epsilon(X))$ samples from this distribution [BJV04], [HG97].

Precision problems. Assume that we want to compute the bias of a noise variable X, which is the sum ($\oplus$ or $\boxplus$) of t other sub-noises $X^{(1)}, \ldots, X^{(t)}$ using the convolution formulae given in Equation (2). If the expected bias is $\epsilon(X) \approx 2^{-p}$, then in practice we would expect to have probability values around $2^{-n} \pm 2^{-p/2-n}$, in average, and then a float data type should be able to maintain at least $O(|p/2|)$ bits of precision for every value of X_k in the time domain, conditioned that the float data type has the exponent field (e.g., data types `float` and `double` in standard C).

For example, when we want to compute a bias $\epsilon > 2^{-512}$ ($p = 512$) then underlying data types for float or integer values should hold *at least* 256 bits of precision. This forces a program to utilize a large number arithmetic (e.g., BIGNUM, Quad, etc), which requires larger RAM and HDD storage, and expensive computation time.

In the following, we show that the bias of X may be computed in the frequency domain without having to switch to the time domain, and the required precision may fit well into the standard type `double` in C/C++.

Theorem 1. For an n-bit random variable X with either normalized or non-normalized probability distribution $(X_0, X_1, ..., X_{N-1})$ and its spectrum $(\hat{X}_0, \hat{X}_1, ..., \hat{X}_{N-1})$, computed either in DFT or WHT, the bias $\epsilon(X)$ can be computed in the frequency domain as the sum of normalized squared magnitudes of all nonzero points, where the zero point, $\hat{X}_0$, serves as the normalization factor, i.e.,

$$\epsilon(X) = \frac{1}{|\hat{X}_0|^2} \sum_{i=1}^{N-1} |\hat{X}_i|^2.$$

Proof. From Equation (1) we get that the normalization factor is $f = |\hat{X}_0|$. The SEI expression can be written as $\epsilon(X) = N \sum_{i=0}^{N-1} (X_i/f - U_i)^2$, where U is the uniform distribution. According to Parseval's theorem, we can derive $\epsilon(X) = N \sum_{i=0}^{N-1} |X_i/f -$

$U_i|^2 = N \cdot \frac{1}{N} \sum_{i=0}^{N-1} |\mathcal{F}(X/f - U)_i|^2 = \sum_{i=0}^{N-1} |\hat{X}_i/f - \hat{U}_i|^2$. Since $\hat{X}_0 = f, \hat{U}_0 = 1$, and $\hat{U}_k = 0$ for $k = 1, 2, \ldots, N-1$, we get that $\epsilon(X) = \sum_{i=1}^{N-1} |\hat{X}_i/f|^2$, from which the proof follows. □

Theorem 1 means that the required precision of values in the frequency domain can be as small as just a few bits, but the exponent value must be correct and preserved. In C/C++ it is therefore good enough to store the spectrum of a distribution in type `double` which has 52 bits of precision and the smallest exponent it can hold is 2^{-1023}. We can barely imagine cryptanalysis where the expected bias will be smaller than that (and if it will, we can always change the factor $\hat{X}_0$ to a larger value).

A similar technique to compute the bias in the frequency domain has been given in [BJV04], but the probability sequence in the time domain there is the probability differences to the uniform distribution, while the probability sequence here is the original probabilities of the variable X. By this, we could further directly compute the bias of the sum ($\boxplus$ or $\oplus$) of several sub-noises in the frequency domain by combining Theorem 1 and Equation (2): the bias of the $\boxplus$-sum of several sub-noises can be computed by

$$\epsilon(X^{(1)} \boxplus \ldots \boxplus X^{(t)}) = \frac{1}{f} \sum_{k=1}^{N-1} |\mathcal{F}(X^{(1)})_k|^2 \cdot \ldots \cdot |\mathcal{F}(X^{(t)})_k|^2 = \frac{1}{f} \sum_{k=1}^{N-1} \left(\prod_{i=1}^{t} |\mathcal{F}(X^{(i)})_k| \right)^2,$$

$$\text{where} \quad f = |\mathcal{F}(X^{(1)})_0|^2 \cdot \ldots \cdot |\mathcal{F}(X^{(t)})_0|^2 = \left(\prod_{i=1}^{t} |\mathcal{F}(X^{(i)})_0| \right)^2, \tag{4}$$

and a similar result holds under the $\oplus$-sum for the WHT case. Note, if we convert each spectrum value $|\mathcal{F}(X^{(i)})_k|$ to $\log_2(|\mathcal{F}(X^{(i)})_k|^2)$ (and, similarly, for the WHT case), then arithmetics in the frequency domain, such as in Equation (4), change from computing products to computing sums. This can give additional speed-up, RAM and storage savings. Later we will show how these results help to find a good approximation.

The main observation and motivation for developing further algorithms. In linear cryptanalysis of stream ciphers where we have an FSM and an LFSR, the approach is usually to first linearly approximate the FSM and get a noise variable X of the linear approximation, then the LFSR contribution in the linear approximation is canceled out by combining several (say t) time instances, such that only noise terms remain. Thus, the final noise is the t-folded noise of X, written as $t \times X$ (i.e., the total noise is the sum of t independent noise variables that follow the same distribution as X), for which the bias is written $\epsilon(t \times X)$. Usually, an attacker tries to maximize this value.

One important observation from Theorem 1 and Equation (4) is that if there is a peak (maximum) value $|\hat{X}_k|$ in the spectrum of X at some nonzero position k, then that peak value will be the dominating contributor to the bias $\epsilon(t \times X)$, as it will contribute $|\hat{X}_k|^{2t}$, while other points in the spectrum of X will have a much less (or even negligible) contribution to the total bias as t grows.

This important observation also affects the case when trying to align the spectrum points from several sub-noises with different distributions to achieve a large bias. We should actually try to move the peak spectrum values of each sub-noise such that they are aligned at some nonzero index k. Then the product of those peak values will result

in a large total bias value. This motivates us to develop further algorithms to permute or linearly mask variables and align them at an expected or desired spectrum location k in the frequency domain. In the next sections we will give new findings and algorithms for WHT and DFT cases, which can be helpful in searching for a good linear approximation for common operations in the nonlinear part of a stream cipher, such as $\boxplus, \oplus$, S-boxes, etc.

3.2 Algorithms for WHT type approximations

Consider the expression

$$X = M^{(1)}X^{(1)} \oplus M^{(2)}X^{(2)} \oplus \ldots \oplus M^{(t)}X^{(t)}, \tag{5}$$

where distribution tables of $X^{(i)}$'s are known, and an attacker can freely select $n \times n$ full-rank Boolean matrices $M^{(i)}, i = 1 \ldots t$; we want to find a method to efficiently search for the choices of $M^{(i)}$'s to maximize the total bias $\epsilon(X)$. Below we first give a theorem and then an algorithm to achieve this.

Theorem 2. Given an n-bit variable X and its distribution, for an $n \times n$ full-rank Boolean matrix M we have

$$\mathcal{W}(M \cdot X)_k = \mathcal{W}(X)_{k \cdot M}$$

Proof. Note that $Pr\{M \cdot X = j\} = Pr\{X = M^{-1} \cdot j\}$, then we have $\mathcal{W}(M \cdot X)_k = \sum_{j=0}^{N-1} X_{M^{-1} \cdot j}(-1)^{k \cdot j} \overset{[i=M^{-1} \cdot j]}{=} \sum_{i=0}^{N-1} X_i(-1)^{k \cdot M \cdot i} = \mathcal{W}(X)_{k \cdot M}$. □

Note that the left-side matrix multiplication $M \cdot X$ is switched to the right-side matrix multiplication $k \cdot M$.

We want to maximize $\epsilon(X)$ in Equation (5) and we know that if spectrum values of $X^{(i)}$'s are aligned after linear masking of $M^{(i)}$'s, we could achieve a large bias. By "aligned" we mean that the largest spectrum magnitudes of each $X^{(i)}$ are at the same location, and this holds for the second largest, the third largest spectrum magnitudes and so on. But in practice, it is unlikely to achieve such a perfect alignment for all spectrum points. Instead, we can try to align n largest spectrum magnitudes and thus getting a decent bias. Algorithm 1 below can be used to achieve this based on Theorem 2.

Intuitively, the main trick in Algorithm 1 happens in the step 12. For example, let us take the first row of K as the integer k, and the first row from Λ as the integer λ. The integer λ will eventually be the first value in the sorted list, $\lambda = \lambda_1$, where we have $|\mathcal{W}(X^{(q)})_{\lambda_1}| \to \max$. Following Theorem 2 we then get that the k'th spectrum point of $M^{(q)}X^{(q)}$, expressed as $\mathcal{W}(M^{(q)}X^{(q)})_k = \mathcal{W}(X^{(q)})_{k \cdot M^{(q)}}$, now actually have the largest spectrum value $\mathcal{W}(X^{(q)})_{\lambda_1}$, since $k \cdot M^{(q)} = \lambda = \lambda_1$ by construction in that step 12.

Algorithm 1 Find $M^{(1)}, \ldots, M^{(t)}$ that maximize spectrum points of X at n indices K

Input The distributions of $X^{(i)}$'s $(1 \leq i \leq t)$ and the index matrix K, which must be an $n \times n$ full-rank Boolean matrix where each row $K_{j,*}$ is a binary form of the j-th spectrum index where we want the best alignment to happen.
Output The $n \times n$ full-rank Boolean matrices $M^{(1)}, M^{(2)}, \ldots, M^{(t)}$

procedure WhtMatrixAlign$(K, X^{(1)}, \ldots, X^{(t)})$
  for $q = 1, \ldots, t$ **do**
    Compute $W = \mathcal{W}(X^{(q)})$
    Let $\{\lambda_1, \lambda_2, \ldots, \lambda_{N-1}\}$ be all nonzero indices sorted as $|W_{\lambda_i}| \geq |W_{\lambda_j}|, i < j$
    Construct the $n \times n$ Boolean matrix Λ in a greedy approach as follows:
    Set a variable $l = 1$ and an $n \times n$ Boolean matrix $\Lambda = \mathbf{0}$
    for $i = 0, 1, \ldots, n-1$ **do**
      do
        Set the i-th row of Λ as $\Lambda_{i,*} = \lambda_l$
        $l := l + 1$
      while $rank(\Lambda) \neq (i + 1)$
    Then we want that $K \cdot M^{(q)} = \Lambda$, from which we derive $M^{(q)} = K^{-1} \cdot \Lambda$.

As a comment, in Algorithm 1 we do not really have to sort and find all $N-1$ indices of λ, it is most likely that the inner loop will use just a bit more than n values of the first "best" λ's. Thus, it is enough to only collect the best $c \cdot n$ indices, for some small $c = 2, 3, 4$, out of which the full-rank matrix Λ can be constructed. We note that the algorithm does not necessarily give the best overall bias, but it guarantees that at least n points in the spectrum of X will have the largest possible peak values.

Linear approximation of S-boxes. S-boxes, which can be regarded as vectorial Boolean functions, are widely used in both stream ciphers and block ciphers, serving as the main nonlinear component to disrupt the linearity. Therefore, linear approximations of S-boxes are widely studied in cryptanalysis. For one dimensional approximations of an S-box, i.e., $ax \oplus bS(x)$ where $a, b \in GF(2^n)$ are linear masks, the common way is to construct a linear approximation table (LAT), by trying all possibilities of a, b values. The complexity is $O(2^{2n})$, which is affordable for small S-boxes, e.g., 4-bit, 8-bit. WHT is usually employed to speed up the process. For multiple (vectorized) linear approximations, i.e., $Ax \oplus BS(x)$, where A, B are $n \times n$ full-rank binary masking matrices, testing every choice of A, B would be impossible, and the main task is rather to find A, B such that the linear approximation would be highly biased. Some papers investigated properties of multiple linear approximations, such as [HN12], [HCN19], but there is not much research on how a linear masking in the time domain would affect the spectrum points in the frequency domain, and how to explore good linear maskings to achieve a highly biased approximation. Below we give some new results in these aspects.

Let $S(x)$ be an S-box that maps $\mathbb{Z}_N \to \mathbb{Z}_N$, and $x \in \mathbb{Z}_N$, $N = 2^n$. For the sake of notation in this section the expression of the kind $\mathcal{W}(F(x))$ means the WHT over the distribution table that is constructed through the function $F(x) : \mathbb{Z}_N \to \mathbb{Z}_N$ by running through all values of x.

For an n-bit S-box $S(x)$ and an n-bit integer k, let us introduce the k-th binary-valued

(i.e., $\pm 1/N$) function, associated with $S(x)$, as follows

$$B^{[k]}_{\{S(x)\}} = 1/N \cdot (-1)^{k \cdot S(x)}, \quad \text{for } x = 0, 1, \ldots, N-1,$$

where $k \cdot S(x)$ is the scalar product of two binary vectors, i.e., $k \cdot S(x) = \bigoplus_{i=0}^{n-1} k_i \cdot S(x)_i$, and $1/N$ is the normalization factor. Such a combination (without the normalization factor $1/N$) is called a component of the S-box, and for a well-chosen S-box, every component should have good cryptographic properties. We can derive the following results.

Theorem 3. For a given S-box $S(x)$ and a full-rank Boolean matrix Q we have

$$\mathcal{W}(S(x) \oplus Q \cdot x)_k = \mathcal{W}(B^{[k]}_{\{S(x)\}})_{k \cdot Q}.$$

Proof. Let X be a *non-normalized* noise distribution of the expression $(S(x) \oplus Q \cdot x)$, where every X_j is the number of different values of x for which $j = S(x) \oplus Q \cdot x$. Then we have:

$$\mathcal{W}(S(x) \oplus Q \cdot x)_k = \frac{1}{N} \sum_{j=0}^{N-1} X_j \cdot (-1)^{k \cdot j} = \frac{1}{N} \sum_{x=0}^{N-1} (-1)^{k \cdot (S(x) \oplus Q \cdot x)} = \sum_{x=0}^{N-1} B^{[k]}_{\{S(x)\}} \cdot (-1)^{k \cdot Q \cdot x},$$

from which the result follows, since the last term is exactly $\mathcal{W}(B^{[k]}_{\{S(x)\}})_{k \cdot Q}$. □

Theorem 3 can now be used to derive a matrix Q such that at least n points in the noise spectrum, where the noise variable is $X = S(x) \oplus Q \cdot x$, will have peak values, thus, making the total bias $\epsilon(X)$ large. Basically, we first search for the $> n$ best one-dimensional linear masks and then we build a matrix Q that contains these best peak values in the spectrum, see Algorithm 2 for details.

In Algorithm 2, the choice of the parameter c should be such that we would not need to generate final rows of K and Λ randomly. Alternatively, one can also modify the algorithm as follows: when a new triple is added to Φ, we run the greedy algorithm and flag records in Φ that are used to construct K and Λ. After that, the first worst triple in Φ (starting from the end of Φ) that was not flagged is removed if the size of Φ reaches the limit.

The algorithm does not guarantee to get the maximum possible overall bias, but it guarantees that at least the maximum possible peak value will be present in the noise spectrum, which would allow to get a fairly large bias in the end. The complexity is $O(N^2 \log N)$, but in practice there are usually other sub-noises that depend solely on k and λ, which can be used to select a subset of "promising" k and λ values for actual probing of the total noise spectrum, as it will be shown later for the ZUC-256 case.

Other useful formulae on spectral analysis of S-boxes can be derived in Corollary 1, based on Theorem 2 and Theorem 3.

Corollary 1. Let M, P, Q be $n \times n$ full-rank Boolean matrices, and let $S(x)$ be a bijective

Algorithm 2 Find Q that maximizes n spectrum points of $S(x) \oplus Q \cdot x$

Input The n-bit S-box $S(x)$

Output The $n \times n$ full-rank Boolean matrix Q

1: **procedure** WhtSBoxApproximation($S(x)$)
2: Let Φ be the sorted list of maximum $c \cdot n$ (for some small $c \approx 4$) best triples (k, λ, ω) sorted by the magnitude of ω, where k is the index of the binary-valued function of the S-box, λ denotes the index of the spectrum points and ω is the corresponding spectrum value. If the list is full and we want to add a new triple then the last (worst) list entry is removed.
3: **for** $k = 1, \ldots, N-1$ **do**
4: Compute $W = \mathcal{W}(B^{[k]}_{\{S(x)\}})$, where $B^{[k]}_{\{S(x)\}} = 1/N \cdot (-1)^{k \cdot S(x)}$
5: **for** $\lambda = 1, \ldots, N-1$ **do**
6: Consider the triple $(k, \lambda, \omega = |W_\lambda|)$. If ω is larger than that in the worst triple of Φ (with the smallest ω), then add $(k, \lambda, \omega = |W_\lambda|)$ to the list.
7: From the list Φ use the greedy approach to construct the $n \times n$ full-rank Boolean matrices K and Λ, similar to how it was done in Algorithm 1.
8: Set $l = 0$
9: **for** $i = 0, 1, \ldots, n-1$ **do**
10: **do**
11: **if** $l = |\Phi|$ **then**
12: generate the remaining rows of K and Λ randomly
13: exit from the for-loop
14: Set the i-th row of K as the k value of the l-th entry of Φ, i.e., $K_{i,*} = \Phi(l).k$
15: Set the i-th row of Λ as the λ value of the l-th entry of Φ, i.e., $\Lambda_{i,*} = \Phi(l).\lambda$
16: $l := l + 1$
17: **while** $rank(K) \neq (i+1)$ or $rank(\Lambda) \neq (i+1)$
18: Set $Q = K^{-1}\Lambda$.

S-box over n-bit integers. Then

$$\mathcal{W}(MS(Px) \oplus Qx)_k = \mathcal{W}(M(S(x) \oplus M^{-1}QP^{-1}x))_k = \mathcal{W}(S(x) \oplus M^{-1}QP^{-1}x)_{k \cdot M} \tag{6}$$

$$= \mathcal{W}(B^{[k \cdot M]}_{\{S(x)\}})_{k \cdot M \cdot M^{-1}QP^{-1}} = \mathcal{W}(B^{[k \cdot M]}_{\{S(x)\}})_{k \cdot QP^{-1}}, \tag{7}$$

$$\mathcal{W}(MS(Px) \oplus Qx)_k = \mathcal{W}(Mx \oplus QP^{-1}S^{-1}(x))_k = \mathcal{W}(B^{[k \cdot QP^{-1}]}_{\{S^{-1}(x)\}})_{k \cdot M}, \tag{8}$$

$$\mathcal{W}(M(S(Px) \oplus Qx))_k = \mathcal{W}(S(x) \oplus QP^{-1}x)_{k \cdot M} = \mathcal{W}(B^{[k \cdot M]}_{\{S(x)\}})_{k \cdot MQP^{-1}}, \tag{9}$$

$$\mathcal{W}(B^{[k \cdot P]}_{\{S(x)\}})_{k \cdot Q} = \mathcal{W}(B^{[k \cdot Q]}_{\{S^{-1}(x)\}})_{k \cdot P}. \tag{10}$$

Theorem 4 (Linear transformation of S-boxes)**.** Let us consider the following k-th binary-valued function at its spectrum point $\lambda = k \cdot M$, for some full-rank Boolean matrix M, where the original S-box $S(x)$ is linearly transformed with other full-rank

Boolean matrices R and Q,

$$\mathcal{W}(B^{[k]}_{\{RS(Qx)\}})_\lambda. \tag{11}$$

We want to find a set of the best m triples $\{(k,\lambda,\epsilon)\}$ sorted by the maximum bias ϵ. Assume we have a fast method to find best m triples $\{(k',\lambda',\epsilon)\}$ for $\mathcal{W}(B^{k'}_{\{S(x)\}})_{\lambda'}$ instead, then that set can be converted to $\{(k,\lambda,\epsilon)\}$ as follows:

$$\{(k,\lambda,\epsilon)\} = \{(k'\cdot R^{-1},\ \lambda'\cdot Q,\ \epsilon)\}.$$

Proof. $\mathcal{W}(B^{[k]}_{\{RS(Qx)\}})_\lambda = \mathcal{W}(RS(Qx)\oplus Mx)_k = \mathcal{W}(RS(x)\oplus MQ^{-1}x)_k =$
$= \mathcal{W}(B^{[k\cdot R]}_{\{S(x)\}})_{k\cdot MQ^{-1}}$, and the result follows. □

Theorem 5 (S-box as a disjoint combination)**.** Let us consider an n-bit S-box constructed from t smaller $n_1, n_2, \ldots, n_t$-bit S-boxes $S_1(x_1), S_2(x_2), \ldots, S_t(x_t)$, such that

$$S(x) = \begin{pmatrix} S_1(x_1) & S_2(x_2) & \ldots & S_t(x_t)\end{pmatrix}^{\mathrm{T}},$$

where the n-bit input integer x is split into t n_i-bit ($n=\sum_i n_i$) disjoint sub-values as $x=(x_1|x_2|\ldots|x_t)$. Let us also split indices k,λ in a similar way as $k=(k_1|k_2|\ldots|k_t)$ and $\lambda=(\lambda_1|\lambda_2|\ldots|\lambda_t)$. Then we have the following result

$$\mathcal{W}(B^{[k]}_{\{S(x)\}})_\lambda = \prod_{i=1}^{t}\mathcal{W}(B^{[k_i]}_{\{S_i(x)\}})_{\lambda_i}.$$

Proof. Since all x_i's are independent from each other, the combined bias at any point λ is the product of sub-biases at corresponding λ_i's for each k_i-th binary-valued function of the corresponding S-box $S_i(x)$, which can be proved by below.

$$\begin{aligned}\prod_{i=1}^{t}\mathcal{W}(B^{[k_i]}_{\{S_i(x)\}})_{\lambda_i} &= (\frac{1}{N_1}\sum_{x_1=0}^{N_1-1}(-1)^{k_1S_1(x_1)\oplus\lambda_1x_1})\cdot\ldots\cdot(\frac{1}{N_t}\sum_{x_t=0}^{N_t-1}(-1)^{k_tS_t(x_t)\oplus\lambda_tx_t})\\ &= \frac{1}{N_1N_2\ldots N_t}\sum_{x_1=0}^{N_1-1}\ldots\sum_{x_t=0}^{N_t-1}(-1)^{k_1S_1(x_1)\oplus\lambda_1x_1\oplus\ldots\oplus k_tS_t(x_t)\oplus\lambda_tx_t}\\ &= \frac{1}{N}\sum_{x=0}^{N-1}(-1)^{kS(x)\oplus\lambda x} = \mathcal{W}(B^{[k]}_{\{S(x)\}})_\lambda.\end{aligned}$$

□

Theorem 4 and Theorem 5 pave the way to compute the bias of any pair (k,λ) in Equation (11) efficiently in time $O(t)$, without even having to construct a large n-bit distribution of the S-box approximation (e.g., $X=RS(Qx)\oplus Mx$), given that $S(x)$ is constructed from smaller S-boxes, which is a common case in cipher designs. E.g., we can simply precompute the tables of $\{(k_i,\lambda_i,\epsilon)\}$ exhaustively for smaller S-boxes, then apply the theorems to compute the bias for a large composite S-box for any pair (k,λ).

For example, let $X = RS(Qx) \oplus Mx$ be the noise variable as the result of an approximation of a large n-bit composite S-box, $RS(Qx)$, where R and Q are some known $n \times n$ Boolean matrices, and Mx is the approximation of that large S-box with a selectable (or given) $n \times n$ full-rank Boolean matrix M. Then, if we want to get the value of some spectrum point k of X we do: compute $\lambda = k \cdot M$ (Theorem 3), then convert the indices as $k' = k \cdot R$ and $\lambda' = \lambda \cdot Q^{-1}$ (Theorem 4), and split them into t $n_1, \ldots, n_t$-bit integers as $k' = (k'_1, \ldots, k'_t)$ and $\lambda' = (\lambda'_1, \ldots, \lambda'_t)$ (Theorem 5). Then, the desired spectrum value at the index k is derived as

$$\mathcal{W}(X)_k = \prod_{i=1}^{t} \mathcal{W}(B^{[k'_i]}_{\{S_i(x)\}})_{\lambda'_i}.$$

Alongside, this also leads to an efficient and fast algorithm to search for the best set of triples $\{(k, \lambda, \epsilon)\}$ in Equation (11), by "reverting" the procedure. These findings have a direct application in the upcoming cryptanalysis of ZUC-256.

General approach of spectral cryptanalysis using WHT. With the tools and methods developed in this subsection, we can now propose a general framework for finding the best approximation, based on probing spectral indices.

1. Derive the total noise expression based on basic approximations and S-box approximations. The noise expression may involve $\oplus$ operations, Boolean matrices multiplications, where some of the matrices can be selected by the attacker.
2. Derive the expression for the k-th spectrum point of the total noise, using the formulae that we found earlier.
3. Convert expressions such as $k \cdot M$, where the matrix M is selectable, to be some parameter λ. If there are more selectable matrices then more λ's can be used.
4. Probe different tuples $(k, \lambda, \ldots)$ to find the maximum peak value in the spectrum for the total noise. The search space for k's and λ's may be shrunk by spectrum values of basic approximations.
5. Convert the best found tuple into the selected matrices, and compute the final multidimensional bias using the constructed matrices.

3.3 Algorithms for DFT type approximations

In this section we provide a few ideas on spectral analysis for DFT type convolutions. Although these methods were not used in the presented attack on ZUC-256, they can be quite helpful in linear cryptanalysis for some other ciphers.

Consider the expression

$$X = c_1 X^{(1)} \boxplus c_2 X^{(2)} \boxplus \ldots \boxplus c_t X^{(t)} \mod N, \tag{12}$$

where, again, $N = 2^n$ and the attacker can choose the constants c_i's, which must be odd, and $X^{(i)}$'s are independent random variables. We will propose the algorithm to find the best combination of the constants c_i's such that the total noise X will have the best peak spectrum value.

The theorem below would help to decide how to rearrange the spectrum points in the frequency domain to achieve a larger total bias, by multiplication with a constant in the time domain, which is a linear masking.

Theorem 6. For a given distribution of X and an odd constant c we have

$$\mathcal{F}(c \cdot X)_k = \mathcal{F}(X)_{k \cdot c \mod N},$$

for any spectrum index $k = 0, 1, \ldots, N-1$.

Proof. $\mathcal{F}(c{\cdot}X)_k = \sum_{n=0}^{N-1} x_{c^{-1}n} \cdot \left(e^{-i2\pi/N}\right)^{kn} = \sum_{n=0}^{N-1} x_n \cdot \left(e^{-i2\pi/N}\right)^{k \cdot c \cdot n} = \mathcal{F}(X)_{k \cdot c \mod N}$. □

Corollary 2. Any spectrum value at index $k = 2^m(1+2q)$, for some $m = 0 \ldots n-1, q = 0 \ldots 2^{n-m-1}-1$, can only be relocated to another index k' of the form $k' = 2^m(1+2q')$, for some $q' = 0 \ldots 2^{n-m-1}-1$.

Proof. The constant c is odd and $c = 1+2r$, for some r. If $\mathcal{F}(c \cdot X)_{k'} = \mathcal{F}(X)_k$, we get that $c \cdot k' \equiv k \mod N$, and then $k' \equiv 2^m(1+2q) \cdot (1+2r)^{-1} \mod N$. □

Corollary 3. Any spectrum value at index $k = 2^m(1+2q)$, for some $m = 0 \ldots n-1, q = 0 \ldots 2^{n-m-1}-1$, can be relocated to the index 2^m in the spectrum by applying the constant $c = 1+2q$.

Proof. $\mathcal{F}(c \cdot X)_{2^m} = \mathcal{F}((1+2q) \cdot X)_{2^m} = \mathcal{F}(X)_{2^m(1+2q)} = \mathcal{F}(X)_k$. □

The results above can be used to solve the problem of finding the constants c_i in Equation (12) such that the spectrum of X would contain the maximum possible peak value.

Algorithm 3 Find c_i's that maximize the peak spectrum point of X in Equation (12)

Input The distributions of $X^{(i)}$, for $i = 1, 2, \ldots, t$.
Output The coefficients c_i, for $i = 1, 2, \ldots, t$.

1: **procedure** DftConstantsAlign($X^{(1)}, X^{(2)}, \ldots, X^{(t)}$)
2: Initialize a $t \times n$ matrix Ψ with 0, each cell of which contains the pair (c, ω).
3: **for** $i = 1, \ldots, t$ **do**
4: Compute $W = \mathcal{F}(X^{(i)})$
5: **for** $m = 0, \ldots, n-1$ and $q = 0, \ldots, 2^{n-m-1}-1$ **do**
6: Set $\omega = |W_{2^m(1+2q)}|$
7: **if** $\omega \geq \Psi_{i,m}.\omega$ (i.e., the ω value of the entry in the i-th row and m-th column in Ψ) **then** set $\Psi_{i,m} = (1+2q, \omega)$
8: Set $m' = 0$ and $\omega' = 0$
9: **for** $m = 0, \ldots, n-1$ **do**
10: Compute $\omega = \prod_{i=1}^{t} \Psi_{i,m}.\omega$
11: **if** $\omega > \omega'$ **then** set $m' = m$ and $\omega' = \omega$
12: **for** $i = 1, \ldots, t$ **do**
13: Assign $c_i = \Psi_{i,m'}.c$ (i.e., the c value of the entry in the i-th row and m'-th column in Ψ)

The complexity of the above algorithm is $O(t \cdot N \log N)$.

4 Linear cryptanalysis of ZUC-256

In this section, we perform linear cryptanalysis on ZUC-256. Normally, the basic idea of linear cryptanalysis is to approximate nonlinear operations as linear ones and further to find some linear relationships between the generated keystream symbols or between keystream symbols and the LFSR state words, and thus respectively resulting into a distinguishing attack and correlation attack. In a distinguishing attack over a binary or an extension field over $GF(2)$, the common way is to find LFSR states at several time instances (usually 3, 4 or 5) which are XOR-ed to be zero such that the LFSR contribution in the linear approximation is canceled out while only noise terms remain which would be biased. This common way, however, does not apply well on ZUC, since the LFSR in ZUC is defined over a prime field $GF(2^{31}-1)$ which is different to the extension field $GF(2^{32})$ in the function F.

In this section, we describe a more general approach where the expression that we use to cancel out the LFSR contribution *is directly included in the full noise expression*, which effectively reduces the total noise, i.e., the final bias is larger. This general approach may be used in cryptanalysis of any other stream cipher where an LFSR is involved.

Below we first give our linear approximation of the full ZUC-256, including the LFSR state cancellation process. Then we describe in details how we employ the spectral tools given in Section 3 and a technique we call "bit-slicing" to efficiently compute the bias. Finally, we use the derived linear approximation to launch a distinguishing attack on ZUC-256.

4.1 Linear approximation

Any LFSR's 31-bit word $s_x^{(t)}$ at a time instance t and a cell index x can be expressed as $s_0^{(t+x)}$, for $0 \le t$ and $0 \le x \le 15$. Thus, in this section, we will omit the lower index and refer to an LFSR's word by using the time instance only, i.e., $s^{(t+x)}$. We then try to find a four-tuple of time instances t_1, t_2, t_3, t_4 such that,

$$s^{(t_1)} + s^{(t_2)} = s^{(t_3)} + s^{(t_4)} \mod p \qquad (\text{where } p = 2^{31}-1). \tag{13}$$

Note that for any time offset δ, Equation (13) also holds since the LFSR update is a linear transformation in the ring $1 + \mathbb{Z}_p$; i.e., if Equation (13) is true then the following is also true:

$$\forall \delta: \quad s^{(t_1+\delta)} + s^{(t_2+\delta)} = s^{(t_3+\delta)} + s^{(t_4+\delta)} \mod p.$$

At each time instance t_i, we define a 32-bit variable $X^{(t_i)}$ which is the concatenation of the low and high 16-bit parts of $s^{(t_i+a)}$ and $s^{(t_i+b)}$, for some constants $0 \le a, b \le 15$, $a \ne b$, following the description of the BR layer in ZUC-256, i.e., $X^{(t_i)}$ is one of $X0^{(t_i)} = (s_H^{(t_i+15)} || s_L^{(t_i+14)})$, $X1^{(t_i)} = (s_L^{(t_i+11)} || s_H^{(t_i+9)})$, $X2^{(t_i)} = (s_L^{(t_i+7)} || s_H^{(t_i+5)})$, or $X3^{(t_i)} = (s_L^{(t_i+2)} || s_H^{(t_i)})$. Then one can derive the following relation for $X^{(t_i)}$'s according

to Equation (13):

$$X^{(t_1)} \boxplus_{16} X^{(t_2)} = X^{(t_3)} \boxplus_{16} X^{(t_4)} \boxplus_{16} C^{(t_1)}, \tag{14}$$

where $\boxplus_{16}$ is the 16-bit arithmetic addition, i.e., addition modulo 2^{16}, of the low and high 16-bit halves of $X^{(t_i)}$'s in parallel. Here $C^{(t_1)} = (C_H^{(t_1)}||C_L^{(t_1)})$ is a 32-bit random carry variable from the approximation of the modulo p, and it can only take the values $C_L^{(t_1)}, C_H^{(t_1)} \in \{0, -1, +1\} \mod 2^{16}$, where the values in the low and high parts of $C^{(t_1)}$ are independent. As an example, the approximation in Equation (14) for $X^{(t_i)} = X1^{(t_i)}$ is then written as:

$$\underbrace{\begin{pmatrix} s_H^{(t_1+9)} \\ s_L^{(t_1+11)} \end{pmatrix}}_{X1^{(t_1)}} \boxplus_{16} \underbrace{\begin{pmatrix} s_H^{(t_2+9)} \\ s_L^{(t_2+11)} \end{pmatrix}}_{X1^{(t_2)}} = \underbrace{\begin{pmatrix} s_H^{(t_3+9)} \\ s_L^{(t_3+11)} \end{pmatrix}}_{X1^{(t_3)}} \boxplus_{16} \underbrace{\begin{pmatrix} s_H^{(t_4+9)} \\ s_L^{(t_4+11)} \end{pmatrix}}_{X1^{(t_4)}} \boxplus_{16} \underbrace{\begin{pmatrix} C1_L^{(t_1)} \\ C1_H^{(t_1)} \end{pmatrix}}_{C1^{(t_1)}}.$$

Next, we would like to derive the distribution of the carries $C_L^{(t_1)}, C_H^{(t_1)}$, and to achieve that, we first give a theorem.

Theorem 7. Let a modulus p be of the form $p = 2^n - 1$, for an integer $n > 1$. Assume $a_1, a_2, a_3, a_4 \in 1 + \mathbb{Z}_p$, such that $a_1 + a_2 = a_3 + a_4 \mod p$. For integers s and t with $0 \leq s < n$ and $1 \leq t \leq (n - s)$, we extract "middle" t-bit values with the bit-offset s as $A_i^{(s)} = \lfloor a_i/2^s \rfloor \mod 2^t$, for $i = 1, 2, 3, 4$. Then we can get the following approximation

$$A_1^{(s)} \boxplus_t A_2^{(s)} = A_3^{(s)} \boxplus_t A_4^{(s)} \boxplus_t Q^{(s)} \mod 2^t, \tag{15}$$

where the carry value $Q^{(s)} \in \{0, -1, +1\}$ [1] and it has the distribution $Pr\{Q^{(s)} = 0\} = (2p^2 + 1)/3p^2$, and $Pr\{Q^{(s)} = -1\} = Pr\{Q^{(s)} = +1\} = (p^2 - 1)/6p^2$.

Proof. The proof is given in Appendix 1. □

Corollary 4. The distribution for $C^{(t_1)}$(note here t_1 denotes the time instance) in Equation (14) is as follows:

$$\begin{aligned} Pr\{C_L^{(t_1)} = 0\} = Pr\{C_H^{(t_1)} = 0\} &\approx 2/3, \\ Pr\{C_L^{(t_1)} = -1\} = Pr\{C_H^{(t_1)} = -1\} &\approx 1/6, \\ Pr\{C_L^{(t_1)} = +1\} = Pr\{C_H^{(t_1)} = +1\} &\approx 1/6. \end{aligned}$$

Proof. Given the relation in Equation (14), we basically need to consider these two 16-bit cases independently (since $a \neq b$): $s_L^{(t_1+a)} \boxplus_{16} s_L^{(t_2+a)} = s_L^{(t_3+a)} \boxplus_{16} s_L^{(t_4+a)} \boxplus_{16} E_L^{(t_1+a)}$ and $s_H^{(t_1+b)} \boxplus_{16} s_H^{(t_2+b)} = s_H^{(t_3+b)} \boxplus_{16} s_H^{(t_4+b)} \boxplus_{16} E_H^{(t_1+b)}$, for some constants $0 \leq a, b \leq 15, a \neq b$, where the carry $C^{(t_1)}$ is either $(E_L^{(t_1+a)}||E_H^{(t_1+b)})$ or $(E_H^{(t_1+b)}||E_L^{(t_1+a)})$.

The distributions of $E_L^{(t_1+a)}$ and $E_H^{(t_1+b)}$ can be respectively proved through Theorem 7 by setting $n = 31, s = 0, t = 16$ and $n = 31, s = 15, t = 16$. The probability values can be approximated as $2/3$ and $1/6$ with an error $< 2^{-63}$. □

[1] In a special case when $t = 1$ bit the values of $Q^{(s)}$ are $\{0, 1\}$, since $(-1) \equiv 1 \mod 2$; the probabilities of $Pr\{Q^{(s)} = -1\}$ and $Pr\{Q^{(s)} = +1\}$ are then combined into a single case when $Q^{(s)} = 1$.

Next, we list the expressions for generating keystream symbols at time instances t and $t+1$ as follows,

$$Z^{(t)} = [(T2^{(t)} \oplus X2^{(t)}) \boxplus ((T1^{(t)} \boxminus X1^{(t)}) \oplus X0^{(t)})] \oplus X3^{(t)},$$
$$Z^{(t+1)} = [SL_2(T2'^{(t)}) \boxplus (SL_1(T1'^{(t)}) \oplus X0^{(t+1)})] \oplus X3^{(t+1)},$$

where $\boxminus$ is the arithmetic subtraction modulo 2^{32} and $(T1', T2') = (T1, T2) \lll 16$ is a cyclic shift 16 bits to the left. Then we give the full approximation of ZUC-256 based on Equation (13) and its approximation in Equation (14) as follows:

$$\begin{aligned}
&M\sigma[Z^{(t_1)} \oplus Z^{(t_2)} \oplus Z^{(t_3)} \oplus Z^{(t_4)}] \oplus [Z^{(t_1+1)} \oplus Z^{(t_2+1)} \oplus Z^{(t_3+1)} \oplus Z^{(t_4+1)}] \\
&\quad = M\sigma N1^{(t_1)} \oplus \bigoplus_{t\in\{t_1,\ldots,t_4\}} M(\underbrace{\sigma T1^{(t)} \oplus \sigma T2^{(t)}}_{=T1'^{(t)} \oplus T2'^{(t)}}) \oplus N2^{(t_1)} \\
&\quad \oplus \bigoplus_{t\in\{t_1,\ldots,t_4\}} (SL_1(T1'^{(t)}) \oplus SL_2(T2'^{(t)})) \\
&\quad = M\sigma N1^{(t_1)} \oplus N2^{(t_1)} \\
&\quad \oplus \bigoplus_{t\in\{t_1,\ldots,t_4\}} \left[M \cdot T1'^{(t)} \oplus SL_1(T1'^{(t)}) \oplus M \cdot T2'^{(t)} \oplus SL_2(T2'^{(t)})\right],
\end{aligned}$$

where σ is the swap of the high and low 16 bits of a 32-bit argument, and M is some 32×32 full-rank Boolean matrix that we can choose, which serves as a linear masking matrix. The expressions for the noise $N1^{(t_1)}$ (we further split $N1^{(t_1)} = N1a^{(t_1)} \oplus N1b^{(t_1)}$) and noise $N2^{(t_1)}$ are as follows:

$$\begin{aligned}
N1a^{(t_1)} &= [((T2^{(t_1)} \oplus X2^{(t_1)}) \boxplus ((T1^{(t_1)} \boxminus X1^{(t_1)}) \oplus X0^{(t_1)}))] \\
&\oplus [((T2^{(t_2)} \oplus X2^{(t_2)}) \boxplus ((T1^{(t_2)} \boxminus X1^{(t_2)}) \oplus X0^{(t_2)}))] \\
&\oplus [((T2^{(t_3)} \oplus X2^{(t_3)}) \boxplus ((T1^{(t_3)} \boxminus X1^{(t_3)}) \oplus X0^{(t_3)}))] \\
&\oplus [((T2^{(t_4)} \oplus (X2^{(t_1)} \boxplus_{16} X2^{(t_2)} \boxminus_{16} X2^{(t_3)} \boxminus_{16} C2^{(t_1)})) \boxplus ((T1^{(t_4)} \\
&\boxminus (X1^{(t_1)} \boxplus_{16} X1^{(t_2)} \boxminus_{16} X1^{(t_3)} \boxminus_{16} C1^{(t_1)})) \\
&\oplus (X0^{(t_1)} \boxplus_{16} X0^{(t_2)} \boxminus_{16} X0^{(t_3)} \boxminus_{16} C0^{(t_1)}))))] \oplus \bigoplus_{t\in\{t_1,\ldots,t_4\}} (T1^{(t)} \oplus T2^{(t)}),
\end{aligned} \tag{16}$$

$$N1b^{(t_1)} = X3^{(t_1)} \oplus X3^{(t_2)} \oplus X3^{(t_3)} \oplus (X3^{(t_1)} \boxplus_{16} X3^{(t_2)} \boxminus_{16} X3^{(t_3)} \boxminus_{16} C3^{(t_1)}),$$

and

$$\begin{aligned}
N2^{(t_1)} &= [[(SL_2(T2'^{(t_1)}) \boxplus (SL_1(T1'^{(t_1)}) \oplus X0^{(t_1+1)})) \oplus X3^{(t_1+1)}] \\
&\oplus [(SL_2(T2'^{(t_2)}) \boxplus (SL_1(T1'^{(t_2)}) \oplus X0^{(t_2+1)})) \oplus X3^{(t_2+1)}] \\
&\oplus [(SL_2(T2'^{(t_3)}) \boxplus (SL_1(T1'^{(t_3)}) \oplus X0^{(t_3+1)})) \oplus X3^{(t_3+1)}] \\
&\oplus [(SL_2(T2'^{(t_4)}) \boxplus (SL_1(T1'^{(t_4)}) \oplus (X0^{(t_1+1)} \boxplus_{16} X0^{(t_2+1)} \\
&\boxminus_{16} X0^{(t_3+1)} \boxminus_{16} C0^{(t_1+1)}))) \oplus (X3^{(t_1+1)} \boxplus_{16} X3^{(t_2+1)} \boxminus_{16} X3^{(t_3+1)}
\end{aligned}$$

$$\boxminus_{16} C3^{(t_1+1)})]] \oplus \bigoplus_{t\in\{t_1,\ldots,t_4\}} (SL_1(T1'^{(t)}) \oplus SL_2(T2'^{(t)})).$$

In our analysis we consider noise variables $N1^{(t_1)}$ and $N2^{(t_1)}$ as independent. By this assumption the attacker actually looses some advantage since there is a dependency between, for example, $T1^{(t_1)}, T2^{(t_1)}$ in $N1^{(t_1)}$ and $SL_1(T1'^{(t_1)}), SL_2(T2'^{(t_1)})$ in $N2^{(t_1)}$. The attack can be stronger if we could take into account these dependencies, since then there will be more information in these noise distributions. However, it is practically hard to compute the bias in that scenario.

Next we want to compute the distribution and the bias of the noise terms. However, as one can note, there are many variables involved in each sub-noise expression. For example, the sub-noise $N1a^{(t_1)}$ involves 17 32-bit variables, and 3 C-carries. In order to compute the distribution of $N1a^{(t_1)}$, a naive loop over all combinations of the involved variables would imply the complexity $O(9^3 \cdot 2^{17\cdot 32})$, which is computationally infeasible.

In the next subsections we make a recap of the bit-slicing technique and show how we adapt it to our case to compute the distributions of the above noise terms.

4.2 Recap on the bit-slicing technique from [MJ05]

Let an n-bit noise variable N be expressed in terms of several n-bit uniformly distributed independent variables, using any combination of bitwise Boolean functions (AND, OR, XOR, etc.) and arithmetical addition $\boxplus$ and subtraction $\boxminus$ modulo 2^n. The distribution of such a noise expression, referred to as a *pseudo-linear function* in [MJ05], can be efficiently derived through the so-called "bit-slicing" technique in complexity $O(k \cdot 2^n + k^2 n \cdot 2^{n/2})$, for some (usually small) k.

The general idea behind the technique is that if we know the set of distributions for $(n-1)$-bit truncated inputs for each possible outcome vector of the sub-carries' values for corresponding arithmetical sub-expressions, then we can easily extend these distributions to the n-bit truncated distributions with a new vector of output sub-carries' values. Then, the algorithm may be viewed as a Markov chain where the nodes are viewed as a vector of probabilities for each combination of sub-carries, and some *transition matrices* are used to go from the $(n-1)$-th state to the n-th state.

Example. Let us explain the technique on a small example. Let $n = 32$ bits and the noise N is expressed in terms of random 32-bit variables A, B, C:

$$N = \underbrace{[\underbrace{(A \boxplus B \boxminus C)}_{\text{inner ADD-1}} \oplus \underbrace{(A \boxplus C)}_{\text{inner ADD-2}}] \boxminus B}_{\text{outer ADD-3}}. \tag{17}$$

For each n-bit value X with $(x_{n-1} \ldots x_1 x_0)$ as its binary form, we will compute the number of combinations of A, B, C such that the value of N is equal to X.

Carries and the state. Here we have 3 arithmetical parts: two inner and one outer. We express the carries using a vector denoted $(c1, c2, c3)$, where $c1 \in \{-1, 0, +1\}, c2 \in \{0, 1\}, c3 \in \{-1, 0\}$. At each bit position $i, 0 \le i \le n-1$, we would have the input carry vector coming from the first $i-1$ bits, $(c1_{in}, c2_{in}, c3_{in})$, and the output carry vector $(c1_{out}, c2_{out}, c3_{out})$ going to the $(i+1)$-th bit position. Introduce a mapping function τ as: $\tau(c1, c2, c3) = ((c1+1) \cdot 2 + c2) \cdot 2 + (c3+1) \in [0 \ldots 11]$, that maps each value of the

carry vector to a unique integer index. Thus, at every step i, we will have a column vector V_i of length $k = 12$, each entry of which corresponds to a certain combination of output carries $(c1_{out}, c2_{out}, c3_{out})$, and the values are respectively the number of combinations of the i-bit truncated variables A, B, C such that the first i bits of N are equal to the first i bits of X. The initial vector is $V_0 = (0, \ldots 0, 1($ in the index $\tau(c1 = 0, c2 = 0, c3 = 0)), 0, \ldots, 0)^{\mathrm{T}}$.

Transition matrices. We will construct two $k \times k$ (12×12) transition matrices, M_0 and M_1, associated with every bit position i for the i-th bit value of X, x_i, being either 0 or 1, such that the vector V_{i+1} is derived by $V_{i+1} = M_{x_i} \cdot V_i$. I.e., when the i-th bit of X, x_i, is 0, we apply M_0, otherwise M_1. These two matrices are constructed as follows: initialize M_0 and M_1 with zeroes; loop through all possible choices of the i-th bits of $A, B, C \in \{0, 1\}^3$ and all possible values of $(c1_{in}, c2_{in}, c3_{in})$; then for each combination we compute the resulting bit $r \in \{0, 1\}$ by evaluating the noise expression, and the vector of output carries $(c1_{out}, c2_{out}, c3_{out})$; we then increase the corresponding matrix cell by 1 as $++M_r[\tau(c1_{out}, c2_{out}, c3_{out})][\tau(c1_{in}, c2_{in}, c3_{in})]$ at the same time. Note, the inner output carries $c1_{out}$ and $c2_{out}$ should not be summed up in the outer output carry $c3_{out}$, while only the resulting 1-bit values of inner sums should go to the outer expression.

In Appendix 2 we give the code in C for computing the transition matrices M_0 and M_1 for the exampled noise expression given in Equation (17).

The general formulae can now be derived as follows:

$$\Pr\{N = (x_{n-1} \ldots x_0)\} = \frac{1}{2^{t \cdot n}} \cdot \underbrace{(1, 1, \ldots, 1) \cdot \prod_{i=n/2}^{n-1} M_{x_i}}_{\text{High part, } H[(x_{n-1} \ldots x_{n/2})]} \cdot \underbrace{\prod_{i=0}^{n/2-1} M_{x_i} \cdot V_0}_{\text{Low part, } L[(x_{n/2-1} \ldots x_0)]} \quad , \qquad (18)$$

where t is the number of involved random variables, in our example $t = 3$, and $1/2^{t \cdot n}$ is the normalization factor for the distribution. The left-side row vector $(1, 1, \ldots, 1)$ due to the last carries are truncated by the modulo 2^n operation and thus all combinations for all carries' outcomes should be summed up to the result.

Precomputed vectors. We intentionally split Equation (18) into two parts, since it shows that the computation of $Pr\{N = X\}$ for all values of $X \in \{0, 1, \ldots, 2^n - 1\}$ can be accelerated by precomputing two tables of the middle sub-vectors in Equation (18) for all possible values of the high $(H[(x_{n-1} \ldots x_{n/2})])$ and low $(L[(x_{n/2-1} \ldots x_0)])$ halves of X, independently. The whole precomputation takes time $O(n \cdot 2^{n/2} \cdot k^2)$. Then the probability $Pr\{N = X\}$ is a simple scalar product, computed in time $O(k)$ as:

$$\Pr\{N = (x_{n-1} \ldots x_0)\} = \frac{1}{2^{t \cdot n}} \cdot H[(x_{n-1} \ldots x_{n/2})] \cdot L[(x_{n/2-1} \ldots x_0)].$$

4.3 Bit-slicing technique adaptation to compute $N1a$, $N1b$ and $N2$

In this section we will describe in more details how we adapt the bit-slicing technique in order to compute the "heaviest" noise $N1a$. The remaining noises are computationally less demanding and can be derived with similar adaptation techniques.

A direct application of the bit-slicing technique to compute $N1a$, given in Equation (16), is complicated due to: (1) we have $\boxplus_{16}$ adders that block some of the sub-carries to propagate between the 15-th and 16-th bits; and (2) we have random C-carries that can have 3 values at the 0-th and 16-th bits.

The first problem is resolved by introducing two special transition matrices $M_r^{(15)}$ (for $r = 0, 1$) which are only applied to the bit 15. In these matrices, output sub-carries in all involved $\boxplus_{16}$ would not propagate to the next bit 16 and are forced to be 0.

The second problem is solved by introducing another two special transition matrices $M_r^{(0)}$ (for $r = 0, 1$) that are only applied to the bits 0 *and* 16. These special matrices take into account C-values that are added to the 0-th and 16-th bits. The important fact here is that all input sub-carries at bit positions where C's are involved are always 0, and this makes it possible to keep the sub-carry values in the expressions like $(X3^{(t_1)} \boxplus_{16} X3^{(t_2)} \boxminus_{16} X3^{(t_3)} \boxminus_{16} C3^{(t_1)})$ in the smaller range $\{-1, 0, +1\}$, since a C-value $\in \{-1, 0, +1\}$ only appears at the first bit under the 16-bit addition/subtraction where input carries are zeroes, and in the next bits $C = 0$. Thus, to construct $M_r^{(0)}$'s, we do the following: loop through the 1-bit random variables involved in the noise expression; loop through sub-carries that propagate over 32 bits; *do not* loop through the carries that are involved in 16-bit propagations; and loop through $C \in \{-1, 0, +1\}$ values. Then, instead of increasing the corresponding entry of $M_r^{(0)}$ by 1, we actually add the product of the probabilities of all involved C-values.

Transition matrices for the remaining bits (except the bits 0, 15, 16) are constructed as usual, but C-values are all 0.

Additional adaptation is done in the part of L/H precomputed tables of vectors. We know that the low and high precomputations meet in the middle at the bits 15 and 16, where all sub-carries in $\boxplus_{16}$ adders vanish to 0. This makes it possible to shrink the size of the vectors and only leave the states with sub-carries that propagate over all 32-bits of the noise expression.

Complexity. For $N1a$ we have the following situation: we have 17 32-bit variables ($T1$ and $T2$ in 4 time instances and $X0, X1, X2$ in 3 time instances); 8 carries that propagate over 32 bits having binary values either $\{0, +1\}$ or $\{-1, 0\}$; 3 carries that propagate over 16 bits in the range $\{-1, 0, +1\}$. Thus we get the dimension of all involved transition matrices by $k = 2^8 \cdot 3^3$, i.e., the matrices are of size $2^{12.8} \times 2^{12.8}$. If each entry of a matrix is of C-type `double` (8 bytes), then one transition matrix occupies around 365Mb of RAM.

The precomputation phase to compute low (L) and high (H) tables of vectors has time complexity around $O(2^{2 \cdot 12.8} \cdot 32 \cdot 2^{16}) = O(2^{46.6})$. The size of the stored L/H vectors was dramatically reduced from the vector lengths $k = 2^{12.8}$ down to the lengths $k' = 2^8$, since there are only 8 binary-valued sub-carries that propagate between the bits 15 and 16, while other carries were "truncated" by applying the matrix $M_r^{(15)}$.

The total time complexity to construct the noise $N1a$ is, therefore, $O(2^{46.6} + 2^{32} \cdot 2^8)$. We compute $N1b$ and $N2$ with a similar adaptation of the bit-slicing technique, but the time complexity there is a lot smaller.

4.4 Spectral analysis to find the matrix M

With the methods presented in Section 3, the spectral analysis becomes rather simple for the ZUC-256 case, and we below give necessary expressions to perform that. Let us recall that the expression for the total noise is:

$$N_{tot}^{(t_1)} = M\sigma N1^{(t_1)} \oplus N2^{(t_1)} \\ \oplus \bigoplus_{t\in\{t_1,\ldots,t_4\}} \left[SL_1(T1'^{(t)}) \oplus M \cdot T1'^{(t)} \oplus SL_2(T2'^{(t)}) \oplus M \cdot T2'^{(t)}\right].$$

The spectrum expression at some point k can thus be derived as follows.

$$\begin{aligned}\mathcal{W}(N_{tot}^{(t_1)})_k &= \mathcal{W}(M\sigma N1)_k \cdot \mathcal{W}(N2)_k \cdot \mathcal{W}(SL_1(x) \oplus Mx)_k^4 \cdot \mathcal{W}(SL_2(x) \oplus Mx)_k^4 \\ &= \mathcal{W}(\sigma N1)_\lambda \cdot \mathcal{W}(N2)_k \cdot \mathcal{W}(B^{[k]}_{\{SL_1(x)\}})_\lambda^4 \cdot \mathcal{W}(B^{[k]}_{\{SL_2(x)\}})_\lambda^4,\end{aligned}$$

where $\lambda = k \cdot M$.

Our strategy for the spectral analysis was as follows. We selected $\approx 2^{24.78}$ "promising" spectrum points for λ where $|\mathcal{W}(\sigma N1)_\lambda|^2 > 2^{-150}$, and also selected $\approx 2^{18}$ "promising" spectrum points for k where $|\mathcal{W}(N2)_k|^2 > 2^{-80}$. Then we tried all combinations of the selected (k, λ) and computed the total spectrum value. For the computation of spectrum points for S-boxes (e.g., for $\mathcal{W}(B^{[k]}_{\{SL_1(x)\}})_\lambda^4$), we utilized Theorem 4 and Theorem 5, so that we did not have to construct the full 32-bit distributions of the S-box approximations, but exploring a spectrum point in time $O(1)$. The total complexity of the analysis is $\approx O(2^{41})$.

We then collected the best pairs $\{(k, \lambda)\}$ in terms of the largest peak spectrum values, and constructed two full-rank 32×32 Boolean matrices K and Λ from the indices (k, λ) with the greedy approach given in Algorithm 2. Then the matrix M was derived as $M = K^{-1} \cdot \Lambda$.

Results. We only used seven pairs from the result of the spectrum analysis (since many other pairs did not give us full-rank matrices K and Λ), and the remaining 25 rows of K, Λ were randomly generated. Thereafter, we tested that matrix M in the full approximation and received the total bias:

$$\epsilon(N_{tot}^{(t_1)}) \approx 2^{-236.380623}.$$

The 32×32 binary matrix M is given below as a vector of 32-bit integers, where the bit $M_{i,j}$, for $0 \le i, j \le 31$, is extracted as $M_{i,j} = \lfloor \texttt{M[}i\texttt{]}/2^j \rfloor \mod 2$, and in standard C it is then $M_{i,j} = \texttt{(M[i]>>j)\&1}$.

4.5 A distinguishing attack on ZUC-256

In a distinguishing attack, an adversary aims to find some linear relationships between the generated keystream symbols by canceling the LFSR contribution in the linear approximation, thus being able to distinguish the keystream sequence from random.

In Section 4.1, we have shown that if we could find four time instances t_1, t_2, t_3, t_4 such that $s^{(t_1)} + s^{(t_2)} = s^{(t_3)} + s^{(t_4)} \mod p$, we can build the keystream samples $M\sigma[Z^{(t_1)} \oplus Z^{(t_2)} \oplus Z^{(t_3)} \oplus Z^{(t_4)}] \oplus [Z^{(t_1+1)} \oplus Z^{(t_3+1)} \oplus Z^{(t_3+1)} \oplus Z^{(t_4+1)}]$ to be biased with a bias of

$2^{-236.38}$. By collecting around $O(2^{236.38})$ such samples, we could distinguish this sample sequence from random, thus resulting in a distinguishing attack.

The remaining problem is how we can find such a time instance tuple t_1, t_2, t_3, t_4 satisfying the requirement, i.e., $s^{(t_1)} + s^{(t_2)} = s^{(t_3)} + s^{(t_4)} \mod p$. This problem is equivalent to finding a weight 4 multiple of the feedback polynomial, for which there already exist a number of research results [LJ14][YJM19]. We use the algorithm in [LJ14] to solve this problem. But here we should find a weight 4 multiple with two coefficients being 1 and the other two being -1. Let us first figure out how far we should run the cipher, i.e., the degree of the multiple, to find such a tuple. Let q be the expected degree. If we consider all $t \leq q$, we could create $\binom{q}{3}$ different combinations of $s^{(t_1)} + s^{(t_2)} - s^{(t_3)} - s^{(t_4)}$ when we fix one time instance. Since there are 2^{496} possible such combinations, we can expect that we need to go to the length such that $\binom{q}{3} \approx 2^{496}$, resulting in $q \approx 2^{167}$. We use the algorithm in [LJ14] to find the time instance tuple, but note that at the last step, instead of keeping $x^{i_1} + x^{i_2} + x^{i_3} + x^{i_4} = 0 \mod P(x)$, we should keep $x^{i_1} + x^{i_2} - x^{i_3} - x^{i_4} = 0 \mod P(x)$. The algorithm requires computational complexity of q and similar storage. For our case, the complexity is 2^{167}. The algorithm to find the time instance tuple can be found in Appendix 3.

Thus we succeed to have a distinguishing attack on ZUC-256, for which we need to run the cipher around 2^{236} iterations and collect 2^{236} samples.

5 Conclusions

In this paper, we give a number of spectral tools for linear cryptanalysis and further apply them to ZUC-256 resulting in a distinguishing attack on ZUC-256 faster than exhaustive key search.

We explored how a linear masking in the time domain would affect the spectrum points in the frequency domain under some commonly used operations in cryptography, such as $\boxplus, \oplus$, and S-boxes, in both WHT and DFT types. We also gave a number of results and algorithms about how to find a good linear masking in the time domain by aligning the spectrum points in the frequency domain.

For the distinguishing attack, we first derive a linear approximation of the non-linear part F and the transformation from $GF(p)$ field in LFSR to $GF(2^{32})$ in F. We then employ the spectral tools to find good linear maskings and adapt the bit-slicing technique to efficiently compute the bias of the approximation. The linear approximation is then used to launch a distinguishing attack by finding a weight 4 multiple of the generating polynomial to cancel the contribution from the LFSR. The complexity of the distinguishing attack is $O(2^{236})$. It indicates that ZUC-256 does not provide a source with full 256-bit entropy in the generated keystream, as would be expected from a 256-bit key.

Acknowledgements

We would like to thank all reviewers for providing valuable comments to the manuscript. This work was in part financially supported by the Swedish Foundation for Strategic Research, grant RIT17-0005. The author Jing Yang is also supported by the scholarship

from the National Digital Switching System Engineering and Technological Research Center, China.

References

[3GP18] 3GPP TSG-SA. Study on the support of 256-bit algorithms for 5G (release 16), November 2018. `https://www.3gpp.org/ftp/tsg_sa/WG3_Security/TSGS3_93_Spokane/Docs`.

[BJV04] Thomas Baigneres, Pascal Junod, and Serge Vaudenay. How far can we go beyond linear cryptanalysis? In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 432–450. Springer, 2004.

[CT12] Thomas M Cover and Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[ETS11a] ETSI/SAGE. Specification of the 3GPP confidentiality and integrity algorithms 128-EEA3 & 128-EIA3. document 2: ZUC specification, 2011.

[ETS11b] ETSI/SAGE. Specification of the 3GPP confidentiality and integrity algorithms 128-EEA3 & 128-EIA3. document 4: Design and evaluation report, 2011.

[GDL13] Jie Guan, Lin Ding, and Shu-Kai Liu. Guess and determine attack on SNOW 3G and ZUC. *Journal of Software*, 6:1324–1333, 2013.

[HCN19] Miia Hermelin, Joo Yeon Cho, and Kaisa Nyberg. Multidimensional linear cryptanalysis. *Journal of Cryptology*, 32(1):1–34, 2019.

[HG97] Helena Handschuh and Henri Gilbert. χ^2 cryptanalysis of the SEAL encryption algorithm. In *International Workshop on Fast Software Encryption*, pages 1–12. Springer, 1997.

[HN12] Miia Hermelin and Kaisa Nyberg. Multidimensional linear distinguishing attacks and Boolean functions. *Cryptography and Communications*, 4(1):47–64, 2012.

[LD16] Yi Lu and Yvo Desmedt. Walsh transforms and cryptographic applications in bias computing. *Cryptography and Communications*, 8(3):435–453, 2016.

[LJ14] Carl Löndahl and Thomas Johansson. Improved algorithms for finding low-weight polynomial multiples in $F_2[x]$ and some cryptographic applications. *Designs, codes and cryptography*, 73(2):625–640, 2014.

[LMVH15] Frédéric Lafitte, Olivier Markowitch, and Dirk Van Heule. SAT based analysis of LTE stream cipher ZUC. *Journal of Information Security and Applications*, 22:54–65, 2015.

[MJ05] Alexander Maximov and Thomas Johansson. Fast computation of large distributions and its cryptographic applications. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 313–332. Springer, 2005.

[NH07] Kaisa Nyberg and Miia Hermelin. Multidimensional Walsh transform and a characterization of bent functions. In *2007 IEEE Information Theory Workshop on Information Theory for Wireless Networks*, pages 1–4. IEEE, 2007.

[STL10] B Sun, XH Tang, and C Li. Preliminary cryptanalysis results of ZUC. In *Proc. of the Record of the 1st International Workshop on ZUC Algorithm*, 2010.

[Tea18] ZUC Design Team. The ZUC-256 Stream Cipher, 2018. `http://www.is.cas.cn/ztzl2016/zouchongzhi/201801/W020180126529970733243.pdf`.

[WHN+12] Hongjun Wu, Tao Huang, Phuong Ha Nguyen, Huaxiong Wang, and San Ling. Differential attacks against stream cipher ZUC. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 262–277. Springer, 2012.

[YJM19] Jing Yang, Thomas Johansson, and Alexander Maximov. Vectorized linear approximations for attacks on SNOW 3G. In *27th Annual Fast Software Encryption Conference, FSE 2020*, 2019.

[ZFL11] Chunfang Zhou, Xiutao Feng, and Dongdai Lin. The initialization stage analysis of ZUC v1. 5. In *International Conference on Cryptology and Network Security*, pages 40–53. Springer, 2011.

Appendices

1 The proof of Theorem 7

Case when $s = 0$**.** Given a_1, a_2 and a_3, the value of a_4 would be fixed. Thus, the total number of all possible combinations of a_i's is p^3.

When $a_1 + a_2 = a_3 + a_4 = k$ for $2 \leq k \leq 2p$, the carry value $Q^{(0)} = 0$ for any t. One can get that there are $(k-1)^2$ solutions for the combinations of a_i's when $2 \leq k \leq p+1$, and $(2p+1-k)^2$ solutions when $p+2 \leq k \leq 2p$. Then the probability $Pr\{Q^{(0)} = 0\}$ in this case is calculated as $(1+2^2+...(p-1)^2+p^2+(p-1)^2+...+2^2+1)/p^3 = (2p^2+1)/3p^2$.

Similarly, we can get that when $a_1 + a_2 = a_3 + a_4 + p$ or $a_1 + a_2 + p = a_3 + a_4$, which are two equally likely events, the carry values of $Q^{(0)}$ would respectively be $\pm p \mod 2^t = \pm(2^n - 1) \mod 2^t = \pm 1 \mod 2^t$, both with equal probability $(1 - (2p^2 + 1)/3p^2)/2 = (p^2 - 1)/6p^2$.

Case when $s \neq 0$**.** Let us define the set $S^{(0)} = \{(a_1, a_2, a_3, a_4) : a_1 + a_2 = a_3 + a_4 \mod p\}$, which corresponds to all p^3 valid combinations of a_i's when $s = 0$, and $S^{(s)} = \{(2^s a_1, 2^s a_2, 2^s a_3, 2^s a_4) : (a_1, a_2, a_3, a_4) \in S^{(0)}\}$ for the case when $s \neq 0$. Clearly, $|S^{(s)}| = |S^{(0)}|$ and each tuple from $S^{(s)}$ also satisfies $2^s a_1 + 2^s a_2 = 2^s a_3 + 2^s a_4 \mod p$, thus, every tuple of $S^{(s)}$ must also be an element of $S^{(0)}$. The mapping $a_i \to 2^s a_i$ is injective since 2^s is invertible modulo p ($2^s \cdot 2^{n-s} = 1 \mod p$). Therefore, we get that $S^{(0)} \to S^{(s)}$ is an injective mapping and the two sets are equal to each other.

Let us pick any tuple $(a_1, a_2, a_3, a_4) \in S^{(0)}$ assuming the case when $s = 0$, then we extract the lower t-bit values of a_i as $A_i^{(0)}$. The corresponding carry value is then derived as $Q^{(0)} = (A_1^{(0)} \boxplus_t A_2^{(0)}) \boxminus_t (A_3^{(0)} \boxplus_t A_4^{(0)}) \mod 2^t$.

Now we observe that with the selected modulus $p = 2^n - 1$, the multiplication $2 \cdot x \mod p$ is just a circular rotation by 1 bit of x to the left. Thus, in the corresponding mapped tuple $(2^s a_1, 2^s a_2, 2^s a_3, 2^s a_4) \in S^{(s)}$ each a_i value is just circularly rotated by s bits to the left. Then the extracted "middle" t-bit values of $2^s a_i$'s, here denoted as $A_i^{'(s)}$'s, are consistent with $A_i^{(0)}$'s. As a consequence, the resulting carry value will also match, $Q'^{(s)} = Q^{(0)}$. I.e., the mapping $S^{(0)} \to S^{(s)}$ is not only injective but also preserves all other properties including the carry values, therefore, the space of carry values and their probabilities for $s \neq 0$ are the same as for the case when $s = 0$.

2 Computation of transition matrices for the exampled noise expression given in Equation (17)

```
// M[0] and M[1] for approximation: N = ((A + B - C) ^ (A + C)) - B
#define tau(c1, c2, c3) (((c3 + 1) * 2 + c2) * 3 + c1 + 1)
long long M[2][12][12];
memset(M, 0, sizeof M);

for(int a=0; a<=1; ++a)
for(int b=0; b<=1; ++b)
for(int c=0; c<=1; ++c)
for(int c1in=-1; c1in<=1; ++c1in)
for(int c2in= 0; c2in<=1; ++c2in)
for(int c3in=-1; c3in<=0; ++c3in)
{
        // process subexpression-1 (inner)
        int expr1   = a + b - c + c1in;
        int result1 = expr1 & 1,  c1out = expr1 >> 1;

        // process subexpression-2 (inner)
        int expr2   = a + c + c2in;
        int result2 = expr2 & 1,  c2out = expr2 >> 1;

        // process subexpression-3 (outer)
        // (!) note that c1out and c2out are not included
        int expr3   = (result1 ^ result2) - b + c3in;
        int result3 = expr3 & 1,  c3out = expr3 >> 1;

        // mapping of in/out carries into indices in the range [0..11]
        int in  = tau(c1in , c2in , c3in );
        int out = tau(c1out, c2out, c3out);

        // add 1 combination to the corresponding in/out carries
        M[result3][out][in] += 1;
}
```

3 The algorithm to find a multiple of $P(x)$

Algorithm 4 Finding a multiple of $P(x)$ with weight 4 and two nonzero coefficients being 1 and the other two being -1

Input Polynomial $P(x)$, a small integer b

Output A polynomial multiple $K(x) = P(x)Q(x)$ of weight 4 and expected degree 2^d with two of the nonzero coefficients being 1 and the other two being -1

1. From $P(x)$, create all residues $x^{i_1} \bmod P(x)$, for $0 \leq i_1 < 2^{d+b}$ and put $(x^{i_1} \bmod P(x), i_1)$ in a list $\mathcal{L}_1$. Sort $\mathcal{L}_1$ according to the residue value of each entry.

2. Create all residues $x^{i_1} + x^{i_2} \bmod P(x)$ such that $\phi(x^{i_1} + x^{i_2} \bmod P(x)) = 0$, for $0 \leq i_1 < i_2 < 2^{d+b}$ and put in a list $\mathcal{L}_2$. Here $\phi()$ means the d least significant bits. This is done by merging the sorted list $\mathcal{L}_1$ by itself and keeping only residues $\phi(x^{i_1} + x^{i_2} \bmod P(x)) = 0$. The list $\mathcal{L}_2$ is sorted according to the residue value.

3. In the final step we merge the sorted list $\mathcal{L}_2$ with itself to create a list $\mathcal{L}$, keeping only residues $x^{i_1} + x^{i_2} - x^{i_3} - x^{i_4} = 0 \bmod P(x)$, i.e., $x^{i_1} + x^{i_2} = x^{i_3} + x^{i_4} \bmod P(x)$.

Paper III

A new SNOW stream cipher called SNOW-V

Abstract

In this paper we are proposing a new member in the SNOW family of stream ciphers, called SNOW-V. The motivation is to meet an industry demand of very high speed encryption in a virtualized environment, something that can be expected to be relevant in a future 5G mobile communication system. We are revising the SNOW 3G architecture to be competitive in such a pure software environment, making use of both existing acceleration instructions for the AES encryption round function as well as the ability of modern CPUs to handle large vectors of integers (e.g. SIMD instructions). We have kept the general design from SNOW 3G, in terms of linear feedback shift register (LFSR) and Finite State Machine (FSM), but both entities are updated to better align with vectorized implementations. The LFSR part is new and operates 8 times the speed of the FSM. We have furthermore increased the total state size by using 128-bit registers in the FSM, we use the full AES encryption round function in the FSM update, and, finally, the initialization phase includes a masking with key bits at its end. The result is an algorithm generally much faster than AES-256 and with expected security not worse than AES-256.

Keywords: SNOW, Stream Cipher, 5G Mobile System Security.

1 Introduction

Stream ciphers have always played an important part in securing the various generations of 3GPP mobile telephony systems, starting with the GSM system employing the A5 suit of ciphers, continuing with the use of SNOW 3G as one of the core algorithm for integrity and confidentiality in both UMTS and LTE. When we now turn to the next generation system, called 5G, we see some fundamental changes in system architecture and security level that in many cases invalidate the previous algorithms. We will focus on the LTE (or 4G, as it is commonly called) system when describing the current state in link protection for mobile systems.

The basis for the link security in all 3GPP generations of mobile telephony systems is a shared secret key between the device (commonly called the User Equipment, UE)

Patrik Ekdahl, Thomas Johansson, Alexander Maximov, and Jing Yang. Anew SNOW stream cipher called SNOW-V.IACR Transactions on Symmetric Cryptology, pages 1–42, 2019.

and the home network, the Mobile Network Operator that the user has a service agreement with, and from whom the user receives the credentials in form of a UICC with a USIM application (often referred to as the SIM-card). The shared key is stored in the Home Subscriber Server (HSS) and in the Secure Element on the UICC. From this key, through a set of key derivations, the home network and the UE both agree on new keys to be used for integrity and confidentiality protection of the control channel, and confidentiality protection of the user data channel. The 4G system defines three different possible algorithms for integrity (128-EIAx) and confidentiality (128-EEAx), based on three different primitives SNOW 3G [SAG06], AES [oST01], and ZUC [SAG11]. The algorithms used in UMTS and LTE are all using the 128-bit key size, and are depicted in Table 1.

	UMTS		LTE	
	Integrity	Encryption	Integrity	Encryption
Kasumi	UIA1	UEA1		
SNOW 3G	UIA2	UEA2	EIA1	EEA1
AES			EIA2	EEA2
ZUC			EIA3	EEA3

Table 1: Base algorithms used in UMTS and LTE for integrity and confidentiality.

The SNOW family of stream ciphers started with the SNOW [EJ01] proposal in the European project NESSIE, a call for new primitives. Two attacks [HR02, CHJ02] were soon discovered and the design was subsequently updated to the SNOW 2.0 [EJ02] design. Attacks on SNOW 2.0 will be more discussed in Section 3. The ETSI Security Algorithm Group of Experts (SAGE) modified the SNOW 2.0 design and proposed the resulting cipher SNOW 3G as one of the algorithms protecting the air interface in 3GPP telecommunication networks.

Although sufficient for 4G system, these 128-EIAx and 128-EEAx algorithms face some challenges in the 5G environment. For the 5G system, the 3GPP standardization organization is looking towards increasing the security level to 256-bit key lengths [SA318]. For ExA1, and ExA2, this does not immediately appear to be a problem, since both the underlying primitives (AES and SNOW) are specified for 256-bit keys. ZUC is currently only specified and evaluated under 128-bit key strength, but another version, ZUC-256, supporting 256-bit keys has recently been presented [Bin]. However, since the design of the radio and core network will also fundamentally change in the 5G system, there are other challenges. Many of the network nodes will become virtualized [3GP] and thus the ability to use specialized hardware for the cryptographic primitives will be reduced. Many newer processors from both Intel and ARM now include instructions to accelerate AES, and it will be fairly easy to reach encryption speeds of 20-25 Gbps for EIA2 and EEA2, but for the stream ciphers SNOW and ZUC, we need to look for other solutions. Current benchmarks on SNOW 3G gives approximately 9 Gbps in a pure software implementation, which is far too low for the targeted speed of 20 Gbps downlink in the 5G system (see, e.g., [ITU17]).

In this paper we revise the SNOW 2.0/ SNOW 3G design to be competitive in a pure software environment, relying on both the acceleration instructions for the AES round function as well as the ability of modern CPUs to handle large vectors of integers

(e.g. SIMD instructions). We have kept most of the design from SNOW 3G, in terms of linear feedback shift register (LFSR) and Finite State Machine (FSM), but both entities are updated to better align with vectorized implementations. We have also increased the total state size by going from 32-bit registers to 128-bit registers in the FSM. Each clocking of SNOW-V (V for Virtualization) now produces 128 bits of keystream.

We also propose an AEAD (Authenticated Encryption with Associated Data) operational mode to provide both confidentiality and integrity protection. The keystream width of 128 bits makes the authentication framework of GMAC [Dwo07] very easy to be adopted to SNOW-V.

This paper is organized as follows. In Section 2, we present the new design, including pseudocode. In Section 3 we give a brief security analysis, describing most of the common attack approaches and how they apply to SNOW-V. In Section 4 we describe how authentication can be included in an AEAD mode of operation. Then software implementation aspects are considered in Section 5, and in Section 6 software performance results and implementation aspects using future SIMD instruction set are presented. We end the paper with some conclusions in Section 7.

2 The design

SNOW-V follows the design pattern of previous SNOW versions and consists of an LFSR part and an FSM part. The overall schematic is shown in Figure 1. The LFSR part is now a circular construction consisting of two shift registers, each feeding into the other. The FSM has three 128-bit registers and two instances of a single AES encryption round function.

Starting with the LFSR part, we will now provide a detailed description of the design. The two LFSRs are named LFSR-A and LFSR-B, both of length 16 and with a cell size of 16 bits. The 32 cells are denoted $a_{15} \dots a_0$ and $b_{15} \dots b_0$ respectively.

Each cell represents an element in $\mathbb{F}_{2^{16}}$, but LFSR-A and LFSR-B have different generating polynomials. The elements of LFSR-A are generated by the polynomial

$$g^A(x) = x^{16} + x^{15} + x^{12} + x^{11} + x^8 + x^3 + x^2 + x + 1 \in \mathbb{F}_2[x] \tag{1}$$

and the elements of LFSR-B are generated by

$$g^B(x) = x^{16} + x^{15} + x^{14} + x^{11} + x^8 + x^6 + x^5 + x + 1 \in \mathbb{F}_2[x]. \tag{2}$$

When we consider these elements of $\mathbb{F}_{2^{16}}$ as words, the x^0 position will be the least significant bit in the word. Let $\alpha \in \mathbb{F}^A_{2^{16}}$ be a root of $g^A(x)$ and $\beta \in \mathbb{F}^B_{2^{16}}$ be a root of $g^B(x)$. At time $t \geq 0$ we denote the states of the LFSRs as $(a^{(t)}_{15}, a^{(t)}_{14}, \dots, a^{(t)}_1, a^{(t)}_0), a^{(t)}_i \in \mathbb{F}^A_{2^{16}}$ and $(b^{(t)}_{15}, b^{(t)}_{14}, \dots, b^{(t)}_1, b^{(t)}_0), b^{(t)}_i \in \mathbb{F}^B_{2^{16}}$ respectively for LFSR-A and LFSR-B. Referring to Figure 1, the elements $a^{(t)}_0$ and $b^{(t)}_0$ are the elements to first exit the LFSRs. The LFSRs produce sequences $a^{(t)}$ and $b^{(t)}, t \geq 0$ which are given by the expressions

$$a^{(t+16)} = b^{(t)} + \alpha a^{(t)} + a^{(t+1)} + \alpha^{-1} a^{(t+8)} \mod g^A(\alpha) \tag{3}$$

and

$$b^{(t+16)} = a^{(t)} + \beta b^{(t)} + b^{(t+3)} + \beta^{-1} b^{(t+8)} \mod g^B(\beta), \tag{4}$$

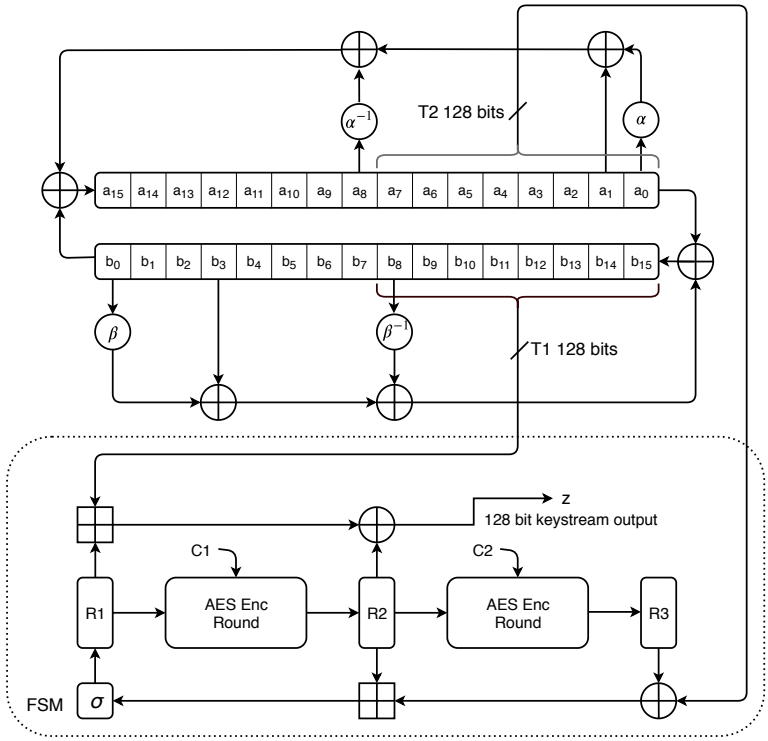

Figure 1: Overall schematics of SNOW-V.

where the initial states of the LFSRs are given by $(a^{(15)}, a^{(14)}, \ldots, a^{(0)})$ and $(b^{(15)}, b^{(14)}, \ldots, b^{(0)})$. We would like to emphasize the notation here; $a^{(t)}$ means the symbol produced by the linear recursion in Equation (3) at time t, whereas $a_i^{(t)}, 0 \leq i \leq 15$ are the values of the cells in the LFSR-A at time t. In the case of α and β, the notation α^{-1} and β^{-1} are the inverses in the respective implemented fields.

As the reader might notice, we are a bit sloppy in Equation (3) and Equation (4) and apply the field addition operation between elements of different fields, but it should be interpreted as an implicit bit pattern preserving conversion between the fields.

Each time we update the LFSR part, we clock LFSR-A and LFSR-B 8 times, i.e., 256 bits of the total 512-bit state will be updated in a *single step*, and the two taps $T1$ and $T2$ will have fresh values. In Appendix 1 we give the proof that this circular construction gives the maximum cycle length of $2^{512} - 1$.

The tap $T1$ is formed by considering $(b_{15}, b_{14}, \ldots, b_8)$ as a 128-bit word where b_8 is the least significant part. Similarly, $T2$ is formed by considering $(a_7, a_6, \ldots, a_0)$ as a 128-bit word where a_0 is the least significant part. The mapping is pictured in Figure 2, and the expressions are given by

$$T1^{(t)} = (b_{15}^{(8t)}, b_{14}^{(8t)}, \ldots, b_8^{(8t)}), \tag{5}$$

$$T2^{(t)} = (a_7^{(8t)}, a_6^{(8t)}, \ldots, a_0^{(8t)}). \tag{6}$$

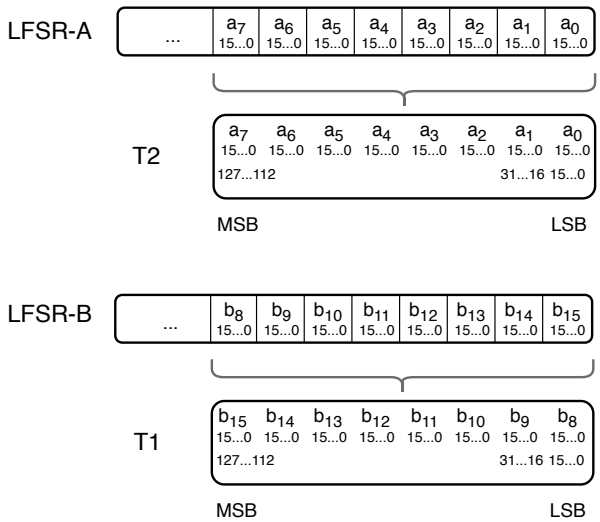

Figure 2: Mapping the 16-bit words of the LFSRs into 128-bit words $T1$ and $T2$.

We will now turn to the FSM. The FSM takes the two blocks $T1$ and $T2$ from the LFSR part as inputs and produces a 128-bit keystream as output. $R1$, $R2$, and $R3$ are 128-bit registers, $\oplus$ denotes a bitwise XOR operation, and $\boxplus_{32}$ denotes a parallel application of four additions modulo 2^{32} over each sub-word. So the four 32-bit parts of the 128-bit words are added with carry, but the carry does not propagate from a lower 32-bit word to the higher.

The output, $z^{(t)}$ at time $t \geq 0$, is given by the expression

$$z^{(t)} = (R1^{(t)} \boxplus_{32} T1^{(t)}) \oplus R2^{(t)}. \tag{7}$$

Registers $R2$ and $R3$ are updated through a full AES encryption round function as shown in Figure 3, see [oST01] for details. Let us denote the AES encryption round function

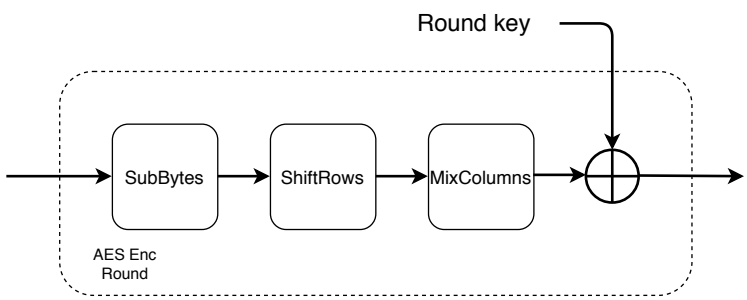

Figure 3: Internal functions of the AES encryption round function.

by $AES^R(IN, KEY)$. The mapping between the 128-bit registers and the state array

of the AES round function follows the definition in [oST01], and is pictured in Figure 4.

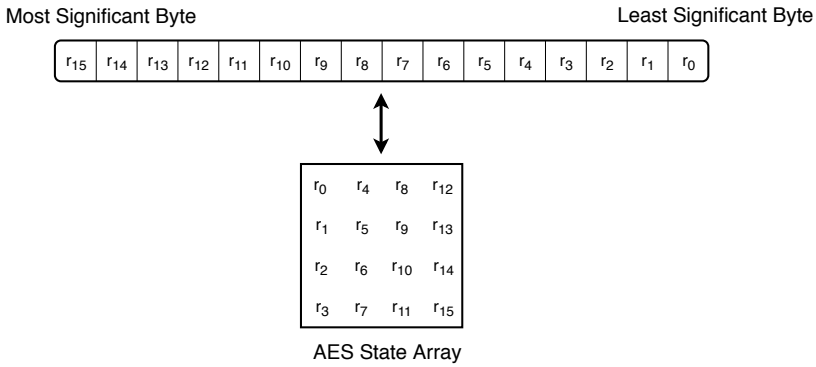

Figure 4: Mapping between a 128-bit register value and the state array of the AES round function.

We can now write the update expressions for the registers as

$$R1^{(t+1)} = \sigma(R2^{(t)} \boxplus_{32} (R3^{(t)} \oplus T2^{(t)})), \tag{8}$$

$$R2^{(t+1)} = AES^R(R1^{(t)}, C1), \tag{9}$$

$$R3^{(t+1)} = AES^R(R2^{(t)}, C2). \tag{10}$$

The values of the two round key constants $C1$ and $C2$ are set to zero, and σ is a byte-oriented permutation given by

$$\sigma = [\ 0,\ 4,\ 8,\ 12,\ 1,\ 5,\ 9,\ 13,\ 2,\ 6,\ 10,\ 14,\ 3,\ 7,\ 11,\ 15\ ]. \tag{11}$$

This should be interpreted as byte 0 is moved to position 0, byte 4 is moved to position 1, byte 8 is moved to position 2, and so on. Position 0 is the least significant byte in accordance to the mapping described above. The chosen σ implements the transposition of the mapped AES state matrix.

2.1 Initialization

Initialization is done as described in this subsection. The algorithm has a 256-bit key K and a 128-bit initialization vector(IV) as inputs. The key is denoted by

$$K = (k_{15}, k_{14}, \ldots, k_1, k_0),$$

where each k_i, $0 \leq i \leq 15$, is a 16-bit vector. The IV vector is denoted by

$$IV = (iv_7, iv_6, \ldots, iv_1, iv_0),$$

where again each iv_i, $0 \leq i \leq 7$, is a 16-bit vector.

The first step of the initialization is to load the key and IV into the LFSRs by assigning

$$(a_{15}, a_{14}, \ldots, a_0) = (k_7, k_6, \ldots, k_0, iv_7, iv_6, \ldots, iv_0)$$

and

$$(b_{15}, b_{14}, \ldots, b_0) = (k_{15}, k_{14}, \ldots, k_8, 0, 0, \ldots, 0).$$

Note that $(b_7, \ldots, b_0)$ will have a non-zero value when SNOW-V is used in AEAD-mode, see Section 4.

Then the initialization consists of 16 steps where the cipher is updated in the same way as in the running-key mode, with the exception that the 128-bit output z is not an output but is xored into the LFSR structure to positions $(a_{15}, a_{14}, \ldots, a_8)$ in every step. Additionally, at the two last steps of the initialization phase, we xor the key into the $R1$ register, inspired by [HK18]. We also limit the keystream length to a maximum of 2^{64} for a single pair of key and IV vectors, and each key may be used with a maximum of 2^{64} different IV vectors.

The pseudocode in Algorithm 1 clarifies the procedure.

Algorithm 1 SNOW-V initialization

procedure INITIALIZATION(K, IV)
  $(a_{15}, a_{14}, \ldots, a_8) \leftarrow (k_7, k_6, \ldots, k_0)$
  $(a_7, a_6, \ldots, a_0) \leftarrow (iv_7, iv_6, \ldots, iv_0)$
  $(b_{15}, b_{14}, \ldots, b_8) \leftarrow (k_{15}, k_{14}, \ldots, k_8)$
  $(b_7, b_6, \ldots, b_0) \leftarrow (0, 0, \ldots, 0)$
  $R1, R2, R3 \leftarrow 0, 0, 0$
  for $t = 1...16$ **do**
    $T1 \leftarrow (b_{15}, b_{14}, \ldots, b_8)$
    $z \leftarrow (R1 \boxplus_{32} T1) \oplus R2$
    $FSMupdate()$
    $LFSRupdate()$
    $(a_{15}, a_{14}, \ldots, a_8) \leftarrow (a_{15}, a_{14}, \ldots, a_8) \oplus z$
    if $t = 15$ **then** $R1 \leftarrow R1 \oplus (k_7, k_6, \ldots, k_0)$
    if $t = 16$ **then** $R1 \leftarrow R1 \oplus (k_{15}, k_{14}, \ldots, k_8)$

This completes the description of SNOW-V, and the full algorithm can be summarized in the pseudocode as in Algorithm 2, Algorithm 3, and Algorithm 4.

3 Security analysis

The main and most important design criterion is the security of the design. This section contains a brief analysis for a number of possible standard attack approaches. Before going into the details of various attacks, we need to have a clear picture of the expected security. We have the target of providing 256-bit security in SNOW-V, by which we mean that we claim that the total cost of finding the secret key given some keystreams is not significantly smaller than 2^{256} simple operations.

Algorithm 2 SNOW-V algorithm

procedure SNOW-V(K, IV)
  INITIALIZATION(K, IV)
  while more keystream blocks needed **do**
    $T1 \leftarrow (b_{15}, b_{14}, \ldots, b_8)$
    $z \leftarrow (R1 \boxplus_{32} T1) \oplus R2$
    $FSMupdate()$
    $LFSRupdate()$
    Output keystream symbol z

Algorithm 3 LFSR update algorithm

procedure $LFSRupdate()$
  for $i = 0...7$ **do**
    $tmp_a \leftarrow b_0 + \alpha a_0 + a_1 + \alpha^{-1} a_8 \bmod g^A(\alpha)$
    $tmp_b \leftarrow a_0 + \beta b_0 + b_3 + \beta^{-1} b_8 \bmod g^B(\beta)$
    $(a_{15}, a_{14}, \ldots, a_0) \leftarrow (tmp_a, a_{15}, \ldots, a_1)$
    $(b_{15}, b_{14}, \ldots, b_0) \leftarrow (tmp_b, b_{15}, \ldots, b_1)$

Algorithm 4 FSM update algorithm

procedure $FSMupdate()$
  $T2 \leftarrow (a_7, a_6, \ldots, a_0)$
  $tmp \leftarrow R2 \boxplus_{32} (R3 \oplus T2)$
  $R3 \leftarrow AES^R(R2)$ ▷ Note that the round keys for these AES
  $R2 \leftarrow AES^R(R1)$ ▷ encryption rounds are $C1 = C2 = 0$
  $R1 \leftarrow \sigma(tmp)$

The use of the algorithm is limited to keystreams of length at most 2^{64} and we also limit the number of different keystreams that are produced for a fixed key to be at most 2^{64}. There seem to be no use cases where it makes sense to violate this limitation. Although attacks beyond these limits are certainly of academic interest, an attack claiming to break the cipher should meet this requirement. In Section 3.1 and Section 3.4 we give some cryptanalysis results also for the case when σ is replaced by the identity mapping σ_0. This may aid the understanding of the strength of different methods of cryptanalysis.

We also frequently compare with AES-256 in the GCM mode. We note that exhaustive key search of AES-256 requires computational cost around 2^{256}. However, if used in the GCM mode, it actually takes complexity (and data) around 2^{64} to distinguish such keystreams from random. For SNOW-V, we claim that the security is never worse than the security of AES-256 in the GCM mode, for any kind of attack on the algorithmic level.

3.1 Initialization attacks through MDM/AIDA/cube attacks

Stream ciphers always have an initialization phase before producing keystream bits, during which the key and IV are loaded and mixed by running the cipher a few rounds (16 for SNOW-V) without giving outputs until the state becomes random-like. It should be difficult for a cryptanalyst to predict the keystream or get some information about the initial key according to the output after initialization. It then becomes vital to make sure that the key/IV loading has no fatal flaws and the initialization round is carefully chosen in order not to result in a resource waste (too many rounds) or some weakness (too few rounds).

A chosen IV attack is one type of attacks targeting this problem [Mj06, EJT07], in which the adversary attempts to build a distinguishing attack to introduce randomness failures in the output by selecting and running through certain IV values. The rationales behind are that: 1) a stream cipher can be regarded as a succession of single-valued boolean functions f_i with each keystream bit as the output and key/IV as the inputs, and 2) any monomial coefficient in the algebraic normal form (ANF) representations of these Boolean functions should appear to be 1 (or 0) with probability 1/2 if f_i is drawn uniformly at random [Sta13]. If one can distinguish the output from a random distribution under some key/IV settings with acceptable complexity, the cipher is believed to be unsafe. In this attack, the adversary fixes the key and a subset of IV bits and runs through all possible values of the non-fixed IV bits, which are called a cube. The truth tables of the boolean functions can be derived, which are further used to compute the monomial coefficients of the ANF and compared with expected values. If there exists a big gap between them, the output is believed to be non-random. The best and most commonly used monomial is the maximum degree monomial (MDM) and the corresponding test is called MDM test. In [Sta10] one even allows setting arbitrary key values to build a non-randomness detector to further check whether the initialization is robust enough. It should be noted that the MDM test and AIDA (algebraic IV differential attack)/cube distinguishers [Vie07, DS09] are various forms of using higher order differentials [Lai94] on stream ciphers.

We employ the greedy MDM test algorithm in [Sta10] to test the SNOW-V initialization. We start with the worst 3-bit set under which the randomness result deviates the most from the expected value and gradually increase to a 24-bit set. Every time when one more bit is added from the remaining bits, we select the bit resulting in the worst randomness result until we get a 24-bit set (larger sets can be tested on more powerful computers). Figure 5 shows the maximum number of initialization rounds failing the MDM test under different bit set sizes for SNOW-V under permutation σ and no permutation σ_0 (identity mapping). The results for 1, 2 and 3-bit sets are obtained through exhaustive search, while for the sets with larger sizes, the results are based on greedy searching from the obtained worst 3-bit set. It can be seen that the performance under σ is better, indicating the cipher is mixed better and faster in this case. In both cases, roughly the first 7 rounds fail the MDM test, ensuring a large security gap to 16 rounds. One can also note that the number of rounds the MDM test can detect grows very slowly with the size of the set of key/IV bits that are exhausted. In an attack, one could consider sets of sizes up to 64 bits. This indicates that the 16 initialization rounds in SNOW-V should be enough for the cipher and that the output of the cipher has become random-like after the initialization. It also indicates that significantly reducing

the number of rounds might be dangerous.

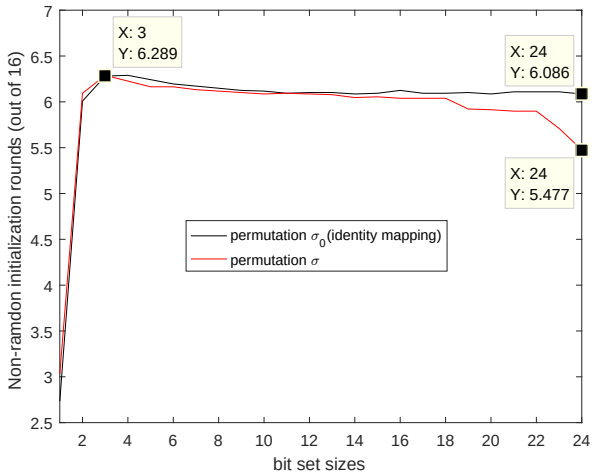

Figure 5: The maximum number of initialization rounds failing the MDM test under different bit set sizes.

Recently, cube attacks based on division property have got a lot of research and application on multiple stream ciphers. Division property, proposed by Todo *et al.* in [Tod15], is a generalization of integral property being used to find integral distinguishers or launch cube attacks. Unlike traditional experimental attacks where the ciphers are regarded as black boxes, division property based attacks explore the internal structures of ciphers and trails of division property according to propagation rules for different operations. These rules could be expressed with some (in)equalities and the attacks are modeled as MILP(Mixed Integer Linear Programming) problems with certain constrains and objective functions. Some optimization tools such as Gurobi, Cplex could then help to solve the problems very efficiently and identify if the attacks are feasible or not in certain cases.

In [TIHM17], division property was introduced into cube attacks to evaluate the set of key bits J involved in the superpoly given a certain cube I. After obtaining the set J, attackers are able to recover the superpoly by building the truth table and further to recover part of the key by querying the encryption oracle. The time complexity of recovering the superpoly is $2^{|I|+|J|}$, which is feasible when $|I| + |J|$ is smaller than the security bit level. Authors in [WHT+18] further improved the attack by exploiting various algebraic properties of superpolies. They introduced a technique to evaluate the upper bound of the algebraic degree, denoted as d, of the superpoly to avoid recovering the coefficients of monomials with degrees larger than d. Hence, only $\binom{|J|}{\leq d}$ coefficients need to be recovered and the time complexity reduces to $2^{|I|} \times \binom{|J|}{\leq d}$.

We evaluate SNOW-V with division property based cube attacks using the method described above. The MILP model of division property for SNOW-V which can evaluate all division trails when the initialization rounds are reduced to R is illustrated in Algorithm 5. We first load key and IV bits to the LFSR state and initialize $R1$, $R2$,

$R3$ to 0 according to the initialization specification of SNOW-V (Algorithm 1). Since K and IV are loaded into LFSR states through one-by-one mapping, we do not introduce more intermediate variables in the model. In every round, we deal with the first and last 7 iterations of LFSR update differently since they have different propagation trails: the function `LFSRupdate` for the last 7 LFSR updates consists of `copy`, `xor`, `multiplication` (with $\alpha, \beta, \alpha^{-1}, \beta^{-1}$), `shift` to update the LFSR state while for the first round, the function `LFSRupdateFirstIter` involves in more complicated `copy` trails to get $T1$ and $T2$. The `funcAES` function is four parallel AES rounds consisting of `sbox`, `shiftrow` and `mixcolumn` whose propagation rules could be found in [Tod15]. The function `funcModAdd` consists of 4 parallel modular additions whose propagation rule has been established in [SWW17] and `funcXor` represents `xor` rule for 128 bits. The functions of `multiplication` and `mixcolumn` have consistent characteristic: the sum of input equals to the sum of output in terms of division property while the functions `shift`, `shiftrow` and `sigma` only permute the division property vectors.

Algorithm 5 MILP model of division property for SNOW-V

procedure SnowvCore(ROUND R)
  Prepare empty MILP Model $\mathcal{M}$
  $\mathcal{M}.var \leftarrow s_i^0$ for $i \in \{0, 1, ..., 511\}$ and $R1_i^0, R2_i^0, R3_i^0$ for $i \in \{0, 1, ..., 127\}$
  $(\mathcal{M}, \boldsymbol{s}^0, \boldsymbol{R1}^0, \boldsymbol{R2}^0, \boldsymbol{R3}^0) =$ `init` $(\mathcal{M}, \boldsymbol{K}, \boldsymbol{IV})$
  for $r = 1$ to $R - 1$ **do**
    $(\mathcal{M}, \boldsymbol{T1}, \boldsymbol{T2}, \boldsymbol{s}^{r,0}) =$ `LFSRupdateFirstIter` $(\mathcal{M}, \boldsymbol{s}^{r-1})$
    $(\mathcal{M}, \boldsymbol{X}) =$ `funcModAdd` $(\mathcal{M}, \boldsymbol{R1}^{r-1}, \boldsymbol{T1})$
    $(\mathcal{M}, \boldsymbol{Z}^r) =$ `funcXor` $(\mathcal{M}, \boldsymbol{X}, \boldsymbol{R2}^{r-1})$
    $(\mathcal{M}, \boldsymbol{Y}) =$ `funcXor` $(\mathcal{M}, \boldsymbol{R3}^{r-1}, \boldsymbol{T2})$
    $(\mathcal{M}, \boldsymbol{U}) =$ `funcModAdd` $(\mathcal{M}, \boldsymbol{R2}^{r-1}, \boldsymbol{Y})$
    $(\mathcal{M}, \boldsymbol{R1}^r) =$ `sigma` $(\mathcal{M}, \boldsymbol{U})$
    $(\mathcal{M}, \boldsymbol{R3}^r) =$ `funcAES` $(\mathcal{M}, \boldsymbol{R2}^{r-1})$
    $(\mathcal{M}, \boldsymbol{R2}^r) =$ `funcAES` $(\mathcal{M}, \boldsymbol{R1}^{r-1})$
    for $i = 1$ to 7 **do**
      $(\mathcal{M}, \boldsymbol{s}^{r,i}) =$ `LFSRupdate` $(\mathcal{M}, \boldsymbol{s}^{r,i-1})$
    $(\mathcal{M}, s^r_{128...255}) =$ `funcXor` $(\mathcal{M}, s^{r,7}_{128...255}, \boldsymbol{Z}^r)$
    $(\mathcal{M}, s^r_{0...127}, s^r_{256...511}) = (\mathcal{M}, s^{r,7}_{0...127}, s^{r,7}_{256...511})$
  $(\mathcal{M}, \boldsymbol{X}) =$ `funcModAdd` $(\mathcal{M}, \boldsymbol{R1}^{R-1}, s^{R-1}_{384...511})$
  $(\mathcal{M}, \boldsymbol{Z}^R) =$ `funcXor` $(\mathcal{M}, \boldsymbol{X}, \boldsymbol{R2}^{R-1})$
  for $i = 0$ to 383 **do**
    $\mathcal{M}.con \leftarrow s_i^{R-1} = 0$
  for $i = 0$ to 127 **do**
    $\mathcal{M}.con \leftarrow R3_i^{R-1} = 0$
  $\mathcal{M}.con \leftarrow \sum Z_i^R = 1$
  return $\mathcal{M}$

We tried different cubes and Table 2 listed some examples under which adversaries have good advantages for permutations σ and σ_0. The time complexity in the table shows the time complexity of superpoly recovery. One can see, all key bits are involved

Rounds	3	4	5	6	7
cube size $\|I\|$	15	40	128	128	128
degree d	17	145(27)	256(106)	256(254)	256
involved key size $\|J\|$	131(113)	256(96)	256	256	256
time complexity	$2^{84.9}(2^{81.1})$	$> 2^{256}(2^{119.5})$	$> 2^{256}$	$> 2^{256}$	$> 2^{256}$

Table 2: Cube attacks on reduced-rounds of SNOW-V (values in round brackets are for the case with σ as identity (σ_0) where the value is different).

in the superpolies from the 4-th round under permutation σ while 5-th round under σ_0, which indicates the good mixing effect after 4(or 5) rounds and guarantees a large enough security margin for 16 rounds. The results match well with the research on division property based distinguishing attacks on AES in [Tod15], where only 4-round distinguisher could be found with 2^{120} plaintexts and the conclusion in [SWW16] that integral distinguishers for AES based on division property covering more than four rounds probably do not exist.

3.2 Other initialization attacks

Another attack possibility is to launch a differential attack, either in the IV bits only, or in combination with key bits. The latter would then lead to a related-key attack. Since the initialization contains 16 rounds, each including two applications of the AES encryption round function, the differential would have to go through a lot of highly nonlinear operations, which makes this approach less successful.

Finally, a further option is the slide attacks [BW99]. Such sliding properties have been considered on previous versions in the SNOW family [KY11]. The idea is to have the same initial state for two different key/IV pairs in different time instances. Then they will produce the same keystream with the difference of a shift in time. Since the required IV values vary with the choice of key bits, it is questionable whether such an approach is useful at all in cryptanalysis, but at least it indicates that the cipher is not to be considered as a random function of both the key and IV. For SNOW-V such properties would still be much more difficult to find, due to the update of 128-bit blocks in each time instance and the use of the $FP(1)$-mode [HK18] in the initialization.

3.3 Time/Memory/Data tradeoff attacks

A Time/Memory/Data tradeoff (TMD-TO) attack is a generic method of inverting ciphers by balancing between spent time, required memory and obtained data, which can be much more efficient and applicable than an exhaustive key search attack. Some stream ciphers are vulnerable to TMD-TO attacks, and their effective key lengths (e.g., n-bit) could then be reduced towards the birthday bound (i.e., $n/2$), typically happening if the state size is small. A well known such attack on A5/1 was given in [BSW01].

The TMD-TO attacks have two phases: a preprocessing phase, during which the mapping table from different secret keys or internal states to keystreams is computed and stored with time complexity P and memory M; and a real-time phase, when attackers have intercepted D keystreams and search them in the table with time complexity

T, expecting to get some matches and further recover the corresponding input. By balancing between parameters P, D, M, and T under some tradeoff curves, attackers can launch attacks according to their available time, memory and data resources. The most popular tradeoffs are Babbage-Golic (BG) [Bab95, Gol97] and Biryukov-Shamir (BS) [BS00] tradeoff with curves $TM = N$, $P = M$ with $T \leq D$; and $TM^2D^2 = N^2$, $P = N/D$ with $T \geq D^2$, where N is the input space, respectively. Attackers can try to reconstruct the internal state at a specific time or recover the secret key.

The rationale behind the TMD-TO attacks that try to reconstruct the internal state is that in many stream ciphers, the internal state update process is invertible, which means that if an attacker manages to reconstruct an internal state at any specific time, it can not only obtain subsequently generated keystreams by running the cipher forwards, but also recover previous states iteratively and further get the underlying secret key by running backwards. But for the SNOW-V case, attackers have no obvious ways to reconstruct the internal state, since SNOW-V has a large internal state with 896 bits ($2 \times$ 256-bit LFSRs + $3 \times$ 128-bit registers), which is 3.5 times the secret key length. The best attack complexity achieved is under BG tradeoff with point $T = M = D = N^{1/2} = 2^{448}$, which is still much worse than the exhaustive key search attack. Actually, SNOW-V satisfies the rule derived from TMD-TO attacks in [Gol97] and widely applied in the design of new ciphers, that the size of the internal state should be at least twice the size of the secret key to get the expected security level.

Moreover, in SNOW-V, attackers would get even less even if they reconstructed an internal state. While computing subsequent keystreams corresponding to that specific IV is still possible, they can not trivially recover the secret key or keystreams under other IV values. This is due to the key masking to the register $R1$ at the last two rounds of initialization, which represents an instantiation of the $FP(1)$-mode introduced in [HK18].

Attackers can also try to recover the secret key directly. To do so, some mappings from different key/IV pairs to generated keystream segments are firstly pre-computed and stored [HS05, DK08]. If attackers get some keystream data under different secret keys corresponding to these IV values, they can search them in the table to expect a collision and further recover some of the secret keys directly. The tradeoff curves are still the same as that to recover the internal states except N is now changed to be the size of the set of all possible (K, IV) pairs. In the SNOW-V case, the sizes of key and IV spaces are 2^{256} and 2^{128}, respectively. Two typical points for BG and BS attacks are $T = D = M = 2^{192}$ and $T = 2^{256}$, $D = M = 2^{128}$. Someone would question that the efficient size of the key in the first tradeoff is reduced from 256 to 192 bits, but actually, no ciphers including AES-256 can be immune to this as long as their IV sizes are smaller than the key sizes. In any case, the corresponding multikey attacks on AES-256 are not more costly.

3.4 Linear distinguishing attacks and correlation attacks

Traditionally, the main threat against stream ciphers has been various types of linear attacks, either in the form of distinguishing attacks on the keystreams, or state recovery attacks through correlation attacks. The basic foundations of correlation attacks can be found in papers like [CJS01, CJM02] and an overview of distinguishing attacks is to be

found in [HJB09].

The basic technique for these types of attacks is to use linear approximations of the nonlinear operations used in the cipher and then derive a linear relationship between output values from different time instances. Such a relationship will then hold only as a very rough approximation, which in turn can be thought of as a linear function of some given output bits being considered as a sample drawn from a nonuniform distribution. This approach may give a distinguishing property for the keystream. If the relationship also involves state bits, the same arguments may give samples that are highly noisy observations of state bits, which in turn may be linear combinations of the original initial state. This may give a way to recover the state and that is the foundation of a correlation attack.

For SNOW 2.0, several distinguishing attacks and correlation attacks have been proposed [NW06, ZXM15]. The basic idea has been to approximate the FSM part through linear masking and then to cancel out the contributions of the registers by combining expressions for several keystream words. We should note that this kind of attacks tend to require an extremely large length of the keystream. Also, no significant attack of this type on SNOW 3G has been published. We now consider a similar approach for making some basic arguments on SNOW-V.

We recall the FSM equations:

$$\begin{aligned} z^{(t)} &= (R1^{(t)} \boxplus_{32} T1^{(t)}) \oplus R2^{(t)}, \\ R1^{(t+1)} &= \sigma(R2^{(t)} \boxplus_{32} (R3^{(t)} \oplus T2^{(t)})), \\ R2^{(t+1)} &= AES^R(R1^{(t)}), \\ R3^{(t+1)} &= AES^R(R2^{(t)}). \end{aligned}$$

A linear approximation of the FSM would then try to cancel out the contribution from the registers, leaving keystream symbols and the LFSR contribution. Assume that the value of the registers at some time t is $(\hat{R1}, \hat{R2}, \hat{R3})$. Then we have

$$\begin{aligned} z^{(t)} &= (\hat{R1} \boxplus_{32} T1^{(t)}) \oplus \hat{R2}, \\ R1^{(t+1)} &= \sigma(\hat{R2} \boxplus_{32} (\hat{R3} \oplus T2^{(t)})), \\ R2^{(t+1)} &= AES^R(\hat{R1}), \\ R3^{(t+1)} &= AES^R(\hat{R2}). \end{aligned}$$

For time $t+1$,

$$z^{(t+1)} = (\sigma(\hat{R2} \boxplus_{32} (\hat{R3} \oplus T2^{(t)})) \boxplus_{32} T1^{(t+1)}) \oplus AES^R(\hat{R1}).$$

It is now straight-forward to see that since $z^{(t)}$ depends only on $\hat{R1}$ and $z^{(t+1)}$ depends on both $\hat{R1}$ and $\hat{R2}$, there can be no biased linear approximation using only $z^{(t)}$ and $z^{(t+1)}$. So the minimum number of equations that needs to be considered is three. To simplify coming derivations, we introduce the expressions for $z^{(t-1)}$, i.e.,

$$z^{(t-1)} = ((AES^R)^{-1}(\hat{R2}) \boxplus_{32} T1^{(t-1)}) \oplus (AES^R)^{-1}(\hat{R3}),$$

and seek a biased expression involving $z^{(t-1)}, z^{(t)}, z^{(t+1)}$.

Consider all 128-bit variables as column vectors of 16 bytes. Then $AES^R(X)$ can be written as $L \cdot S(X)$, where S is an application of the 16 AES S-Boxes, one on each byte of X, and L is a linear transformation over the 16 bytes (including both ShiftRow and MixColumn). Furthermore, $(AES^R)^{-1}(X) = S^{-1}(L^{-1} \cdot X)$. We now introduce simplifications to the SNOW-V algorithm to show the best results on versions weaker than the proposed algorithm.

3.4.1 Analysis of the bias using σ_0 and using byte-wise addition $\boxplus_8$

We assume that there is no byte-wise permutation, i.e., σ_0 is just the identity. Furthermore, instead of $\boxplus_{32}$ we consider a modulo addition which is restricted to each byte, denoted $\boxplus_8$. With this simplification all operations are byte-oriented and the investigation of linear approximations is easier.

Let us now seek a byte-oriented linear approximation. We examine the following three equations.

$$\begin{aligned} z^{(t-1)} &= (S^{-1}(L^{-1} \cdot \hat{R2}) \boxplus_8 T1^{(t-1)}) \oplus S^{-1}(L^{-1} \cdot \hat{R3}), \\ z^{(t)} &= (\hat{R1} \boxplus_8 T1^{(t)}) \oplus \hat{R2}, \\ L^{-1}z^{(t+1)} &= L^{-1}((\hat{R2} \boxplus_8 (\hat{R3} \oplus T2^{(t)})) \boxplus_8 T1^{(t+1)}) \oplus S(\hat{R1}). \end{aligned}$$

Let X_i denote the ith byte of a vector of bytes X. Summing up the three z-terms on the left side and taking byte 0 gives us

$$[z^{(t-1)} \oplus z^{(t)} \oplus L^{-1}z^{(t+1)}]_0 = [N1 \oplus N2 \oplus N3 \oplus T1^{(t)} \oplus T1^{(t-1)} \oplus L^{-1}(T2^{(t)} \oplus T1^{(t+1)})]_0,$$

where

$$\begin{aligned} N1 &= (S^{-1}(L^{-1} \cdot \hat{R2}) \boxplus_8 T1^{(t-1)}) \oplus S^{-1}(L^{-1} \cdot \hat{R2}) \oplus T1^{(t-1)}, \\ N2 &= (\hat{R1} \boxplus_8 T1^{(t)}) \oplus S(\hat{R1}) \oplus T1^{(t)}, \\ N3 &= \underbrace{L^{-1}((\hat{R2} \boxplus_8 (\hat{R3} \oplus T2^{(t)}) \boxplus_8 T1^{(t+1)}) \oplus T2^{(t)} \oplus T1^{(t+1)}) \oplus \hat{R2}}_{\text{Linear part } A} \\ &\quad \oplus S^{-1}(\ \underbrace{L^{-1} \cdot \hat{R2}}_{\text{Linear part } B}\ ) \oplus S^{-1}(\ \underbrace{L^{-1} \cdot \hat{R3}}_{\text{Linear part } C}\ ). \end{aligned}$$

The general idea is that $[N1 \oplus N2 \oplus N3]_0$ is a biased distribution. It is true because the (types of) noise variables $n1, n2, n3$ defined below are biased, which can be checked:

$$\begin{aligned} n1 &= (x \boxplus_8 y) \oplus x \oplus y, \\ n2 &= (x \boxplus_8 y) \oplus S(x) \oplus y, \\ n3 &= (x \boxplus_8 y) \oplus x \oplus S(x) \oplus y. \end{aligned}$$

Each of the noise terms $N1, N2, N3$ given above are of the above types and hence the sum of them can also be a biased distribution.

Computation of the bias: it remains to compute the bias and the $N3_0$ part is the most complicated case (although we are computing the bias of $[N1 \oplus N2 \oplus N3]_0$ as

there is a dependence between $N1$ and $N3$). The noise $N3$ at byte 0 can be rewritten as:

$$N3_0 = \sum_{i=0}^{3}(c_i \cdot U_i) \oplus \hat{R}2_0 \oplus S^{-1}\left(\sum_{i=0}^{3}(c_i \cdot \hat{R}2_i)\right) \oplus S^{-1}\left(\sum_{i=0}^{3}(c_i \cdot \hat{R}3_i)\right),$$

where the coefficients are $c = [e \;\; b \;\; d \;\; 9]$. First note that U_i is the variable corresponding to $U_i = (\hat{R}2_i \boxplus_8 (\hat{R}3_i \oplus T2_i^{(t)}) \boxplus_8 T1_i^{(t+1)}) \oplus T2_i^{(t)} \oplus T1_i^{(t+1)}$, for $0 \leq i \leq 3$. Note also that L^{-1} corresponds to first applying the inverse MixColumn and then the inverse ShiftRow (which does not change the position of byte 0).

The idea of computing the distribution is now simple. We assume that we have 8-bit adders $\boxplus_8$ instead of the original 32-bit ones, keeping all operations within bytes. We try all combinations of the first bytes of the inputs $\hat{R}2_0, \hat{R}3_0, T2_0^{(t)}, T1_0^{(t+1)}$ which leads to a *partial sum* as a 3-byte value denoted $A|B|C$, as marked in the equations above. Thus, we can compute the distribution $D_0(A|B|C)$. We also compute similar distributions for every "slice" of the inputs, $D_k(A|B|C)$, $k = 0, 1, 2, 3$, corresponding to inputs $\hat{R}2_k, \hat{R}3_k, T2_k^{(t)}, T1_k^{(t+1)}$.

The XOR-convolution of the computed distributions of the partial linear expressions gives the combined distribution of the triple $A|B|C$ over all possible 32-bit inputs. Having that total distribution, it is then easy to construct the 8-bit distribution of $N3_0$.

Let D be a distribution of a noise variable X. Then we compute the bias denoted $\epsilon(X) = \epsilon(D)$ using the *Squared Euclidean Imbalance* as in [ZXM15], defined through

$$\epsilon(D) = |D| \sum_{x=0}^{|D|-1} \left(D(x) - \frac{1}{|D|}\right)^2.$$

The number of samples needed to distinguish a source of noise D from random is roughly $O(1/\epsilon)$. The results when we use 8-bit adders $\boxplus_8$, and σ identity are as follows: $\epsilon(N1) > 1$, $\epsilon(N2) \approx 2^{-2.9}$, $\epsilon(N3) \approx 2^{-46.0}$, and

$$\epsilon(N_{tot}) \approx 2^{-53.5}, \quad \epsilon(2 \times N_{tot}) \approx 2^{-106.8}, \quad \epsilon(3 \times N_{tot}) \approx 2^{-160.2}, \quad \epsilon(4 \times N_{tot}) \approx 2^{-213.6}.$$

Here the $N1, N2, N3$ denotes the partial noise as described above and N_{tot} represents the full noise of a single approximation, i.e., $N_{tot} = [N1 \oplus N2 \oplus N3]_0$. Finally, $i \times N_{tot}$ denotes the noise obtained by a sum of i such independent noise terms.

3.4.2 Analysis of the bias using σ_0 and using 32-bit $\boxplus_{32}$

In order to deal with 32-bit adders we should actually compute partial noise distributions D_k that also correspond to different values of input and output carries (0, 1, or 2) and then perform sums and convolutions over matching distribution tables. It is computationally more demanding but not unreachable and we have computed the biases also in this case.

The results when using 32-bit adders $\boxplus$, and σ as identity are as follows: $\epsilon(N1) > 1, \epsilon(N2) \approx 2^{-2.9}, \epsilon(N3) \approx 2^{-46.4}$, and

$$\epsilon(N_{tot}) \approx 2^{-58.7}, \quad \epsilon(2 \times N_{tot}) \approx 2^{-118.4}, \quad \epsilon(3 \times N_{tot}) \approx 2^{-177.8}, \quad \epsilon(4 \times N_{tot}) \approx 2^{-237.1}.$$

3.4.3 Using the bias in a fast correlation attack

A detected bias can be used in a distinguishing attack if one can find, say, a weight 3 or weight 4 multiple of the polynomial corresponding to the byte-oriented sequence $W(t) = [T1^{(t)} \oplus T1^{(t-1)} \oplus L^{-1}(T2^{(t)} \oplus T1^{(t+1)})]_0$. Since the LFSR feedback is defined in $\mathbb{F}_{2^{16}}$, it can be rewritten in $\mathbb{F}_{2^8}$ and there will be a linear recursion relation for $W(t)$ over $\mathbb{F}_{2^8}$. The complexity of finding a weight 3 or weight 4 multiple with general methods is far more than that of exhaustive key search. Instead, a fast correlation attack looks more promising. In such an attack the LFSR state at some initial time is considered as a length 64 byte vector $\mathbf{v} = (v_0, v_1, \ldots v_{63})$, $v_i \in \mathbb{F}_{2^8}$ and every $W(t)$ is written as a linear combination of initial state bytes, i.e., $W(t) = \mathbf{w_t} \cdot \mathbf{v}^T$, where the vector $\mathbf{w_t}$ is computed through the recursion for $W(t)$. If we now look for pairs of vectors $\mathbf{w_t}$ and $\mathbf{w_{t'}}$ such that $\mathbf{w_t} \oplus \mathbf{w_{t'}}$ is zero in the last d entries, we can form the sum

$$[z^{(t-1)} \oplus z^{(t)} \oplus L^{-1}z^{(t+1)}]_0 \ \oplus [z^{(t'-1)} \oplus z^{(t')} \oplus L^{-1}z^{(t'+1)}]_0$$

which approximates $W(t) \oplus W(t')$ with a noise which is the sum of two noise variables of the form $Ntot$, which has a bias of $2^{-118.4}$. In the attack we guess $64 - d$ byte entries of the initial state and then we need to find in the order of 2^{118} pairs of vectors that have the same last d entries. Through a birthday argument, assuming we generate all such values up to t_0, we need to have $t_0^2 \approx 2^{118+8d}$. For example, if $d = 36$ then we need to guess $28 \cdot 8 = 224$ bits and we need to generate output until time $t_0 = 2^{203}$. Following the complexity estimations of e.g. [ZXM15] we end up with a complexity slightly below exhaustive key search. This however requires a keystream of length $t_0 = 2^{203}$, which can be compared to the maximum keystream length allowed which is 2^{64}. The attack also uses memory of size 2^{203} units, each storing the necessary information of one time unit. So the relevance of such attacks is indeed questionable.

3.4.4 Analysis of the bias using σ as proposed

In this scenario, we use σ as given in Section 2. This presumably makes the bias of linear approximations for the FSM to be much smaller, and we will sketch some ideas on how to find good linear approximations. Let us again return to the equations for a three word approximation, and use 8-bit adders $\boxplus_8$ in order to simplify derivations:

$$\begin{aligned} z^{(t-1)} &= (S^{-1}(L^{-1} \cdot \hat{R2}) \boxplus_8 T1^{(t-1)}) \oplus S^{-1}(L^{-1} \cdot \hat{R3}), \\ z^{(t)} &= (\hat{R1} \boxplus_8 T1^{(t)}) \oplus \hat{R2}, \\ z^{(t+1)} &= (\sigma(\hat{R2} \boxplus_8 (\hat{R3} \oplus T2^{(t)})) \boxplus_8 T1^{(t+1)}) \oplus L \cdot S(\hat{R1}). \end{aligned}$$

We consider a byte-oriented linear approximation with left side $\Lambda_0 z^{(t-1)} \oplus \Lambda_1 z^{(t)} \oplus \Lambda_2 z^{(t+1)}$, where Λ_i are length 16 row vectors of bytes viewed as elements in $\mathbb{F}_{2^8}$.

Recall now that if a particular byte is only present once as a linear term or in an S-box expression in the right side, then the approximation will be unbiased. However, if it appears at least twice in different types of expressions, it is likely to give a biased contribution.

The first observation we can do is that the bytes in $\hat{R1}$ appear only in two different expressions: as the direct value $\hat{R1}$ in the expression for $z^{(t)}$ and as $L \cdot S(\hat{R1})$ in the

expression for $z^{(t+1)}$. In turn, this means that $\Lambda_1 z^{(t)} \oplus \Lambda_2 z^{(t+1)}$ depends on $\hat{R1}$ through $\Lambda_1 \hat{R1} \oplus \Lambda_2 L \cdot S(\hat{R1})$. It is biased only if every byte present in $\Lambda_1 \hat{R1}$ is also present in $\Lambda_2 L \cdot \hat{R1}$ and we come to the conclusion that $\Lambda_2 = \Lambda_1 L^{-1}$, in its straightforward case.

Now we consider the contributions from $\hat{R2}$ and $\hat{R3}$. For $\hat{R2}$ we have contributions $\Lambda_0 S^{-1}(L^{-1} \cdot \hat{R2})$, $\Lambda_1 \hat{R2}$ and $\Lambda_1 L^{-1} \sigma(\hat{R2} \boxplus_8 ...)$. When σ_0 was used, we could simply consider byte 0 ($\Lambda_0 = \Lambda_1 = (1, 0, \ldots, 0)$) because it involved four bytes (byte 0-3) and they all appeared at least twice in the above expression. But for the actual σ, this is no longer possible since $L^{-1}\sigma(\hat{R2})$ includes different bytes in a position, compared to $L^{-1} \cdot \hat{R2}$.

We have not been able to find better approximations than the ones that include all 16 bytes of $\hat{R2}$ and $\hat{R3}$, and 4 bytes of $\hat{R1}$ in the approximation. One such example would be $\Lambda_0 = \Lambda_1 = (1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$, which would nearly correspond to the sum of 4 approximations of the previous kind with σ_0 (no-permutation).

In our best attempt to approximate the FSM of the proposed algorithm, we have got the total noise (with $\boxplus_8$) having the bias (more details can be found in Appendix 2):

$$\epsilon(N_{tot}) \approx 2^{-214.8} \quad \text{and} \quad \epsilon(2 \times N_{tot}) \approx 2^{-429.7}.$$

3.5 Algebraic attacks

In an algebraic attack the attacker derives a number of nonlinear equations in either unknown key bits or unknown state bits and solves the system of equations. In general, the problem of solving a system of nonlinear equations is not known to be solvable in polynomial time (even for quadratic equations), but some special cases might be solved efficiently [CKPS00].

For SNOW 2.0 there was a very interesting algebraic attack on a simplified version, given in [BG05]. However, due to the use of three FSM registers instead of two, applying such an approach on SNOW-V does not give such a nice quadratic system as in [BG05].

So for a general algebraic attack, we should either target the key or the state. For the latter, one would need to use equations from 7 keystream blocks to be able to solve for the $7 * 128$ bit internal state. That would involve nonlinearity from 11 AES encryption round functions and 13 $\boxplus_{32}$ operations. Instead, targeting the key bits would require stepping through the equations of the 16 initialization rounds together with the equations of two keystream blocks. Both these approaches give systems of nonlinear equations that appear to be much more difficult to solve than corresponding equations for AES-256. This is due to the use of the $\boxplus_{32}$ operation.

3.6 Guess-and-determine attacks

In a guess-and-determine attack one guesses part of the state and from the keystream equations, determines the value of other parts of the state. The goal is to guess as few bits as possible and determine as many as possible through keystream equations. For the case of SNOW-V, the equation $z^{(t)} = (R1^{(t)} \boxplus_{32} T1^{(t)}) \oplus R2^{(t)}$ involves three unknown values, each of size 128 bits. In order to determine some state bits, one then has to guess two of them, i.e. guessing 256 bits. Then looking at the equation for $z^{(t+1)}$, it would require the guess of one more 128 bit value. This indicates that a guess-and-determine attack would not be successful.

3.7 Other attacks

We have not made any specific design choices to explicitly support implementations that should protect against side-channel attacks and fault attacks. So such attacks, if relevant for an application, have to be considered when the algorithm is implemented. In particular, information leakage from the CPU in a software implementation must be carefully considered.

4 AEAD mode of operation

The GMAC integrity and authentication algorithm specified in [Dwo07] can easily be adopted to work with SNOW-V to define an AEAD mode of operation. We will use notations from [Dwo07] in the following. In GCM, an unspecified block cipher is used in counter mode to encrypt the plaintext. Additionally, the block cipher is used to produce the final authentication tag T, and to derive the key H used in the function $GHASH_H$.

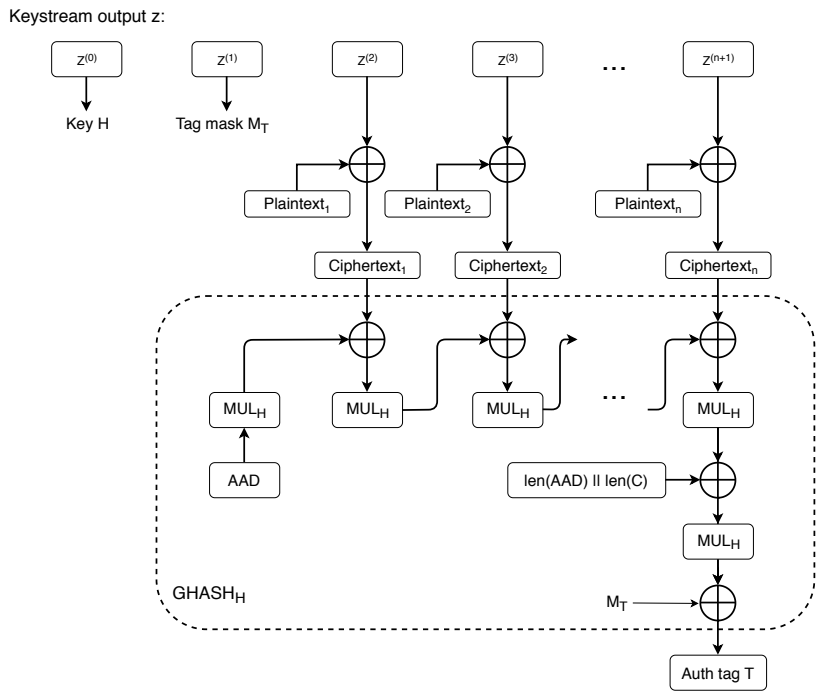

Figure 6: How SNOW-V is used together with $GHASH_H$ to enable AEAD.

When using SNOW-V together with the $GHASH_H$ algorithm, the key H is the very first keystream output $z^{(0)}$. Then we use keystream output $z^{(1)}$ as the final masking for the tag, similarly to the encrypted value of J_0 in [Dwo07]. To encrypt the n plaintext blocks, we use the keystream outputs $z^{(2)}, \ldots, z^{(n+1)}$, feeding the ciphertext blocks into $GHASH_H$.

SNOW-V works as described in Section 2 with a single change. During initialization of the LFSRs, we set the lower part of the LFSR-B to the following hex values:

$$(b_7, b_6, \ldots, b_0) = (6D6F, 6854, 676E, 694A, 2064, 6B45, 7865, 6C41). \tag{12}$$

The hex values are the UTF8 encoding of the names of the authors.

An overview of how SNOW-V is used together with the $GHASH_H$ algorithm is shown in Figure 6. The padding of the Additional Authenticated Data (AAD) and how to concatenate the length of the AAD and the length of the ciphertext C and all other restrictions on plaintext length and change of IV from [Dwo07] remain. We have only defined a new way to derive the counter mode keystream, and the additional key and tag mask needed in the GCM algorithm.

5 Software implementation aspects

One important change in future telecom networks is the virtualization of the network functions. This puts new requirements on the crypto algorithms used to protect the traffic in that it needs to execute fast in a pure software implementation on modern CPUs. According to [ITU17], the minimum requirements related to 5G radio interface are 10 Gbps uplink and 20 Gbps downlink, at peak data rates. Classical encryption algorithms cannot reach these high speeds in pure software without any hardware support.

Nowadays, most of CPU vendors provide large registers and vectorized SIMD instructions, such as AVX2 set of instructions (intrinsics) that can execute over registers of up to 256 bits. Typical instructions include such functions as XOR, AND, nADD32, etc., applied to long registers, where, depending on the instruction, a single register can be represented as a vector of 8/16/32/64-bit values.

AES is one of the most widely used crypto algorithms and it has received special support from CPU vendors in the form of SIMD instructions (AES-NI for Intel) that makes it possible to execute AES quite fast even on user-grade laptops. Crypto ciphers SNOW 3G and ZUC, standardized in 4G, and other ciphers (to our best knowledge), cannot reach the speed even close to AES when AES-NI is used.

SNOW-V is designed to perform very fast in software, with the aim to utilize *currently available* SIMD instructions. However, even without AES-NI, SNOW-V can be implemented quite efficiently with 16/32/64-bit registers. Our take-away is that if a given platform supports AES-NI then other SIMD instructions are also likely supported. If AES-NI is not available then AES-256 will be much slower than SNOW-V, and actually, slower than SNOW 3G as well. This section is written with Intel intrinsics notation, but similar implementations can likely be made on other CPUs, e.g. AMD and ARM. A comprehensive guide on Intel's intrinsics can be found in [Int18].

The FSM part of SNOW-V is quite straightforward to implement using 128-bit registers `__m128i` and AES-NI intrinsic function `_mm_aesenc_si128()`. For 4 parallel arithmetic additions one can use `_mm_add_epi32()`[1]. The 16-byte permutation σ can be done with `_mm_shuffle_epi8()`.

[1]This intrinsic is intended for addition of signed integers but because most CPUs use two's complement representation for negative numbers, it will produce the correct results also for the unsigned addition needed in SNOW-V.

The key to an efficient implementation of the LFSRs is choosing the right data structures. We propose to store the content of the two LFSRs in two 256-bit registers `__m256i hi, lo`, such that:

$$\texttt{lo[127..0 bits]} = \{a_7, \ldots, a_0\} \qquad \texttt{hi[127..0 bits]} = \{a_{15}, \ldots, a_8\}$$
$$\texttt{lo[255..128 bits]} = \{b_7, \ldots, b_0\} \qquad \texttt{hi[255..128 bits]} = \{b_{15}, \ldots, b_8\}$$

To perform a single LFSR update (8 steps), we only need to calculate new values for one register, `hi_new=update(lo, hi)` while the other register update is a copy `lo_new=hi`.

Let `gA=0x990f` represents the generating polynomial $g^A(\alpha)$ of the field $\mathbb{F}^A_{2^{16}}$, without the term α^{16}. Then, multiplication of x by α in $\mathbb{F}^A_{2^{16}}$ can be done as follows: we first shift `x«1`, then, based on the 15th bit of the original `x`, we XOR the result with `gA`. This may be done with only 4 instructions, using 16-bit values

```
mul_alpha(uint16 x, uint16 gA):= (x<<1) xor (((signed int16)x >>15) and gA)
```

Note that the condition wether to `xor` with `gA` or not is implemented with the help of the 16-bit `mask = (signed int16)x » 15`, where the mask is created by the *arithmetical* shift of the *signed* `x` to the right by 15 positions. The arithmetical shift to the right results in propagation of the sign (15th) bit, thus forming the mask either `0xffff` in case the bit 15 was 1, or `0x0000`, otherwise.

The above trick can be applied to the combined 256-bit vector `lo` $= (b_7, \ldots, b_0, a_7, \ldots, a_0)$ to multiply the first half with α from the first base field $\mathbb{F}^A_{2^{16}}$ and the high part with β from the second base field $\mathbb{F}^B_{2^{16}}$, simultaneously. Here we need to use `_mm256_srai_epi16()` that performs *arithmetical* shift to the right of 16 16-bit *signed* integers represented in the combined 256-bit register `lo`. Obviously, the `and` operand should be done with the constant where the low 8 x 16-bit values are `gA=0x990f` and the second half contains `gB=0xc963`.

A similar idea is applied for multiplication of `hi` by α^{-1} and β^{-1}. In our reference implementation we found the way with only 4 instructions with the help of a non-trivial intrinsic `_mm256_sign_epi16()` – however, if that intrinsic is not available then there is an alternative solution with 5 instructions.

The results of the above two steps should be XORed together with the values at tap offsets 1 and 3 for LFSRs A and B, respectively. The latter part is just byte shuffling that can be done with `_mm256_blend_epi32()` and `_mm256_alignr_epi8()`, three instructions in total.

6 Software performance evaluation

In this section we give software performance benchmarks of SNOW-V-(GCM), implemented by us in C++ (Visual Studio 2017) utilizing AVX2/AES-NI/ PCLMULQDQ intrinsics. All performance tests were carried out on a user-grade laptop with Intel i7-8650U CPU @1.90GHz with Turbo Boost up to @4.20GHz, testing each algorithm on a single process/thread and with various lengths of the input plaintext. Before each encryption process, we perform a key/IV setup procedure. In the first place, we should compare SNOW-V with AES since it demonstrates the fastest speed on commodity CPUs

with available AES-NI instructions set. Perhaps, the second best choice algorithm, that is in various places serves as a backup for AES, is ChaCha20, and we should compare with that as well. We also include our best possible implementation of SNOW-3G in order to demonstrate the advantage of the new member of the SNOW family of stream ciphers.

For a fair and most challenging comparison we have downloaded the latest OpenSSL (3.0.0-dev, 2019-04-01) sources, and built it with the latest NASM and Visual Studio 2017 for that certain native x64 machine, with most possible optimizations switched on.

Implementations of the algorithms AES-256-(CBC, CTR, GCM) and ChaCha20-(Poly1305) in OpenSSL are the most recent and *highly optimized assembly codes* that utilize AVX2/AES-NI/PCLMULQDQ, instructions stitching, and other best practice optimization techniques. OpenSSL's command line tool was used for performance evaluation of the selected algorithms; it runs an algorithm for a chosen *number of seconds* and delivers the number of bytes per second processed, out of which we derive the speed in Gigabits/sec (Gbps).

In order to fully align our own measurements of SNOW-V with the numbers given by OpenSSL, we actually extracted and adopted the benchmarking code from OpenSSL's sources and did exactly the same way of measurements of SNOW-V. To negotiate pitfalls from the system and the OS, we benchmarked every considered algorithm several times, for 1-3 seconds, then picked the best values (a similar method is used in SUPERCOP benchmarking approach). The results are presented in Table 3.

Encryption only	Size of input plaintext (bytes)						
	16384	8192	4096	2048	1024	256	64
SNOW-3G-128 (C++)	9.22	9.07	8.89	8.50	7.81	5.38	2.37
AES-256-CBC (asm)	8.50	8.50	8.49	8.48	8.42	8.11	7.07
ChaCha20 (asm)	26.53	26.41	26.29	25.86	24.99	11.80	5.61
AES-256-CTR (asm)	35.06	34.82	34.16	32.94	30.95	22.67	11.32
SNOW-V (C++)	**58.25**	56.98	54.60	50.70	45.28	26.37	9.85
AEAD mode							
ChaCha20-Poly1305 (asm)	18.46	18.24	18.16	17.54	16.99	8.98	4.29
AES-256-GCM (asm)	34.42	33.86	32.74	30.49	27.22	17.32	8.54
SNOW-V-GCM (C++)	**38.91**	37.66	34.86	30.71	26.16	13.93	5.16

Table 3: Performance comparison of SNOW-V-(GCM) and best OpenSSL's algorithms. Performance values are given in Gbps.

For a **large plaintext**, SNOW-V outperforms AES-256-CBC by around 6.5 times, even though AES-256-CBC is implemented in an optimized assembly code with AES-NI. SNOW-V is also 2 times faster than ChaCha20. An encryption in AES-256-CTR can be done in "parallel", so that the technique called "instructions interleaving" makes it possible to speed up a lot. Even here, SNOW-V is 66% faster than AES-256-CTR.

We would like to note that running an algorithm for even 1 second includes a lot of system overhead such as OS's scheduler, switching to other hundreds of OS's processes and services, switching CPUs' contexts and affinity for load balancing, etc. When we tried to measure SNOW-V for a very small fraction of a second (basically, measuring a

single encryption), we have seen the speed goes up to 67Gbps (for encryption of 16384 bytes), which indicates that the OS's overhead is a significant factor.

One could notice from our measurements that AEAD mode of OpenSSL's AES-256-GCM is almost for "free" (35.06Gbps vs. 34.42Gbps). This was achieved by careful instructions interleaving and stitching techniques done in their optimized assembly code. In case of SNOW-V-GCM we, however, did not get GHASH for "free" in our C++ implementation (58.25Gbps vs. 38.91Gbps), but we think that an optimized assembly implementation of SNOW-V-(GCM) could potentially give a better result.

AVX512 is a new set of intrinsics utilizing wider 512-bit registers, and a subset of the AVX512 instructions is currently only available on high-end Intel CPUs. In this new set of intrinsics, there is an instruction to perform 4 AES encryption rounds in parallel `_mm512_aesenc_epi128()`, which would speed up AES-256-CTR by around x4 times.

For SNOW-V it will mainly reduce the number of instructions, as several operations in both the LFSR and the FSM can be combined in a single new AVX512 instruction. A first approximation is that the number of instructions will be approximately halved, but we need to evaluate SNOW-V on a full AVX512 implementation first. Since SNOW-V would only use half of the 512-bit wide registers, the second half could be used to perform another SNOW-V instance in parallel, with its own key and IV. Thus, as a rough estimate, the speed of SNOW-V could be increased by 2-4 times.

We also did small tests for ARM NEON implementations of SNOW-V and AES on an Apple A11 ARM processor[2]. Note that ARM architectures for devices are very different from Intel desktop/server architectures in the sense that there is not a standardized implementation of the SoC. It is up to the SoC designer to decide on e.g. cache configurations. At least, this test gives some indication of relative performance. The SNOW-V implementation is a single threaded code using NEON intrinsics in C, and the AES-CTR implementation is a single threaded assembly code from OpenSSL 1.1.1c. The results are presented in Table 4.

	Size of input plaintext (bytes)						
	16384	8192	4096	2048	1024	256	64
SNOW-V (C)	23.59	23.24	22.38	21.31	19.39	12.31	5.0
AES-CTR (asm)	15.97	15.87	15.59	15.08	14.34	10.62	5.04

Table 4: Performance comparison of SNOW-V and AES-CTR on an Apple A11 ARM processor. Performance values are given in Gbps.

7 Conclusions

A new 128-bit stream cipher called SNOW-V is presented. It follows the design principles of the previous ciphers in the SNOW family, but leverages the AES round function instruction support found in many modern CPUs. In a single thread implementation in software, SNOW-V outperforms AES in all comparable modes of operation for plaintext lengths above 256 bytes. Basic cryptanalysis of the new design is presented and

[2] Tests were run on an Apple iPhone X.

SNOW-V is argued to be resistant against these attacks. Finally, an AEAD mode of operation based on the well known GCM scheme is given. Test vectors and reference implementations are given in Appendix 3, Appendix 4, and Appendix 5. We also provide a brief hardware evaluation of SNOW-V in Appendix 6, including a 64-bit implementation utilizing a *single* AES core.

Acknowledgement

This work was in part financially supported by the Swedish Foundation for Strategic Research, grant RIT17-0005. Jing Yang is also supported by the scholarship from the National Digital Switching System Engineering and Technological Research Center, China.

References

[3GP] 3GPP. Work item on network functions virtualisation. `http://www.3gpp.org/more/1584-nfv`.

[Bab95] Steve Babbage. Improved "exhaustive search" attacks on stream ciphers. In *European Convention on Security and Detection*. IET, 1995.

[BG05] Olivier Billet and Henri Gilbert. Resistance of SNOW 2.0 against algebraic attacks. In Alfred Menezes, editor, *Topics in Cryptology – CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 19–28, San Francisco, CA, USA, February 14–18, 2005. Springer, Heidelberg, Germany.

[Bin] Bin Zhang et al. The ZUC-256 stream cipher. `http://www.is.cas.cn/ztzl2016/zouchongzhi/201801/W020180126529970733243.pdf`.

[BS00] Alex Biryukov and Adi Shamir. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In Tatsuaki Okamoto, editor, *Advances in Cryptology – ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 1–13, Kyoto, Japan, December 3–7, 2000. Springer, Heidelberg, Germany.

[BSW01] Alex Biryukov, Adi Shamir, and David Wagner. Real time cryptanalysis of A5/1 on a PC. In Bruce Schneier, editor, *Fast Software Encryption – FSE 2000*, volume 1978 of *Lecture Notes in Computer Science*, pages 1–18, New York, NY, USA, April 10–12, 2001. Springer, Heidelberg, Germany.

[BW99] Alex Biryukov and David Wagner. Slide attacks. In Lars R. Knudsen, editor, *Fast Software Encryption – FSE'99*, volume 1636 of *Lecture Notes in Computer Science*, pages 245–259, Rome, Italy, March 24–26, 1999. Springer, Heidelberg, Germany.

[CHJ02] D. Coppersmith, S. Halevi, and C.S. Jutla. Cryptanalysis of stream ciphers with linear masking. In M. Yung, editor, *Advances in Cryptology—CRYPTO 2002*, volume 2442, pages 515–532, 2002.

[CJM02] Philippe Chose, Antoine Joux, and Michel Mitton. Fast correlation attacks: An algorithmic point of view. In Lars R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 209–221, Amsterdam, The Netherlands, April 28 – May 2, 2002. Springer, Heidelberg, Germany.

[CJS01] Vladimor V. Chepyzhov, Thomas Johansson, and Ben J. M. Smeets. A simple algorithm for fast correlation attacks on stream ciphers. In Bruce Schneier, editor, *Fast Software Encryption – FSE 2000*, volume 1978 of *Lecture Notes in Computer Science*, pages 181–195, New York, NY, USA, April 10–12, 2001. Springer, Heidelberg, Germany.

[CKPS00] Nicolas Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In Bart Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407, Bruges, Belgium, May 14–18, 2000. Springer, Heidelberg, Germany.

[DK08] Orr Dunkelman and Nathan Keller. Treatment of the initial value in time-memory-data tradeoff attacks on stream ciphers. *Information Processing Letters*, 107(5):133–137, 2008.

[DS09] Itai Dinur and Adi Shamir. Cube attacks on tweakable black box polynomials. In Antoine Joux, editor, *Advances in Cryptology – EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 278–299, Cologne, Germany, April 26–30, 2009. Springer, Heidelberg, Germany.

[Dwo07] Morris J. Dworkin. Sp 800-38d. Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC. Technical report, Gaithersburg, MD, United States, 2007.

[EJ01] Patrik Ekdahl and Thomas Johansson. SNOW – a new stream cipher. In *Proceedings of First Open NESSIE Workshop, KU-Leuven*, 2001.

[EJ02] Patrik Ekdahl and Thomas Johansson. A New Version of the Stream Cipher SNOW. In Kaisa Nyberg and Howard M. Heys, editors, *Selected Areas in Cryptography*, volume 2595 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 2002.

[EJT07] Håkan Englund, Thomas Johansson, and Meltem Sönmez Turan. A framework for chosen IV statistical analysis of stream ciphers. In K. Srinathan, C. Pandu Rangan, and Moti Yung, editors, *Progress in Cryptology - INDOCRYPT 2007: 8th International Conference in Cryptology in India*, volume 4859 of *Lecture Notes in Computer Science*, pages 268–281, Chennai, India, December 9–13, 2007. Springer, Heidelberg, Germany.

[Gol97] Jovan Dj Golić. Cryptanalysis of alleged A5 stream cipher. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 239–255. Springer, 1997.

[HJB09] Martin Hell, Thomas Johansson, and Lennart Brynielsson. An overview of distinguishing attacks on stream ciphers. *Cryptography and Communications*, 1(1):71–94, 2009.

[HK18] Matthias Hamann and Matthias Krause. On stream ciphers with provable beyond-the-birthday-bound security against time-memory-data tradeoff attacks. *Cryptography and Communications*, 10(5):959–1012, 2018.

[HR02] P. Hawkes and G.G. Rose. Guess-and-determine attacks on SNOW. In K. Nyberg and H. Heys, editors, *Selected Areas in Cryptography—SAC 2002*, volume 2595, pages 37–46, 2002.

[HS05] Jin Hong and Palash Sarkar. New applications of time memory data tradeoffs. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 353–372. Springer, 2005.

[Int18] Intel Corporation. Intel Intrinsics Guide. Technical report, 2018. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`.

[ITU17] ITU. Minimum requirements related to technical performance for IMT-2020 radio interface(s). Version 1.0, ITU, 2017. `https://www.itu.int/pub/R-REP-M.2410-2017`.

[KY11] Aleksandar Kircanski and Amr M Youssef. On the sliding property of SNOW 3G and SNOW 2.0. *IET Information Security*, 5(4):199–206, 2011.

[Lai94] Xuejia Lai. Higher order derivatives and differential cryptanalysis. In *Communications and Cryptography*, pages 227–233. Springer, 1994.

[Max19] Alexander Maximov. AES MixColumn with 92 XOR gates. Cryptology ePrint Archive, Report 2019/833, 2019. `https://eprint.iacr.org/2019/833`.

[Mj06] O Saarinen Markku-juhani. Chosen-IV statistical attacks on eSTREAM stream ciphers. In *eSTREAM, ECRYPT Stream Cipher Project, Report 2006/013*. Citeseer, 2006.

[NW06] Kaisa Nyberg and Johan Wallén. Improved linear distinguishers for SNOW 2.0. In Matthew J. B. Robshaw, editor, *Fast Software Encryption – FSE 2006*, volume 4047 of *Lecture Notes in Computer Science*, pages 144–162, Graz, Austria, March 15–17, 2006. Springer, Heidelberg, Germany.

[oST01] National Institute of Standards and Technology. Advanced encryption standard. *NIST FIPS PUB 197*, 2001.

[RMTA18] Arash Reyhani-Masoleh, Mostafa Taha, and Doaa Ashmawy. Smashing the implementation records of AES S-box. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):298–336, May 2018. `https://tches.iacr.org/index.php/TCHES/article/download/884/835/`.

[SA318] 3GPP SA3. TR 33.841 study on supporting 256-bit algorithms for 5G., 2018. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3422.

[SAG06] SAGE. Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2. Version 1.1, ETSI/SAGE, 2006. https://www.gsma.com/aboutus/wp-content/uploads/2014/12/snow3gspec.pdf.

[SAG11] SAGE. Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. document 2: ZUC specification. Version 1.6, ETSI/SAGE, 2011. https://www.gsma.com/aboutus/wp-content/uploads/2014/12/eea3eia3zucv16.pdf.

[Sam00] Samsung Electronics Co., Ltd. STD90/MDL90 0.35μm 3.3V CMOS Standard Cell Library for Pure Logic/MDL Products Databook, 2000. https://www.digchip.com/datasheets/download_datasheet.php?id=935791&part-number=STD90.

[Sta10] Paul Stankovski. Greedy distinguishers and nonrandomness detectors. In Guang Gong and Kishan Chand Gupta, editors, *Progress in Cryptology - INDOCRYPT 2010: 11th International Conference in Cryptology in India*, volume 6498 of *Lecture Notes in Computer Science*, pages 210–226, Hyderabad, India, December 12–15, 2010. Springer, Heidelberg, Germany.

[Sta13] Paul Stankovski. *Cryptanalysis of Selected Stream Ciphers*, volume 50. Department of Electrical and Information Technology, Lund University, 2013.

[SWW16] Ling Sun, Wei Wang, and Meiqin Wang. MILP-aided bit-based division property for primitives with non-bit-permutation linear layers. *IACR Cryptology ePrint Archive*, 2016:811, 2016.

[SWW17] Ling Sun, Wei Wang, and Meiqin Wang. Automatic search of bit-based division property for ARX ciphers and word-based division property. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 128–157, Hong Kong, China, December 3–7, 2017. Springer, Heidelberg, Germany.

[TIHM17] Yosuke Todo, Takanori Isobe, Yonglin Hao, and Willi Meier. Cube attacks on non-blackbox polynomials based on division property. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part III*, volume 10403 of *Lecture Notes in Computer Science*, pages 250–279, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.

[Tod15] Yosuke Todo. Structural evaluation by generalized integral property. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 287–314, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany.

[UMHA16] Rei Ueno, Sumio Morioka, Naofumi Homma, and Takafumi Aoki. A high throughput/gate AES hardware architecture by compressing encryption and decryption datapaths — toward efficient CBC-mode implementation. Cryptology ePrint Archive, Report 2016/595, 2016. `https://eprint.iacr.org/2016/595`.

[Vie07] Michael Vielhaber. Breaking ONE.FIVIUM by AIDA an algebraic IV differential attack. Cryptology ePrint Archive, Report 2007/413, 2007. `http://eprint.iacr.org/2007/413`.

[WHT+18] Qingju Wang, Yonglin Hao, Yosuke Todo, Chaoyun Li, Takanori Isobe, and Willi Meier. Improved division property based cube attacks exploiting algebraic properties of superpoly. In *Annual International Cryptology Conference*, pages 275–305. Springer, 2018.

[ZXM15] Bin Zhang, Chao Xu, and Willi Meier. Fast correlation attacks over extension fields, large-unit linear approximation and cryptanalysis of SNOW 2.0. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 643–662, Santa Barbara, CA, USA, August 16–20, 2015. Springer, Heidelberg, Germany.

Appendices

1 Remarks about the maximum period of the LFSR structure

We can denote the LFSR state at time $t \geq 0$ as

$$S^{(t)} = (a_0^{(t)}, a_1^{(t)}, ..., a_{14}^{(t)}, a_{15}^{(t)}, b_0^{(t)}, b_1^{(t)}, ..., b_{14}^{(t)}, b_{15}^{(t)})$$

with 32 16-bit cells, i.e., 512 bits in total. If we consider the binary representation of the state, then the next state at $t+1$, $S^{(t+1)}$ can be written as,

$$S^{(t+1)} = S^{(t)}M$$

where M is the 512×512 state transition matrix.

Every cell of the next state except $a_{15}^{(t+1)}, b_{15}^{(t+1)}$ is determined by a shift from the neighboring cell, that is $a_i^{(t+1)} = a_{i+1}^{(t)}, b_i^{(t+1)} = b_{i+1}^{(t)}$ for $i = 0, 1, ...14$, and the corresponding binary state transition submatrices for such update are identity matrix M_I with size 16×16. As for a_{15}, b_{15}, we can rewrite them in the polynomial form. Suppose the bases for finite fields $\mathbb{F}_{2^{16}}^A$ and $\mathbb{F}_{2^{16}}^B$ are respectively $(1, \alpha, ..., \alpha^{15})$,$(1, \beta, ..., \beta^{15})$, then every state element can be expressed as a polynomial corresponding to the two bases.

For instance, a certain element $e \in \mathbb{F}_{2^{16}}^A$ can be interpreted as $e = e_0 + e_1\alpha + , ..., + e_{14}\alpha^{14} + e_{15}\alpha^{15}$, where e_i denotes the value at the i-th position of e. Then,

$$e\alpha \bmod g^A(\alpha) = (e_{15}\alpha^{16} + e_{14}\alpha^{15} + , ..., + e_1\alpha^2 + e_0\alpha) \bmod g^A(\alpha)$$

Since

$$\alpha^{16} \bmod g^A(\alpha) = \alpha^{15} + \alpha^{12} + \alpha^{11} + \alpha^8 + \alpha^3 + \alpha^2 + \alpha + 1,$$

$e\alpha \bmod g^A(\alpha)$ can be expanded and rearranged as,

$$\begin{aligned}
&= e_{15}(\alpha^{15}+\alpha^{12}+\alpha^{11}+\alpha^8+\alpha^3+\alpha^2+\alpha+1) + e_{14}\alpha^{15}+, ..., +e_1\alpha^2 + e_0\alpha \\
&= (e_{15}+e_{14})\alpha^{15} + e_{13}\alpha^{14} + e_{12}\alpha^{13} + (e_{15}+e_{11})\alpha^{12} + (e_{15}+e_{10})\alpha^{11} + \\
&e_9\alpha^{10} + e_8\alpha^9 + (e_{15}+e_7)\alpha^8 + e_6\alpha^7 + e_5\alpha^6 + e_4\alpha^5 + e_3\alpha^4 + (e_{15}+e_2)\alpha^3 \\
&+(e_{15}+e_1)\alpha^2 + (e_{15}+e_0)\alpha + e_{15} \\
&= (e_0, e_1, ..., e_{15}) M_\alpha (1, \alpha, ..., \alpha^{15})^{\mathrm{T}}
\end{aligned}$$

From which we can deduce the matrix

$$M_\alpha = \begin{matrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1
\end{matrix}$$

With the same method, we can also derive $M_{\alpha^{-1}}, M_\beta, M_{\beta^{-1}}$. Then we can rewrite the update for $a_{15}^{(t+1)}, b_{15}^{(t+1)}$ in a matrix form,

$$\begin{aligned}
a_{15}^{(t+1)} &= b_0^{(t)} M_I + a_0^{(t)} M_\alpha + a_1^{(t)} M_I + a_8^{(t)} M_{\alpha^{-1}} \\
b_{15}^{(t+1)} &= a_0^{(t)} M_I + b_0^{(t)} M_\beta + b_3^{(t)} M_I + b_8^{(t)} M_{\beta^{-1}}
\end{aligned}$$

Then the complete binary transition matrix for the LFSR update can be written as,

$$
M = \begin{array}{c|cccccccccccccccc}
 & 0 & 1 & ... & 7 & ... & 14 & 15 & ... & 18 & ... & 23 & ... & 30 & 31 \\
0 & & & & & & & M_\alpha & & & & & & & M_I \\
1 & M_I & & & & & & M_I & & & & & & & \\
2 & & M_I & & & & & & & & & & & & \\
... & & & ... & & & & & & & & & & & \\
8 & & & & M_I & & & M_{\alpha^{-1}} & & & & & & & \\
... & & & & & ... & & & & & & & & & \\
15 & & & & & & M_I & & & & & & & & \\
16 & & & & & & & M_I & & & & & & & M_\beta \\
... & & & & & & & & ... & & & & & & \\
19 & & & & & & & & & M_I & & & & & M_I \\
... & & & & & & & & & & ... & & & & \\
24 & & & & & & & & & & & M_I & & & M_{\beta^{-1}} \\
... & & & & & & & & & & & & ... & & \\
31 & & & & & & & & & & & & & M_I &
\end{array}
$$

where every element in the 32×32 matrix is a 16×16 matrix and all the other empty places are 16×16 zero matrices. Then we can get the 512×512 transition matrix and some mathematical tools, such as Sagemath, can be employed to verify whether it is primitive. We employ the built-in function `charpoly()` in Sagemath to get the characteristic polynomial, which is,

$$
\begin{aligned}
&m(x) = \\
&x^{512} + x^{491} + x^{489} + x^{480} + x^{478} + x^{475} + x^{474} + x^{473} + x^{472} + x^{468} + x^{467} + \\
&x^{466} + x^{464} + x^{455} + x^{453} + x^{452} + x^{445} + x^{444} + x^{443} + x^{441} + x^{438} + x^{437} + \\
&x^{434} + x^{433} + x^{429} + x^{426} + x^{425} + x^{424} + x^{423} + x^{422} + x^{420} + x^{419} + x^{418} + \\
&x^{417} + x^{416} + x^{415} + x^{410} + x^{409} + x^{407} + x^{405} + x^{404} + x^{402} + x^{394} + x^{393} + \\
&x^{391} + x^{390} + x^{385} + x^{384} + x^{383} + x^{382} + x^{381} + x^{380} + x^{374} + x^{371} + x^{369} + \\
&x^{368} + x^{367} + x^{366} + x^{365} + x^{363} + x^{361} + x^{360} + x^{358} + x^{357} + x^{354} + x^{351} + \\
&x^{345} + x^{344} + x^{341} + x^{339} + x^{337} + x^{336} + x^{334} + x^{330} + x^{325} + x^{324} + x^{321} + \\
&x^{317} + x^{315} + x^{314} + x^{313} + x^{311} + x^{310} + x^{309} + x^{308} + x^{307} + x^{305} + x^{302} + \\
&x^{299} + x^{296} + x^{292} + x^{291} + x^{284} + x^{283} + x^{281} + x^{280} + x^{279} + x^{276} + x^{275} + \\
&x^{273} + x^{271} + x^{267} + x^{264} + x^{263} + x^{262} + x^{260} + x^{259} + x^{258} + x^{257} + x^{256} + \\
&x^{254} + x^{253} + x^{251} + x^{249} + x^{248} + x^{247} + x^{246} + x^{245} + x^{243} + x^{242} + x^{240} + \\
&x^{238} + x^{236} + x^{229} + x^{225} + x^{218} + x^{217} + x^{216} + x^{215} + x^{214} + x^{209} + x^{208} + \\
&x^{207} + x^{205} + x^{204} + x^{203} + x^{201} + x^{198} + x^{193} + x^{192} + x^{190} + x^{189} + x^{187} + \\
&x^{186} + x^{185} + x^{180} + x^{178} + x^{176} + x^{173} + x^{170} + x^{169} + x^{167} + x^{165} + x^{164} + \\
&x^{163} + x^{162} + x^{160} + x^{159} + x^{155} + x^{152} + x^{151} + x^{150} + x^{149} + x^{148} + x^{147} + \\
&x^{145} + x^{144} + x^{142} + x^{141} + x^{136} + x^{134} + x^{131} + x^{126} + x^{125} + x^{123} + x^{122} + \\
&x^{121} + x^{118} + x^{117} + x^{114} + x^{113} + x^{109} + x^{106} + x^{105} + x^{104} + x^{103} + x^{101} + \\
&x^{100} + x^{96} + x^{95} + x^{94} + x^{91} + x^{87} + x^{86} + x^{85} + x^{83} + x^{82} + x^{81} + x^{78} + \\
&x^{76} + x^{74} + x^{73} + x^{69} + x^{68} + x^{67} + x^{66} + x^{64} + x^{63} + x^{62} + x^{61} + x^{59} + \\
&x^{56} + x^{54} + x^{53} + x^{50} + x^{49} + x^{47} + x^{42} + x^{38} + x^{36} + x^{35} + x^{33} + x^{25} + \\
&x^{24} + x^{23} + x^{20} + x^{16} + x^{15} + x^{14} + x^{13} + x^{11} + x^9 + x^6 + x + 1
\end{aligned}
$$

Then we can verify it primitive by Sagemath, which indicates the LFSR structure has the maximum period $2^{512}-1$.

2 Details on the exampled linear approximation of the FSM for the proposed algorithm

In this Section we provide more details on the exampled approximation given in Section 3.4.4. We, again, assume 8-bit adders $\boxplus_8$, instead of $\boxplus_{32}$. Recall the exampled approximation where $\Lambda_0 = \Lambda_1 = [1\ 1\ 1\ 1\ 0\ 0\ ...\ 0]$, and $\Lambda_2 = \Lambda_1 \cdot L^{-1}$. We can thus analyse the following expressions on three consecutive keystream words $z^{(t-1)}$, $z^{(t)}$, and $z^{(t+1)}$:

$$
\begin{aligned}
z^{(t-1)} &= (S^{-1}(L^{-1} \cdot \hat{R2}) \boxplus_8 T1^{(t-1)}) \oplus S^{-1}(L^{-1} \cdot \hat{R3}),\\
z^{(t)} &= (\hat{R1} \boxplus_8 T1^{(t)}) \oplus \hat{R2},\\
L^{-1}z^{(t+1)} &= L^{-1}(\sigma(\hat{R2} \boxplus_8 (\hat{R3} \oplus T2^{(t)})) \boxplus_8 T1^{(t+1)}) \oplus S(\hat{R1}).
\end{aligned}
$$

The exampled approximation is the sum ($\oplus$) of the first 4 bytes of the above 3 expressions. In order to make it easier to follow our derivations we give explicit matrices for L^{-1} and $L^{-1}\sigma$ in Listing 1.

Listing 1: Matrices L^{-1} (left) and $L^{-1}\sigma$ (right).

```
e b d 9| 0 0 0 0| 0 0 0 0| 0 0 0 0      e 0 0 0| b 0 0 0| d 0 0 0| 9 0 0 0
0 0 0 0| 0 0 0 0| 0 0 0 0| 9 e b d      0 0 0 9| 0 0 0 e| 0 0 0 b| 0 0 0 d
0 0 0 0| 0 0 0 0| d 9 e b| 0 0 0 0      0 0 d 0| 0 0 9 0| 0 0 e 0| 0 0 b 0
0 0 0 0| b d 9 e| 0 0 0 0| 0 0 0 0      0 b 0 0| 0 d 0 0| 0 9 0 0| 0 e 0 0
-----------------------------------      -----------------------------------
0 0 0 0| e b d 9| 0 0 0 0| 0 0 0 0      0 e 0 0| 0 b 0 0| 0 d 0 0| 0 9 0 0
9 e b d| 0 0 0 0| 0 0 0 0| 0 0 0 0      9 0 0 0| e 0 0 0| b 0 0 0| d 0 0 0
0 0 0 0| 0 0 0 0| 0 0 0 0| d 9 e b      0 0 0 d| 0 0 0 9| 0 0 0 e| 0 0 0 b
0 0 0 0| 0 0 0 0| b d 9 e| 0 0 0 0      0 0 b 0| 0 0 d 0| 0 0 9 0| 0 0 e 0
-----------------------------------      -----------------------------------
0 0 0 0| 0 0 0 0| e b d 9| 0 0 0 0      0 0 e 0| 0 0 b 0| 0 0 d 0| 0 0 9 0
0 0 0 0| 9 e b d| 0 0 0 0| 0 0 0 0      0 9 0 0| 0 e 0 0| 0 b 0 0| 0 d 0 0
d 9 e b| 0 0 0 0| 0 0 0 0| 0 0 0 0      d 0 0 0| 9 0 0 0| e 0 0 0| b 0 0 0
0 0 0 0| 0 0 0 0| 0 0 0 0| b d 9 e      0 0 0 b| 0 0 0 d| 0 0 0 9| 0 0 0 e
-----------------------------------      -----------------------------------
0 0 0 0| 0 0 0 0| 0 0 0 0| e b d 9      0 0 0 e| 0 0 0 b| 0 0 0 d| 0 0 0 9
0 0 0 0| 0 0 0 0| 9 e b d| 0 0 0 0      0 0 9 0| 0 0 e 0| 0 0 b 0| 0 0 d 0
0 0 0 0| d 9 e b| 0 0 0 0| 0 0 0 0      0 d 0 0| 0 9 0 0| 0 e 0 0| 0 b 0 0
b d 9 e| 0 0 0 0| 0 0 0 0| 0 0 0 0      b 0 0 0| d 0 0 0| 9 0 0 0| e 0 0 0
```

We first note that with $\boxplus_8$ for any 16-byte expression W we have:

$$
\sum_{i=0}^{3}[L^{-1}\sigma \cdot W]_i = \sum_{i=0}^{3}\sum_{j=0}^{3} c_{4j+(-i \bmod 4)} W_{4j+(-i \bmod 4)} = \sum_{i=0}^{3}\sum_{j=0}^{3} c_{4i+j} W_{4i+j},
$$

where the coefficients are:

$$
c = \texttt{[e b d 9 b d 9 e d 9 e b 9 e b d]}.
$$

Then we get the following linear approximation of the FSM:

$$\sum_{i=0}^{3}\left[L^{-1}z^{(t+1)}\oplus z^{(t)}\oplus z^{(t-1)}\right]_i = \sum_{i=0}^{3}\left(\underbrace{S(\hat{R}1_i)\oplus(\hat{R}1_i\boxplus_8 T1_i^{(t)})\oplus T1_i^{(t)}}_{\text{Noise } N1_i}\oplus T1_i^{(t)}\right)$$

$$\oplus\sum_{i=0}^{3}\left[\underbrace{\hat{R}2_i\oplus\sum_{j=0}^{3}c_{4i+j}[(\hat{R}2_{4i+j}\boxplus_8(\hat{R}3_{4i+j}\oplus\hat{X}_{i,j})\boxplus_8\hat{Y}_{i,j})\oplus\hat{X}_{i,j}\oplus\hat{Y}_{i,j}}_{\text{Linear part } C_i \text{ of } N2_i}\oplus\hat{X}_{i,j}\oplus\hat{Y}_{i,j}]\right]$$

$$\oplus\sum_{i=0}^{3}\left[\left(S^{-1}(\underbrace{\sum_{j=0}^{3}c_{4i+j}\hat{R}2_{4i+j}}_{\text{Linear part } A_i \text{ of } N2_i})\boxplus_8 T1_i^{(t-1)}\right)\oplus S^{-1}(\underbrace{\sum_{j=0}^{3}c_{4i+j}\hat{R}3_{4i+j}}_{\text{Linear part } B_i \text{ of } N2_i})\right],$$

where $\hat{X}_{i,j}=T2_{4i+j}^{(t)}$ and $\hat{Y}_{i,j}=(\sigma^{-1}T1^{(t+1)})_{4i+j}$.

In the above it now becomes clear that we need to compute byte-oriented distributions of two *independent* noise variables $N1$ and $N2$. The first noise is trivial to compute, while the second $N2$ is a bit more complicated, but it can be computed with the technique mentioned in Section 3.4.1. I.e., for every i we first compute 4 partial joint distributions $D_{i,j}(A_{i,j}|B_{i,j}|C_{i,j})$ of linear parts $A|B|C$ for each $j=0...3$, then we use Fast Walsh-Hadamard Transform to perform the XOR-convolutions in order to get the complete 24-bit joint distribution $D_i(A_i|B_i|C_i)$ for all possible 32-bit input arguments running over $j=0,1,2,3$, for that certain i. Having that joint distribution $D_i(A_i|B_i|C_i)$ being computed it is then trivial to compute the 8-bit sub-noise distribution $N2_i$: we simply loop over all possible choices of $A_i, B_i, C_i, T1_i^{(t-1)}$, and approximate $...\boxplus_8 T1_i^{(t-1)}$ by adding that term outside of S^{-1}, near by the linear part A. I.e., we compute

$$\Pr\{N2_i=(S^{-1}(A_i)\boxplus_8 T1_i^{(t-1)})\oplus T1_i^{(t-1)}\oplus S^{-1}(B_i)\oplus C_i\}\ +=\ \frac{1}{2^8}\cdot D_i(A_i|B_i|C_i).$$

We repeat the above for each i, and the total noise $N2$ is the XOR-convolution of the four *independent* sub-distributions $N2_i$.

We would like to note that the first sub-distribution $N2_0$ has a smaller bias since 4 bytes of the term $\hat{R}2_i$ in the linear part C of $N2$ are added to $N2_0$, while the remaining 3 sub-distributions are free from these terms. This means that $N2_0$ includes 4 approximations of type $n3$ with a smaller bias ($\approx 2^{-3.3}$) than those of the types $n1(\approx 2^{3.1})$ and $n2(\approx 2^{-2.9})$. Our computation results are as follows: $\epsilon(N2_{i=0})\approx 2^{-53.828334}$,

$\epsilon(N2_{i=1..3}) \approx 2^{-30.382642}$, $\epsilon(N1_{i=0..3}) \approx 2^{-2.920807}$, and

$$\epsilon(\sum_{i=0}^{3} N1_i) \approx 2^{-26.446376},$$
$$\epsilon(\sum_{i=0}^{3} N2_i) \approx 2^{-187.562693},$$
$$\epsilon(N_{tot}) \approx 2^{-214.848865},$$
$$\epsilon(2 \times N_{tot}) \approx 2^{-429.674887}.$$

3 Test Vectors

This section presents test vectors for SNOW-V with three different keys and IVs. The vectors are written with the **least significant byte** of the 128-bit word appearing to the left in the row. For the keys, the lower 128-bit part is written on the first row, followed by the high part on the second row.

Listing 2: Test vectors for SNOW-V.

```
== SNOW-V test vectors #1:
key = 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
iv  = 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Initialization phase, z =
      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
      a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5
      ea ea ea ea eb eb eb eb eb eb eb eb eb eb eb eb
      55 f7 f7 c2 e8 e8 dd 4a e8 dd 4a e8 dd 4a e8 e8
      c7 2a 23 bf e8 93 73 30 23 bc 66 ec 94 d2 eb b2
      a7 dd ca f3 13 87 61 02 6e ad f4 2b 54 e3 ef cf
      6a 67 62 3e 6f 8a f9 79 1e cd 81 83 c5 86 8e 3a
      45 10 1e 83 a2 c6 dd eb 40 86 38 2d ac fb 3b 65
      3c c4 df 56 ec bf c1 06 6d ac 02 c5 0a 68 3c fe
      0c cb e1 de 2e 41 af da 70 98 d5 60 19 20 06 98
      53 cd 98 69 c7 78 ca de d7 db 45 9b 6f 45 8b 10
      8d 94 0b e5 9f bd b1 61 c1 21 fc 29 7a 3d 0a 15
      26 13 2c 14 9e af 12 cc d3 2f 35 76 f6 43 68 94
      0e 75 be 09 54 18 1e f5 8a 60 a9 a9 54 3a 05 ff
      dc 77 a4 97 23 eb 65 6a e1 8f 28 2c f1 de 1d 00
Keystream phase, z =
      69 ca 6d af 9a e3 b7 2d b1 34 a8 5a 83 7e 41 9d
      ec 08 aa d3 9d 7b 0f 00 9b 60 b2 8c 53 43 00 ed
      84 ab f5 94 fb 08 a7 f1 f3 a2 df 18 e6 17 68 3b
      48 1f a3 78 07 9d cf 04 db 53 b5 d6 29 a9 eb 9d
      03 1c 15 9d cc d0 a5 0c 4d 5d bf 51 15 d8 70 39
      c0 d0 3c a1 37 0c 19 40 03 47 a0 b4 d2 e9 db e5
      cb ca 60 82 14 a2 65 82 cf 68 09 16 b3 45 13 21
      95 4f df 30 84 af 02 f6 a8 e2 48 1d e6 bf 82 79


== SNOW-V test vectors #2:
key = ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
```

```
       ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
iv  = ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
Initialization phase, z =
       ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
       d3 07 d2 07 d3 07 d2 07 d3 07 2d f8 2e f8 2d f8
       65 f6 62 f6 65 f6 62 f6 65 f6 62 f6 65 f6 62 f6
       fe 86 fe 86 f5 2d f2 2d 31 96 d7 54 6a e8 6a e8
       8b d8 8a a5 c8 29 c6 26 7c 51 37 97 bf 9a c8 7c
       21 c0 4a 14 e4 1c 34 95 d0 9c 96 e5 48 60 89 81
       7c ce 64 29 1a cf 8f 4a 06 ca 55 65 3f c4 93 97
       0a f9 1c 75 0f d3 80 e3 48 6b ff e5 c7 bb e3 d4
       89 60 89 a2 e6 f0 7c 2c 92 ed 62 ed 9d 43 61 98
       ff 04 bf 72 41 c0 7f 6b 17 fd 90 c8 8a 61 bf ca
       97 88 78 33 20 08 2f f6 f9 34 45 18 6e 71 bc bc
       7e 17 b4 ff 42 3a 2e 2c c7 c5 0f 84 5d 9b b3 ee
       32 40 8c 85 58 e0 d2 7e f5 a3 a8 d7 63 32 25 dc
       a2 93 73 c3 48 2b 3f 1a d3 3b b4 57 a3 0d 7f e4
       72 e0 95 5b 9a 83 3a 3f db 98 68 56 35 80 b4 b0
       94 9f be 85 a4 e5 35 7f bf 75 e9 86 4d 2c 7b a1
Keystream phase, z =
       30 76 09 fb 10 10 12 54 4b c1 75 e3 17 fb 25 ff
       33 0d 0d e2 5a f6 aa d1 05 05 b8 9b 1e 09 a8 ec
       dd 46 72 cc bb 98 c7 f2 c4 e2 4a f5 27 28 36 c8
       7c c7 3a 81 76 b3 9c e9 30 3b 3e 76 4e 9b e3 e7
       48 f7 65 1a 7c 7e 81 3f d5 24 90 23 1e 56 f7 c1
       44 e4 38 e7 77 11 a6 b0 ba fb 60 45 0c 62 d7 d9
       b9 24 1d 12 44 fc b4 9d a1 e5 2b 80 13 de cd d4
       86 04 ff fc 62 67 6e 70 3b 3a b8 49 cb a6 ea 09


== SNOW-V test vectors #3:
key = 50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
       0a 1a 2a 3a 4a 5a 6a 7a 8a 9a aa ba ca da ea fa
iv  = 01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10
Initialization phase, z =
       0a 1a 2a 3a 4a 5a 6a 7a 8a 9a aa ba ca da ea fa
       66 d4 2d 92 ac 52 b6 44 63 3c c3 71 c3 91 c6 24
       a2 d7 ea be 3f 04 8e 50 00 b1 7b 74 2f 34 5e 49
       96 a7 34 ed fd 07 46 9d c8 f9 a2 91 fc 13 76 73
       58 c8 70 73 d8 a2 a1 bd 03 e7 a1 4c c7 b7 db 89
       7e 86 eb 71 d6 dc 00 99 d1 31 e3 1b 54 c5 3e f8
       a8 ca ff 06 0d c0 9e 67 cc 95 62 16 17 19 8c f2
       c0 99 3a 55 f3 e2 d7 8d 6a f7 e1 57 0f a1 63 02
       39 8f a0 7e ab a2 73 89 94 f9 ac 3e 8e b1 ff 64
       15 32 31 6a 42 5c 12 a6 39 ce 79 cb 30 43 47 1e
       2e 7a 44 fd ad 23 77 5a f1 61 1c ca 5b b2 1e 95
       93 69 c8 20 a9 37 d5 c8 b6 7a df 84 45 5e 13 c3
       c1 0f 8d b5 fb 37 08 31 11 d1 c8 44 6e a2 ac 9e
       13 ac 34 20 7b 01 b7 ab d3 57 02 a1 ed 98 9b dc
       0b 15 43 a4 74 26 2c 76 a3 e2 73 57 28 4b dc 67
       7b 79 91 96 cf 6b 76 27 f8 dd a1 89 bb af dc 93
Keystream phase, z =
       aa 81 ea fb 8b 86 16 ce 3e 5c e2 22 24 61 c5 0a
       6a b4 48 77 56 de 4b d3 1c 90 4f 3d 97 8a fe 56
       33 4f 10 dd df 2b 95 31 76 9a 71 05 0b e4 38 5f
       c2 b6 19 2c 7a 85 7b e8 b4 fc 28 b7 09 f0 8f 11
       f2 06 49 e2 ee f2 49 80 f8 6c 4c 11 36 41 fe d2
       f3 f6 fa 2b 91 95 12 06 b8 01 db 15 46 65 17 a6
```

```
      33 0a dd a6 b3 5b 26 5e fd 72 2e 86 77 b4 8b fc
      15 b4 41 18 de 52 d0 73 b0 ad 0f e7 59 4d 62 91
```

Listing 3: Test vectors for SNOW-V-GCM.

```
== SNOW-V test vectors #1:
key = 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
iv  = 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Initialization phase, z =
      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
      a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5
      ea ea ea ea eb eb eb eb eb eb eb eb eb eb eb eb
      55 f7 f7 c2 e8 e8 dd 4a e8 dd 4a e8 dd 4a e8 e8
      c7 2a 23 bf e8 93 73 30 23 bc 66 ec 94 d2 eb b2
      a7 dd ca f3 13 87 61 02 6e ad f4 2b 54 e3 ef cf
      6a 67 62 3e 6f 8a f9 79 1e cd 81 83 c5 86 8e 3a
      45 10 1e 83 a2 c6 dd eb 40 86 38 2d ac fb 3b 65
      3c c4 df 56 ec bf c1 06 6d ac 02 c5 0a 68 3c fe
      0c cb e1 de 2e 41 af da 70 98 d5 60 19 20 06 98
      53 cd 98 69 c7 78 ca de d7 db 45 9b 6f 45 8b 10
      8d 94 0b e5 9f bd b1 61 c1 21 fc 29 7a 3d 0a 15
      26 13 2c 14 9e af 12 cc d3 2f 35 76 f6 43 68 94
      0e 75 be 09 54 18 1e f5 8a 60 a9 a9 54 3a 05 ff
      dc 77 a4 97 23 eb 65 6a e1 8f 28 2c f1 de 1d 00
Keystream phase, z =
      69 ca 6d af 9a e3 b7 2d b1 34 a8 5a 83 7e 41 9d
      ec 08 aa d3 9d 7b 0f 00 9b 60 b2 8c 53 43 00 ed
      84 ab f5 94 fb 08 a7 f1 f3 a2 df 18 e6 17 68 3b
      48 1f a3 78 07 9d cf 04 db 53 b5 d6 29 a9 eb 9d
      03 1c 15 9d cc d0 a5 0c 4d 5d bf 51 15 d8 70 39
      c0 d0 3c a1 37 0c 19 40 03 47 a0 b4 d2 e9 db e5
      cb ca 60 82 14 a2 65 82 cf 68 09 16 b3 45 13 21
      95 4f df 30 84 af 02 f6 a8 e2 48 1d e6 bf 82 79


== SNOW-V test vectors #2:
key = ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
      ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
iv  = ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
Initialization phase, z =
      ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
      d3 07 d2 07 d3 07 d2 07 d3 07 2d f8 2e f8 2d f8
      65 f6 62 f6 65 f6 62 f6 65 f6 62 f6 65 f6 62 f6
      fe 86 fe 86 f5 2d f2 2d 31 96 d7 54 6a e8 6a e8
      8b d8 8a a5 c8 29 c6 26 7c 51 37 97 bf 9a c8 7c
      21 c0 4a 14 e4 1c 34 95 d0 9c 96 e5 48 60 89 81
      7c ce 64 29 1a cf 8f 4a 06 ca 55 65 3f c4 93 97
      0a f9 1c 75 0f d3 80 e3 48 6b ff e5 c7 bb e3 d4
      89 60 89 a2 e6 f0 7c 2c 92 ed 62 ed 9d 43 61 98
      ff 04 bf 72 41 c0 7f 6b 17 fd 90 c8 8a 61 bf ca
      97 88 78 33 20 08 2f f6 f9 34 45 18 6e 71 bc bc
      7e 17 b4 ff 42 3a 2e 2c c7 c5 0f 84 5d 9b b3 ee
      32 40 8c 85 58 e0 d2 7e f5 a3 a8 d7 63 32 25 dc
      a2 93 73 c3 48 2b 3f 1a d3 3b b4 57 a3 0d 7f e4
      72 e0 95 5b 9a 83 3a 3f db 98 68 56 35 80 b4 b0
      94 9f be 85 a4 e5 35 7f bf 75 e9 86 4d 2c 7b a1
```

```
Keystream phase, z =
       30 76 09 fb 10 10 12 54 4b c1 75 e3 17 fb 25 ff
       33 0d 0d e2 5a f6 aa d1 05 05 b8 9b 1e 09 a8 ec
       dd 46 72 cc bb 98 c7 f2 c4 e2 4a f5 27 28 36 c8
       7c c7 3a 81 76 b3 9c e9 30 3b 3e 76 4e 9b e3 e7
       48 f7 65 1a 7c 7e 81 3f d5 24 90 23 1e 56 f7 c1
       44 e4 38 e7 77 11 a6 b0 ba fb 60 45 0c 62 d7 d9
       b9 24 1d 12 44 fc b4 9d a1 e5 2b 80 13 de cd d4
       86 04 ff fc 62 67 6e 70 3b 3a b8 49 cb a6 ea 09


== SNOW-V test vectors #3:
key = 50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
      0a 1a 2a 3a 4a 5a 6a 7a 8a 9a aa ba ca da ea fa
iv  = 01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10
Initialization phase, z =
       0a 1a 2a 3a 4a 5a 6a 7a 8a 9a aa ba ca da ea fa
       66 d4 2d 92 ac 52 b6 44 63 3c c3 71 c3 91 c6 24
       a2 d7 ea be 3f 04 8e 50 00 b1 7b 74 2f 34 5e 49
       96 a7 34 ed fd 07 46 9d c8 f9 a2 91 fc 13 76 73
       58 c8 70 73 d8 a2 a1 bd 03 e7 a1 4c c7 b7 db 89
       7e 86 eb 71 d6 dc 00 99 d1 31 e3 1b 54 c5 3e f8
       a8 ca ff 06 0d c0 9e 67 cc 95 62 16 17 19 8c f2
       c0 99 3a 55 f3 e2 d7 8d 6a f7 e1 57 0f a1 63 02
       39 8f a0 7e ab a2 73 89 94 f9 ac 3e 8e b1 ff 64
       15 32 31 6a 42 5c 12 a6 39 ce 79 cb 30 43 47 1e
       2e 7a 44 fd ad 23 77 5a f1 61 1c ca 5b b2 1e 95
       93 69 c8 20 a9 37 d5 c8 b6 7a df 84 45 5e 13 c3
       c1 0f 8d b5 fb 37 08 31 11 d1 c8 44 6e a2 ac 9e
       13 ac 34 20 7b 01 b7 ab d3 57 02 a1 ed 98 9b dc
       0b 15 43 a4 74 26 2c 76 a3 e2 73 57 28 4b dc 67
       7b 79 91 96 cf 6b 76 27 f8 dd a1 89 bb af dc 93
Keystream phase, z =
       aa 81 ea fb 8b 86 16 ce 3e 5c e2 22 24 61 c5 0a
       6a b4 48 77 56 de 4b d3 1c 90 4f 3d 97 8a fe 56
       33 4f 10 dd df 2b 95 31 76 9a 71 05 0b e4 38 5f
       c2 b6 19 2c 7a 85 7b e8 b4 fc 28 b7 09 f0 8f 11
       f2 06 49 e2 ee f2 49 80 f8 6c 4c 11 36 41 fe d2
       f3 f6 fa 2b 91 95 12 06 b8 01 db 15 46 65 17 a6
       33 0a dd a6 b3 5b 26 5e fd 72 2e 86 77 b4 8b fc
       15 b4 41 18 de 52 d0 73 b0 ad 0f e7 59 4d 62 91
```

4 SNOW-V 32-bit Reference Implementation in C/C++

```
// SNOW-V 32-bit Reference Implementation (Endianness-free)
#include <stdint.h>
#include <stdlib.h>

typedef uint8_t u8;
typedef uint16_t u16;
typedef uint32_t u32;

u8 SBox[256] =
{
       0x63,0x7C,0x77,0x7B,0xF2,0x6B,0x6F,0xC5,0x30,0x01,0x67,0x2B,0xFE,0xD7,0xAB,0x76,
```

```
12          0xCA,0x82,0xC9,0x7D,0xFA,0x59,0x47,0xF0,0xAD,0xD4,0xA2,0xAF,0x9C,0xA4,0x72,0xC0,
13          0xB7,0xFD,0x93,0x26,0x36,0x3F,0xF7,0xCC,0x34,0xA5,0xE5,0xF1,0x71,0xD8,0x31,0x15,
14          0x04,0xC7,0x23,0xC3,0x18,0x96,0x05,0x9A,0x07,0x12,0x80,0xE2,0xEB,0x27,0xB2,0x75,
15          0x09,0x83,0x2C,0x1A,0x1B,0x6E,0x5A,0xA0,0x52,0x3B,0xD6,0xB3,0x29,0xE3,0x2F,0x84,
16          0x53,0xD1,0x00,0xED,0x20,0xFC,0xB1,0x5B,0x6A,0xCB,0xBE,0x39,0x4A,0x4C,0x58,0xCF,
17          0xD0,0xEF,0xAA,0xFB,0x43,0x4D,0x33,0x85,0x45,0xF9,0x02,0x7F,0x50,0x3C,0x9F,0xA8,
18          0x51,0xA3,0x40,0x8F,0x92,0x9D,0x38,0xF5,0xBC,0xB6,0xDA,0x21,0x10,0xFF,0xF3,0xD2,
19          0xCD,0x0C,0x13,0xEC,0x5F,0x97,0x44,0x17,0xC4,0xA7,0x7E,0x3D,0x64,0x5D,0x19,0x73,
20          0x60,0x81,0x4F,0xDC,0x22,0x2A,0x90,0x88,0x46,0xEE,0xB8,0x14,0xDE,0x5E,0x0B,0xDB,
21          0xE0,0x32,0x3A,0x0A,0x49,0x06,0x24,0x5C,0xC2,0xD3,0xAC,0x62,0x91,0x95,0xE4,0x79,
22          0xE7,0xC8,0x37,0x6D,0x8D,0xD5,0x4E,0xA9,0x6C,0x56,0xF4,0xEA,0x65,0x7A,0xAE,0x08,
23          0xBA,0x78,0x25,0x2E,0x1C,0xA6,0xB4,0xC6,0xE8,0xDD,0x74,0x1F,0x4B,0xBD,0x8B,0x8A,
24          0x70,0x3E,0xB5,0x66,0x48,0x03,0xF6,0x0E,0x61,0x35,0x57,0xB9,0x86,0xC1,0x1D,0x9E,
25          0xE1,0xF8,0x98,0x11,0x69,0xD9,0x8E,0x94,0x9B,0x1E,0x87,0xE9,0xCE,0x55,0x28,0xDF,
26          0x8C,0xA1,0x89,0x0D,0xBF,0xE6,0x42,0x68,0x41,0x99,0x2D,0x0F,0xB0,0x54,0xBB,0x16
27  };
28  u8 Sigma[16] = {0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15};
29  u32 AesKey1[4] = { 0, 0, 0, 0 };
30  u32 AesKey2[4] = { 0, 0, 0, 0 };
31
32  #define MAKEU32(a, b) (((u32)(a) << 16) | ((u32)(b) ))
33  #define MAKEU16(a, b) (((u16)(a) << 8) | ((u16)(b) ))
34
35  struct SnowV32
36  {
37          u16 A[16], B[16]; // LFSR
38          u32 R1[4], R2[4], R3[4]; // FSM
39
40          void aes_enc_round(u32 * result, u32 * state, u32 * roundKey)
41          {
42  #define ROTL32(word32, offset) ((word32 << offset) | (word32 >> (32 - offset)))
43  #define SB(index, offset)  (((u32)(sb[(index) % 16])) << (offset * 8))
44  #define MKSTEP(j)\
45          w = SB(j * 4 + 0, 3) | SB(j * 4 + 5, 0) | SB(j * 4 + 10, 1) | SB(j * 4 + 15, 2)↩
                ;\
46          t = ROTL32(w, 16) ^ ((w << 1) & 0xfefefefeUL) ^ (((w >> 7) & 0x01010101UL) * 0↩
                x1b);\
47          result[j] = roundKey[j] ^ w ^ t ^ ROTL32(t, 8)
48
49                  u32 w, t;
50                  u8 sb[16];
51                  for (int i = 0; i < 4; i++)
52                          for (int j = 0; j < 4; j++)
53                                  sb[i * 4 + j] = SBox[(state[i] >> (j * 8)) & 0xff];
54
55                  MKSTEP(0);
56                  MKSTEP(1);
57                  MKSTEP(2);
58                  MKSTEP(3);
59          }
60
61          u16 mul_x(u16 v, u16 c)
62          { if (v & 0x8000)
63                          return(v << 1) ^ c;
64                  else
65                          return (v << 1);
66          }
67
```

```
68      u16 mul_x_inv(u16 v, u16 d)
69      { if (v & 0x0001)
70                      return(v >> 1) ^ d;
71              else
72                      return (v >> 1);
73      }
74
75      void permute_sigma(u32 * state)
76  { u8 tmp[16];
77
78      for (int i = 0; i < 16; i++)
79          tmp[i] = (u8)(state[Sigma[i] >> 2] >> ((Sigma[i] & 3) << 3));
80
81              for (int i = 0; i < 4; i++)
82                      state[i] = MAKEU32(MAKEU16(tmp[4 * i + 3], tmp[4 * i + 2]),
83                                      MAKEU16(tmp[4 * i + 1], tmp[4 * i]));
84  }
85
86      void fsm_update(void)
87      { u32 R1temp[4];
88              memcpy(R1temp, R1, sizeof(R1));
89
90              for (int i = 0; i < 4; i++)
91              { u32 T2 = MAKEU32(A[2 * i + 1], A[2 * i]);
92                      R1[i] = (T2 ^ R3[i]) + R2[i];
93              }
94              permute_sigma(R1);
95              aes_enc_round(R3, R2, AesKey2);
96              aes_enc_round(R2, R1temp, AesKey1);
97      }
98
99      void lfsr_update(void)
100     {
101             for (int i = 0; i < 8; i++)
102             { u16 u = mul_x(A[0], 0x990f) ^ A[1] ^ mul_x_inv(A[8], 0xcc87) ^ B[0];
103                     u16 v = mul_x(B[0], 0xc963) ^ B[3] ^ mul_x_inv(B[8], 0xe4b1) ^ A↩
                            [0];
104
105                     for (int j = 0; j < 15; j++)
106                     { A[j] = A[j + 1];
107                             B[j] = B[j + 1];
108                     }
109
110                     A[15] = u;
111                     B[15] = v;
112             }
113     }
114
115     void keystream(u8 * z)
116     {
117             for (int i = 0; i < 4; i++)
118             { u32 T1 = MAKEU32(B[2 * i + 9], B[2 * i + 8]);
119                     u32 v = (T1 + R1[i]) ^ R2[i];
120                     z[i * 4 + 0] = (v >> 0) & 0xff;
121                     z[i * 4 + 1] = (v >> 8) & 0xff;
122                     z[i * 4 + 2] = (v >> 16) & 0xff;
123                     z[i * 4 + 3] = (v >> 24) & 0xff;
124             }
```

```
125
126                 fsm_update();
127                 lfsr_update();
128         }
129
130         void keyiv_setup(u8 * key, u8 * iv, int is_aead_mode)
131         {
132                 for (int i = 0; i < 8; i++)
133                 { A[i] = MAKEU16(iv[2 * i + 1], iv[2 * i]);
134                         A[i + 8] = MAKEU16(key[2 * i + 1], key[2 * i]);
135                         B[i] = 0x0000;
136                         B[i + 8] = MAKEU16(key[2 * i + 17], key[2 * i + 16]);
137                 }
138
139         if(is_aead_mode == 1)
140         { B[0] = 0x6C41;
141             B[1] = 0x7865;
142             B[2] = 0x6B45;
143             B[3] = 0x2064;
144             B[4] = 0x694A;
145             B[5] = 0x676E;
146             B[6] = 0x6854;
147             B[7] = 0x6D6F;
148         }
149
150                 for (int i = 0; i < 4; i++)
151                         R1[i] = R2[i] = R3[i] = 0x00000000;
152
153                 for (int i = 0; i < 16; i++)
154                 { u8 z[16];
155                         keystream(z);
156
157                         for (int j = 0; j < 8; j++)
158                                 A[j + 8] ^= MAKEU16(z[2 * j + 1], z[2 * j]);
159
160                         if (i == 14)
161                                 for (int j = 0; j < 4; j++)
162                                         R1[j] ^= MAKEU32(MAKEU16(key[4 * j + 3], key[4 * j ↩
                                                + 2]),
163                                                         MAKEU16(key[4 * j + 1], key[4 * j + ↩
                                                             0]));
164                         if (i == 15)
165                                 for (int j = 0; j < 4; j++)
166                                         R1[j] ^= MAKEU32(MAKEU16(key[4 * j + 19], key[4 * j↩
                                                + 18]),
167                                                         MAKEU16(key[4 * j + 17], key[4 * j +↩
                                                              16]));
168                 }
169         }
170 };
```

```
1 // AEAD mode: SNOW-V-GCM in C++ (Endianness-free)
2 #include <stdint.h>
3 #include <stdlib.h>
4 #include "SNOWV.h"
```

```
#include "ghash.h"

#define min(a, b) (((a) < (b)) ? (a) : (b))

void snowv_gcm_encrypt(u8 * A, u8 * ciphertext, u8 * plaintext, u64 plaintext_sz,
                       u8 * aad, u64 aad_sz, u8 * key32, u8 * iv16)
{
    u8 Hkey[16], endPad[16];
    struct SnowV32 snowv;
    memset(A, 0, 16);
    snowv.keyiv_setup(key32, iv16, 1);
    snowv.keystream(Hkey);
    snowv.keystream(endPad);
    ghash_update(Hkey, A, aad, aad_sz);

    for (u64 i = 0; i < plaintext_sz; i += 16)
    { u8 key_stream[16];
       snowv.keystream(key_stream);
       for(u8 j = 0; j < min(16, plaintext_sz - i); j++)
            ciphertext[i + j] = key_stream[j] ^ plaintext[i + j];
    }

    ghash_update(Hkey, A, ciphertext, plaintext_sz);
    ghash_final(Hkey, A, aad_sz, plaintext_sz, endPad);
}

void snowv_gcm_decrypt(u8 * A, u8 * ciphertext, u8 * plaintext, u64 ciphertext_sz,
                       u8 * aad, u64 aad_sz, u8 * key32, u8 * iv16)
{
    u8 Hkey[16], endPad[16], auth[16] = {0x00};
    struct SnowV32 snowv;
    snowv.keyiv_setup(key32, iv16, 1);
    snowv.keystream(Hkey);
    snowv.keystream(endPad);
    ghash_update(Hkey, auth, aad, aad_sz);
    ghash_update(Hkey, auth, ciphertext, ciphertext_sz);
    ghash_final(Hkey, auth, aad_sz, ciphertext_sz, endPad);
    for(int i = 0; i < 16; i++)
        if(auth[i] != A[i])
               { printf("Authentication Failed!");
                  exit(1);
               }

    for (u64 i = 0; i < ciphertext_sz; i += 16)
    { u8 key_stream[16];
       snowv.keystream(key_stream);
       for(u8 j = 0; j < min(16, ciphertext_sz - i); j++)
            plaintext[i + j] = key_stream[j] ^ ciphertext[i + j];
    }
}
```

```
// Informative: an exampled implementation of GHASH core (C++)
#define XOR2x64(dst, src) ((u64*)(dst))[0] ^= ((u64*)(src))[0], \
                              ((u64*)(dst))[1] ^= ((u64*)(src))[1]
#define XOR3x64(dst, src1, src2) ((u64*)(dst))[0] = ((u64*)(src1))[0] ^ ((u64*)(src2))↩
    [0], \
```

```
                                ((u64*)(dst))[1] = ((u64*)(src1))[1] ^ ((u64*)(src2))[1]

void ghash_mult(u8 * out, const u8 * x, const u8 * y)
{ char tmp[17];
        u64 c0, c1, u0 = ((u64*)y)[0], u1 = ((u64*)y)[1];
        memset(out, 0, 16);

        for (int i = 0; i < 16; i++)
                for (int j = 7; j >= 0; j--)
                { if ((x[i] >> j) & 1) ((u64*)out)[0] ^= u0, ((u64*)out)[1] ^= u1;
                        c0 = (u0 << 7) & 0x8080808080808080ULL;
                        c1 = (u1 << 7) & 0x8080808080808080ULL;
                        u0 = (u0 >> 1) & 0x7f7f7f7f7f7f7f7fULL;
                        u1 = (u1 >> 1) & 0x7f7f7f7f7f7f7f7fULL;
                        ((u64*)(tmp + 1))[0] = c0;
                        ((u64*)(tmp + 1))[1] = c1;
                        tmp[0] = (tmp[16] >> 7) & 0xe1;
                        u0 ^= ((u64*)tmp)[0];
                        u1 ^= ((u64*)tmp)[1];
                }
}

void ghash_update(const u8 * H, u8 * A, const u8 * data, long long length)
{ u8 tmp[16];
        for( ;length >= 16; length -=16, data += 16)
        { XOR3x64(tmp, data, A);
                ghash_mult(A, tmp, H);
        }
        if(!length) return;
        memset(tmp, 0, 16);
        memcpy(tmp, data, length);
        XOR2x64(tmp, A);
        ghash_mult(A, tmp, H);
}

void ghash_final(const u8 * H, u8 * A, u64 lenAAD, u64 lenC, const u8 * maskingBlock)
{
    u8 tmp[16];
    lenAAD <<= 3;
    lenC <<= 3;
    for(int i=0; i<8; ++i)
    { tmp[7-i] = (u8)(lenAAD >> (8 * i));
        tmp[15-i] = (u8)(lenC >> (8 * i));
    }
    XOR2x64(tmp, A);
    ghash_mult(A, tmp, H);
    XOR2x64(A, maskingBlock); /* The resulting AuthTag is in A[] */
}
```

5 SNOW-V Reference Implementation with SIMD

```
// SNOW-V Reference Implementation utilizing AES-NI, SSE2, SSSE3, AVX, AVX2 (Little ↩
    endian)
#include <immintrin.h>
#define vpset16(value) _mm256_set1_epi16(value)
```

```
4  const __m256i _snowv_mul = _mm256_blend_epi32(vpset16( 0x990f), vpset16( 0xc963), 0xf0)↩
       ;
5  const __m256i _snowv_inv = _mm256_blend_epi32(vpset16(-0xcc87), vpset16(-0xe4b1), 0xf0)↩
       ;
6  const __m128i _snowv_aead = _mm_lddqu_si128((__m128i*)"AlexEkd JingThom");
7  const __m128i _snowv_sigma= _mm_set_epi8(15,11,7,3,14,10,6,2,13,9,5,1,12,8,4,0);
8  const __m128i _snowv_zero = _mm_setzero_si128();
9
10 struct SnowV256
11 {
12         __m256i hi, lo; // LFSR
13         __m128i R1, R2, R3; // FSM
14
15         inline __m128i keystream(void)
16         {
17                 // Extract the tags T1 and T2
18                 __m128i T1 = _mm256_extracti128_si256(hi, 1);
19                 __m128i T2 = _mm256_castsi256_si128(lo);
20
21                 // LFSR Update
22                 __m256i mulx = _mm256_xor_si256(_mm256_slli_epi16(lo, 1),
23                                                 _mm256_and_si256(_snowv_mul, _mm256_srai_epi16(↩
                                                        lo, 15)));
24                 __m256i invx = _mm256_xor_si256(_mm256_srli_epi16(hi, 1),
25                                                 _mm256_sign_epi16(_snowv_inv, _mm256_slli_epi16(↩
                                                        hi, 15)));
26                 __m256i hi_old = hi;
27                 hi = _mm256_xor_si256(
28                         _mm256_xor_si256(
29                                 _mm256_blend_epi32(
30                                         _mm256_alignr_epi8(hi, lo, 1 * 2),
31                                         _mm256_alignr_epi8(hi, lo, 3 * 2), 0xf0),
32                                 _mm256_permute4x64_epi64(lo, 0x4e)),
33                         _mm256_xor_si256(invx, mulx));
34                 lo = hi_old;
35
36                 // Keystream word
37                 __m128i z = _mm_xor_si128(R2, _mm_add_epi32(R1, T1));
38
39                 // FSM Update
40                 __m128i R3new = _mm_aesenc_si128(R2, _snowv_zero);
41                 __m128i R2new = _mm_aesenc_si128(R1, _snowv_zero);
42                 R1 = _mm_shuffle_epi8(_mm_add_epi32(R2, _mm_xor_si128(R3, T2)), ↩
                        _snowv_sigma);
43                 R3 = R3new;
44                 R2 = R2new;
45                 return z;
46         }
47
48         template<int aead_mode = 0>
49         inline void keyiv_setup(const unsigned char * key, const unsigned char * iv)
50         {
51                 R1 = R2 = R3 = _mm_setzero_si128();
52                 hi = _mm256_lddqu_si256((const __m256i*)key);
53                 lo = _mm256_zextsi128_si256(_mm_lddqu_si128((__m128i*)iv));
54                 if (aead_mode)
55                         lo = _mm256_insertf128_si256(lo, _snowv_aead, 1);
56
```

```
                for (int i = 0; i < 15; ++i)
                        hi = _mm256_xor_si256(hi, _mm256_zextsi128_si256( keystream() ));

                R1 = _mm_xor_si128(R1, _mm_lddqu_si128((__m128i*)(key + 0)));
                hi = _mm256_xor_si256(hi, _mm256_zextsi128_si256( keystream() ));
                R1 = _mm_xor_si128(R1, _mm_lddqu_si128((__m128i*)(key + 16)));
        }
};
```

```
// AEAD mode: SNOW-V-GCM with SIMD (Little Endian)
#define SNOWV_ENCDEC(snowv, out, in) _mm_storeu_si128((__m128i*)(out),\
        _mm_xor_si128(_mm_lddqu_si128((__m128i*)(in)), snowv.keystream()))

// Any external implementation of GHASH
struct ghash_context;
extern void ghash_init (ghash_context & gh, __m128i H);
extern void ghash_update(ghash_context & gh, const u8 * data, long long length);
extern __m128i ghash_final (ghash_context & gh, u64 lenAAD, u64 lenC, __m128i endPad);

// Note: ciphertext must reserve [plaintext_sz + 16] bytes
long long snowv_gcm_encrypt(u8 * ciphertext, const u8 * plaintext, u64 plaintext_sz,
                    const u8 * aad, u64 aad_sz, const u8 * key32, const u8 * iv16)
{
        ghash_context gh;
        SnowV256 snowv;
        snowv.keyiv_setup<1>(key32, iv16); // init with AEAD mode
        ghash_init(gh, snowv.keystream() ); // GHASH key H
        __m128i endPad = snowv.keystream(); // ending pad
        ghash_update(gh, aad, aad_sz); // push AAD into GHASH

        // SNOW-V Encryption
        for (long long i = 0; i < plaintext_sz; i += 16)
                SNOWV_ENCDEC(snowv, ciphertext + i, plaintext + i);

        // Push ciphertext into GHASH
        ghash_update(gh, ciphertext, plaintext_sz);

        // Finalize GCM mode and add the authorization tag to the end of the ciphertext
        _mm_storeu_si128((__m128i*)(ciphertext + plaintext_sz),
                ghash_final(gh, aad_sz, plaintext_sz, endPad));

        // return the total length of the ciphertext
        return plaintext_sz + 16;
}

// Note: plaintext must reserve [ciphertext_sz - 16] bytes
long long snowv_gcm_decrypt(u8 * plaintext, const u8 * ciphertext, u64 ciphertext_sz,
                    const u8 * aad, u64 aad_sz, const u8 * key32, const u8 * iv16)
{
        ghash_context gh;
        SnowV256 snowv;
        snowv.keyiv_setup<1>(key32, iv16); // init with AEAD mode
        ghash_init(gh, snowv.keystream() ); // GHASH key H
        ghash_update(gh, aad, aad_sz); // push AAD into GHASH
        ghash_update(gh, ciphertext, (ciphertext_sz -= 16) ); // push ciphertext to ↩
            GHASH

        // Finalize GCM mode and verify the authorization tag
```

```
49          __m128i auth = ghash_final(gh, aad_sz, ciphertext_sz, snowv.keystream());
50          auth = _mm_xor_si128(auth, _mm_lddqu_si128((__m128i*)(ciphertext + ciphertext_sz↩
                ));
51
52          if (!_mm_test_all_zeros(auth, auth))
53                  return -1; // auth tag is not correct? return a negative value
54
55          // SNOW-V Decryption
56          for (long long i = 0; i < ciphertext_sz; i += 16)
57                  SNOWV_ENCDEC(snowv, ciphertext + i, plaintext + i);
58
59          // return the total length of the plaintext
60          return ciphertext_sz;
61  }
```

6 Hardware implementation aspects

When designing new algorithms targeting existing systems, reusability of hardware components is important to reduce area and cost of the ASICs. Many systems dealing with network communication security implement some form of AES acceleration, either in a specialized ASIC or as specialized CPU instructions. SNOW-V leverages this coexistence by using two full AES encryption rounds as the main nonlinear element. A hardware implementation of SNOW-V can utilize either one or two external AES cores, if present, or implement its own AES encryption rounds in a stand-alone design for maximum speed. Although a 128-bit implementation is straight-forward from the algorithm description, it has some drawbacks when we only have one single external AES core available, as is the case in many constraint implementations. In this section we will consider how to implement SNOW-V using a single AES core with a 64-bit hardware architecture. We will refer to the 64-bit and 128-bit hardware implementations as the 64-SNOW-V and 128-SNOW-V respectively.

6.1 SNOW-V 64-bit Hardware Architecture

In this section we propose a 64-bit hardware architecture where SNOW-V requires a *single* AES encryption core (external or built-in), and each clocking of 64-SNOW-V produces 64 bits of the keystream.

Cons: additional two 64-bit delay registers $D1$ and $D2$ are needed; the logic needs additional 6 64-bit multiplexers; two clocks to produce 128 bits of keystream that actually halves the speed.

Pros: a single AES encryption core is needed; produces 64 bits of keystream at *each clock*; all basic operations in both FSM and LFSR, such as XOR and ADD, are now halved in size.

In order to utilize a single AES core the FSM update function should be split into two steps. The *main critical path* is the AES EncRound, which means that while splitting FSM into two stages we should avoid any extra logic on the input and output signals of the AES core. Thus, input to and output from the AES core must be registers.

Let us split all 128-bit registers and all 128-bit signals of the FSM block, say X, into two 64-bit halves as X_a (low) and X_b (high). We also assume that the tap values $T1$

and $T2$ from the LFSRs also arrive in 64-bit chunks, such that every even clock FSM gets $T1_a$ and $T2_a$, and every odd clock $T1_b$ and $T2_b$.

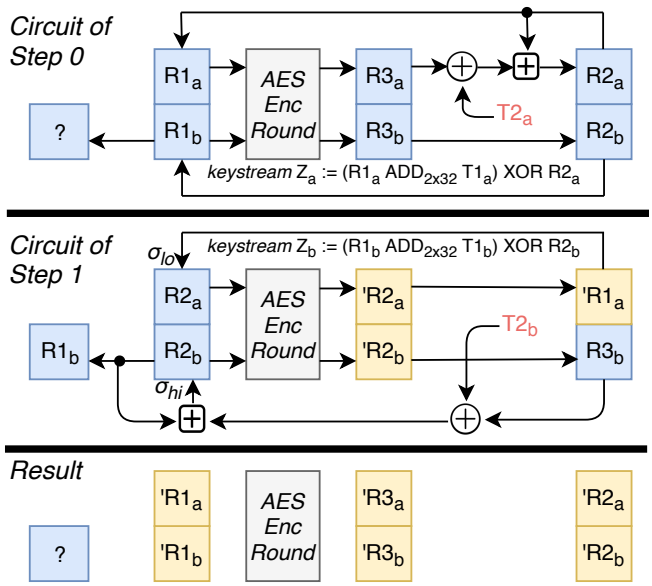

Figure 7: Splitting of FSM into two steps in order to utilize only one AES core (excluding initialization steps).

In Figure 7 we propose a possible way to split the FSM such that it contains the two circuits for even and odd steps, 0 and 1 resp. (excluding the gates needed for initialization). One can notice that after these two steps the content of the registers $R1, R2, R3$ become updated to new 128-bit values $'R1,' R2,' R3$, and ready to process the next 128 bits of data with the same two steps. The above two circuits are then combined into a single circuit using multiplexers.

In Figure 8 the complete hardware architecture for 64-bit SNOW-V is presented. There are 7 64-bit multiplexers in total, and we denote the control signal to them by $M_1..M_7$, respectively. There are also 5 64-bit AND gates, the purpose of which is to either bypass the signal or block it. Those AND blocks are controlled by four signals G_A, G_Z, G_K, G_F, the latter controls 2x64 AND-blocks. The Control Unit in Figure 8 generates the control signals for the multiplexers and AND gates depending on the state of SNOW-V.

Critical path. Our primary assumption is that the AES encryption round would be the *main critical path (MCP)*. However, one can easily determine that the *secondary critical path (SCP)* would be the sequence AND-MUX-XOR-ADD-XOR-MUX over 2x32-bit integers, denoted by red wires in Figure 8. Thus, when selecting 32-bit adders one should make sure that they are fast enough so that the *MCP* is sustained.

The algorithm has 3 stages:

Stage 1 – Loading. The design is constructed in such a way that the registers do not need to have any `RESET` signal. Instead, all registers will be sequentially loaded with

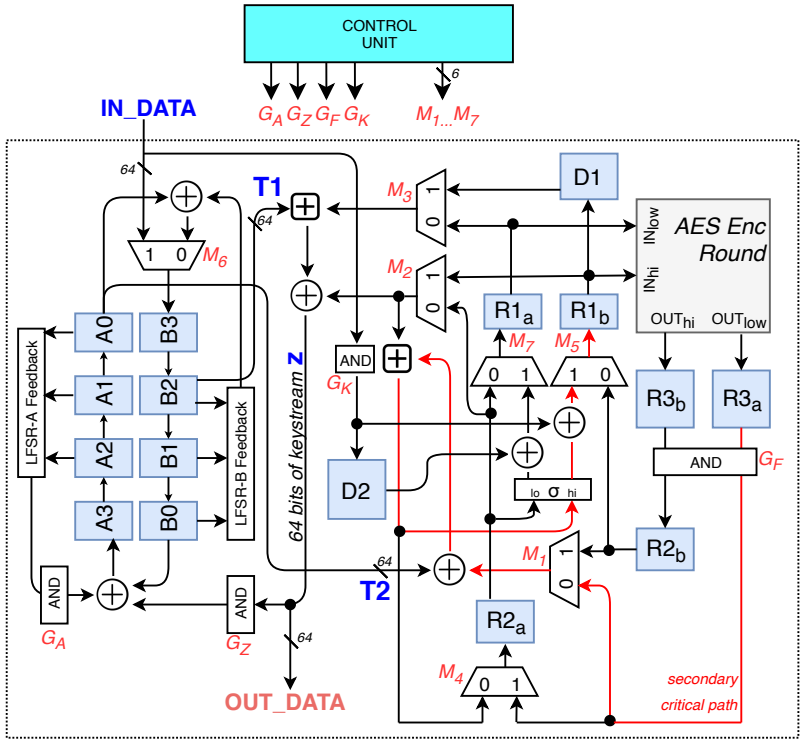

Figure 8: Hardware architecture of 64-bit SNOW-V with a single AES core.

the key and IV, and the remaining registers will be zeroized, during this stage.

The stage begins with a strobe signal on `LOAD`, and the first 64-bit chunk of data is expected on the `IN_DATA` bus. In total, the stage expects to receive 8 64-bit words each clock in the following order: $\{iv_0, iv_1, k_0, k_1, 0, 0, k_2, k_3\}$.

In this stage, the control unit should block AND gates $G_Z = G_A = 0$, and set $M_6 = 1$, in order to concatenate LFSRs A and B into a single large LFSR while shifting in the initialization data. In order to zeroize FSM registers, the control unit should block $G_F = 0$ and also enforce the multiplexer inputs $M_4 = 1, M_5 = M_7 = 0$. G_K is set to 0.

After the 8 clocks where the key and IV are loaded, we proceed to stage 2.

Stage 2 – Initialization. In this stage, the FSM works in the same way as when it produces keystream output symbols, i.e. the multiplexer control signals switches according to even/odd clock cycle as explained previously. The LFSRs are connected together by setting $G_Z = G_A = 1$ and switching $M_6 = 0$ to disable any external input.

Note that we placed the AND gating *after* the registers $R3_a, R3_b$, so that we do not add extra depth to the critical path of AES core, hence these registers will not be zeroized. To overcome this problem the control unit generates $G_F = 0$ in the first clock of this stage, and then sets $G_F = 1$ until the end of stage 2. We keep $G_K = 0$ for the first 28 clocks. In the remaining 4 clocks we need to XOR the key K to $R1$ according to

the initialization procedure. So we enable $G_K = 1$ and expect to receive $\{k_0, k_1, k_2, k_3\}$ consecutively from the input bus `IN_DATA`. After this, the circuit is ready to produce keystream words.

Stage 3 – Keystream generation. Both LFSR and FSM operate normally. The control unit in this stage detaches the Z signal from being feeded into LFSR-A by setting $G_Z = 0$. The input bus is also detached by setting $G_K = M_6 = 0$.

6.2 Theoretical Analysis of 64/128-bit SNOW-V in Hardware

The area will be estimated in terms of **gate equivalence (GE)**, where 1GE = size of a NAND gate. The speed will be estimated in terms of Gigabits per second (Gbps), based on known speed results of AES circuits. We will use GE values given in [Sam00] for 1-speed technology elements.

For comparison with AES, we will use one of the more recent results from [UMHA16] where an area-speed optimized AES-128 (10 rounds) on NanGate 15nm technology runs with the speed 71.19 Gbps and has the area **17232 GE**. This means that having the same design, AES-256 (14 rounds) would run with the speed of **50.85 Gbps**.

Our basic assumption is that the AES core is the critical path of the SNOW-V circuit. Thus, if SNOW-V would utilize a single AES core as above, the speed of 64-SNOW-V could be as high as **356 Gbps**. The speed of 128-SNOW-V with two AES cores is therefore as high as **712 Gbps**. What remains is to calculate the hardware cost of SNOW-V, excluding the external AES core, but including the cost of integration into that external AES core. We will also exclude the control unit, as this can be implemented with a very few gates and latches and every implementation will have slightly different needs of control and ready signaling.

State Registers. For 64-SNOW-V, there are 512 registers for the LFSR and 6x64+2x64 registers for the FSM. Since our 64-bit implementation does not require complex latches (e.g., no `RESET`), we can use the simplest D-latch with Q-output only from [Sam00] [FD1Q]. The total cost is $1024 * 4.33 =$ **4434 GE**.

For 128-SNOW-V we also need 512 registers for the LFSRs without reset, and 3x128 registers with `RESET` [FD2Q], thus resulting in 512*4.33+(3*128)*5.67 = **4394 GE**.

For **arithmetical 32-bit adders** we suggest to take, for example, a Han-Carlson 32-bit adder, as it has a low area overhead (15%-25% larger than Ripple-Carry adders) and a very small delay $O(log(n))$ – which is important in order to keep the critical path upper bounded by the AES round function. We can estimate these components as 4x(30FADD3 + 2HADD2)+20%= 4(30∗6.33+2∗3.67)∗1.20 = **947 GE** for 64-SNOW-V and **1894 GE** for 128-SNOW-V.

The remaining part of the **FSM update logic** therefore contains 3x64AND2 + 6x64MUX2 + 4x64XOR2 = $3 * 64 * 1.33 + 6 * 64 * 2.33 + 4 * 64 * 2.33 =$ **1747 GE** for 64-SNOW-V and (128AND2+3x128XOR2)=**1065 GE** for 128-SNOW-V.

LFSR Update logic involves two circuits for the feedback functions. 16-bit field multiplications by $\alpha, \alpha^{-1}, \beta, \beta^{-1}$ can be done with 8 XORs in each case, since the Hamming weight of both $g^A(\alpha)$ and $g^B(\beta)$ is 8.

However, let us have a closer look on how each bit of, e.g. a_{16} is calculated. Each bit $a_{16}[i], 14 \geq i \geq 1$ is unconditionally depending on four bits, namely

$$a_{16}[i] \quad : \quad a_0[i-1] + a_1[i] + a_8[i+1] + b_0[i] \tag{13}$$

The end bits are easy to work out too. Some of the bits of a_{16} are also depending on $a_0[15]$ and $a_8[0]$, due to the multiplication with α and α^{-1}. Table 5 gives a full overview of the dependencies for both a_{16} and b_{16}.

i	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Depending on
$a_{16}[i]$	✓			✓	✓			✓					✓	✓	✓	✓	$a_0[15]$
	✓	✓			✓	✓			✓					✓	✓	✓	$a_8[0]$
$b_{16}[i]$	✓	✓			✓			✓		✓	✓				✓	✓	$b_0[15]$
	✓	✓	✓			✓			✓		✓	✓				✓	$b_8[0]$

Table 5: Bit dependencies due to multiplications for a_{16} and b_{16}.

This means that in order to compute $a_{16}[i]$, we have to XOR 4, 5, or 6 different input bits. For example, in the table above we see that the $a_{16}[13]$ is only dependent on the basic input bits in Appendix 6.2, and the XOR gate needs 4 inputs:

$$a_{16}[13] = a_0[12] + a_1[13] + a_8[14] + b_0[13].$$

On the other hand, $a_{16}[11]$ needs to XOR 6 inputs:

$$a_{16}[11] = a_0[10] + a_1[11] + a_8[12] + b_0[11] + a_0[15] + a_8[0].$$

since the multiplication with α and α^{-1} will both influence that bit.

Following the hardware architecture of 64-SNOW-V given in Figure 8 we have to split the calculation of the feedback function LFSR-A due to the control AND-gateway. Also, the circuit should compute 4 16-bit updates in parallel. Summarizing, we get (a) LFSR-A feedback function, *excluding* input from b_0: 4x(5XOR3 + 6XOR4 + 5XOR5) $\approx 4*(5*4.00+6*6.00+5*8.00)$ = **384 GE** ; (b) LFSR-B feedback function, *including* input from a_0: 4x(4XOR4 + 8XOR5 + 4XOR6) $\approx 4*(4*6.00+8*8.00+4*10.00)$ = **512 GE**; (c) the remaining part of LFSR block: 2x64AND2 + 64XOR3 + 64MUX2 $= 64*(2*1.33+4.00+2.33)$ = **575 GE**. For 128-SNOW-V we simply double the above numbers.

Integration into an external AES Engine requires input multiplexers for 128 bits of the plaintext and 128 bits for the round key. However, the AES round keys $C1$ and $C2$ are zeroes so that we can use 128AND gates, instead. In total we get 128MUX2 + 128AND2 = $128*(2.33+1.33)$ = **468 GE** for 64-SNOW-V. 128-SNOW-V requires two such integration circuits.

In case we decide to implement SNOW-V with its own internal AES EncRound, the hardware cost could be as small as 16 AES SBoxes, plus some logic for MixColumn. Also note that in this case the critical path decreases since we only need the forward SBox and thus any outer multiplexing logic for a combined forward and inverse SBox can be removed. This could lead to a potential speed up for 128-SNOW-V.

The part MixColumn of AES encryption round, applied to the AES state $\{r_{i,j}\}$ for

$0 \leq i, j \leq 3$, is the following matrix multiplication.

$$\begin{bmatrix} r'_{0,j} \\ r'_{1,j} \\ r'_{2,j} \\ r'_{3,j} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} r_{0,j} \\ r_{1,j} \\ r_{2,j} \\ r_{3,j} \end{bmatrix}, 0 \leq j \leq 3.$$

That can be computed in a depth 2 circuit, for each $0 \leq j \leq 3$, as $t_0 = r_0 + r_1$, $t_1 = r_1 + r_2$, $t_2 = r_2 + r_3$, $t_3 = r_3 + r_0$, and then $r'_0 = 2t_0 + t_2 + r_1$, $r'_1 = 2t_1 + t_3 + r_2$, $r'_2 = 2t_2 + t_0 + r_3$, $r'_3 = 2t_3 + t_1 + r_0$, where multiplication $2t_i$ is the multiplication by x in the Rijndael field and can be implemented with 3XOR2. The cost of MixColumn is therefore 4x4x(8XOR2 + 8XOR3 + 3XOR2) = **922 GE**[3].

The cost of a single forward SBox is around 220 GE (see, e.g., [RMTA18]). Thus, for a **single internal AES EncRound** the total cost is $16 * 220 + 922 =$ **4442 GE**. Summarizing the above we can derive the comparison given in Table 6 .

Hardware design	AES-256 from [UMHA16]	64-SNOW-V 1xAES ext. core	64-SNOW-V 1xAES int. round	128-SNOW-V 2xAES ext. cores	128-SNOW-V 2xAES int. rounds
Area	17232 GE	9067 GE	13041 GE	11231 GE	19179 GE
Speed	50.85 Gbps	358 Gbps	358+ Gbps	712 Gbps	712+ Gbps

Table 6: Theoretical comparison of four SNOW-V versions vs AES-256 in hardware.

[3]Recent results in [Max19] suggest that MixColumn can be implemented with 4x92 gates. However, we believe that the proposed classical solution with 4x108 gates is a better choice since it has a lower depth of the critical path, thus allowing SNOW-V to perform faster.

Paper IV

Improved guess-and-determine and distinguishing attacks on SNOW-V

Abstract

In this paper, we investigate the security of SNOW-V, demonstrating two guess-and-determine (GnD) attacks against the full version with complexities 2^{384} and 2^{378}, respectively, and one distinguishing attack against a reduced variant with complexity 2^{303}. Our GnD attacks use enumeration with recursion to explore valid guessing paths, and try to truncate as many invalid guessing paths as possible at early stages of the recursion by carefully designing the order of guessing. In our first GnD attack, we guess three 128-bit state variables, determine the remaining four according to four consecutive keystream words. We finally use the next three keystream words to verify the correct guess. The second GnD attack is similar but exploits one more keystream word as side information helping to truncate more guessing paths. Our distinguishing attack targets a reduced variant where 32-bit adders are replaced with exclusive-OR operations. The samples can be collected from short keystream sequences under different (key, IV) pairs. These attacks do not threaten SNOW-V, but provide more in-depth details for understanding its security and give new ideas for cryptanalysis of other ciphers.

Keywords: SNOW-V, Guess-and-determine attack, Distinguishing attack.

1 Introduction

SNOW-V [EJMY19] is a new member of the SNOW family of stream ciphers, proposed in 2019 in response to the new requirements of the confidentiality and integrity algorithms in 5G and beyond from 3GPP [3GP19]. First, the 256-bit security level is expected in 5G to resist against attackers equipped with quantum computing capability, while the predecessor SNOW 3G being used in 4G was only specified for 128-bit key length. If the key length in SNOW 3G would be increased to 256 bits, there exist academic attacks against it much faster than exhaustive key search, see e.g., [YJM19]. Besides, the algorithms are expected to achieve high throughput in software environments, as many of the network nodes in 5G can be virtualised and the ability to use specialised hardware for cryptographic primitives will thus be reduced. The targeted speed for

Jing Yang, Thomas Johansson, and Alexander Maximov. Improved guess-and-determine and distinguishing attacks on SNOW-V. IACR Transactions on Symmetric Cryptology, pages 54-83, 2021.

downlink transmission in 5G is 20 Gbps, while current performance benchmarks for SNOW 3G only give approximately 9 Gbps in a pure software environment [YJ20]. 3GPP has asked ETSI SAGE (Security Algorithms Group of Experts) to select and evaluate efficient confidentiality and integrity algorithms for 5G use [3GP19]. SNOW-V is designed given these motivating facts and aims to provide a 256-bit security level and perform fast enough in software environments. It has been submitted to SAGE and is under evaluation [SAG20].

SNOW-V follows the design principles of the SNOW family, with a linear part consisting of LFSRs (Linear Feedback Shift Registers) to serve as the source of pseudo-randomness, and a non-linear part called FSM (Finite State Machine) to disrupt the linearity. Both parts are redesigned and better aligned to adapt to the higher performance and stronger security demands in 5G. The FSM part is now increased to a larger size and accommodates two AES encryption rounds to serve as two large S-boxes providing non-linearity, thus taking full advantage of the intrinsic instruction of AES encryption round supported by most mainstream CPUs. SNOW-V can achieve rates up to 58 Gbps for encryption in a pure software environment [EJMY19] and more than 1 Tbps in hardware [CBB20].

Since proposed, SNOW-V has received internal and external evaluations [EJMY19, CDM20], which exhaustively visit all the promising cryptanalysis techniques of stream ciphers and ensure that none of them applies to SNOW-V faster than exhaustive key search. After that, more in-depth and focused studies followed, e.g., [JLH20, GZ21, HII$^+$21]. For example, the paper [HII$^+$21] investigates the security of the initialisation of SNOW-V, using MILP (Mixed-integer linear programming) model to efficiently search for integral and differential characteristics. The resulting distinguishing and key recovery attacks are applicable to SNOW-V with reduced initialisation rounds of five, out of the original 16, which indicates that the initialisation has a good security margin. Below we give a more detailed introduction to the guess-and-determine attacks [JLH20, CDM20] and linear cryptanalysis [GZ21] against SNOW-V, and present our contribution.

Guess-and-determine (GnD) attacks. A basic GnD attack of complexity 2^{512} is proposed in the evaluation report [CDM20]. In this attack, one has to guess three out of the seven internal 128-bit state variables and derive another three using three consecutive keystream words. Although not all derivation details were given, it was assumed that the derivation is possible with a negligible time, leading to recovering six state variables in time complexity 2^{384}. The last seventh state variable is recovered by guessing, thus the total complexity is 2^{512}. The next four keystream words are thereafter used to verify the correct guess. The authors in [JLH20] propose a byte-based GnD attack against SNOW-V with complexity 2^{406} using seven keystream words. In their attack, the state variables are split into bytes with some carriers introduced, and dynamic programming tool is used to help search a good guessing path that requires guessing as few bytes as possible. Both GnD attacks require seven consecutive 128-bit keystream words which looks reasonable, as the internal state has seven 128-bit unknown variables that needs to be either guessed or determined.

Our contribution. Our GnD attacks follow the research line of the GnD attack in [CDM20] and fill the gaps of it. In our first GnD attack, we find an efficient *recursive enumeration* technique in a byte-wise manner to determine three more state variables

given three guesses and three consecutive keystream words, such that the complexity of deriving six state variables is still 2^{384}. We then use the same enumeration way to derive the last seventh state variable with negligible overhead, thus the total attack complexity is 2^{384}. This improves the GnD attacks both in [CDM20] (2^{512}) and [JLH20] (2^{406}).

In our recursive enumeration technique, we take full advantage of the observation that some guessing values will not give valid solutions at some point in the middle of the guessing process, and one can immediately terminate this guessing branch and trace back to guess another value. Thus, some efforts of going deeper can be saved. The earlier and more often one can find such cases, the more efforts can be saved. We carefully design the guessing order of the guesses, such that most guessing paths would be truncated at some point without going into the end, resulting in the total GnD attack complexity 2^{384}.

In our second GnD attack, we use one additional "backward" keystream word as side information to impose more constraints and truncate more "forward" guessing paths, thus further reduce the complexity to 2^{378}. In order to retrieve the side information efficiently (e.g., instantly) we need a volatile table of size 2^{128} bits, which might be implemented in RAM or HDD. The improvement factor over the first GnD attack is not so significant, but the idea of using side information to refute more guessing paths and thus reduce the overall time complexity is interesting in general.

Linear cryptanalysis. The SNOW family of stream ciphers is constructed from two components – the LFSR, serving as the source of pseudo-randomness, and the FSM, providing nonlinearity. In typical linear cryptanalysis of such a construction, the nonlinear part FSM is approximated by a linear expression between some keystream words and LFSR variables plus a biased noise variable $N^{(t)}$, while the variables in the FSM are cancelled. In a distinguishing attack, such linear expressions at k time instances corresponding to either the feedback polynomial or a low-weight (usually 3 or 4) multiple of it will be combined (typically through exclusive-OR operation), such that the contribution terms from the LFSR are cancelled. Hence, the linear expressions involve only the keystream symbols and noises, making a reorganised keystream sample sequence biased and distinguishable from random, given enough number of samples. As the FSM approximation expression is repeated k times, the total noise would then involve k sub-noises.

For example, in SNOW 2.0 [EJ02], the LFSR has a feedback polynomial of weight four in the time frame of width 17 over $\mathbb{F}_{2^{32}}$.The authors of [WBDC03, NW06] found a very strong approximation of the FSM such that the bias is large, and combining four such approximations to cancel out the LFSR contribution led to a distinguishing attack of overall complexity 2^{225} in [WBDC03] and further improved to 2^{174} in [NW06]. Note that in both papers the feedback polynomial is used to find the four time instances such that the LFSR contribution can be cancelled, and the required samples can be collected from many short keystreams under different key and IV (Initialisation Vector) pairs.

A straightforward prevention of above situation is to increase the number of taps in the LFSR update function. In this case the direct usage of the feedback polynomial would involve many more sub-noises, and the bias of the total noise would be very small. However, there is a possibility to find a theoretical low-weigh multiple of any feedback polynomial, due to the birthday paradox, such that one can still construct a biased noise

sample from several keystream words but far apart in time instances, i.e., the attacker needs a long keystream sequence to collect one single sample. This strategy was used in, e.g., the recent cryptanalysis of ZUC-256 [YJM20] and SNOW 3G [YJM19], in which weight-4 multiples are used. In SNOW-V, an *equivalent* 32×16-bit LFSR has a feedback polynomial of weight 12.

In [GZ21], the authors perform linear cryptanalysis of SNOW-V and propose *correlation attacks* against three reduced variants of it, in which either a permutation operation is omitted or 32-bit arithmetic additions are replaced with 8-bit ones. The closest variant is SNOW-V$_{\boxplus_{32},\boxplus_8}$, in which one $\boxplus_{32}$ (four parallel 32-bit adders) is replaced by $\boxplus_8$ (16 parallel 8-bit adders), and the complexity of the correlation attack against it is 2^{377}. Correlation attacks are focused on recovering the internal state and thus require a single long keystream under a fixed (key, IV) pair.

Our contribution. In our distinguishing attack, we target a reduced variant of SNOW-V, denoted SNOW-V$_{\oplus}$, in which the 32-bit adders are replaced with exclusive-OR. Unlike the classical approach, e.g., the above mentioned, where one first approximates the FSM and thereafter cancels the LFSR contribution by combining expressions at several time instances, we do it vice-versa and cancel the LFSR contribution directly without combining several approximations. We explore the fact that three LFSR registers appear *twice* in three consecutive keystream words – the minimum needed for the FSM approximation in SNOW-V – and moreover, they happen to contribute linearly in SNOW-V$_{\oplus}$ thus can be directly cancelled.

Therefore, we consider three consecutive 128-bit keystream words and linearly combine the bytes in these keystream words, such that the contribution from the LFSR is directly cancelled. We then explore linear masking coefficients in an efficient way to cancel out as many S-box approximations in the FSM as possible, thus to make the bias larger. We find a bias evaluated using Squared Euclidean Imbalance (SEI) around 2^{-303} and give a distinguishing attack with complexity 2^{303}. A single noise sample is collected from just *three 128-bit consecutive keystream words*, and the samples can be collected from many short keystream sequences under different (key, IV) pairs.

Though none of existing and our cryptanalysis efforts result in a valid attack against SNOW-V faster than exhaustive key search, they are still of great importance for fully understanding the security of the cipher. Table 1 lists the main existing cryptanalysis results against SNOW-V, and comparison with new results in this paper.

Outline. We first provide some notations and expressions in Section 2, together with a brief description of SNOW-V. We then demonstrate two guess-and-determine attacks in Section 3 and Section 4, respectively. In Section 5, we perform linear cryptanalysis against SNOW-V and propose a distinguishing attack against the reduced variant SNOW-V$_{\oplus}$. We end the paper with conclusions in Section 6.

Attack	Complexity	Data	Reference
Guess-and-Determine	2^{512}	7 keystream words	[CDM20]
	2^{406}	7 keystream words	[JLH20]
	2^{384}	7 keystream words	Section 3
	2^{378*}	8 keystream words	Section 4
Linear Cryptanalysis	2^{377**}	long keystream 2^{254}	[GZ21]
	2^{303***}	short keystreams	Section 5
Integral Distinguisher	2^{48} (5 rounds)	2^{48}	[HII+21]
Differential Distinguisher	2^{97} (4 rounds)	2^{97}	[HII+21]
Differential Key Recovery	2^{154} (4 rounds)	2^{27}	

* The attack has memory complexity 2^{128} as it needs a volatile table of size 2^{128} bits.

** The attack is applied on the reduced variant SNOW-V$_{\boxplus_{32},\boxplus_8}$.

*** The attack is applied on the reduced variant SNOW-V$_{\oplus}$.

Table 1: Attacks against SNOW-V and its variants.

2 Preliminaries

2.1 Notations

The exclusive-OR and addition modulo 2^m are denoted by $\oplus$ and $\boxplus_m$, respectively. $||$ denotes the concatenation operation. The m-dimensional binary extension field is denoted as $\mathbb{F}_{2^m}$. For two variables $x, y \in \mathbb{F}_{2^m}$, xy denotes the multiplication over $\mathbb{F}_{2^m}$. Given two vectors $\mathbf{a}, \mathbf{b}$ of length t, $\mathbf{a} = (a_{t-1}, \ldots, a_1, a_0)$ and $\mathbf{b} = (b_{t-1}, \ldots, b_1, b_0)$, where $a_i, b_i \in \mathbb{F}_{2^m}$ for $0 \leq i \leq t-1$, we use $\mathbf{ab}$ to denote the point-wise multiplication computed as $\mathbf{ab} = \oplus_{i=0}^{t} a_i b_i$, where $a_i b_i$ is the multiplication over $\mathbb{F}_{2^m}$. We sometimes also use $(a_{t-1}, \ldots, a_1, a_0) \cdot (b_{t-1}, \ldots, b_1, b_0)$ to denote the same point-wise multiplication. If $m = 1$, $\mathbf{ab}$ is the standard inner product.

The variables throughout the paper are normally 128-bit long, unless otherwise specified. For a 128-bit variable x, we can express it as a byte vector $(x_{15}, x_{14}, \ldots, x_1, x_0)$, where x_i $(0 \leq i \leq 15)$ is the i-th byte. We use several subscripts to indicate several bytes of a variable. For example, $x_{1,5,7}$ denotes the 1-st, 5-th, 7-th bytes of x. To express the vector of these bytes, we add a notation $[\cdot]$ outside. For example, $[x_{1,5,7}]$ denotes the byte vector (x_1, x_5, x_7). We add $||$ between subscript indices to denote the concatenation of these bytes, e.g., $x_{1||5||7}$ denotes $x_1||x_5||x_7$.

2.2 Introduction to SNOW-V

In this section, we give a brief introduction to SNOW-V and predefine some notations and expressions which will be frequently used in the subsequent cryptanalysis. The overall schematic of SNOW-V is depicted in Figure 1. It follows the design principles of the SNOW-family, consisting of the LFSR part and the FSM.

The LFSR part is a new circular construction consisting of two 256-bit registers, named LFSR-A and LFSR-B, feeding to each other. Both LFSRs have 16 cells, each of

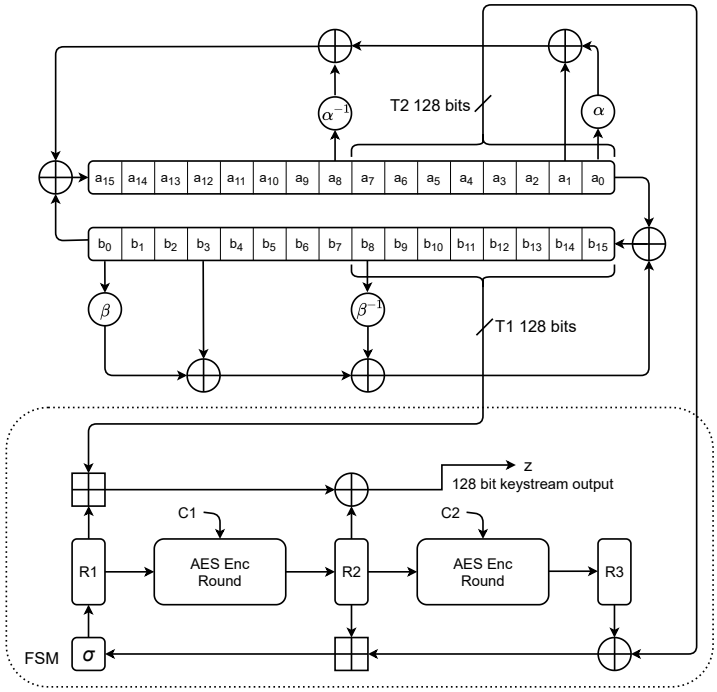

Figure 1: Overall schematic of SNOW-V [EJMY19].

which holds an element from the finite field $\mathbb{F}_{2^{16}}$. These elements in LFSR-A, denoted $a_{15}, \ldots, a_0$, and LFSR-B, denoted $b_{15}, \ldots, b_0$, are generated according to the generating polynomials $g^A(x)$ and $g^B(x)$, respectively, which are expressed as below:

$$g^A(x) = x^{16} + x^{15} + x^{12} + x^{11} + x^8 + x^3 + x^2 + x + 1 \in \mathbb{F}_2[x],$$
$$g^B(x) = x^{16} + x^{15} + x^{14} + x^{11} + x^8 + x^6 + x^5 + x + 1 \in \mathbb{F}_2[x].$$

Denote the state of LFSR-A and LFSR-B at clock t by $(a_{15}^{(t)}, \ldots, a_0^{(t)})$ and $(b_{15}^{(t)}, \ldots, b_0^{(t)})$, respectively. Every time when clocking, the value in a cell is shifted to the next cell with a smaller index and $a_0^{(t)}, b_0^{(t)}$ exit the LFSRs. The values in cell a_{15}, b_{15} are updated as:

$$a^{(t+16)} = b^{(t)} + \alpha a^{(t)} + a^{(t+1)} + \alpha^{-1} a^{(t+8)} \mod g^A(\alpha),$$
$$b^{(t+16)} = a^{(t)} + \beta b^{(t)} + b^{(t+3)} + \beta^{-1} b^{(t+8)} \mod g^B(\beta),$$

where α, β are roots of the two generating polynomials $g^A(\alpha)$ and $g^B(\beta)$, respectively. Such a construction has the maximum cycle of length $2^{512} - 1$.

Every time when updating the LFSR part, LFSR-A and LFSR-B are clocked eight times, thus half of the states will be updated. After that, the two taps $T1$ and $T2$, which are formed by considering $(b_{15}, b_{14}, \ldots, b_9, b_8)$, and $(a_7, a_6, \ldots, a_1, a_0)$ as two 128-bit words, are fed to the FSM.

The FSM has three 128-bit registers, denoted $R1, R2$ and $R3$. It takes $T1, T2$ as inputs and produces a 128-bit keystream word z by the expression below,

$$z^{(t)} = (R1^{(t)} \boxplus_{32} T1^{(t)}) \oplus R2^{(t)}. \tag{1}$$

The three registers are then updated as follows:

$$R2^{(t+1)} = \mathrm{AES}_R(R1^{(t)}), \tag{2}$$

$$R3^{(t+1)} = \mathrm{AES}_R(R2^{(t)}), \tag{3}$$

$$R1^{(t+1)} = \sigma(R2^{(t)} \boxplus_{32} (R3^{(t)} \oplus T2^{(t)})), \tag{4}$$

where $\mathrm{AES}_R()$ is one AES encryption round and σ is a byte-oriented permutation defined as $\sigma = [0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15]$. The AES encryption rounds and $\boxplus_{32}$ provide the source of non-linearity.

The design document has also specified the initialisation phase and AEAD (Authenticated encryption with associated data) mode; as they are not relevant to our attacks, we skip the details but refer to the design document [EJMY19].

Notations and Expressions. We give some notations and expressions here which will be frequently used in the guess-and-determine attacks and linear cryptanalysis.

We use $(R1, R2, R3)$ and $(A0, A1, B0, B1)$ to denote the values of the registers in FSM and in LFSR, respectively, at some specific time t, where $A0$ ($B0$) and $A1$ ($B1$) are the low and high 128 bits of LFSR-A (LFSR-B), respectively. Thus, these seven variables are all 128-bit long and represents the whole state of the cipher. We can then get the following expressions:

$$\begin{array}{lll} B1^{(t-1)} = B0, & B0^{(t+1)} = B1, & B1^{(t+1)} = A0 \oplus l_\beta(B0) \oplus h_\beta(B1), \\ A0^{(t+1)} = A1, & R1^{(t-1)} = \mathrm{AES}_R^{-1}(R2), & R2^{(t-1)} = \mathrm{AES}_R^{-1}(R3), \end{array}$$

where $\mathrm{AES}_R^{-1}()$ is the inverse of one AES encryption round. Here l_β and h_β are two linear operations relevant to the update of the LFSR, and are defined as below:

$$l_\beta(X) = (\beta(X_{15||14}) \,||\cdots||\, \beta(X_{1||0})) \oplus X_{\gg 3\cdot 2}, \tag{5}$$

$$h_\beta(X) = (\beta^{-1}(X_{15||14}) \,||\cdots||\, \beta^{-1}(X_{1||0})) \oplus X_{\ll 5\cdot 2}, \tag{6}$$

where X is a 128-bit variable and $X_{\ll k}, X_{\gg k}$ denote the left and right shift by k bytes, respectively. The multiplication operation with β or β^{-1} are applied to every 16-bit word independently over the field of LFSR-B. They can be expressed as the multiplication of the bit vector of the word and the binary 16×16-bit matrices of β or β^{-1}. The binary matrix representations of β and β^{-1} are given in Appendix 1. The explicit expressions of $l_\beta(X)$ and $h_\beta(X)$ in bytes are given in Appendix 2.

The expressions for three consecutive keystream words at clock $t-1, t$ and $t+1$, which will be frequently used in our attacks, are derived as follows:

$$\begin{aligned} z^{(t-1)} &= (\mathrm{AES}_R^{-1}(R2) \boxplus_{32} B0) \oplus \mathrm{AES}_R^{-1}(R3)), \\ z^{(t)} &= (R1 \boxplus_{32} B1) \oplus R2, \\ z^{(t+1)} &= (\sigma(R2 \boxplus_{32} (R3 \oplus A0)) \boxplus_{32} (A0 \oplus l_\beta(B0) \oplus h_\beta(B1))) \oplus \mathrm{AES}_R(R1). \end{aligned} \tag{7}$$

3 The first guess-and-determine attack ($T = 2^{384}$)

In this section, we fill the gaps of the GnD attack in [CDM20] and improve the complexity from 2^{512} there down to 2^{384}. We first introduce some basics about guess-and-determine attacks, which apply to our second GnD attack in Section 4 as well. We then describe the attack in details and discuss its complexity.

3.1 Basics about guess-and-determine attacks

In a guess-and-determine attack, one *guesses* some variables and *determines* others according to some predefined relationships. In a GnD attack against a stream cipher, if all the variables in the whole state could be determined through guessing a number t of bits, where t is smaller than the security level, the attack is then faster than exhaustive key search. Knowing the whole state of a stream cipher at a certain time allows to trivially recover the whole keystream corresponding to the specific secret key and IV. If the initialisation phase has no special protection, one can even recover the secret key.

In this paper, we call every ordered tuple of values of the guessed and further determined variables a *guessing path* or a *guess-and-determine path*, and use *end-nodes* to denote the end points of the guessing paths. Usually, the complexity of a GnD attack is computed as 2^t, if one simply loops over all the possible values of the chosen variables for guessing. However, we notice that by guessing the variables in a careful order, one can either guess fewer variables or truncate some guessing paths in which the already known (either guessed or determined) variables fail to satisfy some equation constraints in the middle. In the latter case, we can immediately trace back without going further and turn to guess another value, thus the complexity could be reduced.

```
T = 0;
for(x=0; x<256; x++)
for(y=0; y<256; y++)
for(z=0; z<256; z++)
{   T = T + 1;
...
}
```

Listing 1: A simple GnD loop.

For example, consider the simplest loop in the pseudo-code in Listing 1, where x, y, z are three 8-bit variables, it is straightforward to get that the complexity is $T = 2^{24}$.

```
T = 0;
for(x=0; x<256; x++)
for(y = L1[x].first; y!=NULL; y=y->next)
for(z = L2[x, y].first; z!=NULL; z=z->next)
{   T = T + 1;
...
}
```

Listing 2: A more complex GnD loop.

However, for a different loop shown in Listing 2, where $L1[x]$ are lists depending on the specific values of x and $L2[x, y]$ are lists depending on the values of x, y, the size of the loop is not fixed but rather depends on the lengths of lists $L1[x]$ and $L2[x, y]$. For example, for a specific value of x, after we have gone through every value of $L1[x]$ for y (and correspondingly subsequent z), we can immediately trace back to another x value,

instead of considering all the 256 values of y. In this case, the complexity is not simply 2^{24}, but instead the number of valid looping paths.

Thus the complexity of a guess-and-determine attack could be expressed as $c \cdot T$, where c is some constant coefficient which we will explain later, and T is not just the size of the guessing loop, but rather dominated by the number of guessing paths that the attack algorithm will reach an end-node. If the exact value of T is infeasible to compute, the average value of it over the guessed variables is instead considered.

We will use the term **enumeration** to denote going through all the valid guess-and-determine paths, and the size/length of such an enumeration will decide the GnD complexity T. We would like to mention that the organisation of an *enumeration* may not be only plain loops, but some more sophisticated algorithms, e.g., *enumeration by recursion*, in which we adopt a recursion algorithm to explore all the solutions satisfying a certain equation or a system of conditions.

The other term c indicates some constant complexity, which solely depends on the concrete platform and the operations how other values are determined from the known ones. For example, the value of c for computing D given A, B through $D = A \oplus B$ or $A = (D \boxplus B) \oplus (D \oplus S(B))$ (S denotes S-box operation) will be different. Obviously, the complexity for the former example can be ignored as it almost consumes nothing, thus $c = 1$; however, for the latter case, it is not trivial to get the value of D directly, and *enumerations* or some other techniques are required. Thus, the cost for simple *derivations* are normally ignored, while if a derivation involves *enumeration*, the complexity of it should be included.

3.2 Steps of the first GnD attack

In our first GnD attack, we guess three 128-bit state variables $R1, R2, B0$ and use three consecutive keystream words to determine three more, $R3, B1$ and $A0$. The guessing path is quite similar to the one in [CDM20], which guesses $R1, R2, R3$ instead, and then similarly deriving $B0, B1, A0$. The derivations for $R3$ and $B1$ are simple, while tricky for $A0$. It is assumed in [CDM20] that $A0$ can be derived efficiently with negligible complexity but no details are provided. We will fill this gap in Section 3.2.2 by breaking down $A0$ into bytes in a similar manner as in [JLH20], but handling the order of derivations and carries in a better way. After that, we use one more keystream word $z^{(t+2)}$ to determine the final state variable $A1$ using the same way for deriving $A0$ with negligible time, instead of purely guessing it with complexity 2^{128} as done in [CDM20], which helps to reduce the total complexity 2^{512} there to 2^{384}. Finally we use three additional keystream words to verify the correct guess. In total, seven 128-bit keystream words are required to determine the seven 128-bit state variables. A simplified flowchart of this GnD attack can be found in Appendix 6.

3.2.1 Initial guessing set and derivations

We consider the three consecutive keystream words given in Equation (7) and introduce two intermediate 128-bit variables, C and D, which are defined as follows:

$$C = l_\beta(B0) \oplus h_\beta(B1), \tag{8}$$

$$D = \sigma(R2 \boxplus_{32} (R3 \oplus A0)) \boxplus_{32} (A0 \oplus C). \tag{9}$$

Correspondingly, the three keystream words can be rewritten as:

$$\begin{aligned} z^{(t-1)} &= (\mathrm{AES}_R^{-1}(R2) \boxplus_{32} B0) \oplus \mathrm{AES}_R^{-1}(R3), \\ z^{(t)} &= (R1 \boxplus_{32} B1) \oplus R2, \\ z^{(t+1)} &= \mathrm{AES}_R(R1) \oplus D. \end{aligned} \tag{10}$$

There are six unknown variables in Equation (10), and to determine all of them, one has to guess not less than three. Since $R1$ and $R2$ appear twice in the expressions, we prefer to first guess them. Let us initially guess $(R1, R2, B0)$ with complexity 2^{384}. Then the variables $R3, B1$ and D will be directly determined from Equation (10), respectively. Thus, all the variables in Equation (10) are known, either through guessing or determining. Besides, the intermediate variable C in Equation (8) is also determined, and our last step is to determine the values of the remaining two state variables, $A0$ and $A1$.

If we find an efficient way to enumerate all the solutions for $A0$ (and $A1$) without additional guesses, the overall GnD complexity will be exactly 2^{384}. We next show how we efficiently find the solutions of $A0$ in Section 3.2.2, and use the same method to derive the last state variable $A1$ with negligible complexity in Section 3.2.3.

3.2.2 Deriving $A0$ using a 10-step recursive enumeration

$A0$ is determined using Equation (9), while we mention that even when all other variables in Equation (9) are fixed, the value of $A0$ might not be uniquely or directly determined as $A0$ appears twice in the equation with non-linear operations in between. So the task now is to efficiently find the solutions for $A0$ in Equation (9), and we next show how we achieve it in a byte-wise fashion. Each byte of D, D_i $(15 \geq i \geq 0)$ is expressed as:

$$D_i = (R2_j \boxplus_8 (R3_j \oplus A0_j) \boxplus_8 u_j) \boxplus_8 (A0_i \oplus C_i) \boxplus_8 v_i, \quad j = \sigma(i), \tag{11}$$

where $u_j, v_i \in \{0, 1\}$ are carry bits that arrive from arithmetic additions of the previous bytes. We call these byte-wise equations as *D-equations*. Note that some carry values are already known: $u_k = v_k = 0$ for $k = 0, 4, 8, 12$. For other carriers, we do not have to guess them if we derive the bytes of $A0$ in a careful order in 10 steps as given in Table 2.

Table 2: The 10 steps to derive $A0$.

Step 0:	D_0	$= (R2_0 \boxplus_8 (R3_0 \oplus A0_0) \boxplus_8 u_0) \boxplus_8 (A0_0 \oplus C_0) \boxplus_8 v_0$
	where	$u_0 = v_0 = 0$
	derive	$\rightarrow (A0_0, u_1, v_1)$
Step 1:	D_1	$= (R2_4 \boxplus_8 (R3_4 \oplus A0_4) \boxplus_8 u_4) \boxplus_8 (A0_1 \oplus C_1) \boxplus_8 v_1$
	D_4	$= (R2_1 \boxplus_8 (R3_1 \oplus A0_1) \boxplus_8 u_1) \boxplus_8 (A0_4 \oplus C_4) \boxplus_8 v_4$
	where	$u_4 = v_4 = 0$ and u_1, v_1 are known from Step 0
	derive	$\rightarrow (A0_1, A0_4, u_2, v_2, u_5, v_5)$
Step 2:	D_5	$= (R2_5 \boxplus_8 (R3_5 \oplus A0_5) \boxplus_8 u_5) \boxplus_8 (A0_5 \oplus C_5) \boxplus_8 v_5$
	where	u_5, v_5 are known from Step 1
	derive	$\rightarrow (A0_5, u_6, v_6)$

Step 3:	D_2	$= (R2_8 \boxplus_8 (R3_8 \oplus A0_8) \boxplus_8 u_8) \boxplus_8 (A0_2 \oplus C_2) \boxplus_8 v_2$
	D_8	$= (R2_2 \boxplus_8 (R3_2 \oplus A0_2) \boxplus_8 u_2) \boxplus_8 (A0_8 \oplus C_8) \boxplus_8 v_8$
	where	$u_8 = v_8 = 0$ and u_2, v_2 are known from Step 1
	derive	$\rightarrow (A0_2, A0_8, u_3, v_3, u_9, v_9)$
Step 4:	D_3	$= (R2_{12} \boxplus_8 (R3_{12} \oplus A0_{12}) \boxplus_8 u_{12}) \boxplus_8 (A0_3 \oplus C_3) \boxplus_8 v_3$
	D_{12}	$= (R2_3 \boxplus_8 (R3_3 \oplus A0_3) \boxplus_8 u_3) \boxplus_8 (A0_{12} \oplus C_{12}) \boxplus_8 v_{12}$
	where	$u_{12} = v_{12} = 0$ and u_3, v_3 are known from Step 3
	derive	$\rightarrow (A0_3, A0_{12}, u_{13}, v_{13})$
Step 5:	D_6	$= (R2_9 \boxplus_8 (R3_9 \oplus A0_9) \boxplus_8 u_9) \boxplus_8 (A0_6 \oplus C_6) \boxplus_8 v_6$
	D_9	$= (R2_6 \boxplus_8 (R3_6 \oplus A0_6) \boxplus_8 u_6) \boxplus_8 (A0_9 \oplus C_9) \boxplus_8 v_9$
	where	u_6, v_6, u_9, v_9 are known from Steps 2 and 3
	derive	$\rightarrow (A0_6, A0_9, u_7, v_7, u_{10}, v_{10})$
Step 6:	D_{10}	$= (R2_{10} \boxplus_8 (R3_{10} \oplus A0_{10}) \boxplus_8 u_{10}) \boxplus_8 (A0_{10} \oplus C_{10}) \boxplus_8 v_{10}$
	where	u_{10}, v_{10} are known from Step 5
	derive	$\rightarrow (A0_{10}, u_{11}, v_{11})$
Step 7:	D_7	$= (R2_{13} \boxplus_8 (R3_{13} \oplus A0_{13}) \boxplus_8 u_{13}) \boxplus_8 (A0_7 \oplus C_7) \boxplus_8 v_7$
	D_{13}	$= (R2_7 \boxplus_8 (R3_7 \oplus A0_7) \boxplus_8 u_7) \boxplus_8 (A0_{13} \oplus C_{13}) \boxplus_8 v_{13}$
	where	u_7, v_7, u_{13}, v_{13} are known from Steps 4 and 5
	derive	$\rightarrow (A0_7, A0_{13}, u_{14}, v_{14})$
Step 8:	D_{11}	$= (R2_{14} \boxplus_8 (R3_{14} \oplus A0_{14}) \boxplus_8 u_{14}) \boxplus_8 (A0_{11} \oplus C_{11}) \boxplus_8 v_{11}$
	D_{14}	$= (R2_{11} \boxplus_8 (R3_{11} \oplus A0_{11}) \boxplus_8 u_{11}) \boxplus_8 (A0_{14} \oplus C_{14}) \boxplus_8 v_{14}$
	where	$u_{11}, v_{11}, u_{14}, v_{14}$ are known from Steps 6 and 7
	derive	$\rightarrow (A0_{11}, A0_{14}, u_{15}, v_{15})$
Step 9:	D_{15}	$= (R2_{15} \boxplus_8 (R3_{15} \oplus A0_{15}) \boxplus_8 u_{15}) \boxplus_8 (A0_{15} \oplus C_{15}) \boxplus_8 v_{15}$
	where	u_{15}, v_{15} are known from Step 8
	derive	$\rightarrow (A0_{15})$

For each of the 2^{384} values of the initial guessing set $(R1, R2, B0)$, we could have different numbers, either zero or nonzero, of solutions for $A0$. Most of the guessing values will not even pass the first step in Table 2 as no valid solutions exist for the first D-equation, and we can immediately trace back to guess another value of $(R1, R2, B0)$; while other guessing values could have more than one solutions. However, we will show in Section 3.3.1 that the average number of solutions over $(R1, R2, B0)$ is exactly one.

The simplest way to enumerate all solutions is to use a recursion procedure. For example, we can loop for all values of $A0_0$ in the first step, and for each valid solution we recursively call the second step, and so on. If we only use simple loops for enumerating all the solutions in each step in Table 2, the constant c in the complexity will be quite big ($c \approx 2^8$), but later in Section 3.3.3 we will show how to reduce c to much smaller in a number of efficient ways.

3.2.3 Deriving $A1$ and final verification

After the above initial guessing and enumeration, we now know six out of seven 128-bit variables of the state. There will be 2^{384} guessing paths that arrive to this final stage of the attack. In order to derive the final 128-bit state variable $A1$, we use the fourth

keystream word $z^{(t+2)}$:

$$z^{(t+2)} = (R1^{(t+2)} \boxplus_{32} B1^{(t+2)}) \oplus R2^{(t+2)},$$

where

$$\begin{aligned}
R1^{(t+2)} &= \sigma(R2^{(t+1)} \boxplus_{32} (R3^{(t+1)} \oplus A1)) = \sigma(\mathrm{AES}_R(R1) \boxplus_{32} (\mathrm{AES}_R(R2) \oplus A1)),\\
R2^{(t+2)} &= \mathrm{AES}_R(R1^{(t+1)}) = \mathrm{AES}_R(\sigma(R2 \boxplus_{32} (R3 \oplus A0))),\\
B1^{(t+2)} &= A0^{(t+1)} \oplus l_\beta(B0^{(t+1)}) \oplus h_\beta(B1^{(t+1)})\\
&= A1 \oplus l_\beta(B1) \oplus h_\beta(A0 \oplus l_\beta(B0) \oplus h_\beta(B1)).
\end{aligned}$$

Denote $C' = l_\beta(B1) \oplus h_\beta(A0 \oplus l_\beta(B0) \oplus h_\beta(B1))$, then we can get the equation for $A1$:

$$z^{(t+2)} \oplus R2^{(t+2)} = \sigma(R2^{(t+1)} \boxplus_{32} (R3^{(t+1)} \oplus A1)) \boxplus_{32} (A1 \oplus C').$$

One can see that the equation above has exactly the same form as the expression for $A0$ in Equation (9), and therefore, we could use the ten steps in Table 2 to enumerate all solutions for $A1$. The distribution of the number of solutions will be the same and there will be one solution in average for each tuple of values of the other variables.

So far, we have guessed three state variables and determined the remaining four, such that the values of the seven 128-bit words satisfy the four consecutive 128-bit keystream words. The number of valid combinations of values is 2^{384} and in order to decide which one is correct, we use the subsequent three keystream words for verification. The verification only involves simple derivations thus the cost can be ignored.

3.3 Discussion on the complexity

3.3.1 Study of the two types of D-equations in the 10 steps

In this section, we compute the distribution of the number of solutions for the D-equations in Table 2 and show that the average value is exactly one.

In Equation (11), the input carry bits u_j, v_i can be removed by setting $R2'_j = R2_j \boxplus u_j$ and $D'_i = D_i \boxminus v_i$, respectively, which will not influence the distribution or the average value of the number of solutions. The ten steps in Table 2 can be divided into two equivalent types, which we denote by *Type-1* and *Type-2* D-equations.

Type-1 equations have the form:

$$A = (B \boxplus_n (C \oplus X)) \boxplus_n (X \oplus D),$$

where (A, B, C, D) are n-bit variables and X is the unknown which we need to enumerate. Such *Type-1* equations appear in Steps $\{0, 2, 6, 9\}$.

Type-2 equations have the form:

$$\begin{aligned}
A_1 &= (B_1 \boxplus_n (C_1 \oplus X_1)) \boxplus_n (X_2 \oplus D_1),\\
A_2 &= (B_2 \boxplus_n (C_2 \oplus X_2)) \boxplus_n (X_1 \oplus D_2),
\end{aligned}$$

where X_1, X_2 are two unknown variables that we want to enumerate, while others are n-bit known variables. Such *Type-2* equations appear in Steps $\{1, 3, 4, 5, 7, 8\}$.

For both types of equations, we have computed the distribution tables of the number of solutions for the unknown X-bytes, given that other known variables are uniformly distributed. We exhaustively (with some optimisations and cut-offs) try all the values of the known variables, and count the number of solutions for the unknowns.

Table 3 presents the probabilities of X having different numbers of solutions for *Type-1* equations corresponding to a random tuple (A, B, C, D) over $\mathbb{F}_{2^n}$. The probabilities are derived through $p = x/f$, where x's are the integers in the table and f is the corresponding normalisation factor. The probability of having at least one solution when $n = 8$ can be computed easily as $2^{-3.91}$. This means that in Equation (9), only $2^{-3.91}$ of the combinations of $(R2, R3, C, D)$ will result into valid solutions and continue with Step 1, and so on; while for the remaining majority of the combinations we just stop and trace back to the last step of the recursion. We can further compute the average number of solutions, Avr, as below:

$$Avr = \sum_{i=0}^{2^n-1} i \cdot Pr\{\#Solutions = i\}.$$

The computed average value is exactly one.

#Solutions normalisation factor f	n=1 2^1	n=2 2^3	n=3 2^5	n=4 2^7	n=5 2^9	n=6 2^{11}	n=7 2^{13}	n=8 2^{15}
0	1	5	23	101	431	1805	7463	30581
2	1	2	4	8	16	32	64	128
4		1	4	12	32	80	192	448
8			1	6	24	80	240	672
16				1	8	40	160	560
32					1	10	60	280
64						1	12	84
128							1	14
256								1

Table 3: Distribution table of the number of solutions of X for *type-1* equations.

We also derive the distribution and average value of the number of solutions for *Type-2* equations using a similar technique. The distribution table under different n values is given in Appendix 4. The probability of having at least one solution is $2^{-3.53}$ and the average number of solutions is one as well.

Since the ten tuples of equations are independent to each other (except the carriers, but the carriers do not influence the probability of having solutions), the probability of $A0$ having at least one solution is computed as $2^{-3.53\times 6-3.91\times 4} = 2^{-36.82}$. This means that only a small fraction, i.e., $2^{-36.82}$, of the 2^{384} initial guesses of $(R1, R2, B0)$ will actually have solutions for $A0$, while for other guessing values, the guessing process can be just terminated here. However, when $A0$ has valid solutions, the number of solutions will be around $2^{36.82}$ in average, and the overall average number of solutions is still one.

3.3.2 The total attack complexity

As it was mentioned earlier, the large fraction of the guesses $2^{384} \cdot (1 - 2^{-3.91})$ will fail in Step 0 in Table 2, as it involves solving a *Type-1* equation and the probability of having at least one solution there is $2^{-3.91}$. The remaining small fraction, $2^{384} \cdot 2^{-3.91} \approx 2^{380.09}$, of the guesses will advance to Step 1. The number of solutions in Step 0 will be $2^{3.91}$ in average, thus the total number of guessing paths that will arrive Step 1 is again $2^{380.09} \cdot 2^{3.91} \approx 2^{384}$. The same observation applies to every step in Table 2. Thus for 2^{384} input combinations to the recursive enumeration algorithm for deriving $A0$, we will get 2^{384} possible end-nodes, exactly one per guessing tuple $(R1, R2, B0)$ *in average*.

For the final step to determine $A1$, the situation will be the same, i.e., the majority of the derived six-word tuple will fail the first step, and only a small fraction will advance to the next step, and so on. The average number of solutions is again one and there are 2^{384} valid guessing paths. Thus the total complexity of the GnD attack is 2^{384}.

3.3.3 Further reducing the complexity constant c

The complexity is written as $c \cdot 2^{384}$ where c is the complexity of operations involved in each guessing path, mainly lies in solving the D-equations of either type.

Bit-wise enumeration recursion instead of byte-wise. Recall that the simplest way to enumerate all solutions for $A0$ is to make a byte-wise recursion of depth ten, and in each step we loop over the unknown *byte(s)* of $A0$, thus the overall enumeration recursion will have a constant factor $c = 256$ steps. However, we can change the recursion to be deeper with depth $10 \cdot 8$ and search for solutions of each *bit(s)* of $A0$. This will shrink the constant c from 256 down to 2 [1], since now we only need to test the binary bit-value(s) before going to the next recursion depth while considering the resulting carriers from the current step. So we can enumerate the 128-bit unknown $A0$ by deriving one or two bits in each recursive step. We have actually implemented such a bit-wise recursive enumeration algorithm, see Appendix 3. Note that the proposed recursion is linear with a fixed depth, and may as well be organised as a number of (many) nested loops.

Precomputed lookup tables. Another approach is to precompute lookup tables helping to instantly give the list of sub-solutions for each tuple of D-equations. The tables record all the possible values of the known variables and the corresponding solutions for the unknowns.

The smallest table will be of size $2^{32} \rightarrow 256 \times 10$ bits for Step 0, where each entry corresponds to one value of the known variables, 256 is the maximum number of possible solutions corresponding to one entry, and 10 bits correspond to the value of one unknown (one byte) and two carriers (two bits). There will be exactly 2^{32} valid records of size 10 bits in the table. An example of the smallest table is as below:

$$T_0[R2_0, R3_0, C_0, D_0] \rightarrow \{A0_0, u_1, v_1\}.$$

[1] For a Type-2 D-equation we can loop over the first unknown bit-value (0 or 1, thus $c = 2$), then derive the second unknown bit-value using the first equation, and then test the pair of the bits using the second equation.

The largest table is of size $2^{68} \rightarrow 256 \times 20$ bits in Step 5:

$$T_5[R2_6, R3_6, C_6, D_6, u_6, v_6, R2_9, R3_9, C_9, D_9, u_9, v_9] \rightarrow \{A0_6, A0_9, u_7, v_7, u_{10}, v_{10}\}.$$

Truncating guessing paths reaching the 10-step stage for deriving $A0$. The number of guessing paths that reach the 10-step stage for deriving $A0$ can be further reduced by guessing the variables in the initial set in bytes, instead of 128 bits, in a careful order. We give a simple example here, and there exist some more tricky ones. We first guess the following 25 bytes and 2 bits in complexity 2^{202}:

$$R1_{0,1,3,4,5,9,10,14,15},\ R2_{0,1,4,5},\ R3_{0,1,4,5},\ B0_{0,1,4,5,6,7,10,11},\ w_{0,4},$$

where $w_{0,4}$ are two carry bits for 32-bit additions. Then the following variables can be derived:

$$\begin{aligned}
D_0 &= z_0^{(t+1)} \oplus (2 \cdot S(R1_0) \oplus 3 \cdot S(R1_5) \oplus 1 \cdot S(R1_{10}) \oplus 1 \cdot S(R1_{15})),\\
D_1 &= z_1^{(t+1)} \oplus (1 \cdot S(R1_0) \oplus 2 \cdot S(R1_5) \oplus 3 \cdot S(R1_{10}) \oplus 1 \cdot S(R1_{15})),\\
D_4 &= z_4^{(t+1)} \oplus (1 \cdot S(R1_3) \oplus 2 \cdot S(R1_4) \oplus 3 \cdot S(R1_9) \oplus 1 \cdot S(R1_{14})),\\
D_5 &= z_5^{(t+1)} \oplus (1 \cdot S(R1_3) \oplus 1 \cdot S(R1_4) \oplus 2 \cdot S(R1_9) \oplus 3 \cdot S(R1_{14})),\\
B1_{0||1} &= (z_{0||1}^{(t)} \oplus R2_{0||1}) \boxminus_{32} R1_{0||1} \boxminus_{32} w_0,\\
B1_{4||5} &= (z_{4||5}^{(t)} \oplus R2_{4||5}) \boxminus_{32} R1_{4||5} \boxminus_{32} w_4,\\
C_{0||1} &= \beta B0_{0||1} \oplus B0_{6||7} \oplus \beta^{-1} B1_{0||1},\\
C_{4||5} &= \beta B0_{4||5} \oplus B0_{10||11} \oplus \beta^{-1} B1_{4||5}.
\end{aligned}$$

With the set of the guessed and determined values, we can now check whether a solution for bytes $A0_0, A0_1, A0_4, A0_5$ exists, in the first three steps in Table 2. The

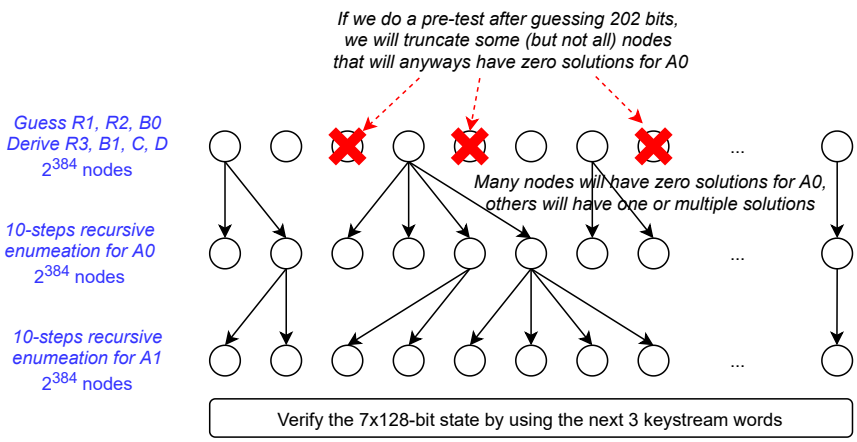

Figure 2: Illustration of the first GnD attack.

probability of valid solutions, denoted p_{0-2}, can be computed as $p_{0-2} = 2^{-3.91\times 2-3.52} = 2^{-11.34}$. If no solutions exist, we just roll back and make another guess; otherwise we guess the remaining 23 ($= 48 - 25$) bytes of the initial guessing set and run the 10-step algorithm to enumerate all values of $A0$. The total number of nodes T' that will arrive to the 10-step stage will be:

$$T' = 2^{200+2} + (p_{0-2} \cdot 2^{200+2}) \cdot 2^{184-2} = 2^{202} + p_{0-2} \cdot 2^{384}.$$

This means that only $2^{372.66}$ guessing paths (out of 2^{384}) will reach the 10-step stage for enumerating $A0$. However, the total complexity will still be 2^{384}, as the fact that there are 2^{384} solutions satisfying the three consecutive keystream words remains unchanged. Figure 2 gives an illustration of the first GnD attack and the "effect" of the idea to do a pre-test after guessing only 202 bits.

4 The second guess-and-determine attack ($T = 2^{378.16}$)

In this section, we provide a second guess-and-determine attack which can further reduce the complexity by using one additional "backward" keystream block $z^{(t-2)}$ as *side information* to truncate more "forward" guessing paths. Thus, this approach needs eight keystream words. The improvement over the first GnD attack is not so significant, but the idea of exploiting more equation constraints to truncate guessing paths itself is interesting, and our second GnD attack serves as a direct illustration of it.

4.1 Use $z^{(t-2)}$ to truncate more guessing paths

If we want to further reduce the complexity of the first GnD attack, we could try to see if we can use some additional information, *besides* those seven keystream words that are already involved in the first attack. With such additional information, we can truncate some portion of the guessing paths that have solutions for the D-equations while not for the additional information, *proportionally*. Thus, the average number of end-points 2^{384} will be reduced proportionally as well. Specifically, we use one additional keystream word at clock $t-2$, i.e., $z^{(t-2)}$, to impose more constraints and truncate more guessing paths. The expression of $z^{(t-2)}$ is shown below:

$$z^{(t-2)} = (R1^{(t-2)} \boxplus_{32} B1^{(t-2)}) \oplus R2^{(t-2)},$$

where $R1^{(t-2)}, B1^{(t-2)}, R2^{(t-2)}$ are derived as follows:

$$\begin{aligned}
R1^{(t-2)} &= \mathrm{AES}_R^{-1}(R2^{(t-1)}) = \mathrm{AES}_R^{-1}(\mathrm{AES}_R^{-1}(R3)),\\
B1^{(t-2)} &= B0^{(t-1)},\\
R2^{(t-2)} &= \mathrm{AES}_R^{-1}(R3^{(t-1)}) = \mathrm{AES}_R^{-1}((\sigma(R1) \boxminus_{32} R2^{(t-1)}) \oplus A0^{(t-1)})\\
&= \mathrm{AES}_R^{-1}((\sigma(R1) \boxminus_{32} \mathrm{AES}_R^{-1}(R3)) \oplus A0^{(t-1)}).
\end{aligned}$$

According to the LFSR update function, we can derive:

$$A0^{(t-1)} = B1 \oplus l_\beta(B0^{(t-1)}) \oplus h_\beta(B1^{(t-1)}) = B1 \oplus l_\beta(B0^{(t-1)}) \oplus h_\beta(B0).$$

Thus $z^{(t-2)}$ can be written as an equation in one unknown variable $B0^{(t-1)}$ (given that other variables are either known, guessed, or determined):

$$z^{(t-2)} = (\underbrace{\mathrm{AES}_R^{-1}(\mathrm{AES}_R^{-1}(R3))}_{X} \boxplus_{32} B0^{(t-1)}) \\ \oplus \mathrm{AES}_R^{-1}(\underbrace{(\sigma(R1) \boxminus_{32} \mathrm{AES}_R^{-1}(R3)) \oplus h_\beta(B0) \oplus B1}_{Y} \oplus l_\beta(B0^{(t-1)})).$$

Using X, Y to denote the expressions in the brackets, we could simplify the above equation as $z^{(t-2)} = (X \boxplus_{32} B0^{(t-1)}) \oplus \mathrm{AES}_R^{-1}(Y \oplus l_\beta(B0^{(t-1)}))$. Similar to the situation for $A0$ in our first GnD attack, $B0^{(t-1)}$ appears twice with non-linear operations in between, thus it can have different numbers of solutions given specific values of X, Y. If we change to initially guess the two 128-bit variables X and Y, the expression of $z^{(t-2)}$ can help to truncate more guessing paths that have no valid solutions for $B0^{(t-1)}$. Specifically, for each guessing value of (X, Y), if we can immediately give a binary answer, i.e,. Yes or No, about whether there is at least one solution for $B0^{(t-1)}$, we can discard those (X, Y) values with no solutions, and only continue guessing the third 128-bit variable for the others. Note that we do not enumerate solutions for $B0^{(t-1)}$ in $z^{(t-2)}$, otherwise we would get the same complexity 2^{384} as the first GnD attack, and we will later show how we efficiently get the binary answer in Section 4.2.1. Actually, we will guess $(X, B0^{(t-1)})$ instead of (X, Y) there, but we still first describe the idea by guessing (X, Y) as it is easier to illustrate how $z^{(t-2)}$ is exploited.

Let p_z denote the probability that $B0^{(t-1)}$ has solutions in the equation of $z^{(t-2)}$, then the total complexity of the second GnD attack would be computed as:

$$T = \underbrace{2^{256}}_{\text{guess } X,Y} \cdot (\underbrace{(1-p_z)}_{\text{“No”}} + \underbrace{p_z}_{\text{“Yes”}} \cdot \underbrace{2^{128}}_{\text{3rd guess}}) \approx p_z \cdot 2^{384}.$$

We have derived the specific value of p_z in Appendix 7, which is $2^{-5.84}$, thus the total complexity of the second GnD attack is around $2^{384-5.84} \approx 2^{378.16}$.

4.2 Scenario of the second GnD attack

The flowchart of the second GnD attack is given in Appendix 6, which follows the steps below:

(1) Guess X and Y with complexity 2^{256}.

(2) For each (X, Y) value, check if $B0^{(t-1)}$ in $z^{(t-2)}$ has solutions: if yes, continue with guessing the third variable in the next step; otherwise roll back to the last step.

(3) Guess $B0$ in complexity 2^{128} and further derive $R2, R3$ as below:

$$R3 \text{ from: } X = \mathrm{AES}_R^{-1}(\mathrm{AES}_R^{-1}(R3)),$$
$$R2 \text{ from: } z^{(t-1)} = (\mathrm{AES}_R^{-1}(R2) \boxplus_{32} B0) \oplus \mathrm{AES}_R^{-1}(R3)).$$

This step will be entered $p_z \cdot 2^{256}$ times in average.

(4) For each valid combination of $(X, Y, B0)$, we get the following two equations in two unknowns $R1$ and $B1$:

$$z^{(t)} \oplus R2 = R1 \boxplus_{32} B1,$$
$$Y \oplus h_\beta(B0) = (\sigma(R1) \boxminus_{32} \mathrm{AES}_R^{-1}(R3)) \oplus B1. \tag{12}$$

We check if $B1, R1$ have valid solutions given other variables, and roll back if the answer is negative, otherwise we **enumerate** all solutions recursively. We have computed the distribution and average value of the number of solutions using the similar way for the D-equations in the first GnD attack, and the details are given in Appendix 5. There is again one solution in average for each combination of the known variables. Similarly, lookup tables can be precomputed to help enumerate solutions efficiently.

(5) **Enumerate** all solutions for $A0$ as done in Section 3.2.2 in the first GnD attack;

(6) **Enumerate** all solutions for $A1$ as done in Section 3.2.3 in the first GnD attack;

(7) Use the next three keystream words to verify the correct guess.

4.2.1 Guess $(X, B0^{(t-1)})$ instead of (X, Y)

In the first step, we need to give a binary answer about whether solutions exist for $B0^{(t-1)}$ in the equation:

$$z^{(t-2)} = (B0^{(t-1)} \boxplus_{32} X) \oplus \mathrm{AES}_R^{-1}(l_\beta(B0^{(t-1)}) \oplus Y).$$

One simple way to achieve this is to run an enumeration algorithm on $B0^{(t-1)}$, and whenever a solution is found, we stop and return "Yes". This is similar to the process of computing p_z in Appendix 7. The process is actually an enumeration algorithm on $B0^{(t-1)}$ with complexity $2^{48-5.84}$, resulting in the total complexity even higher than 2^{384}.

However, we can actually guess $(X, B0^{(t-1)})$ instead of (X, Y), and Y can be uniquely determined given $(X, B0^{(t-1)})$. But it could happen that for different $(X, B0^{(t-1)})$ pairs, the values of (X, Y) are the same. So for every new X we must ensure that the value of Y is new, and skip the cases when the pair (X, Y) has already been considered. Thus, for each new value of X we make a binary vector of length 2^{128} in which we flag (i.e., set to 1) those Y's that have already been considered for that specific value of X. Thus, in step (1) in Section 4.2, we guess $(X, B0^{(t-1)})$ and determine Y, and in step (2), we check if (X, Y) pair has already been flagged: if so, we roll back to guess another value; otherwise, continue with guessing $B0$ in step (3). Other steps are just the same as before.

```
T = 0;
N = pow(2, 128); // 2 to the power of 128
char flag[N];
for(X = 0; X < N; ++X)
{   for(i = 0; i < N; ++i)
        flag[i] = 0;
    for(B0 = 0; B0 < N; ++B0) // B0 at clock t-1
    {   derive Y;
        if(flag[Y] == 0)
        {   // we enter this branch with probability p_z in average
```

```
                flag[Y] = 1;
                for(B0t = 0; B0t < N; ++B0t)
                // guess the third unknown B0t: B0 at clock t
                {   T = T + 1; // complexity to enumerate all guess basis
                    (*)...further derivation and enumerations, Steps 3-7
                }
            }
        }
}
```

Listing 3: Outline of the second GnD attack.

Listing 3 gives the pseudo-code of the second GnD attack. It is easy to see that the number of times that the GnD attack arrives to the point (*) is $T \approx 2^{256} \cdot p_z \cdot 2^{128}$ where $p_z = 2^{-5.84}$, thus the complexity is about 2^{378}. However, in order to gain the advantage in time complexity over the first GnD attack we have to use memory of size 2^{128} bits.

5 Linear cryptanalysis of SNOW-V$_{\oplus}$

The basic idea of linear cryptanalysis is to approximate the non-linear operations of a cipher as linear ones, and further to explore linear relationships either between keystream words, or between keystream words and initial states, which could result into a distinguishing attack or a correlation attack, respectively. Usually, such a linear approximation will introduce a noise, and the quality of the linear approximation is measured by the bias of this noise, which will directly influence the attack complexity. There are many ways to define the bias and derive the complexity, and in our attack, we use SEI as defined in [BJV04]. For a variable with distribution D, the SEI of it is computed as:

$$\epsilon(D) = |D| \cdot \sum_{i=0}^{|D|-1} \left(D[i] - \frac{1}{|D|} \right)^2,$$

where $D[i]$ is the occurrence probability of the value in the i-th entry. For a distribution with SEI $\epsilon(D)$, the number of samples required to distinguish it from the uniform random distribution is in the order of $1/\epsilon(D)$ [BJV04].

In this section, we perform linear cryptanalysis of SNOW-V and propose a distinguishing attack with complexity 2^{303} against a reduced version SNOW-V$_{\oplus}$ in which the 32-bit adders are replaced with exclusive-OR. In the attack we explore the feature that three consecutive keystream words contain the contribution from the LFSR linearly and redundantly, due to the chosen tap positions of $T1$ and $T2$ in the design. Thus, unlike the linear attacks against the predecessors SNOW 2.0 (e.g., [WBDC03, NW06]), and SNOW 3G [YJM19], where one first approximates the FSM and then cancels out the contribution from the LFSR either according to the feedback polynomial or a multiple of it, here we do it vice-versa. We will first cancel the LFSR variables locally within these three keystream words without combining several time instances, and thereafter construct a noise expression based on the remaining expressions over the FSM variables.

5.1 Linear approximation

We first express the operations in the AES encryption round as $L \cdot S$, where S denotes S-box operation and L is the combination of the `ShiftRow` and `MixColumn` operations.

Similarly, the inverse AES encryption round can be expressed as $S^{-1} \cdot L^{-1}$, where S^{-1} denotes the inverse S-box operation and L^{-1} is the combination of inverse `MixColumn` and inverse `ShiftRow` operations. L and L^{-1} can be expressed as two 16×16-byte matrices, in which each entry is an element from $\mathbb{F}_{2^8}$. The expressions of L and L^{-1} are given in Appendix 1. Besides, we replace $\boxplus_{32}$ with $\oplus$, and make a substitution of the variables $R2, R3$ as $L \cdot R2, L \cdot R3$, respectively. Hence, $R2, R3$ are not the original variables, but for ease of reading, we still use the original notations. Then the expressions of the three consecutive keystream words in Equation (7) can be rewritten as follows:

$$\begin{aligned} z^{(t-1)} &= S^{-1}(R2) \oplus B0 \oplus S^{-1}(R3), \\ z^{(t)} &= R1 \oplus B1 \oplus L \cdot R2, \\ z^{(t+1)} &= \sigma L \cdot R2 \oplus \sigma L \cdot R3 \oplus (\sigma A0 \oplus A0) \oplus l_\beta(B0) \oplus h_\beta(B1) \oplus L \cdot S(R1). \end{aligned}$$

The variables $B0, B1, A0$ are contributions from the LFSR, and we would like to cancel them out first. To achieve so, we apply two linear operations l_β, h_β, which can be expressed as two 128×128 binary matrices, to $z^{(t)}$ and $z^{(t-1)}$, respectively, and introduce a new 128-bit variable W defined as below:

$$W = l_\beta(z^{(t-1)}) \oplus h_\beta(z^{(t)}) \oplus z^{(t+1)}. \tag{13}$$

The contribution from the variables $B0$ and $B1$ is cancelled in W, and what remains from the LFSR is only $(\sigma A0 \oplus A0)$. Now let us introduce ten byte-based variables from W, shown below:

$$\begin{aligned} &E_0 = W_0, \quad E_1 = W_1 \oplus W_4, \quad E_2 = W_5, \quad E_3 = W_2 \oplus W_8, \quad E_4 = W_6 \oplus W_9, \\ &E_5 = W_{10}, \quad E_6 = W_3 \oplus W_{12}, \quad E_7 = W_7 \oplus W_{13}, \quad E_8 = W_{11} \oplus W_{14}, \quad E_9 = W_{15}, \end{aligned}$$

where W_i is the i-th byte of W. Each byte-wise expression E_k $(0 \le k \le 9)$ cancels out the contribution from $A0$, and only the byte variables from registers $R1, R2, R3$ remain. Each of the above E_k terms can be expressed in a form as below:

$$\begin{aligned} E_k = &\bigoplus_{i=0}^{15} [l_{k,i}^{(1)} \cdot R1_i \oplus n_{k,i}^{(1)} \cdot S(R1_i)] \\ &\oplus [l_{k,i}^{(2)} \cdot R2_i \oplus n_{k,i}^{(2)} \cdot S^{-1}(R2_i)] \oplus [l_{k,i}^{(3)} \cdot R3_i \oplus n_{k,i}^{(3)} \cdot S^{-1}(R3_i)], \end{aligned} \tag{14}$$

where $l_{k,i}^{(j)}, n_{k,i}^{(j)}$ $(j \in \{1,2,3\}, 0 \le k \le 9, 0 \le i \le 15)$ are 8×8 binary matrices that can be derived following the expressions of W and E terms. This means that each E_k can contain up to 48 independent noise terms of the form $ax \oplus bS(x)$, i.e., up to 48 approximations of the S-boxes or the inverse S-boxes. We can derive the expression for the total noise N as a linear combination of these ten E-bytes as follows:

$$N = c_0 \cdot E_0 \oplus c_1 \cdot E_1 \oplus \cdots \oplus c_9 \cdot E_9,$$

where c_i's are linear masking coefficients or binary matrices that an attacker can freely choose. It is computationally infeasible to exhaust all the values of these matrices, and below we show how we efficiently search them to achieve a decent bias.

Since we have ten byte expressions each of which can have up to 48 S-box approximations, it is possible to find some linear combinations of these ten bytes such that some S-box approximations could be removed in N, i.e., the coefficients of the linear part and the S-box part of some bytes both become zero. Now we are interested in the maximum number of S-box approximations that can be removed, as it can give a higher bias.

We first use MILP (Mixed-Integer Linear Programming) to help find a lower bound on the number of active S-boxes, as done in [ENP19]. By solving the MILP problem, we get a first insight that there will be not less than 37 active S-boxes. We next show how we explore linear masking coefficients to remove as many S-box approximations as possible.

5.2 Exploring maskings to remove S-box approximations

We can construct a w-bit noise N_w using the ten 8-bit E-expressions, which is expressed in a matrix form as below:

$$N_w = \begin{pmatrix} c_0 & c_1 & \dots & c_9 \end{pmatrix}_{w\times 10\cdot 8} \cdot \begin{pmatrix} E_0 \\ E_1 \\ \vdots \\ E_9 \end{pmatrix}_{10\cdot 8} = \mathbf{c}\cdot\mathbf{E},$$

where c_i's, $0 \le i \le 9$, are $w \times 8$ binary matrices that the attacker can choose freely, but with the constraint that the rank of $\mathbf{c}$ is w, i.e., all w rows are nonzero and linearly independent. For simplicity, let us introduce 96 8-bit variables as follows:

$$\begin{aligned} \text{for } i = 0, \dots, 15: \quad X_i &= R1_i, & Y_i &= S(R1_i), \\ X_{16+i} &= R2_i, & Y_{16+i} &= S^{-1}(R2_i), \\ X_{32+i} &= R3_i, & Y_{32+i} &= S^{-1}(R3_i). \end{aligned}$$

Note that every X_j $(0 \le j \le 47)$ can be regarded as a uniformly distributed random variable, and Y_j is the corresponding value after applying the S-box or inverse S-box. Thus, an expression of the form $a \cdot X_j \oplus b \cdot Y_j$, where a, b are two linear maskings, can be possibly biased only when $a \neq 0, b \neq 0$. When $a = 0, b \neq 0$ or $a \neq 0, b = 0$, the expression will be uniform; and when $a = 0, b = 0$, this approximation can be removed. Since every E_i is a linear expression of the X, Y variables, the expression of the noise N_w can be rewritten as:

$$\begin{aligned} N_w &= \begin{pmatrix} c_0 & c_1 & \dots & c_9 \end{pmatrix}_{w\times 10\cdot 8} \cdot \left[\mathbf{A}_{10\cdot 8\times 48\cdot 8} \cdot \begin{pmatrix} X_0 \\ \vdots \\ X_{47} \end{pmatrix}_{48\cdot 8} \oplus \mathbf{B}_{10\cdot 8\times 48\cdot 8} \cdot \begin{pmatrix} Y_0 \\ \vdots \\ Y_{47} \end{pmatrix}_{48\cdot 8} \right] \\ &= \mathbf{c}\cdot[\mathbf{A}\cdot\mathbf{X} \oplus \mathbf{B}\cdot\mathbf{Y}], \end{aligned}$$

where $\mathbf{A}$ and $\mathbf{B}$ are two $10\cdot 8 \times 48\cdot 8$ binary matrices derived from the ten E-expressions in Equation (14). It is therefore clear that the total w-bit noise N_w consists of at most 48 sub-noise parts:

$$N_w = \bigoplus_{i=0}^{47} \underbrace{(\mathbf{c}\cdot\mathbf{A})_{[0:w-1;\ 8i:8i+7]}}_{a_i} \cdot X_i \oplus \underbrace{(\mathbf{c}\cdot\mathbf{B})_{[0:w-1;\ 8i:8i+7]}}_{b_i} \cdot Y_i,$$

where a_i and b_i are $w \times 8$ binary sub-matrices, constituted from the w rows and the eight columns from $8i$ to $8i+7$ of the matrices $\mathbf{c} \cdot \mathbf{A}$ and $\mathbf{c} \cdot \mathbf{B}$, respectively. There are in total 96 such matrices.

Obviously, if $a_i = b_i = 0$, the i-th sub-noise part vanishes to zero, and thus the total noise will have a larger bias. If, on the other hand, only one of the two matrices is zero, the contribution of that i-th sub-noise will make some or all bits of N_w pure random, thus these bits will have no contribution to the bias. If all bits are affected and become random, the total bias will be 0. Therefore, we are interested in selecting the masking matrix $\mathbf{c}$ such that we can cancel as many S-box approximations out of 48 as possible, meanwhile guaranteeing that the xor-sum of the remaining sub-noises is biased. Next we show how we achieve this.

Algorithm to derive the linear masking matrix c. Let us select k distinct indices $\{i_1, i_2, \ldots, i_k\} \in \{0, 1, \ldots, 47\}$, and we want to cancel the sub-noise parts corresponding to these k indices, i.e., to make $a_{i_j} = b_{i_j} = 0$ for $j = 1, 2, \ldots, k$, by carefully choosing the linear masking c_i's. We can construct a matrix $\mathbf{K}$ that consists of the corresponding 8-bit columns taken from the matrices $\mathbf{A}$ and $\mathbf{B}$:

$$\mathbf{K}_{10\cdot 8 \times 2k\cdot 8} = \begin{pmatrix} \mathbf{A}_{[0:7;\ 8i_1:8i_1+7]} & \mathbf{B}_{[0:7;\ 8i_1:8i_1+7]} & \cdots & \mathbf{A}_{[0:7;\ 8i_k:8i_k+7]} & \mathbf{B}_{[0:7;\ 8i_k:8i_k+7]} \\ \mathbf{A}_{[8:15;\ 8i_1:8i_1+7]} & \mathbf{B}_{[8:15;\ 8i_1:8i_1+7]} & \cdots & \mathbf{A}_{[8:15;\ 8i_k:8i_k+7]} & \mathbf{B}_{[8:15;\ 8i_k:8i_k+7]} \\ & & \vdots & & \\ \mathbf{A}_{[72:79;\ 8i_1:8i_1+7]} & \mathbf{B}_{[72:79;\ 8i_1:8i_1+7]} & \cdots & \mathbf{A}_{[72:79;\ 8i_k:8i_k+7]} & \mathbf{B}_{[72:79;\ 8i_k:8i_k+7]} \end{pmatrix},$$

and we want to find a nonzero matrix $\mathbf{c}$ such that:

$$\mathbf{c}_{w\times 80} \cdot \mathbf{K}_{80\times 2k\cdot 8} = \mathbf{0}_{w\times 2k\cdot 8}.$$

First of all, if the rank r of the matrix $\mathbf{K}$ is 80, there are no valid solutions of $\mathbf{c}$ satisfying $\mathbf{c} \cdot \mathbf{K} = \mathbf{0}$. While if $r < 80$, there exist $w = 80 - r$ nonzero linear combinations that will map through $\mathbf{K}$ to zero. This also explains how the size w for the total noise N_w was derived in our attack.

In order to search for the kernel linear combinations, we initially set $\mathbf{c}$ as a square identity matrix $\mathbf{c}_{80\times 80} = \mathbf{I}_{80\times 80}$, then perform the standard Gaussian elimination on the binary matrix $\mathbf{K}$ to transform it to the row echelon form $\mathbf{K}'$, and apply the same operations to the matrix $\mathbf{c}_{80\times 80}$. This is quite similar to the steps of deriving an inverse matrix of $\mathbf{K}$, if $\mathbf{K}$ would be a square matrix.

In the end, we get the row echelon form $\mathbf{K}' = \mathbf{c} \cdot \mathbf{K}$, where the last $w = 80 - r$ rows of $\mathbf{K}'$ are zeroes, while the matrix $\mathbf{c}$ will be of the full-rank 80. Then we keep the last w rows of $\mathbf{c}$ and discard all other r rows, thus deriving the desired $\mathbf{c}_{w\times 80}$ satisfying $\mathbf{c}\cdot\mathbf{K} = \mathbf{0}$.

Search strategy for a good linear approximation. It is now clear that a larger bias of the total noise can be achieved by removing as many S-box approximations (out of 48) as possible. We can do it by exhaustively selecting k indices in $\binom{48}{k}$ ways, then applying the algorithm above to check if a solution for the matrix $\mathbf{c}$ exists for the selected sub-noises, and if so, derive w and the corresponding linear masking matrix $\mathbf{c}$.

Then given the derived $\mathbf{c}_{w\times 80}$, we construct the distribution of the total w-bit noise N_w and compute the bias. We pick the solution for which the total bias is the largest.

Correction approach. For many k-tuples of indices we would get a full-rank $\mathbf{K}$, and thus we do not have to continue further computations. However, another step of cutting out k-tuples is to do a *correction* approach for the matrix $\mathbf{c}$. If w is shrunk down to 0 during such a correction, there is no need to continue further computations and we jump to the next k-tuple. The *correction* idea is as follows.

Given a derived masking matrix $\mathbf{c}_{w\times 80}$, we can meet the situation when some of the 48 sub-noises will have $a_i = 0$ and $b_i \neq 0$ (or vice versa), which means that some bits of the w-bit total noise become uniformly distributed. In such a case, we can try to *correct* the masking matrix $\mathbf{c}_{w\times 80}$ by removing those rows where the rows of b_i are nonzero. In this way we shrink w down but get $a_i = b_i = 0$. If w becomes 0 at the end of this procedure, we proceed to the next k-tuple.

If for all 48 sub-noises we get either $a_i = 0, b_i = 0$ or $a_i \neq 0, b_i \neq 0$, the resulting linear masking matrix $\mathbf{c}$ may lead to a biased total noise. We then construct the distribution of the total noise N_w and compute the corresponding bias. When constructing the distribution, we can utilise the Walsh-Hadamard Transforms to speed up the convolution of the 48 w-bit sub-noises [MJ05, YJM19].

Results. In our simulations we managed to find a 16-bit approximation N_{16}, i.e., $w = 16$, and the masking matrix $\mathbf{c}_{16\times 80}$ can effectively eliminate nine S-box approximations. The received bias (SEI) is

$$\epsilon(N_{16}) \approx 2^{-303}.$$

The linear masking $\mathbf{c}_{16\times 80}$ is given in Listing 4, where the bits are encoded as 64-bit unsigned integers in C/C++, and are mapped to the bits of $\mathbf{c}$ as follows:

$$\mathbf{c}_{16\times 80}[i,j] = (C[i][j/64] \gg (j\%64))\&1.$$

```
uint64_t C[16][2] = {
{ 0x0000020200020000ULL, 0x0000ULL}, { 0x94730000005e0000ULL, 0x0000ULL},
{ 0x0000080800080000ULL, 0x0000ULL}, { 0x48c4159600fa0120ULL, 0x0002ULL},
{ 0x48c421a200ce0120ULL, 0x0002ULL}, { 0x0000444400440000ULL, 0x0000ULL},
{ 0x3c1581000022080ULL, 0x0001ULL}, { 0x0000000000000022ULL, 0x0000ULL},
{ 0x40c1000000600000ULL, 0x0100ULL}, { 0x0000000000000008ULL, 0x0000ULL},
{ 0x0000000000000060ULL, 0x0000ULL}, { 0x0000000000000021ULL, 0x0000ULL},
{ 0x0000000000000004ULL, 0x0000ULL}, { 0x0000000000000010ULL, 0x0000ULL},
{ 0x4b39000000ee0000ULL, 0x0000ULL}, { 0x54cc000000fe0000ULL, 0x8000ULL}};
```

Listing 4: The linear masking $\mathbf{c}_{16\times 80}$.

We also tested if there exists a linear masking that can eliminate ten or more S-box approximations. We ran our exhaustive search program with $k = 10$ for all the $\binom{48}{10} \approx 2^{32.6}$ 10-tuples, but with no valid results returned. By this we confirm that at most nine S-box approximations can be removed from the total noise expression.

5.3 Distinguishing attack

If all arithmetic additions are substituted with exclusive-OR, we could have a distinguishing attack against this variant with data complexity 2^{303}. Specifically, one should

collect around 2^{303} different triples of consecutive keystream words and construct the sequence of 16-byte words $\{W\}$ of length 2^{303} by applying Equation (13) for each triple; then build the sequence of 10-byte words $\{E\}$ from $\{W\}$; and, finally, apply the linear masking $\mathbf{c_{16\times 80}}$ given in Listing 4 to each word in $\{E\}$, thus receiving the sequence of length 2^{303} of biased 16-bit noise samples $\{N_{16}\}$, which can be distinguished from random.

The bias derived in our attack does not depend on the key or IV, and the time width to build a single sample is just three keystream words, which means that the data in our attack can be collected from many short keystream sequences under different (key, IV) pairs. Though the data complexity is still out of reach in practice, the attacking scenario is more relevant to the practical situation. The attack can also be used to recover some unknown bits of a plaintext encrypted a large number of times with different IVs and potentially different keys, e.g., in a broadcast setting [SSS+19].

Discussion on the full version. If we take the 32-bit adders into consideration, the bias would change. However, how the bias would vary is not clear, as the $\boxplus_{32}$ operations can be seen as part of multiple S-boxes and their approximations. On the other hand, it is computationally difficult to compute the bias by exhaustive looping. We do not have a good idea about how to compute that bias in practice, and leave it as an open question for further research.

6 Conclusions

In this paper, we investigate the security of SNOW-V and propose two guess-and-determine attacks with complexities 2^{384} and 2^{378}, respectively, and one distinguishing attack against a reduced version, in which the 32-bit adders are replaced with exclusive-OR, with complexity 2^{303}. These attacks do not threaten the full SNOW-V, but provide deeper understanding into its security. Besides, our attacks provide new ideas for cryptanalysis against other ciphers. Specifically, we recommend that in a guess-and-determine attack, instead of simple looping, one should carefully design the order of the guessing and always truncate those paths invalidating some equation constraints. In this way, one can save the cost for going through the invalid guessing paths and thus the complexity can be reduced. A very interesting open problem would be to investigate whether there are possible speed-ups for these kind of GnD attacks using quantum computers. For linear cryptanalysis against LFSR-based stream ciphers, it might be interesting to check if the LFSR contribution can be cancelled locally first, then the remaining equations on FSM variables may be used to construct a biased noise.

Acknowledgements

We thank the reviewers for valuable comments and questions that made it possible to improve this paper at a great extent. This work was in part financially supported by the Swedish Foundation for Strategic Research, grant RIT17-0005 and the ELLIIT program. Jing Yang is also supported by the scholarship from the National Digital Switching System Engineering and Technological Research Center, China.

References

[3GP19] 3GPP. TS 33.841 (V16.1.0): 3rd generation partnership project; technical specification group services and systems aspects; security aspects; study on the support of 256-bit algorithms for 5G (release 16). March 2019. `https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3422`.

[BJV04] Thomas Baigneres, Pascal Junod, and Serge Vaudenay. How far can we go beyond linear cryptanalysis? In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 432–450. Springer, 2004.

[CBB20] Andrea Caforio, Fatih Balli, and Subhadeep Banik. Melting SNOW-V: improved lightweight architectures. *Journal of Cryptographic Engineering*, pages 1–21, 2020.

[CDM20] Carlos Cid, Matthew Dodd, and Sean Murphy. A security evaluation of the SNOW-V stream cipher. 4 June 2020. Quaternion Security Ltd. `https://www.3gpp.org/ftp/tsg_sa/WG3_Security/TSGS3_101e/Docs/S3-202852.zip`.

[EJ02] Patrik Ekdahl and Thomas Johansson. A new version of the stream cipher SNOW. In *International Workshop on Selected Areas in Cryptography*, pages 47–61. Springer, 2002.

[EJMY19] Patrik Ekdahl, Thomas Johansson, Alexander Maximov, and Jing Yang. A new SNOW stream cipher called SNOW-V. *IACR Transactions on Symmetric Cryptology*, pages 1–42, 2019.

[ENP19] Maria Eichlseder, Marcel Nageler, and Robert Primas. Analyzing the linear keystream biases in AEGIS. *IACR Transactions on Symmetric Cryptology*, pages 348–368, 2019.

[GZ21] Xinxin Gong and Bin Zhang. Resistance of SNOW-V against fast correlation attacks. *IACR Transactions on Symmetric Cryptology*, pages 378–410, 2021.

[HII+21] Jin Hoki, Takanori Isobe, Ryoma Ito, Fukang Liu, and Kosei Sakamoto. Distinguishing and key recovery attacks on the reduced-round SNOW-V and SNOW-Vi. Cryptology ePrint Archive, Report 2021/546, 2021. `https://eprint.iacr.org/2021/546`.

[JLH20] Lin Jiao, Yongqiang Li, and Yonglin Hao. A guess-and-determine attack on SNOW-V stream cipher. *The Computer Journal*, 63(12):1789–1812, 2020.

[MJ05] Alexander Maximov and Thomas Johansson. Fast computation of large distributions and its cryptographic applications. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 313–332. Springer, 2005.

[NW06] Kaisa Nyberg and Johan Wallén. Improved linear distinguishers for SNOW 2.0. In *International Workshop on Fast Software Encryption*, pages 144–162. Springer, 2006.

[SAG20] ETSI SAGE. 256-bit algorithms based on SNOW 3G or SNOW V. 4 November 2020. LS S3-203338. `https://www.3gpp.org/ftp/tsg_sa/WG3_Security/TSGS3_101e/Docs/S3-203338.zip`.

[SSS+19] Danping Shi, Siwei Sun, Yu Sasaki, Chaoyun Li, and Lei Hu. Correlation of quadratic Boolean functions: Cryptanalysis of all versions of full MORUS. In *Annual International Cryptology Conference*, pages 180–209. Springer, 2019.

[WBDC03] Dai Watanabe, Alex Biryukov, and Christophe De Canniere. A distinguishing attack of SNOW 2.0 with linear masking method. In *International Workshop on Selected Areas in Cryptography*, pages 222–233. Springer, 2003.

[YJ20] Jing Yang and Thomas Johansson. An overview of cryptographic primitives for possible use in 5G and beyond. *Science China Information Sciences*, 63(12):1–22, 2020.

[YJM19] Jing Yang, Thomas Johansson, and Alexander Maximov. Vectorized linear approximations for attacks on SNOW 3G. *IACR Transactions on Symmetric Cryptology*, pages 249–271, 2019.

[YJM20] Jing Yang, Thomas Johansson, and Alexander Maximov. Spectral analysis of ZUC-256. *IACR transactions on symmetric cryptology*, pages 266–288, 2020.

Appendices

1 The matrices

```
0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 1     1 1 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 1     0 0 1 0 0 0 0 0 | 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 0     0 0 0 1 0 0 0 0 | 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 | 0 0 0 0 0 0 0 0 0     0 0 0 0 1 0 0 0 | 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 | 0 0 0 0 0 0 0 0 0     1 0 0 0 0 1 0 0 | 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 | 0 0 0 0 0 0 0 0 1     1 0 0 0 0 0 1 0 | 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 | 0 0 0 0 0 0 0 0 1     0 0 0 0 0 0 0 1 | 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 | 0 0 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0 | 1 0 0 0 0 0 0 0
---------------------------------     ---------------------------------
0 0 0 0 0 0 0 1 | 0 0 0 0 0 0 0 0 1     0 0 0 0 0 0 0 0 | 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 | 1 0 0 0 0 0 0 0 0     0 0 0 0 0 0 0 0 | 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 | 0 1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0 | 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 | 0 0 1 0 0 0 0 0 1     0 0 0 0 0 0 0 0 | 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 | 0 0 0 1 0 0 0 0 0     0 0 0 0 0 0 0 0 | 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 | 0 0 0 0 1 0 0 0 0     1 0 0 0 0 0 0 0 | 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 | 0 0 0 0 0 1 0 0 1     1 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 1 1 1     1 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0
```

Listing 5: The 16×16 binary matrices for β (left) and β^{-1} (right).

```
e b d 9| 0 0 0 0| 0 0 0 0| 0 0 0 0      2 0 0 0| 0 3 0 0| 0 0 1 0| 0 0 0 1
0 0 0 0| 0 0 0 0| 0 0 0 0| 9 e b d      1 0 0 0| 0 2 0 0| 0 0 3 0| 0 0 0 1
0 0 0 0| 0 0 0 0| d 9 e b| 0 0 0 0      1 0 0 0| 0 1 0 0| 0 0 2 0| 0 0 0 3
0 0 0 0| b d 9 e| 0 0 0 0| 0 0 0 0      3 0 0 0| 0 1 0 0| 0 0 1 0| 0 0 0 2
-----------------------------------      -----------------------------------
0 0 0 0| e b d 9| 0 0 0 0| 0 0 0 0      0 0 0 1| 2 0 0 0| 0 3 0 0| 0 0 1 0
9 e b d| 0 0 0 0| 0 0 0 0| 0 0 0 0      0 0 0 1| 1 0 0 0| 0 2 0 0| 0 0 3 0
0 0 0 0| 0 0 0 0| 0 0 0 0| d 9 e b      0 0 0 3| 1 0 0 0| 0 1 0 0| 0 0 2 0
0 0 0 0| 0 0 0 0| b d 9 e| 0 0 0 0      0 0 0 2| 3 0 0 0| 0 1 0 0| 0 0 1 0
-----------------------------------      -----------------------------------
0 0 0 0| 0 0 0 0| e b d 9| 0 0 0 0      0 0 1 0| 0 0 0 1| 2 0 0 0| 0 3 0 0
0 0 0 0| 9 e b d| 0 0 0 0| 0 0 0 0      0 0 3 0| 0 0 0 1| 1 0 0 0| 0 2 0 0
d 9 e b| 0 0 0 0| 0 0 0 0| 0 0 0 0      0 0 2 0| 0 0 0 3| 1 0 0 0| 0 1 0 0
0 0 0 0| 0 0 0 0| 0 0 0 0| b d 9 e      0 0 1 0| 0 0 0 2| 3 0 0 0| 0 1 0 0
-----------------------------------      -----------------------------------
0 0 0 0| 0 0 0 0| 0 0 0 0| e b d 9      0 3 0 0| 0 0 1 0| 0 0 0 1| 2 0 0 0
0 0 0 0| 0 0 0 0| 9 e b d| 0 0 0 0      0 2 0 0| 0 0 3 0| 0 0 0 1| 1 0 0 0
0 0 0 0| d 9 e b| 0 0 0 0| 0 0 0 0      0 1 0 0| 0 0 2 0| 0 0 0 3| 1 0 0 0
b d 9 e| 0 0 0 0| 0 0 0 0| 0 0 0 0      0 1 0 0| 0 0 1 0| 0 0 0 2| 3 0 0 0
```

Listing 6: The L^{-1} (left) and L (right) matrices.

2 The operations of l_β and h_β in bytes

$l_\beta(B0)$ can be expressed in bytes as below:

$$
\begin{aligned}
l_\beta(B0)_{0||1} &= \beta B0_{0||1} \oplus B0_{6||7}, & l_\beta(B0)_{2||3} &= \beta B0_{2||3} \oplus B0_{8||9},\\
l_\beta(B0)_{4||5} &= \beta B0_{4||5} \oplus B0_{10||11}, & l_\beta(B0)_{6||7} &= \beta B0_{6||7} \oplus B0_{12||13},\\
l_\beta(B0)_{8||9} &= \beta B0_{8||9} \oplus B0_{14||15}, & l_\beta(B0)_{10||11} &= \beta B0_{10||11},\\
l_\beta(B0)_{12||13} &= \beta B0_{12||13}, & l_\beta(B0)_{14||15} &= \beta B0_{14||15}.
\end{aligned}
$$

$h_\beta(B1)$ can be expressed in bytes as below:

$$
\begin{aligned}
h_\beta(B1)_{0||1} &= \beta^{-1} B1_{0||1}, & h_\beta(B1)_{2||3} &= \beta^{-1} B1_{2||3},\\
h_\beta(B1)_{4||5} &= \beta^{-1} B1_{4||5}, & h_\beta(B1)_{6||7} &= \beta^{-1} B1_{6||7}\\
h_\beta(B1)_{8||9} &= \beta^{-1} B1_{8||9}, & h_\beta(B1)_{10||11} &= \beta^{-1} B1_{10||11} \oplus B1_{0||1}\\
h_\beta(B1)_{12||13} &= \beta^{-1} B1_{12||13} \oplus B1_{2||3}, & h_\beta(B1)_{14||15} &= \beta^{-1} B1_{14||15} \oplus B1_{4||5}.
\end{aligned}
$$

3 Recursion implementation for the 10-steps algorithm

Note that for a random choice of inputs $C, D, R2, R3$, the probability of having at least one solution of $A0$ is $2^{-36.82}$. However, if solutions exist, the average number of solutions will be $2^{36.82}$. Therefore, in the code below we also include the flag `solvable=0/1` as the argument to the method `Dequation::random()` that generates either a fully random input where $A0$ may possibly have a solution, or a random input where $A0$ is guaranteed to have a solution – that is for testing and simulation purposes.

```
struct Dequation
{
    u8 R2[16], R3[16], C[16], D[16]; // input
    u8 u[16], v[16]; // internal
```

```
    u8 A0[16]; // result

    void computeD(u8 * Dr)
    {
        u8 T1[16];
        for (int i = 0; i < 4; i++)
        ((u32*)T1)[i] = ((u32*)R2)[i] + (((u32*)R3)[i] ^ ((u32*)A0)[i
]);
        for (int i = 0; i < 16; i++)
        Dr[i] = T1[((i >> 2) | (i << 2)) & 0xf];
        for (int i = 0; i < 4; i++)
        ((u32*)Dr)[i] += ((u32*)A0)[i] ^ ((u32*)C)[i];
    }

    void random(int solvable=0)
    {   memset(this, 0xff, sizeof(*this));
        for (int i = 0; i < 16; i++)
        {   R2[i] = rand();
            R3[i] = rand();
            C[i]  = rand();
            A0[i] = rand();
            D[i]  = rand();
        }
        if(solvable) computeD(D);
    }


    int expr(int i, int j, int Xi, int Xj)
    {   return D[i] ^ ((R2[j] + (R3[j] ^ Xj) + u[j]) + (Xi ^ C[i]) + v
[i]);
    }

    void solve1(int step, int i, int X=0, int bit=-1)
    {
        if (bit >= 0 && (expr(i, i, X, X) & (1 << bit))) return;
        if (bit == 7)
        {
            A0[i] = X;
            next_carries(i, i);
            solve(step + 1);
            return;
        }
        solve1(step, i, X, ++bit);
        solve1(step, i, X ^ (1 << bit), bit);
    }

    void solve2(int step, int i, int j, int Xi = 0, int Xj=0, int bit
=-1)
    {
        if (bit>=0 && ((expr(i, j, Xi, Xj)|expr(j, i, Xj, Xi)) & (1<<
bit)))
        return;

        if (bit == 7)
        {
            A0[i] = Xi;
            A0[j] = Xj;
            next_carries(i, j);
```

```
                next_carries(j, i);
                solve(step + 1);
                return;
            }
            int t = (1 << ++bit);
            solve2(step, i, j, Xi, Xj, bit);
            solve2(step, i, j, Xi ^ t, Xj, bit);
            solve2(step, i, j, Xi, Xj ^ t, bit);
            solve2(step, i, j, Xi ^ t, Xj ^ t, bit);
        }

        void next_carries(int i, int j)
        {
            int nu = ((int)R2[j] + (int)(R3[j] ^ A0[j]) + (int)u[j]);
            int nv = (nu & 0xff) + (int)(A0[i] ^ C[i]) + (int)v[i];
            ++i, ++j;
            if (j & 3) u[j] = nu >> 8;
            if (i & 3) v[i] = nv >> 8;
        }

        void solve(int step = 0)
        {
            static int S[10] = { 0, 1, 2, 5, 3, 6, 10, 7, 11, 15 };
            if (step == 0)
            u[0] = u[4] = u[8] = u[12] = v[0] = v[4] = v[8] = v[12] = 0;

            if (step == 10)
            {
                // A solution for A0 is found! do something with it...
                u8 ver[16]; // we just verify that the solution is correct
                computeD(ver);
                if (memcmp(D, ver, 16))
                printf("ERROR: Verification of the derived A0 failed!\n");
                return;
            }

            int i = S[step], j = ((i >> 2) | (i << 2)) & 0xf; // j = sigma
    (i)
            if (i == j) solve1(step, i);
            else solve2(step, i, j);
        }

    };
```

Listing 7: A possible recursion organisation for 10-steps.

4 The distribution table of solutions for *Type-2* equations

Consider n-bit variables $A_{1,2}, B_{1,2}, C_{1,2}, D_{1,2}, X_{1,2}$ and two n-bit equations:

$$A_1 = (B_1 \boxplus_n (C_1 \oplus X_1)) \boxplus_n (X_2 \oplus D_1),$$
$$A_2 = (B_2 \boxplus_n (C_2 \oplus X_2)) \boxplus_n (X_1 \oplus D_2).$$

Table 4 contains the probabilities of the pair (X_1, X_2) having k solutions for a random tuple $(A_{1,2}, B_{1,2}, C_{1,2}, D_{1,2})$, which are derived through $p = x/f$, where x's are the integers in the table and f is the corresponding normalisation factor. For the GnD attack against SNOW-V we are interested in the distribution where $n = 8$.

#Solutions factor $f \rightarrow$	n=1 2^2	n=2 2^3	n=3 2^7	n=4 2^{10}	n=5 2^{14}	n=6 2^{18}	n=7 2^{22}	n=8 2^{26}
0	1	5	91	793	13484	225652	3734648	61316512
2	1	2	16	64	512	4096	32768	262144
4		1	18	119	1377	14759	150417	1478903
8			3	43	803	12265	166035	2071185
12				1	29	529	7761	100077
16				4	162	3978	76314	1256786
20					1	33	661	10405
24					5	205	5001	94273
28					1	33	661	10405
32					10	536	16552	385832
36						3	117	2691
40						5	225	5901
44						1	37	809
48						18	978	30258
52						1	37	809
56						5	225	5901
60						1	41	985
64						24	1632	61440
68							1	41
72							19	981
76							1	41
80							18	1050
84							5	217
88							5	245
92							1	41
96							56	3864
100							1	43
104							5	245
108							1	49
112							18	1050
116							1	41
120							5	273
124							1	41
128							56	4688
132								5
136								5
140								5
144								82
148								1
152								5

156								5
160								56
164								1
168								33
172								1
176								18
180								5
184								5
188								1
192								160
196								3
200								5
204								1
208								18
212								1
216								5
220								1
224								56
228								1
232								5
236								1
240								18
244								1
248								5
252								1
256								128

Table 4: Distribution table for *Type-2* equations.

5 The probability of valid solutions in Equation 12

In this section, we compute the probability of valid solutions in Equation 12. We recall that the equations are:

$$z^{(t)} \oplus R2 = R1 \boxplus_{32} B1,$$
$$Y \oplus h_\beta(B0) = (\sigma(R1) \boxminus_{32} \mathrm{AES}_R^{-1}(R3)) \oplus B1,$$

where $R1$ and $B1$ are the two unknowns. First we note that $z^{(t)}$ and $R2$ are independent from the rest variables, looping over the xor-sum of $z^{(t)}$ and $R2$ is equivalent to looping over one random variable. Thus, we use a new variable U to denote $z^{(t)} \oplus R2$. Similarly, Y and $B0$ are independent from the rest variables, and we can regard $Y \oplus h_\beta(B0)$ as a new variable V. Here we should be careful about $h_\beta(B0)$: since h_β is a full-rank matrix, when $B0$ takes all the values, $h_\beta(B0)$ will also take all the values. $\mathrm{AES}_R^{-1}(R3)$ can also be regarded as a random variable W as is is a bijective mapping.

Thus we have a simplified system of equations:

$$\begin{aligned} U &= R1 \boxplus_{32} B1, \\ V &= (\sigma(R1) \boxminus_{32} W) \oplus B1. \end{aligned} \tag{15}$$

According to Equation (15), we have $B1 = (\sigma(R1) \boxminus_{32} W) \oplus V$, and further get:

$$U = R1 \boxplus_{32} ((\sigma(R1) \boxminus_{32} W) \oplus V). \tag{16}$$

The distributions of number of solutions of Equation (15) and Equation (16) are the same, since $B1$ is uniquely determined given V, W and $R1$. We have experimentally verified this observation over smaller dimensions. Thus we can use Equation (16) to get the distribution of number of solutions of $R1$ and $B1$.

Similarly, we would have two types of equations, the first type with the form below,

$$U_0 = R1_0 \boxplus_8 ((R1_0 \boxminus_8 W_0 \boxminus_8 v_0) \oplus V_0) \boxplus_8 u_0,$$

and the second type with the form:

$$\begin{aligned} U_1 &= R1_1 \boxplus_8 ((R1_4 \boxminus_8 W_1 \boxminus_8 v_1) \oplus V_1) \boxplus_8 u_1 \\ U_4 &= R1_4 \boxplus_8 ((R1_1 \boxminus_8 W_4 \boxminus_8 v_4) \oplus V_4) \boxplus_8 u_4. \end{aligned}$$

We have experimentally computed the distributions of solutions for these two types of equations, and the probabilities of having solutions are $2^{-3.91}$ and $2^{-3.53}$, respectively. The average number of solutions is exactly one for each combination of other variables. The results are just the same to the ones of the D-equations for $A0$ in the first GnD attack.

6 The flowcharts of the guess-and-determine attacks

Figure 3 presents simple illustrations of the proposed GnD attacks.

7 The probability p_z

In this section, we derive the probability p_z of $B0^{(t-1)}$ having solutions in the equation of $z^{(t-2)}$. Recall that the equation of $z^{(t-2)}$ is expressed as below:

$$z^{(t-2)} = (B0^{(t-1)} \boxplus_{32} X) \oplus \mathrm{AES}_R^{-1}(l_\beta(B0^{(t-1)}) \oplus Y),$$

where $\mathrm{AES}_R^{-1}(X)$ can be expressed as $S^{-1}(L^{-1} \cdot X)$, and l_β operation is defined in Equation (5). We temporarily replace $\boxplus_{32}$ with $\boxplus_8$. For simplicity, we denote $Y' = L^{-1}Y$ and ignore the time notations, then we can simplify the equation as:

$$z = (B0 \boxplus_8 X) \oplus S^{-1}(L^{-1} l_\beta(B0) \oplus Y').$$

Now our task is to compute the probability of $B0$ having solutions given z, X, Y'. We use an enumeration algorithm to achieve this by considering four groups of equations

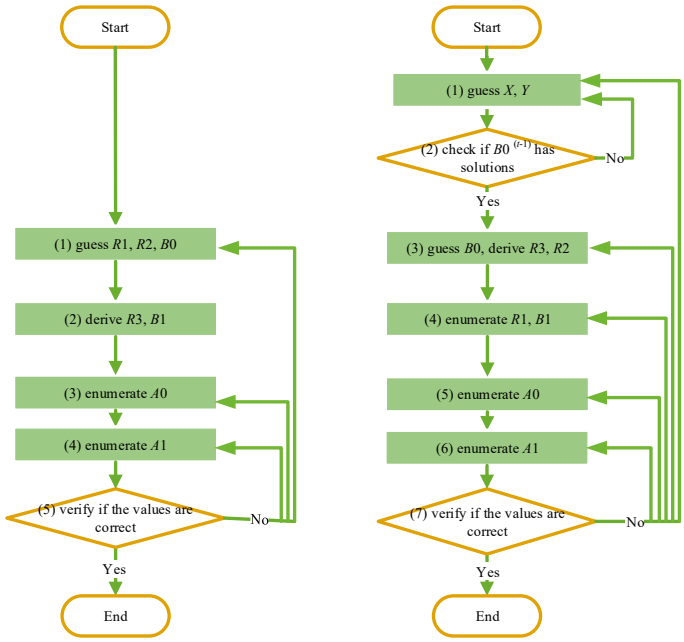

Figure 3: Illustration of the GnD attacks (left: first; right: second).

recursively, which are given below.

Step 1. Before giving the first group of equations, we first use z_{12} as an example to illustrate how to derive each byte of z in details. z_{12} can be expressed as:

$$z_{12} = (B0_{12} \boxplus_8 X_{12}) \\ \oplus S^{-1}((e,b,d,9) \cdot (\beta(B0_{12||13})_0, \beta(B0_{12||13})_1, \beta(B0_{14||15})_0, \beta(B0_{14||15})_1) \oplus Y'_{12}),$$

where $\beta(B0_{i||i+1})_j, i \in \{12,14\}, j \in \{0,1\}$ is the j-th byte of $\beta(B0_i||B0_{i+1})$.

For simplicity of expressions, we use $[B0_{i,i+1,i+2,i+3}]$ to denote the vector of the four bytes $(B0_i, B0_{i+1}, B0_{i+2}, B0_{i+3})$ and $[\psi B0_{i,i+1,i+2,i+3}]$ to denote the vector of the four bytes after multiplying with β, i.e.,

$$[\psi B0_{i,i+1,i+2,i+3}] = (\beta(B0_{i||i+1})_0, \beta(B0_{i||i+1})_1, \beta(B0_{i+2||i+3})_0, \beta(B0_{i+2||i+3})_1),$$

for $i = 0, 4, 8, 12$.

Now consider the first group of equations:

$$z_{12} = (B0_{12} \boxplus_8 X_{12}) \oplus S^{-1}((e,b,d,9) \cdot [\psi B0_{12,13,14,15}] \oplus Y'_{12}),$$
$$z_{11} = (B0_{11} \boxplus_8 X_{11}) \oplus S^{-1}((b,d,9,e) \cdot [\psi B0_{12,13,14,15}] \oplus Y'_{11}),$$
$$z_{6} = (B0_{6} \boxplus_8 X_{6}) \oplus S^{-1}((d,9,e,b) \cdot [\psi B0_{12,13,14,15}] \oplus Y'_{6}),$$
$$z_{1} = (B0_{1} \boxplus_8 X_{1}) \oplus S^{-1}((9,e,b,d) \cdot [\psi B0_{12,13,14,15}] \oplus Y'_{1}).$$

Given the bytes of z, X, Y', we can freely choose the values of $B0_{13,14,15}$, then in z_{12} only $B0_{12}$ remains unknown. Once $B0_{12}$ is further determined, $B0_{1,6,11}$ will be derived uniquely from $z_{1,6,11}$, thus there is always a solution for these bytes if $B0_{12}$ in z_{12} has solutions. So the main task now is to compute the probability of $B0_{12}$ having solutions in z_{12}. According to the expression of β matrix given in Appendix 1, z_{12} can be further derived as:

$$z_{12} = (B0_{12} \boxplus_8 X_{12}) \oplus S^{-1}(e \cdot (B0_{12} \ll 1) \oplus b \cdot (B0_{12} \gg 7) \oplus Y''_{12}),$$

where Y''_{12} is a new variable, which is the linear combination of $Y'_{12}, B0_{13}, B0_{14}, B0_{15}$. We can compute the probability of $B0_{12}$ having at least one solution, denoted $p_z(B0_{12})$, which is:

$$p_z(B0_{12}) \approx 0.363230705.$$

Thus, in Step 1 we can loop over $B0_{13,14,15}$, solve $B0_{12}$ with valid solutions of probability $p_z(B0_{12})$, and further derive $B0_{1,6,11}$ correspondingly.

Step 2. Consider the second group of equations:

$$\begin{aligned}
z_{13} &= (B0_{13} \boxplus_8 X_{13}) \oplus S^{-1}((9, e, b, d) \cdot ([\psi B0_{8,9,10,11}] \oplus (B0_{14}, B0_{15}, 0, 0)) \oplus Y'_{13}),\\
z_8 &= (B0_8 \boxplus_8 X_8) \oplus S^{-1}((e, b, d, 9) \cdot ([\psi B0_{8,9,10,11}] \oplus (B0_{14}, B0_{15}, 0, 0)) \oplus Y'_8),\\
z_7 &= (B0_7 \boxplus_8 X_7) \oplus S^{-1}((b, d, 9, e) \cdot ([\psi B0_{8,9,10,11}] \oplus (B0_{14}, B0_{15}, 0, 0)) \oplus Y'_7),\\
z_2 &= (B0_2 \boxplus_8 X_2) \oplus S^{-1}((d, 9, e, b) \cdot ([\psi B0_{8,9,10,11}] \oplus (B0_{14}, B0_{15}, 0, 0)) \oplus Y'_2).
\end{aligned}$$

Here we can only freely choose $B0_{9,10}$, as the values of $B0_{11,14,15}$ have already been considered in Step 1. We add the linear combinations of these known variables to the Y'-terms, resulting in new Y'' variables, and use a new variable X'_{13} to denote $B0_{13} \boxplus X_{13}$, which is also known. Thus we need to find solutions of $B0_8$ that satisfies the two equations below:

$$\begin{aligned}
z_{13} &= X'_{13} \oplus S^{-1}(9 \cdot (B0_8 \ll 1) \oplus e \cdot (B0_8 \gg 7) \oplus Y''_{13}),\\
z_8 &= (B0_8 \boxplus_8 X_8) \oplus S^{-1}(e \cdot (B0_8 \ll 1) \oplus b \cdot (B0_8 \gg 7) \oplus Y''_8).
\end{aligned}$$

We have computed that the probability of valid solutions for $B0_8$ is:

$$p_z(B0_8) \approx 0.363230705 \cdot 2^{-8}.$$

This can be understood in another way: the probability of $B0_8$ having solutions in z_8 is 0.363230705, and the solutions will satisfy the equation of z_{13} with probability around 2^{-8}. After we have solved $B0_8$, we can further derive $B0_7$ and $B0_2$ uniquely. Thus in Step 2 we can loop over $B0_{9,10}$, solve $B0_8$ with valid solutions of probability $p_z(B0_8)$, and further derive $B0_{2,7}$.

Step 3. We further consider the next group of equations:

$$\begin{aligned}
z_{14} &= (B0_{14} \boxplus_8 X_{14}) \oplus S^{-1}((d, 9, e, b) \cdot ([\psi B0_{4,5,6,7}] \oplus [B0_{10,11,12,13}]) \oplus Y'_{14}),\\
z_9 &= (B0_9 \boxplus_8 X_9) \oplus S^{-1}((9, e, b, d) \cdot ([\psi B0_{4,5,6,7}] \oplus [B0_{10,11,12,13}]) \oplus Y'_9),\\
z_4 &= (B0_4 \boxplus_8 X_4) \oplus S^{-1}((e, b, d, 9) \cdot ([\psi B0_{4,5,6,7}] \oplus [B0_{10,11,12,13}]) \oplus Y'_4),\\
z_3 &= (B0_3 \boxplus_8 X_3) \oplus S^{-1}((b, d, 9, e) \cdot ([\psi B0_{4,5,6,7}] \oplus [B0_{10,11,12,13}]) \oplus Y'_3).
\end{aligned}$$

The known bytes $B0_{6,7,10,11,12,13}$ are added to the Y'-terms, while the bytes $B0_{9,14}$ are added to the X-terms. We can freely loop over $B0_5$ and solve the following equation in $B0_4$:

$$z_4 = (B0_4 \boxplus_8 X_4) \oplus S^{-1}(e \cdot (B0_4 \ll 1) \oplus b \cdot (B0_4 \gg 7) \oplus Y_4'').$$

The probability of valid $B0_4$ solutions in z_4 is again computed as 0.363230705, and such solutions will satisfy z_{14}, z_9 with probability around 2^{-16}. Thus the total probability of valid $B0_4$ solutions, denoted $p_z(B0_4)$, is computed as:

$$p_z(B0_4) \approx 0.363230705 \cdot 2^{-16}.$$

After $B0_4$ having been solved, $B0_3$ can be uniquely determined according to z_3. Thus in Step 3 we can loop over $B0_5$, solve $B0_4$ with valid solutions of probability $p_z(B0_4)$, and further derive $B0_3$.

Step 4. The last group of equations contain the remaining four byte expressions $z_{0,5,10,15}$ in only one unknown variable $B0_0$, while other variables are already known:

$$\begin{aligned}
z_0 &= (B0_0 \boxplus_8 X_0) \oplus S^{-1}((e,b,d,9) \cdot ([\psi B0_{0,1,2,3}] \oplus [B0_{6,7,8,9}]) \oplus Y_0'),\\
z_5 &= (B0_5 \boxplus_8 X_5) \oplus S^{-1}((9,e,b,d) \cdot ([\psi B0_{0,1,2,3}] \oplus [B0_{6,7,8,9}]) \oplus Y_5'),\\
z_{10} &= (B0_{10} \boxplus_8 X_{10}) \oplus S^{-1}((d,9,e,b) \cdot ([\psi B0_{0,1,2,3}] \oplus [B0_{6,7,8,9}]) \oplus Y_{10}'),\\
z_{15} &= (B0_{15} \boxplus_8 X_{15}) \oplus S^{-1}((b,d,9,e) \cdot ([\psi B0_{0,1,2,3}] \oplus [B0_{6,7,8,9}]) \oplus Y_{15}').
\end{aligned}$$

Similarly, $B0_0$ will have valid solutions with probability 0.363230705 in z_0, and these solutions will satisfy z_5, z_{10}, z_{15} with probability 2^{-24}. Thus the probability of valid $B0_0$ solutions, denoted $p_z(B0_0)$, is:

$$p_z(B0_0) \approx 0.363230705 \cdot 2^{-24}.$$

Summary. We can freely choose six bytes of $B0$, i.e., $B0_{5,9,10,13,14,15}$, of total size 2^{48}, which will result into valid solutions for bytes $B0_{0,4,8,12}$ with probability $p_z(B0_0) \cdot p_z(B0_4) \cdot p_z(B0_8) \cdot p_z(B_{12})$. Other bytes will be further uniquely determined. Thus the total probability p_z is computed as:

$$p_z = 2^{48} \cdot p_z(B0_0) \cdot p_z(B0_4) \cdot p_z(B0_8) \cdot p_z(B_{12}) \approx 2^{-5.84}.$$

We cannot really compute an exact success probability for 32-bit adders $\boxplus_{32}$, but one can expect that it would be very similar to the derived probability, as only several carrier bits need to be further considered.

Paper V

SNOW-Vi: An Extreme Performance Variant of SNOW-V for Lower Grade CPUs

Abstract

SNOW 3G is a stream cipher used as one of the standard algorithms for data confidentiality and integrity protection over the air interface in the 3G and 4G mobile communication systems. SNOW-V is a recent new version that was proposed as a candidate for inclusion in the 5G standard. In this paper, we propose a faster variant of SNOW-V, called SNOW-Vi, that can reach the targeted speeds for 5G in a software implementation on a larger variety of CPU architectures. SNOW-Vi differs in the way how the LFSR is updated and also introduces a new location of the tap $T2$ for stronger security, while everything else is kept the same as in SNOW-V. The throughput in a software environment is increased by around 50% in average, up to 92 Gbps. This makes the applicability of the cipher much wider and more use cases are covered. The security analyses previously done for SNOW-V are not affected in most aspects, and SNOW-Vi provides the same 256-bit security level as SNOW-V.

Keywords: SNOW, stream cipher, 5G mobile system security.

1 Introduction

Symmetric ciphers play an important role in securing the transmitted data in various generations of 3GPP mobile telephony systems. The stream cipher SNOW 3G is one of the core algorithms for integrity and confidentiality protection in both UMTS and LTE, together with AES and ZUC. In the current generation system, called 5G, we see fundamental changes in the system architecture and new demands in security, which pose several challenges to existing cryptographic algorithms [3GP20].

Firstly, the 3GPP standardisation organisation is aiming at increasing the security level to 256-bit key length [3GP19]. Although there exist academic attacks [BKR11] (attacks faster than exhaustive key search, but still beyond the practical capability or regulations), this change is relatively straightforward for AES, as the 256-bit variant has been known and used for a long time. For ZUC and SNOW 3G, the situation is

Patrik Ekdahl, Thomas Johansson, Alexander Maximov, and Jing Yang. SNOW-Vi: an extreme performance variant of SNOW-V for lower grade CPUs. In Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks, pages 261-272, 2021.

somewhat different: neither of the two ciphers was originally specified for 256-bit key length. There are simple ways to increase the key length of both ZUC and SNOW 3G (in fact a 256-bit version of ZUC was announced in 2018), but they also become susceptible to some academic attacks [YJM19, YJM20].

Secondly, the changes in the radio and core network in the 5G system will also introduce some challenges for the cryptographic algorithms. It is expected that many network nodes in 5G will become virtualised and thus the ability to use special hardware (e.g., IP cores) for cryptographic primitives is limited. This might not be a problem for AES, as many processors from Intel, ARM and AMD have included special instructions to accelerate AES, and it will be easy to reach encryption speeds of more than 20 Gbps, which is the targeted speed of the downlink in 5G. Thus, one can expect that AES could be kept in 5G. However, for SNOW 3G and ZUC, such high rates cannot be achieved in a pure software environment.

In response to these challenges, a new member of the SNOW family of stream ciphers, called SNOW-V [EJMY19], was developed, with the design goal to be fast in virtualised environments and provide 256-bit security. It is proposed for consideration as a candidate for inclusion in the 5G standard. The algorithm takes advantage of the AES instructions in the CPU as well as vectorised SIMD (Single Instruction Multiple Data) instructions, such as the AVX2 (Advanced Vector Extensions 2) set of instructions, and achieves rates up to 58 Gbps for encryption.

However, SNOW-V may not perform as good on CPUs with limited vector register widths or instruction sets. For example, there might be a transitional network deployment scenario where the 5G encryption layer (PDCP) is not yet virtualised, but processed in software on the base station, where typically there is a mixture of dedicated hardware and general CPU resources. These CPUs are normally not server-grade but something more suitable for embedding in a base station. By running the encryption layer in software we are then forced to perform fast air encryption on CPUs with limited vector register widths and simpler SIMD instruction sets (but enough capability to serve in a base station) as well. This possible use case was only partially covered by the SNOW-V design goals, and in this work we present a way to speed up SNOW-V and thus to extend its usage.

We propose SNOW-Vi[1] – an extreme performance variant of SNOW-V, that reaches much higher speeds on a wider variety of platforms. The basis for SNOW-Vi are not only cloud hosting CPUs with SIMD registers of 256 bits or wider, but also platforms with only 128-bit registers. With this new variant we can tackle the speed requirements also in these lower-grade CPUs. The encryption speed of SNOW-Vi is increased by around 50% than that of SNOW-V, in average, while the security stands on the same level. The minimum requirement for the CPU is that it supports the AES round function as an instruction, and at least 128-bit SIMD registers.

This paper is organised as follows. Firstly, we briefly present the design of SNOW-V in Section 2 and in Section 3 we show the modifications and design rationale to form SNOW-Vi. Secondly, in Section 4 we evaluate the security of SNOW-Vi by revisiting all known analyses for SNOW-V and applying them to this new design, making sure that it still fulfils the security goals. Finally, in Section 5 we perform an extensive software evaluation under different platforms. We end the paper with a short conclusion

[1] "Vi" stands for "Virtualisation, improved".

in Section 6.

2 The SNOW-V stream cipher

The algorithm SNOW-V follows the design principles of the SNOW-family. It consists of two parts – the LFSR (Linear Feedback Shift Register) and FSM (Finite State Machine), but both are redesigned in order to adapt to the higher performance demands in 5G. The overall schematic of the algorithm is depicted in Figure 1.

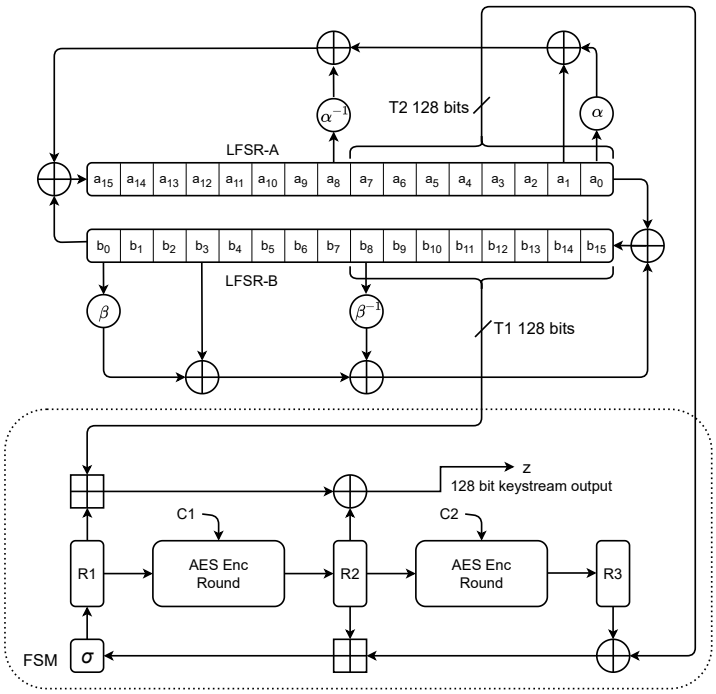

Figure 1: Overall schematic of SNOW-V [EJMY19].

The LFSR is a new circular construction consisting of two 256-bit registers, namely LFSR-A and LFSR-B. Each sub-LFSR consists of 16 16-bit cells, where each cell holds an element of a finite field $GF(2^{16})$. These elements in LFSR-A and LFSR-B are respectively generated according to the generating polynomials defined below:

$$g^A(x) = x^{16} + x^{15} + x^{12} + x^{11} + x^8 + x^3 + x^2 + x + 1 \in \mathbb{F}_2[x],$$
$$g^B(x) = x^{16} + x^{15} + x^{14} + x^{11} + x^8 + x^6 + x^5 + x + 1 \in \mathbb{F}_2[x].$$

Denote the states of LFSR-A and LFSR-B at time clock t as $(a_0^{(t)}, \ldots, a_{15}^{(t)})$ and $(b_0^{(t)}, \ldots, b_{15}^{(t)})$, respectively. Every time when clocking, a value in a cell is shifted to the next cell with a smaller index and $a_0^{(t)}, b_0^{(t)}$ exit the LFSRs. The new values in cells

a_{15}, b_{15} are derived as follows:

$$a^{(t+16)} = b^{(t)} + \alpha a^{(t)} + a^{(t+1)} + \alpha^{-1} a^{(t+8)} \mod g^A(\alpha),$$
$$b^{(t+16)} = a^{(t)} + \beta b^{(t)} + b^{(t+3)} + \beta^{-1} b^{(t+8)} \mod g^B(\beta),$$

where α, β are roots of $g^A(x), g^B(x)$, respectively, and α^{-1}, β^{-1} are the corresponding inverses. The multiplications are operated over the corresponding fields.

Every time when updating the LFSR part, the LFSRs are clocked eight times, such that the two 128-bit values of the taps $T1$ and $T2$, which are formed by considering $(b_{15}, \cdots, b_8)$, and $(a_7, \cdots, a_0)$ as two 128-bit words, are "fresh" to update the FSM and generate a keystream word.

The FSM part is 128-bit oriented and consists of three 128-bit registers $R1, R2$, and $R3$. It takes $T1, T2$ as inputs and produces a 128-bit keystream word as below:

$$z^{(t)} = (R1^{(t)} \boxplus_{32} T1^{(t)}) \oplus R2^{(t)}.$$

The three registers in FSM are then updated as follows:

$$R1^{(t+1)} = \sigma(R2^{(t)} \boxplus_{32} (R3^{(t)} \oplus T2^{(t)})),$$
$$R2^{(t+1)} = AES_R(R1^{(t)}, C1),$$
$$R3^{(t+1)} = AES_R(R2^{(t)}, C2),$$

where $AES_R()$ is one single AES round with the round key being set to zero, i.e., $C1 = C2 = 0$; $\boxplus_{32}$ is four parallel 32-bit arithmetical additions; and σ is a byte-wise permutation – the transposition of the mapped AES's 4×4-byte matrix state, i.e., $\sigma = [0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15]$.

We skip other details, e.g. the initialisation procedure, AEAD mode of operation, and refer to the original paper [EJMY19] for the complete description.

The SNOW-V design has received internal and external evaluations [EJMY19, CDM20, GZ21, JLH20], which show that there are no identified weaknesses in the design resulting in attacks faster than exhaustive key search.

3 The design of SNOW-Vi

The design of SNOW-Vi, in the parts of keystream generation and initialisation procedure, is exactly the same as in SNOW-V, with the only differences in the LFSR update function and the tap position of $T2$, which is now moved to the higher half of LFSR-A – these changes dramatically improve the speed in software implementations and strengthen the security of the cipher. The new LFSR is depicted in Figure 2 and the new updates are as follows:

$$a^{(t+16)} = b^{(t)} + \alpha a^{(t)} + a^{(t+7)} \mod g^A(\alpha),$$
$$b^{(t+16)} = a^{(t)} + \beta b^{(t)} + b^{(t+8)} \mod g^B(\beta),$$

where α and β are respectively the roots of two new fields' generating polynomials, which are defined as follows:

$$g^A(x) = x^{16} + x^{14} + x^{11} + x^9 + x^6 + x^5 + x^3 + x^2 + 1 \in \mathbb{F}_2[x],$$
$$g^B(x) = x^{16} + x^{15} + x^{14} + x^{11} + x^{10} + x^7 + x^2 + x + 1 \in \mathbb{F}_2[x].$$

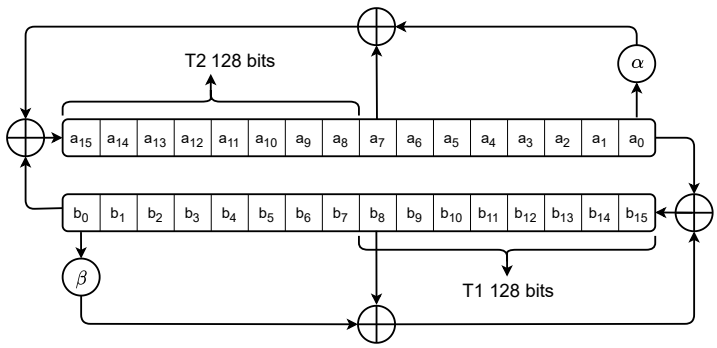

Figure 2: LFSR construction in SNOW-Vi.

3.1 Design rationale

In this section we discuss the design rationale for SNOW-Vi in brief, and some of the arguments are also valid for SNOW-V.

128-bit based design. For modern ciphers with intent to run fast in software environments, implementation aspects are highly important in order to reach a high throughput. SNOW 3G is a 32-bit oriented design that produces a 32-bit keystream word per clock. In order to speed up SNOW 3G, one could imagine producing four such 32-bit keystream words in parallel, using a 128-bit register – wide registers are supported by many CPUs. This means that the FSM unit should also hold four parallel 32-bit words so that it can produce 128 bits in its output. The LFSR can keep its size but should be redesigned in order to update at least 128 bits at a time. This effectively leads to the 128-bit based designs of original SNOW-V and current SNOW-Vi variant, in which the FSM holds three 128-bit words, LFSR can be split into four 128-bit words, and the generated keystream word is 128-bit.

Cycle length. In the SNOW family of stream ciphers, LFSR is a linear block of size 512 bits that serves as the source of pseudo-randomness. One important security property is that the LFSR should not have short cycles and, ideally, has the maximum cycle length of $2^{512} - 1$, excluding the zero state. The proposed LFSR has such a maximum cycle length which can be verified by the same methods as in [EJMY19, CDM20]. The characteristic polynomial is primitive and has 209 terms, see Appendix 1 for details.

Note that the probability of at least one LFSR having a zero state after initialisation, considering all possible (Key, IV) pairs, is negligibly low (i.e., 2^{-128}) for both SNOW-V and SNOW-Vi.

Circular LFSR. It is well-known that in order to prevent trivial linear attacks with multiple short keystreams, the number of taps t to be used for the LFSR update function should be at least three, and preferably even more (around 5-6) depending on how well one can approximate the FSM. However, all the tap values must be extracted and then processed in the LFSR update function, which means the code and time complexity grows linearly with the number of taps.

In a circular LFSR construction we have two sub-LFSRs with t_a and t_b taps involved, respectively. The total number of taps is $t_a + t_b$, but there is a multiplicative effect for the number of taps for an equivalent LFSR that can be used in a linear attack. The equivalent LFSR can have up to $(1 + t_a) \cdot (1 + t_b)$ taps. Thus we can reach the basic security goals by having a smaller number of taps, and the time complexity to clock the LFSR is therefore smaller. I.e., if a classical LFSR has t taps, a circular-LFSR may have as low as $2 \times (\sqrt{t} - 1)$ taps to reach the same security goals, thus reducing the implementation and time complexity for the update function. For example, in SNOW-V the total number of taps is $3 + 3$, but an equivalent LFSR has 11 taps (see Section 4.1 for more details), so that instead of extracting and performing operations on 11 taps in a classical LFSR, we only need to operate with six taps in the circular construction.

Size and type of the base fields. In SNOW 3G, the LFSR update function is built over an extended 32-bit field $GF((2^8)^4)$, where the ground element is an 8-bit subfield $GF(2^8)$. This particular choice made it possible to implement the multiplication by $\alpha \in GF((2^8)^4)$ with a lookup table $2^8 \rightarrow 2^{32}$, two shifts and one XOR. Although the LFSR in SNOW 3G can be parallelised to produce 128 bits of the tap values, it is still hard to implement four multiplications in that 32-bit base field by using only 128-bit registers, and without lookup tables. Moreover, it is not desirable to use lookup tables in modern ciphers since it may become a vulnerability to cache-based side-channel attacks.

Considering the above, the extension field $GF((2^8)^4)$ was abandoned and, instead, a binary field $GF(2^n)$ for some smaller n is introduced in the design, which is more suitable for parallelisation, i.e., we can perform a parallel multiplication of $128/n$ n-bit elements by the primitive element $\alpha \in GF(2^n)$ using 128-bit registers and only four SIMD instructions. In order to shuffle as many bits as possible, n should be rather small, and, ideally, the field size should be $n = 8$ bits – in this case there will be more "decisive" bits to be involved in the update process. However, there is currently no widely spread SIMD instruction, in lower grade CPUs, that can perform an arithmetical shift to the right of 16 8-bit signed values, needed for the implementation of the multiplication by α in $GF(2^8)$. Instead, there is an instruction `_mm_srai_epi16()` for eight 16-bit signed values that we can use for implementation. Therefore, the ground fields were selected to be $GF(2^{16})$.

LFSR update rate. Since the two sub-LFSRs are also the source of 128-bit taps $T1$ and $T2$, to compute the keystream and update the FSM, we would like to make sure that these tap values are "fresh" in each clock. Therefore, we need to update 128 bits in both sub-LFSRs. For 16-bit base fields this implies 8 clocks for a single LFSR update step.

Base fields. Let us take an extreme situation – if the base fields would flip only 1 bit of data during the reduction, an attacker may, perhaps, use the fact that the bits of the

LFSR elements are changed rarely. On the other hand, if the reduction would flip 15 bits out of 16, it is a similar situation since the attacker knows that almost all bits are flipped in most of the reduction times.

Thus, the field generating polynomials proposed for SNOW-Vi both have weights eight (excluding x^{16}), so that if a reduction happens, exactly half of the 16 bits will be flipped. Additionally, the base fields are selected such that they have exactly four coinciding bits, four bits where flips are not happening, and two 4-bit sets where only one of the two fields flips the bits.

Taps positions for the LFSR update function. With a reduced number of taps in SNOW-Vi we should carefully select the update tap positions to meet both an efficient implementation and a good mixing effect. If the content of the LFSRs are denoted as four 128-bit registers $(A0, A1, B0, B1)$, where $A0, A1(B0, B1)$ are the low and high 128 bits of LFSR-A (LFSR-B), respectively, and we want to update 128 bits of each sub-LFSR in a single step, there should be no taps taken from either $(a_9, \ldots, a_{15})$ or $(b_9, \ldots, b_{15})$.

The first clear choice for tap positions is from $A0$ and $B0$ since these 128-bit values are already in the state's 128-bit registers, and we get them for free. Fields multiplications by α and β should be placed "symmetrically" since then we can perform multiplications in both fields in parallel, in case 256-bit registers are available: we simply represent the LFSR state in a "butterfly" manner as $lo = (A0, B0)$ and $hi = (A1, B1)$, where we then multiply lo by (α, β) with SIMD instructions in parallel, thus double the speed. Besides, in such a 256-bit oriented data structure we only have to compute the new value for the hi part, while the new value for lo part is just a single register copy $lo^{(t+1)} = hi^{(t)}$.

The middle state values $(A1, B1)$ would be good "free" taps (a_8, b_8) for the update function, but then it becomes impossible to get a full-cycle LFSR. However, we can take one middle tap as $B1$, and the second middle tap must be byte-unaligned, one of $\{a_1, \ldots, a_7\}$.

When analysing the mixing effect, one can compare the tap positions a_1 vs. a_7, where the latter tap would involve more bytes in the update of the LFSR-A than the former tap. Therefore, we conclude that the middle pair of tap positions (a_7, b_8) seems the best possible choice for a good overall mixing effect.

The final choice of the base fields. So far we have put a lot of constrains and desired properties on the tap positions, field size, placement of multiplications, full cycle length, etc. We coded a search algorithm that first creates a list of 16-bit base field candidates (primitive polynomials of degree 16 and weight 8+1), then tries to select a pair of the base fields satisfying the other criteria (that the intersection of the base fields is also statistically balanced), and finally verifies that the LFSR has a full cycle. In the end, we still received a number of options to choose from. Since we were running out of more criteria, we made our final selection choice intuitively, based on how well the bits of the base fields are spread across the 16 bits.

3.2 The new tap position of $T2$

While we propose a simplified update function in the LFSR for better performance, we also have to ensure the security of the new proposal. By moving the tap $T2$ to the

higher half of LFSR-A, we believe that the security is strengthened. Below we give more details on motivations for this particular design choice.

From linear analysis perspectives. Let us assume that the content of the LFSR is $(A1, A0)$ and $(B1, B0)$, which are four 128-bit words. The three consecutive keystream words at clock $t-1, t$ and $t+1$ can be expressed as follows:

$$\begin{aligned} z^{(t-1)} &= (AES_R^{-1}(\hat{R2}) \boxplus_{32} T1^{(t-1)}) \oplus AES_R^{-1}(\hat{R3}), \\ z^{(t)} &= (\hat{R1} \boxplus_{32} T1^{(t)}) \oplus \hat{R2}, \\ z^{(t+1)} &= (\sigma(\hat{R2} \boxplus_{32} (\hat{R3} \oplus T2^{(t)})) \boxplus_{32} T1^{(t+1)}) \oplus AES_R(\hat{R1}), \end{aligned}$$

where $\hat{R1}, \hat{R2}, \hat{R3}$ are the values of the three registers in the FSM at time clock t. Any choice of the LFSR update function, for the particular circular-LFSR construction, would result in the following linear relations:

$$\begin{aligned} B1^{(t+1)} &= A0^{(t)} \oplus f_\beta(B0^{(t)}, B1^{(t)}), \\ A1^{(t+1)} &= B0^{(t)} \oplus f_\alpha(A0^{(t)}, A1^{(t)}), \\ B0^{(t+1)} &= B1^{(t)}, \\ A0^{(t+1)} &= A1^{(t)}, \end{aligned}$$

where f_α and f_β are two linear functions that correspond to the LFSR update procedure. These expressions are generic for both SNOW-V and SNOW-Vi.

In SNOW-V, the taps are $T1 = B1$ and $T2 = A0$, which implies that in three consecutive keystream expressions the contribution from the LFSR involves three out of four 128-bit words:

$$\begin{aligned} T1^{(t)} &= B1^{(t)}, \\ T1^{(t-1)} &= B1^{(t-1)} = B0^{(t)}, \\ T2^{(t)} &= A0^{(t)}, \\ T1^{(t+1)} &= B1^{(t+1)} = A0^{(t)} \oplus f_\beta(B0^{(t)}, B1^{(t)}). \end{aligned}$$

Note that those three LFSR words, i.e., $B0^{(t)}, B1^{(t)}$ and $A0^{(t)}$, appear in the three keystream expressions twice, thus there is a chance to explore a biased noise expression by considering only these three consecutive 128-bit keystream words in linear cryptanalysis.

We, however, believe that there is no immediate security threat for SNOW-V as it is most likely that up to 48 SBoxes and many arithmetical additions will be involved in a hypothetical noise expression. The bias there is expected to be very small (e.g., 48 SBoxes would already give the bias $\epsilon(48 \times [x \oplus S(x)]) \approx 2^{-286.4}$), and not enough for mounting a linear attack on SNOW-V.

On the other hand, we have noticed that if we take the pair of taps $(T1, T2)$ from either $(A0, B0)$ or $(A1, B1)$, the three consecutive keystream expressions would involve all four 128-bit words of the LFSR, and, moreover, at least 256 bits of them (values from $A1$ and $A0$) will appear in the keystream expressions only once. For example, if the taps are taken as $T1 = B1$ and $T2 = A1$, it implies: $T1^{(t)} = B1^{(t)}$, $T1^{(t-1)} = B0^{(t)}$,

$T2^{(t)} = A1^{(t)}$, $T1^{(t+1)} = A0^{(t)} \oplus f_\beta(B0^{(t)}, B1^{(t)})$. In this case, one has to collect at least 512 bits of the keystream in order to have *some* nonzero bias. That bias is expected to be even smaller than that in SNOW-V since it would involve more SBoxes and arithmetical additions.

From initialisation analysis perspectives. When we discovered that a new tap position would suggest strengthened security from the linear analysis arguments, we then started to look on what would be the most promising combination, by trying all possible variants and performing a brief MDM (maximum degree monomial) test [EJST07] for each of them. The MDM test can examine the non-random initialisation rounds by checking the distribution of the coefficient of the maximum degree monomial in the Boolean functions of the keystream bits.

Taps		#non-random
T1	T2	rounds
A1	B1	5.69 - 6.21
B1	A1	5.43 - 5.75
A0	B0	6.25 - 7.24
B0	A0	9.16 - 10.8

Table 1: Number of nonrandom initialisation rounds (out of 16) when T_1, T_2 are tapped at different positions under the worst cubes of size three.

In Table 1, for each variant of the tap positions, we get the ranges of non-random initialisation rounds under the worst cubes of size three (the ranges also depend on the key/IV-loading scheme). The smaller values indicate better mixing effect. A good mixing effect also contributes to a better mixing during the keystream generation phase. The obvious choice is to pick the variant $(B1, A1)$ for SNOW-Vi, while keeping key/IV-loading scheme unchanged.

From implementation perspectives. In addition to other implementation tricks, the new tap position $T2 = A1$ makes it possible to first update the LFSR once, then update the FSM twice, since then the two consecutive values of $T1$ and $T2$ become directly available in the content of the LFSR.

4 Security analysis

In this section we perform a step-by-step security re-evaluation of SNOW-Vi based on previously known analyses of SNOW-V, given in [EJMY19, CDM20, GZ21, JLH20].

4.1 Linear attacks

Assume that α and β are 16×16 binary matrices that represent multiplication in corresponding fields. Then we can have the following expressions:

$$\begin{aligned} \beta a^{(t+16)} &= \beta b^{(t)} + \beta\alpha a^{(t)} + \beta a^{(t+7)}, \\ a^{(t+24)} &= b^{(t+8)} + \alpha a^{(t+8)} + a^{(t+15)}, \\ a^{(t+32)} &= b^{(t+16)} + \alpha a^{(t+16)} + a^{(t+23)}. \end{aligned}$$

Since $b^{(t+16)} = \beta b^{(t)} + b^{(t+8)} + a^{(t)}$, adding the three expressions above, we could get the recurrence for a-terms in SNOW-Vi as below:

$$\begin{aligned} 0 &= (x^{16} + x^8 + \beta)(x^{16} + x^7 + \alpha) + 1 \\ &= x^{32} + x^{24} + x^{23} + (\alpha + \beta)x^{16} + x^{15} + \alpha x^8 + \beta x^7 + (1 + \beta\alpha), \end{aligned}$$

to be compared with the feedback recurrence in SNOW-V:

$$0 = (x^{16} + \alpha^{-1}x^8 + x^1 + \alpha)(x^{16} + \beta^{-1}x^8 + x^3 + \beta) + 1.$$

I.e., we have an 8-weight recurrence in SNOW-Vi and a 12-weight one in SNOW-V.

For standard linear distinguishing and correlation attacks one has to find a multiple of the above recurrence of weight 3 or 4. Thus, we believe that 8 is also good enough to be resistant against linear cryptanalysis. In [EJMY19] the complexity of a linear distinguishing attack is around 2^{645} based on a 3-weight multiple. In [GZ21], the authors propose correlation attacks against three reduced variants of SNOW-V, and for the closest variant SNOW-V$_{\boxplus_{32},\boxplus_8}$ the complexity is 2^{377}. Since these linear cryptanalyses focus on approximating the non-linear FSM, these results would also apply to SNOW-Vi. However, both attacks are far more complex than exhaustive key search.

4.2 Attacks on the initialisation

As done for SNOW-V, we use the MDM test and cube attack based on division property to check if the key and IV bits are fully mixed after the initialisation.

4.2.1 MDM tests

In a MDM test, each output keystream bit is regarded as a random Boolean function of the key and IV bits, and the MDM coefficient in the algebraic normal form (ANF) of the Boolean function should follow a random uniform distribution between {0, 1}. However, in the initial few rounds of the initialisation, the mixing effect is not enough and the MDMs of the corresponding Boolean functions are much more likely to be zero than one, thus resulting into a zero sequence before they become random-like. The MDM test checks how long this zero sequence persists throughout the full initialisation rounds. As done for SNOW-V, we start with a relatively small set (size four) of Key/IV bits under which the randomness result deviates the most from the expected value (i.e., the longest zero sequence) and greedily increase to a 24-bit set, i.e., in each step, we add the bit to the existing set which results in the longest zero sequence among all the remaining bits. We also tried adding two bits in each step.

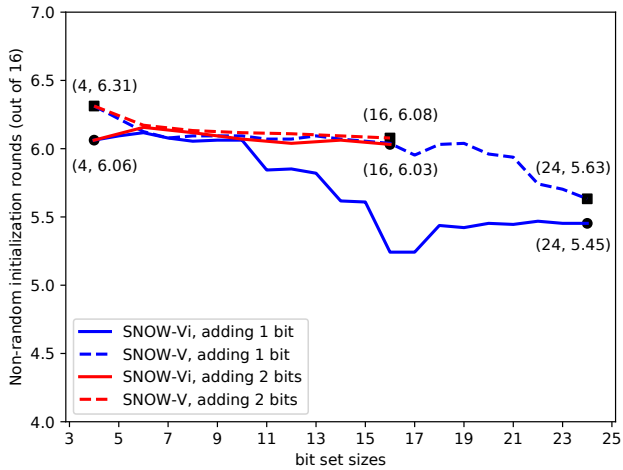

Figure 3: The number of rounds failing the MDM test.

Figure 3 shows the number of rounds failing the MDM test under different bit set sizes compared to SNOW-V when greedily adding one and two bits in each step. From the result, one can see that the randomness of the initialisation output of SNOW-Vi is even better than SNOW-V. Specifically, for the worst set of size four, there are 6.06 rounds that are not random for SNOW-Vi, while 6.31 for SNOW-V. When adding two bits in each step, the difference between SNOW-Vi and SNOW-V is smaller than that when adding one bit. The difference might be larger if the worst bit set of a larger size is explored, or more bits are considered during the greedy steps. However, this is computationally demanding. Next, we use a more fine-grained way based on division property to check the initialisation.

4.2.2 Cube attacks based on division property

Cube attacks based on division property evaluate the set of involved key bits J in the superpoly given a certain cube I (the set of all the possible values of some chosen IV bits while the values of other IV bits are fixed), and recover the superpoly if feasible. The propagation rules of division property for different operations in a cipher can be modelled by some (in)equalities of a MILP (Mixed Integer Linear Programming) problem. By solving the MILP problem using some optimisation tools, one can get the involved key bit set J and the upper bound of the algebraic degree d of the superpoly; the larger $|J|$ and d are, the better the mixing effect is. The time complexity for recovering the superpoly is given as $2^{|I|} \times \binom{|J|}{<d}$ [WHT+18].

The MILP model of SNOW-Vi is generally similar with that for SNOW-V, given in Algorithm 5 in [EJMY19]; while only the modelling for the update of the LFSR should be modified. We tried different cubes and tested the involved key bits and the maximum degrees of the corresponding superpolies under different rounds. The results

Rounds	3		4		≥ 5	
Versions	-Vi	-V	-Vi	-V	-Vi	-V
$\|I\|$	4	15	128	40	128	128
d	28	17	242	145	256	256
$\|J\|$	100	131	256	256	256	256
C	$2^{86.7}$	$2^{84.9}$	$> 2^{256}$	$> 2^{256}$	$> 2^{256}$	$> 2^{256}$

Table 2: Comparison of cube attacks on reduced-round SNOW-Vi and SNOW-V ($|I|, d, |J|$, and C denote the cube size, the degree, the number of involved key bits, and attacking complexity, respectively).

are presented in Table 2 and one can see that the mixing effect of SNOW-Vi is better than SNOW-V. Specifically, after four rounds, for a cube size 40 in SNOW-V, all key bits are involved and the maximum degree is 145. When the cube size goes larger, the number of involved key bits and degree would both reduce. However, in SNOW-Vi, for the cube of all IV bits, all key bits are involved, and the maximum degree is 242. This can be expected since when $T2$ is moved to the higher part of LFSR-A, the new update results of IV and key bits are immediately fed to the FSM, making the mixing faster. After five rounds, all key bits and IV bits are fully mixed just like SNOW-V. These results match well with the results from the MDM test.

4.3 Algebraic attacks

In algebraic attacks one expresses the cipher output as algebraic equations over the unknown key (or state) bits, and tries to solve the resulting system of nonlinear equations. The only source of non-linearity during a normal update iteration of SNOW-Vi is from the FSM, and that is unchanged from SNOW-V. In the algebraic attack analysis of SNOW-V in [CDM20], the authors make use of the fact that the tap values $T1^{(t)}$ and $T2^{(t)}$ are linear combinations of the first values $T1^{(-1)}, T1^{(0)}, T2^{(-1)}, T2^{(0)}$ and each iteration of the cipher can be written as

$$\begin{aligned}
T1^{(t+1)} &= Lin_{\beta}(T1^{(t-1)}, T1^{(t)}, T2^{(t-1)}, T2^{(t)}),\\
T2^{(t+1)} &= Lin_{\alpha}(T1^{(t-1)}, T1^{(t)}, T2^{(t-1)}, T2^{(t)}),\\
R1^{(t+1)} &= \sigma(R2^{(t)} \boxplus_{32} (R3^{(t)} \oplus T2^{(t)}),\\
R2^{(t+1)} &= AES_R(R1^{(t)}),\\
R3^{(t+1)} &= AES_R(R2^{(t)}),\\
z^{(t+1)} &= (R1^{(t)} \boxplus_{32} T1^{(t)}) \oplus R2^{(t)}.
\end{aligned}$$

We can see that these equations are still valid in SNOW-Vi. Following the arguments in [CDM20] we note that the linear parts of the cipher can be *"effectively disregarded when determining the number of nonlinear equations and the number of associated variables"*. Hence the proposed change in linear update functions for $T1$ and $T2$ does not affect the complexity of mounting an algebraic attack using quadratic (or higher degree) equations.

The conclusion is that both linearisation methods and Gröbner basis algorithms remain unfeasible for algebraic attacks on SNOW-Vi.

4.4 Guess-and-determine attacks

In guess-and-determine attacks one guesses part of the state and from the keystream equations determines the other parts. One aims to guess as few bits as possible and then determines as many bits as possible through given equations. For the case of SNOW-Vi the situation is very similar to SNOW-V. The equation $z^{(t)} = (R1^{(t)} \boxplus_{32} T1^{(t)}) \oplus R2^{(t)}$ involves three unknowns, each of size 128 bits. One has to guess two of them (256 bits) in order to determine the remaining one. Looking at the equation for the next keystream word, it requires guessing another 128 bits. This illustrates that a guess-and-determine attack on SNOW-Vi is still of large complexity.

A straightforward guess-and-determine attack is given in [CDM20], which requires guessing 512 bits within three consecutive keystream words to recover the full 896 state bits. The attack there applies to SNOW-Vi exactly the same. Thus we could first get an upper bound on the complexity of the guess-and-determine attack against SNOW-Vi, which is 2^{512}.

In January 2020, Jiao *et al* in [JLH20] gave a byte-based guess-and-determine attack against SNOW-V with complexity 2^{406} using seven keystream words. In their attack, the registers in LFSR and FSM are split into bytes and the update operations are correspondingly transformed to byte-based, while with some carriers introduced. The attack first presets an initial guessing set and runs some algorithm to explore guessing paths and thus driving a guessing basis. This process is repeated several times to remove possible redundant bytes. Though the details of the guess-and-determine attacks against SNOW-V and SNOW-Vi under their attack would be different, the general guessing route could be the same.

The final initial guessing set used in [JLH20] has 24 byte variables, and these variables are all from the FSM registers or the higher halves of the LFSR registers, while the variables which are tapped for update are not used. Thus we could use the same initial guessing set and have a similar guessing path. During the guessing process, 12 more bytes from the FSM registers and 13 more bytes from LFSR are guessed. Since there are three taps for LFSR-A and LFSR-B in SNOW-V while two in SNOW-Vi, we make the worst assumption that when 13 bytes in LFSR are required for guessing in SNOW-V, only around eight bytes are needed in SNOW-Vi. In this case, one still needs to guess $24 + 12 + 8 = 44$ bytes, which are 352 bits. Besides these bytes, some additional carriers must be guessed. Thus the complexity of the guess-and-determine attack against SNOW-Vi is larger than 2^{352}. We can make an even worse assumption that the guessed variables in LFSR can be freely derived, resulting in guessing $24 + 12 = 36$ bytes all from FSM registers, i.e., 288 bits, for which the complexity is still larger than 2^{256}.

Thus we conclude that the guess-and-determine attack would not be faster than exhaustive key search against SNOW-Vi.

4.5 Other analyses

From studying [CDM20], we note that most of the results received for SNOW-V are not affected by the new LFSR:

- the transfer of key entropy (Section 2.1 in [CDM20]), the injectiveness of initialisation (Section 2.4 in [CDM20]), and time-memory-data trade-off attacks (Section 6 in [CDM20]) are not affected since the grounds for these types of analyses are the state size, the key and IV lengths, which are not changed in SNOW-Vi;
- related key-IV attacks (Section 7 in [CDM20]) is not affected since the Key/IV loading scheme is the same as in SNOW-V, which does not create additional entropy in the initial state that could be used to search for collisions in Key/IVs;
- side-channel attacks (Section 8 in [CDM20]) is not affected since modifications in SNOW-Vi do not create any message-dependant routines, and the construction is similar to SNOW-V;
- AEAD mode (Section 9 in [CDM20]) is not affected since it is exactly the same as in SNOW-V;
- In fact, even derivations (Section 3.1 in [CDM20]) on correlation attacks remain true for SNOW-Vi, since the FSM part is not changed in SNOW-Vi, and linear derivations in [CDM20] were performed for a circular-LFSR construction without consideration of the exact positions of $T1$ and $T2$.

Hardware evaluations. In [CBB20] the authors performed a thorough hardware evaluation of SNOW-V, where they looked at three different implementations and reached the throughput rate over 1 Tbps and the energy consumption as low as 12.7 pJ per 128 bits of keystream. This can be compared with AES-256-CTR where the best throughput received is only 80 Gbps and the energy consumption is 952.5 pJ per 128 bits of an encryption block.

For SNOW-Vi, we expect minor changes in hardware compared to SNOW-V. Our assessment is that the throughput rate should not be affected at all, since the critical path is actually in the FSM which is unchanged. The area size and the energy consumption in SNOW-Vi should be slightly better (i.e., lower) than that in SNOW-V, since the new LFSR has a reduced number of gates for its feedback update function, and therefore consumes less power.

5 Software evaluation

Performance of SNOW-Vi heavily depends on the ability to reduce the number of instructions, as well as careful consideration of hardware peculiarities, such as CPU interleaving capabilities, use of registers, instructions latency and throughput characteristics. In this section we analyse SNOW-Vi from the software point of view, considering different implementation techniques and various target platforms.

5.1 Implementations and notations

Algorithms. We have done a dozen of different implementations in C/C++ of SNOW-V and SNOW-Vi, that we can use for relative comparison on various platforms. We also used OpenSSL tools on test targets to measure the performance of AES-256-CTR for

comparison. The notation **AES-256-CTR/ver** will refer to AES-256-CTR in OpenSSL version **ver**.

Registers. In both SNOW-V and SNOW-Vi we have implementations that utilise: only 128-bit registers (e.g., XMM on Intel platforms), and up to 256-bit registers (e.g., YMM). ARM NEON only supports 128-bit registers.

Instruction sets. We have implementation versions with different restrictions in instruction sets. For Intel platforms, we start with the most restricted SSE4.1 set and then add more capabilities as we try implementations utilising AVX2 and AVX-512. For ARM platforms, we only have the NEON instruction set. All implementations and platforms use the AES round function instruction. We present C/C++ versions using Intel intrinsics below, but it is relatively straightforward to convert to NEON.

Code generation. In SSE-type of code generation, the CPU can only handle instructions of the form $x = x + y$, i.e., the value of one input register is changed to hold the result. In AVX-type of code generation, CPU instructions can have 3 arguments, i.e. $x = y + z$, thus the values of the input registers are preserved.

Unrolled versions. By design, both SNOW algorithms would simply have bulk encryption in a loop that process 16 bytes in each step (if we ignore unaligned bytes). That is the same situation as with AES-256-CTR. We call these implementations as 1-unrolled versions. However, there might be a performance gain if each step of such an encryption loop would process 4×16 bytes instead, and the key/IV initialisation is also partly or fully unrolled. We call these implementations 4-unrolled versions.

Notation. We adopt the following notation to indicate a specific case that we were testing: **[Alg/Unroll/Regs-Inst]**, where: **Alg** is the algorithm name – {SNOW-V, SNOW-Vi, AES-256-CTR}; **Unroll** determines if the implementation is a plain one or unrolls four 16-bytes blocks in the encryption loop – {1, 4}; **Regs** determines the maximum size of the registers being used – {128, 256, 512}; **Inst** determines the type of code generation and the maximum instruction sets being used – {SSE, AVX, AVX2, AVX512, NEON}. For 128-SSE case we use up to SSE4.1 instructions.

Examples: SNOW-Vi/1/128-SSE, SNOW-V/4/256-AVX512.

5.2 New test environment

In order to perform a wide software evaluation on various platforms we decided to make a simple, but generic test environment where we utilise the standard C function `time(NULL)`. The granularity of `time()` function is 1 second, so that before each test we are waiting for the start of a "fresh" second, then in the loop we are waiting for the start of the next second, while performing a lot of encryptions with a selected algorithm in a loop and counting the number of encryptions processed. This, of course, has some impact on the received performance numbers. We, however, tried to balance it by calling the function `time()` only after 1024 encryptions. The total count is still magnitudes higher so this approach should not affect the accuracy of the measurements, but partly reduces the impact of the system calls of `time()` function.

In Figure 4 we present the bar chart comparing previous results from [EJMY19] and the new results under the new benchmarking system. One can clearly see that SNOW-Vi can achieve higher speeds than both AES-256 and SNOW-V, and the advantage is larger for longer plaintexts. When the plaintext length is 16 Kbytes or larger, SNOW-Vi can

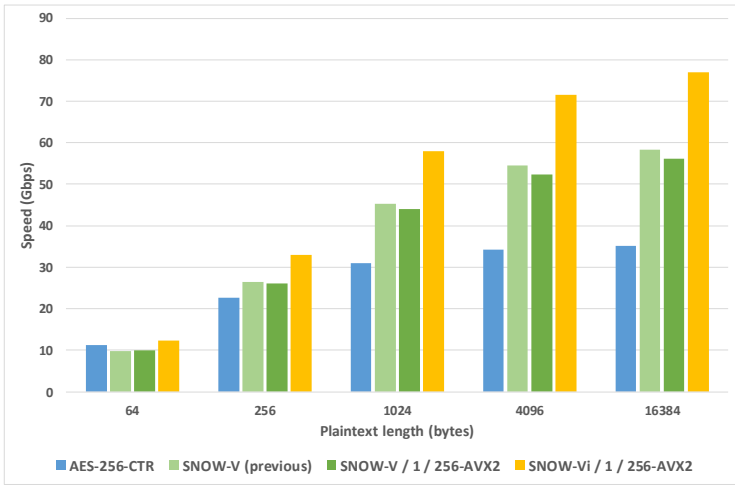

Figure 4: Previous and new benchmarks (platform: work laptop, Win10, Intel i7-8650U @ 4.2GHz / AVX2).

reach the speed of 77 Gbps on the given platform. The exact values of the speeds are given in Appendix 4, Table 3.

5.3 Impact of unrolling and code generations

In Figure 5 we demonstrate the difference between a "usual" and "unrolled" implementations with basically the same 128-bit friendly core code for SNOW-Vi. We can see a significant speedup when unrolling loops, especially in SSE-type of code generation. The exact values of these speeds are given in Appendix 4, Table 4.

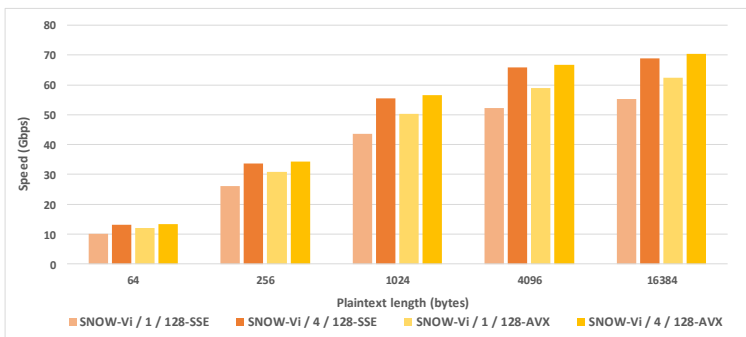

Figure 5: Impact of unrolling (platform: work laptop, Win10, Intel i7-8650U @ 4.2GHz / AVX2).

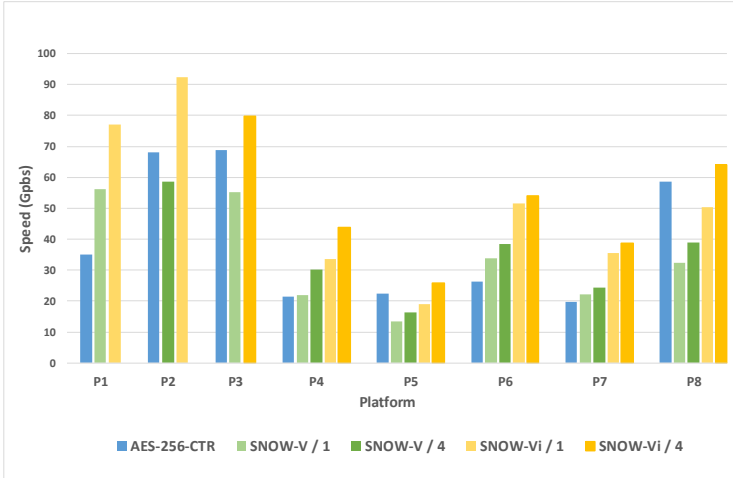

Figure 6: Performance measurements on various platforms (P1∼P8) for plaintext with length 16 Kbytes.

5.4 Performance results

We also tested the performance of different algorithms on a number of other platforms and for various use cases. Figure 6 provides an illustration of the performance comparison under these platforms when the length of the plaintext is 16 Kbytes. We can see that around +50% speed up in throughput of the fastest SNOW-Vi with respect to the fastest SNOW-V is achieved, in average. The detailed information of the eight different platforms P1∼P8 and more comprehensive performance benchmarks under different plaintext lengths are given in Appendix 4, Table 5. We will see SNOW-Vi generally achieves higher speeds on all the tested platforms.

5.5 Implementation optimisations

In Listing 1 we introduce a number of macros, in order to simplify our further C++ listings.

Listing 1: SIMD macros.

```
#define XOR(a, b)      _mm_xor_si128(a, b)
#define AND(a, b)      _mm_and_si128(a, b)
#define ADD(a, b)      _mm_add_epi32(a, b)
#define SET(v)         _mm_set1_epi16((short)v)
#define SLL(a)         _mm_slli_epi16(a, 1)
#define SRA(a)         _mm_srai_epi16(a, 15)
#define TAP7(Hi, Lo)   _mm_alignr_epi8(Hi, Lo, 7 * 2)
#define SIGMA(a)       \
_mm_shuffle_epi8(a, _mm_set_epi64x( \
0x0f0b07030e0a0602ULL, 0x0d0905010c080400ULL));
#define AESR(a, k)     _mm_aesenc_si128(a, k)
#define ZERO()         _mm_setzero_si128()
```

```
#define LOAD(src)     \
_mm_loadu_si128((const __m128i*)(src))
#define STORE(dst, x) \
_mm_storeu_si128((__m128i*)(dst), x)
```

In Appendix 2 we give an "easy-to-read" reference implementation of SNOW-Vi, with test vectors given in Appendix 3. However, a faster implementation can employ additional tricks, such as the call of the AES round function with $T2$ as the round key, thus XORing $T2$ with $R3$ is "for free". One can also optimise the order of instructions for a better performance on a selected platform, see Listing 2 as an example of such efforts for SSE-type of code generation.

Listing 2: Optimised implementation of SNOW-Vi utilising XMM registers for SSE platforms.

```
#define SnowVi_XMM_ROUND(mode, offset)\
T1 = B1, T2 = A1;\
A1 = XOR(XOR(XOR(TAP7(A1,A0), B0), AND(SRA(A0),\
SET(0x4a6d))), SLL(A0));\
B1 = XOR(XOR(B1, AND(SRA(B0), SET(0xcc87))),\
XOR(A0, SLL(B0)));\
A0 = T2; B0 = T1;\
if (mode == 0) A1 = XOR(A1, XOR(ADD(T1, R1), R2));\
else STORE(out + offset, XOR(ADD(T1, R1),\
XOR(LOAD(in + offset), R2)));\
T2 = ADD(R2, R3);\
R3 = AESR(R2, A1);\
R2 = AESR(R1, ZERO());\
R1 = SIGMA(T2);

// Note: here the length must be 16-bytes aligned
inline void SnowVi_encdec(int length, u8 * out,
u8 * in, u8 * key, u8 * iv)
{   __m128i A0, A1, B0, B1, R1, R2, R3, T1, T2;

    // key/IV loading
    B0 = R1 = R2 = ZERO();
    A0 = LOAD(iv);
    R3 = A1 = LOAD(key);
    B1 = LOAD(key + 16);

    // Initialisation
    for (int i = -14; i < 2; ++i)
    {   SnowVi_XMM_ROUND(0, 0);
        if (i < 0) continue;
        R1 = XOR(R1, LOAD(key + i * 16));
    }

    // Bulk encryption
    for (int i = 0; i <= length - 16; i += 16)
    {   SnowVi_XMM_ROUND(1, i); }
}
```

A better optimisation may be achieved on the assembly level. At our best try, a single encryption/decryption of a 16-byte block data may be done with as low as 15 assembly instructions by utilising 12 XMM/YMM registers and up to AVX512 instruction sets. In the initialisation loop the main code can be shrunk down to 13 assembly instructions,

see Listing 3; however, there we omit 2-3 extra instructions that are usually also needed to organise the loop itself.

Listing 3: Sketch for an assembly implementation.

```
;Note: for a 256-bit register the pair of two 128-bit values are (Hi|
Lo)
;Input State:
;ymm1 = hi = (B[128..255] | A[128..255])
;ymm2 = lo = (B[0..127] | A[0..127])
;xmm7 = R1
;xmm8 = R2
;xmm9 = R3 xor A[128..255]
;
;Constants & Derivatives:
;ymm5 = (A[0..127] | B[0..127])
;     = _mm256_permute4x64_epi64(lo, 0x4e)
;ymm4 = _mm256_set_epi64x(
;       0xcc87cc87cc87cc87ULL, 0xcc87cc87cc87cc87ULL,
;       0x4a6d4a6d4a6d4a6dULL, 0x4a6d4a6d4a6d4a6dULL);
;xmm10 = _mm_setzero_si128()
;xmm11 = _mm_set_epi64x(
;        0x0f0b07030e0a0602ULL, 0x0d0905010c080400ULL)
;Load the mask register k1 with 0x0000ffff, e.g.:
;   mov        eax, 65535
;   kmovd      k1, eax
;
;Encryption/Decryption Loop for one 16-byte block:
1.  vmovdqu    ymm3, ymm1
2.  vpsraw     ymm6, ymm2, 15
3.  vpternlogd ymm6, ymm4, ymm5, 106
4.  vpalignr   ymm1 {k1}, ymm1, ymm2, 14
5.  vpsllw     ymm2, ymm2, 1
6.  vpternlogd ymm1, ymm2, ymm6, 150
7.  vmovdqu    xmm2, XMMWORD PTR[r8+rdx]; load in[i*16]
8.  vpermq     ymm5, ymm3, 78
9.  vpaddd     xmm12, xmm7, xmm5
10. vpternlogd ymm2, ymm8, ymm12, 150
11. vpaddd     xmm12, xmm8, xmm9
12. vaesenc    xmm9, xmm8, xmm1
13. vaesenc    xmm8, xmm7, xmm10
14. vmovdqu    XMMWORD PTR[rdx], xmm2; store out[i*16]
15. vpshufb    xmm7, xmm12, xmm11

;Output State: same registers as inputs, except that the new ymm2 is
now actually ymm3. One solution could be to add vmovdqu ymm2, ymm3;
but a better way is to call the above code with swapped registers xmm2
/ymm2 and xmm3/ymm3. I.e., a 2-unrolled loop would be more efficient.

;Initialisation Loop: remove steps 7 and 14, and in step 10 change
ymm2 to ymm1 (=hi). In the last 2 rounds one should XOR the key to the
 register xmm7 (=R1).
```

Implementation tricks. The presented sketch of an assembly code has just a single 256-bit "swap" instruction `vpermq` (step 8) and no `vextractf128` for extracting the taps, thus saving CPU latency since these instructions are costly. There is only one register copy `vmovdqu` (step 1), that we believe is the minimum and unavoidable. We use one of the AES round calls (step 12) with the next clock's value of the tap $T2$ as the "round

key", thus we can skip one XOR instruction ($R3$ `xor` $T2$) during the next clock. We also efficiently utilise the fact that XMM/YMM registers are shared (steps 9 and 10 in the initialisation loop) and we use AVX512's mask register `k1` (step 4) to avoid an extra `vpblendd`. [2] The above code adopts AVX512's ternary logic `vpternlogd` (steps 3, 6, 10) that effectively removes three extra instructions if we would do these steps with the AVX2 set, instead. We can avoid the ending register copy (`vmovdqu ymm2, ymm3`) by implementing 2x-unrolled loops. The above 15 assembly steps demonstrate all these tricks.

Nevertheless, we would like to note that the smallest number of assembly instructions does not always mean the fastest speed in reality, since there are other things to take care about such as instructions interleaving and stitching techniques. For example, one could utilise more than 12 registers to convey a better instructions stitching and thus achieve a higher performance.

6 Conclusions

In this paper we present a slightly modified version of the SNOW-V stream cipher called SNOW-Vi. The purpose of this change is to better accommodate a fast implementation in software on CPUs which only supports 128-bit wide SIMD registers or a limited SIMD instruction set. The only change made, is a small modification to the linear update function and the tap position for $T2$. We thoroughly investigate the security implications of this change and go through all previously known analyses of SNOW-V, applying the changes to these security results. The conclusion is that the high security provided by SNOW-V is still intact, and in some cases even improved. Furthermore, we provide a very detailed software evaluation, comparing SNOW-Vi to both SNOW-V and AES-256-CTR on various CPU architectures. The results show that SNOW-Vi is significantly faster than SNOW-V on all platforms.

Acknowledgement

We would like to thank all anonymous reviewers for their highly valuable comments and questions to us, which helped to improve this article at a great extent.

This work was in part financially supported by the Swedish Foundation for Strategic Research, grant RIT17-0005 and the ELLIIT program. Jing Yang is also supported by the scholarship from the National Digital Switching System Engineering and Technological Research Center, China.

References

[3GP19] 3GPP. Ts 33.841 (v16.1.0): 3rd generation partnership project; technical specification group services and systems aspects; security aspects;

[2] One may also try to use SSE-legacy instruction in step 4: `palignr xmm1, xmm2, 14` – that would modify the lower half of `ymm1` while preserving it's upper half, as we actually want here; however, there might be a timely AVX-SSE switch penalty.

study on the support of 256-bit algorithms for 5g (release 16). March 2019. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3422.

[3GP20] 3GPP. Ts 33.501: 3rd generation partnership project; technical specification group services and system aspects; security architecture and procedures for 5g system. December 2020. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3169.

[BKR11] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique cryptanalysis of the full AES. In *International conference on the theory and application of cryptology and information security*, pages 344–371. Springer, 2011.

[CBB20] Andrea Caforio, Fatih Balli, and Subhadeep Banik. Melting SNOW-V: improved lightweight architectures. *Journal of Cryptographic Engineering*, 4 December 2020.

[CDM20] Carlos Cid, Matthew Dodd, and Sean Murphy. A Security Evaluation of the SNOW-V Stream Cipher. 4 June 2020. Quaternion Security Ltd. https://www.3gpp.org/ftp/tsg_sa/WG3_Security/TSGS3_101e/Docs/S3-202852.zip.

[EJMY19] Patrik Ekdahl, Thomas Johansson, Alexander Maximov, and Jing Yang. A new snow stream cipher called SNOW-V. *IACR Transactions on Symmetric Cryptology*, pages 1–42, 2019.

[EJST07] Håkan Englund, Thomas Johansson, and Meltem Sönmez Turan. A framework for chosen IV statistical analysis of stream ciphers. In K. Srinathan, C. Pandu Rangan, and Moti Yung, editors, *Progress in Cryptology – INDOCRYPT 2007*, pages 268–281, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[GZ21] Xinxin Gong and Bin Zhang. Resistance of SNOW-V against fast correlation attacks. *IACR Transactions on Symmetric Cryptology*, (1):378–410, March 2021.

[JLH20] Lin Jiao, Yongqiang Li, and Yonglin Hao. A Guess-And-Determine Attack On SNOW-V Stream Cipher. *The Computer Journal*, 63(12):1789–1812, 03 2020.

[WHT+18] Qingju Wang, Yonglin Hao, Yosuke Todo, Chaoyun Li, Takanori Isobe, and Willi Meier. Improved division property based cube attacks exploiting algebraic properties of superpoly. In *Annual International Cryptology Conference*, pages 275–305. Springer, 2018.

[YJM19] Jing Yang, Thomas Johansson, and Alexander Maximov. Vectorized linear approximations for attacks on SNOW 3G. *IACR Transactions on Symmetric Cryptology*, pages 249–271, 2019.

[YJM20] Jing Yang, Thomas Johansson, and Alexander Maximov. Spectral analysis of ZUC-256. *IACR Transactions on Symmetric Cryptology*, pages 266–288, 2020.

Appendices

1 Characteristic polynomial for the LFSR in SNOW-Vi

The characteristic polynomial $m(x)$ for the proposed LFSR is $m(x) = \sum_{i=1}^{|T|} x^{T_i}$, where:

$$\begin{aligned} T = [&512, 496, 488, 480, 472, 462, 455, 448, 444, 439, 438, 437, 430, 426,\\ &422, 421, 420, 419, 414, 412, 408, 404, 403, 402, 401, 399, 398, 394, 392,\\ &390, 387, 386, 385, 384, 382, 381, 380, 373, 371, 369, 367, 366, 359, 358,\\ &353, 351, 350, 349, 347, 346, 341, 340, 339, 336, 335, 334, 333, 329, 319,\\ &318, 317, 316, 314, 313, 312, 311, 310, 309, 305, 304, 303, 302, 301, 300,\\ &298, 297, 296, 295, 291, 290, 289, 287, 281, 280, 278, 277, 276, 275, 273,\\ &272, 270, 267, 266, 263, 262, 261, 258, 257, 254, 252, 246, 245, 243, 235,\\ &233, 231, 229, 228, 226, 225, 224, 223, 222, 221, 220, 219, 218, 216, 215,\\ &214, 212, 211, 210, 207, 201, 199, 198, 197, 196, 195, 194, 192, 191, 190,\\ &189, 185, 184, 181, 179, 175, 173, 170, 169, 168, 166, 160, 158, 156, 155,\\ &152, 147, 146, 145, 143, 140, 137, 136, 134, 133, 132, 131, 128, 127, 125,\\ &116, 111, 109, 108, 105, 103, 101, 100, 99, 98, 95, 94, 90, 86, 84, 82, 80,\\ &79, 75, 74, 73, 70, 69, 68, 67, 57, 55, 52, 51, 50, 49, 48, 45, 43, 42, 36, 32,\\ &23, 22, 21, 16, 14, 8, 7, 0]. \end{aligned}$$

2 Reference implementation

A 128-SSE friendly C/C++ code of SNOW-Vi is given in Listing 4. It is not optimised for performance benchmarking but rather serves as an "easy-to-read" reference implementation.

Listing 4: Reference implementation of SNOW-Vi.

```
#include <intrin.h> // or <x86intrin.h> for gcc
#define XOR(a, b)      _mm_xor_si128(a, b)
#define AND(a, b)      _mm_and_si128(a, b)
#define ADD(a, b)      _mm_add_epi32(a, b)
#define SET(v)         _mm_set1_epi16((short)v)
#define SLL(a)         _mm_slli_epi16(a, 1)
#define SRA(a)         _mm_srai_epi16(a, 15)
#define TAP7(Hi, Lo)   _mm_alignr_epi8(Hi, Lo, 7 * 2)
#define SIGMA(a)       \
```

```
_mm_shuffle_epi8(a, _mm_set_epi64x( \
0x0f0b07030e0a0602ULL, 0x0d0905010c080400ULL));
#define AESR(a, k)    _mm_aesenc_si128(a, k)
#define ZERO()        _mm_setzero_si128()
#define LOAD(src)     \
_mm_loadu_si128((const __m128i*)(src))
#define STORE(dst, x) \
_mm_storeu_si128((__m128i*)(dst), x)

struct SnowVi
{   __m128i A0, A1, B0, B1;  // LFSR
    __m128i R1, R2, R3;      // FSM

    inline __m128i keystream(void)
    {   // Taps
        __m128i T1 = B1, T2 = A1;
        // LFSR-A/B
        A1 = XOR(XOR(XOR(TAP7(A1, A0), B0),\
        SLL(A0)), AND(SET(0x4a6d), SRA(A0)));
        B1 = XOR(XOR(SLL(B0), A0), XOR(B1,\
        AND(SET(0xcc87), SRA(B0))));
        A0 = T2;
        B0 = T1;
        // Keystream word
        __m128i z = XOR(R2, ADD(R1, T1));
        // FSM Update
        T2 = ADD(XOR(T2, R3), R2);
        R3 = AESR(R2, ZERO());
        R2 = AESR(R1, ZERO());
        R1 = SIGMA(T2);
        return z;
    }

    template<int aead_mode = 0> inline void keyiv_setup(
    const unsigned char * key, const unsigned char * iv)
    {   B0 = R1 = R2 = R3 = ZERO();
        A0 = LOAD(iv);
        A1 = LOAD(key);
        B1 = LOAD(key + 16);
        if (aead_mode)
        B0 = LOAD("AlexEkd JingThom");
        for (int i = 0; i < 15; ++i)
        A1 = XOR(A1, keystream());
        R1 = XOR(R1, LOAD(key));
        A1 = XOR(A1, keystream());
        R1 = XOR(R1, LOAD(key + 16));
    }
};

// ... some test program
#include <stdio.h>
int main()
{   SnowVi s;
    unsigned char key[32] = { 0 }, iv[16] = { 0 };
    s.keyiv_setup(key, iv);

    for (int t = 0; t < 4; t++)
    {   unsigned char ks[16];
```

```
            STORE(ks, s.keystream());
            for (int i = 0; i < 16; i++)
            printf("%02x ", (unsigned int)ks[i]);
            printf("\n");
        }
        return 0;
    }
```

In a standard stream cipher, the encryption (and decryption) algorithm is an XOR of the keystream with the plaintext (ciphertext). Unused bytes of the last keystream word are simply discarded.

3 Test vectors

Listing 5: Test vectors for SNOW-Vi.

```
== SNOW-Vi test vectors #1:
key =
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

iv  =
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Initialisation phase, z =
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5
4f 4f 4f 4f 4f 4f 4f 4f 4f 4f 4f 4f 4f 4f 4f 4f
7a 5b 5a 5a 79 5b 5a 5a 5a 5a 5a 5b 5a 5a 5a 5a
4d 51 be 6e 19 0a 0a a9 a4 fe fe ae f4 d6 d6 d6
7a 97 fb 47 d2 57 62 46 8a ca df 1e d1 48 4d 3c
8c 97 87 3e 90 00 38 d5 2d f3 46 c3 2f f7 97 0c
10 89 37 a1 02 46 61 0a 67 07 b5 4e 94 1e 0e 3b
94 36 b9 e3 3b 0f 10 9a dc 89 b3 d5 a3 ae f8 2d
ba ea 9f d0 68 b9 a1 1e 43 62 67 f8 7f 4a 05 ac
0c 15 12 c2 38 80 09 46 5a 55 ef f8 89 81 6c 97
75 82 9e c8 a8 73 70 38 cd 5e c5 7e 21 9d 98 16
ed 45 92 3c 43 7a d7 b0 e5 22 61 72 85 47 dc be
e9 38 ac 0b 70 5c b9 85 2a 42 49 ba 0e 87 37 c3
65 28 2c ef ab 7c a9 57 ae f8 d9 4e 29 38 c8 cd

Keystream phase, z =
50 17 19 e1 75 e4 9f b7 41 ba bf 6b a5 de 60 fe
cd a8 b3 4d 7e c4 c6 42 97 55 c1 9d 2f 67 18 71
89 57 d3 26 cb 46 50 2c eb 81 4c cd 6e a5 3a ae
dd 6c 92 fb f3 92 1e 8b d7 31 7b e2 20 15 31 bb
09 3e e8 72 e9 eb 40 34 e9 b7 1a 4a c2 b5 4b d9
f0 0f 5a dc 06 d2 e6 b5 9f b7 5a 01 be f6 13 14
1c 8a b2 02 ee 38 e2 85 0c ca 60 6a b8 75 cd 12
41 03 b3 2f a5 14 5d df 54 e7 a0 7b 0f 3e b7 7a


== SNOW-Vi test vectors #2:
key =
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
```

```
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff

iv  =
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
Initialisation phase, z =
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
9c 9c 9c 9c 9c 9c 9c 9c 9c 9c 9c 9c 9c 9c 9c 9c
cf 09 cc 09 d6 10 d7 10 cc 36 cd 36 d6 10 d7 10
e5 31 72 b0 e8 91 53 dd 75 b0 3e 31 54 dd d2 91
22 b3 31 da e5 05 d7 91 66 7b 7d fb 3f 84 a3 ff
cd d6 c9 02 9e 24 76 3a 19 82 bc 3c 79 d1 9d 62
e1 a3 fb ac ea 2b 6d 68 a1 a7 51 04 3a 46 0b db
b2 30 52 68 82 4b 88 09 ac 92 d5 7d 00 7e ad 0c
79 74 7c eb 01 95 02 a7 1a 2f f5 07 7c 89 96 ad
a1 06 eb d4 c1 d8 5f 12 61 81 e1 a9 55 1b 3b df
aa 5d ff 5a 66 a3 67 16 f7 dc c2 ec 3f da 64 3d
ad 4d ee 83 27 29 15 0a 3e f3 3c 9e d5 79 d9 79
50 a4 a0 dd 21 a0 1c 40 68 31 e6 2e 9d 38 ef 0d
d3 3c c5 72 1b 4d fa 2f 2c cd c9 1f b1 73 fb f3
e8 d0 e3 f1 14 e3 2a 20 ff 56 df 09 7c ab f8 04
1e 24 ae 32 56 9f 7b 08 82 30 4d 80 37 cb 23 b2

Keystream phase, z =
18 71 53 c0 88 1d 00 e8 bf a0 e2 fa fe 71 5e a3
8d e7 fd 87 a6 76 17 1c a1 5e 47 5b 4d a7 b8 7d
ad 86 fc fd 9e 0f bb be ef 6a f4 5f 39 29 c1 23
9b f3 e5 ef b7 d6 90 e6 9d 60 7d c5 c0 4f f4 77
4c 9f 06 a2 b6 36 3e 52 fc b3 0b 8f d3 9f e7 6e
11 64 a6 bd a4 73 4a 76 ee 5f e9 ff 28 ff c1 39
f9 c6 f1 7d 48 43 0c 18 df 3c f4 5d 23 5e dc b3
f6 d4 d1 0b f6 75 f4 ac c4 fb b0 88 cc 5e c4 90


== SNOW-Vi test vectors #3:
key =
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
0a 1a 2a 3a 4a 5a 6a 7a 8a 9a aa ba ca da ea fa

iv  =
01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10

Initialisation phase, z =
0a 1a 2a 3a 4a 5a 6a 7a 8a 9a aa ba ca da ea fa
38 ee a4 da 77 24 62 90 a5 ff 09 e3 6c 85 50 29
c3 62 78 ce 97 43 29 97 7e b0 df 7c 2e 5b 9b a2
ea ba cd 10 4a 5f 1d dd 71 58 96 16 11 e9 59 6e
98 e8 c1 c4 30 18 9d f2 97 f0 0d ce 37 a1 69 bc
d9 82 ee 9c db 03 04 cc 23 22 5e d1 8b dc ae ab
30 00 67 12 44 dd 55 52 12 f4 ae 68 a0 da a3 d0
87 48 b7 ac f4 67 00 37 ce 67 a7 42 71 4e e1 18
91 27 9b f8 ca 8e a1 2d 82 6b 6c f7 b7 ef a9 ce
b4 f0 16 c9 9d d9 7a 3e 76 30 71 f0 99 24 01 a7
24 aa b3 0e d4 fc cf e8 41 8a c5 74 8f 53 c4 47
14 7b fa 54 f5 2f ad 01 ab 96 d6 cc da 01 ee 86
23 fd d5 4f 2b 8d d6 0d 6c d0 b3 de da 70 42 e1
0c 73 a0 0f e2 87 78 1f 5c 1b 92 0c 00 16 b8 0c
b1 49 b2 9c df da 0c 95 b9 d3 18 96 91 81 a2 ec
ea ba d3 84 90 c8 cf b6 a1 f5 80 e0 6f d7 74 33
```

```
Keystream phase, z =
3a 40 f5 40 f5 47 f0 0f 2d 6f e3 d0 01 c1 40 3a
c7 05 9a 39 19 78 4f ab 41 4b be f7 59 25 e5 23
7e 12 45 4a ea 9e 01 1c e4 46 29 ad f3 f7 a8 bb
7e 26 bd 6c 42 95 ce 62 6a 70 b6 4b 41 48 f7 b3
b4 e2 33 57 5a f9 ba 7a 76 34 a6 bb 22 c7 40 77
3e be eb ed 5a 94 94 d5 3a 2b 95 86 03 0d 68 7d
28 f9 7e c9 83 fd 76 41 3e d6 55 1b df 89 f1 eb
30 c2 4d 1c 61 2d 5a 93 14 d7 64 d8 22 7e 4d bf
```

4 Performance Tables

The comprehensive performance benchmarks under different plaintext lengths on various platforms are given in Table 3, Table 4 and Table 5.

Encryption speed (Gbps)	Plaintext length				
	16384	4096	1024	256	64
P1(a): Work laptop, Win10, Intel i7-8650U @ 4.2GHz / AVX2					
	Previous benchmarks from [EJMY19]				
AES-256-CTR/1.1.1j	35.06	34.16	30.95	22.67	11.32
SNOW-V (C++)	58.25	54.60	45.28	26.37	9.85
	New code and test environment				
SNOW-V/1/256-AVX2	56.10	52.28	44.05	26.10	9.94
SNOW-Vi/1/256-AVX2	**77.04**	71.54	57.95	33.01	12.25

Table 3: New test environment, previous and new benchmarks.

Encryption speed (Gbps)	Plaintext length				
	16384	4096	1024	256	64
P1(b): Work laptop, Win10, Intel i7-8650U @ 4.2GHz / AVX2					
SNOW-Vi/1/128-SSE	55.16	52.14	43.52	26.11	10.04
SNOW-Vi/4/128-SSE	68.85	65.93	55.42	33.60	13.12
SNOW-Vi/1/128-AVX	62.28	58.82	50.31	30.93	12.12
SNOW-Vi/4/128-AVX	70.33	66.71	56.59	34.36	13.31

Table 4: Impact of unrolling and SSE/AVX instruction encodings with 128-bit code.

Table 5: Performance measurements on various platforms.

Encryption speed (Gbps)	Plaintext length				
	16384	4096	1024	256	64
P1: Work laptop, Win10, Intel Core i7-8650U @ 4.2GHz / AVX2 (speed up +37%)					
AES-256-CTR/OpenSSL 1.1.1j	35.06	34.16	30.95	22.67	11.32
SNOW-V/1/256-AVX2	56.10	52.28	44.05	26.10	9.94
SNOW-Vi/1/256-AVX2	77.04	71.54	57.95	33.01	12.25

P2: Home laptop, Win10, Intel Core i7-1065 G7 @ 3.9GHz / AVX512 (+58%)					
AES-256-CTR/OpenSSL 3.0.0	68.09	66.07	57.85	38.73	16.42
SNOW-V/4/256-AVX512	58.52	55.57	45.92	27.16	10.33
SNOW-Vi/1/256-AVX512	**92.34**	85.97	69.16	38.60	14.12
P3: Work Station, Ubuntu, AMD Ryzen 5 3600 @ 4.2GHz / AVX2 (+44%)					
AES-256-CTR/OpenSSL 1.1.1f	68.84	67.03	58.35	33.69	18.89
SNOW-V/1/256-AVX2	55.16	51.77	42.45	24.05	8.81
SNOW-Vi/4/128-AVX	79.79	75.77	64.65	40.88	16.56
P4: Remote VM, Ubuntu, Intel Xeon E3-12xx / AVX (+45%)					
AES-256-CTR/OpenSSL 1.1.1	21.57	20.93	19.89	15.81	7.84
SNOW-V/1/128-SSE	22.01	20.87	17.84	11.13	4.37
SNOW-V/4/128-AVX	30.20	28.69	23.71	14.28	5.50
SNOW-Vi/1/128-SSE	33.55	31.57	25.85	16.06	6.18
SNOW-Vi/4/128-AVX	43.75	41.91	35.63	22.25	8.86
P5: Intel NUC7JY, Ubuntu, Intel Pentium Silver J5005 @ 2.8GHz / SSE4.2 (+59%)					
AES-256-CTR/OpenSSL 1.1.1	22.46	21.81	20.12	15.08	7.29
SNOW-V/1/128-SSE	13.56	12.92	10.91	7.14	2.94
SNOW-V/4/128-SSE	16.24	15.23	12.60	7.41	2.82
SNOW-Vi/1/128-SSE	19.06	18.15	15.49	10.57	4.35
SNOW-Vi/4/128-SSE	25.90	24.60	21.05	13.43	5.54
P6: Older laptop, Win7, Intel i7-3540M @ 3GHz / AVX (+40%)					
AES-256-CTR/OpenSSL 1.1.1i	26.33	25.62	23.25	16.77	7.41
SNOW-V/4/128-SSE	33.96	32.01	26.44	15.33	5.73
SNOW-V/4/128-AVX	38.52	36.57	30.30	17.96	6.79
SNOW-Vi/4/128-SSE	51.54	48.96	41.19	25.18	9.87
SNOW-Vi/4/128-AVX	53.96	51.14	43.08	26.19	10.18
P7: Mobile phone, iPhone X, ARM-based A11 Bionic @ 2.39GHz / NEON (+58%)					
AES-256-CTR/OpenSSL 1.1.1i	19.74	19.53	17.86	13.74	8.94
SNOW-V/1/128-NEON	22.25	21.39	18.51	11.72	4.80
SNOW-V/4/128-NEON	24.46	23.54	19.85	12.47	5.19
SNOW-Vi/1/128-NEON	35.42	34.07	29.79	19.18	8.11
SNOW-Vi/4/128-NEON	38.70	37.42	32.69	21.66	10.12
P8: Apple Mini, macOS, ARM-based Apple M1 @ 3.2GHz / NEON (+64%)					
AES-256-CTR/OpenSSL 1.1.1i	58.61	57.44	55.13	45.73	24.97
SNOW-V/1/128-NEON	32.48	30.97	26.47	16.74	6.80
SNOW-V/4/128-NEON	39.06	37.31	31.68	19.78	7.95
SNOW-Vi/1/128-NEON	50.47	48.15	41.21	26.09	10.84
SNOW-Vi/4/128-NEON	64.16	61.10	51.39	31.46	12.78

Paper VI

Revisiting the Concrete Security of Goldreich's Pseudorandom Generator

Abstract

Local pseudorandom generators are a class of fundamental cryptographic primitives having very broad applications in theoretical cryptography. Following Couteau et al.'s work at ASIACRYPT 2018, this paper further studies the concrete security of one important class of local pseudorandom generators, i.e., Goldreich's pseudorandom generators. Our first attack is of the guess-and-determine type. Our result significantly improves the state-of-the-art algorithm proposed by Couteau et al., in terms of both asymptotic and concrete complexity, and breaks all the challenge parameters they proposed. For instance, for a parameter set suggested for 128 bits of security, we could solve the instance faster by a factor of about 2^{77}, thereby destroying the claimed security completely. Our second attack further exploits the extremely sparse structure of the predicate P_5 and combines ideas from iterative decoding. This novel attack, named guess-and-decode, substantially improves the guess-and-determine approaches for cryptographic-relevant parameters. All the challenge parameter sets proposed in Couteau et al.'s work in ASIACRYPT 2018 aiming for 80-bit (128-bit) security levels can be solved in about 2^{58} (2^{78}) operations. We suggest new parameters for achieving 80-bit (128-bit) security with respect to our attacks. We also extend the attacks to other promising predicates and investigate their resistance.

Keywords: Goldreich's pseudorandom generators, guess-and-determine, guess-and-decode, iterative decoding, P_5.

1 Introduction

Pseudorandom generators (PRGs) are one fundamental construction in cryptography, which derive a long pseudorandom output string from a short random string. One particular interesting question about PRGs is the existence of constructions in complexity class $\mathtt{NC}^0$, i.e., each output bit depends on a constant number of input bits. Such constructions, named local pseudorandom generators, can be computed in parallel with

Jing Yang, Qian Guo, Thomas Johansson, and Michael Lentmaier. Revisiting the Concrete Security of Goldreich's Pseudorandom Generator. IEEE Transactions on Information Theory, accepted, 2021.

constant-depth circuits, thus being highly efficient. Considerable research effort has been devoted to this problem [Gol00, CM01, MST03, AIK06, AIK08, App13, ABR16, OW14].

In 2000, Goldreich suggested a simple candidate one-way function based on expander graphs [Gol00], which has inspired a promising construction for PRGs in $\mathtt{NC}^0$. It is constructed as below: given a secret seed x of length n and a well-chosen predicate P with locality $d(n)$: $\{0,1\}^{d(n)} \mapsto \{0,1\}$, choose m subsets $(\sigma^1, \sigma^2, ..., \sigma^m)$, where each subset contains $d(n)$ disjoint indices of x which are chosen randomly and independently. Let $x[\sigma]$ denote the subset of the bits of x indexed by σ and by applying P on $x[\sigma]$, one output bit $P(x[\sigma])$ is obtained. The output string is generated by applying P to all the subsets of bits of x indexed by the sets $(\sigma^1, \sigma^2, ..., \sigma^m)$, i.e., the output string is $P(x[\sigma^1]), P(x[\sigma^2]), \ldots, P(x[\sigma^m])$. Goldreich advocates $m = n$ and depth $d(n)$ in $O(\log n)$ or $O(1)$, and conjectures that it should be infeasible to invert such a construction for a well-chosen predicate P in polynomial time. The case of $d(n)$ in $O(1)$, which puts the construction into the complexity class $\mathtt{NC}^0$, has received more attention due to the high efficiency.

Cryan and Miltersen first considered the existence of PRGs in $\mathtt{NC}^0$ [CM01] and obtained some results: they applied statistical linear tests on the output bits and ruled out the existence of PRGs in $\mathtt{NC}^0_3$ (i.e., each output bit depends on three input bits) for $m \geq 4n$. Mossel et al. further extended this non-existence to $\mathtt{NC}^0_4$ for $m \geq 24n$ with a polynomial-time distinguisher, but provided some positive results for PRGs in $\mathtt{NC}^0_5$ [MST03]. Specifically, they gave a candidate PRG in $\mathtt{NC}^0_5$ instantiated on a degree-2 predicate, which is usually called P_5 defined as:

$$P_5(x_1, x_2, x_3, x_4, x_5) = x_1 \oplus x_2 \oplus x_3 \oplus x_4 x_5,$$

with superlinear stretch while exponentially small bias. Such local PRGs are now commonly known as Goldreich's PRGs, and the ones instantiated on P_5 achieve the best possible locality and have received much attention. The existence of PRGs in $\mathtt{NC}^0$ (as low as $\mathtt{NC}^0_4$) was essentially confirmed in [AIK06] by showing the possibility of constructing low-stretch ($m = O(n)$) PRGs through compiling a moderately easy PRG using randomized encodings. Applebaum et al. in [AIK08] further gave the existence of PRGs with linear stretches by showing that the existence can be related to some hardness problems in, e.g., Max 3SAT (satisfiability problem).

Other than the initial motivation for the efficiency reasoning, i.e., realizing cryptographic primitives that can be evaluated in constant time by using polynomially many computing cores, PRGs in $\mathtt{NC}^0$ with polynomial stretches i.e., $m = \mathrm{poly}(n)$, have numerous more emerging theoretical applications, such as *secure computation with constant computational overhead* [IKOS08, ADI+17], *indistinguishability obfuscation (iO)* [LT17, GJLS20], *multiparty-computation (MPC)-friendly primitives* [MJSC16, GRR+16, ARS+15, CCF+18], and *cryptographic capsules* [BCG+17]. For example, a two-party computation protocol with constant computational overhead was proposed in [IKOS08], on the assumption of the existence of a PRG in $\mathtt{NC}^0$ with a polynomial stretch, together with an arbitrary oblivious transfer protocol. Thus, the existence of poly-stretch PRGs in $\mathtt{NC}^0$ is much attractive.

In [App13], Applebaum considered PRGs with long stretches and low localities and provided the existence of PRGs with linear stretches and weak PRGs with polynomial stretches, e.g., $m = n^s$ for some $s > 1$, with a distinguishing gap at most $1/n^s$. This

work was later strengthened in [ABR16] by showing a dichotomy of different predicates: all non-degenerate predicates yield small-bias generators with output length $m = n^s$ for $s < 1.25$ while degenerate predicates are not secure against linear distinguishers for most graphs. The stretch was later extended to $s < 1.5$ for the special case P_5 in [OW14].

The mentioned works above all focus on checking the existence of potential PRGs in $\mathtt{NC}^0$, establishing asymptotic security guarantees for them, and exploring appealing theoretical applications based on them; one main obstacle, however, for these advanced cryptographic primitives towards being practical comes from the lack of a stable understanding on the concrete security of these PRGs. In [CDM+18], Gouteau et al. first studied the concrete security of Goldreich's PRGs, especially the important instantiation on the P_5 predicate. Specifically, they developed a guess-and-determine-style attack and gave more fine-grained security guarantees for them. In the last part of their presentation at ASIACRYPT 2018, an open problem was raised:

"Can we improve the security bounds for P_5?"

In this paper we focus on this open problem and give an affirmative answer.

Before stating our main cryptanalytic methods and results, we first review the common cryptanalysis techniques against local PRGs.

1.1 Related Work in Cryptanalysis

The main cryptanalysis tools for local PRGs include myopic backtracking algorithms, linear cryptanalysis, and algebraic cryptanalysis.

1.1.1 Myopic Backtracking Algorithms

A Goldreich's PRG can be viewed as a random constraint satisfaction problem (CSP), thus the inversion of the PRG is equivalent to finding a planted solution for a CSP. Thus, some techniques and results from solving CSPs, e.g., 3-SAT, can be adopted. The so-called myopic backtracking method is such one commonly used algorithm. The basic idea is to gradually assign values to some input variables in every step based on newly read t constraints and previous observations until a contradiction is introduced. Every time when the new partial assignments contradict with some constraints, the algorithm backtracks to the latest assigned variable, flips the assigned value and continues the process. After sufficiently many steps, the algorithm will surely recover the secret.

Goldreich considered the myopic backtracking method on the proposed one-way function [Gol00], by reading one output bit at each step and computing all possible values of input bits which would produce the read output bits. The results show that the expected size of the possible values is exponentially large. Alekhnovich et al. further gave exponential lower bounds of the running time for myopic algorithms in [AHI06]. It showed that Goldreich's function is secure against the myopic backtracking algorithm when it is instantiated on a 3-ary predicate $P(x_1, x_2, x_3) = x_1 \oplus x_2 \oplus x_3$. Since a predicate should be non-linear (otherwise a system can be easily solved using Gaussian elimination), the predicate is extended to a more general case involving a non-linear term: $P_d(x_1, ..., x_d) = x_1 \oplus x_2 \oplus \cdots \oplus x_{d-2} \oplus x_{d-1}x_d$. They showed that for most d-ary predicates and some t, the expected success probability of the basic t-myopic algorithm (i.e., reading t constraints at each step) in inverting is $e^{-\Omega(n)}$. The results were verified over small Goldreich's PRGs using the SAT solver MiniSat.

1.1.2 Linear Cryptanalysis

Each constraint of a local PRG can be viewed as a linear equation with a noise, i.e., $y_i = \sum_{j \in \hat{\sigma}^i} x_j + e_i$, where $\hat{\sigma}^i$ is the subset of linear terms of the i-th subset σ^i, while e_i is a biased noise, whose distribution is determined by the chosen predicate. For the P_5 predicate, e_i has the distribution $P(e_i = 1) = 1/4$ and $P(e_i = 0) = 3/4$.

One simple way of linear cryptanalysis could be to find equations sharing the same linear variables and view the noises as independent and identically distributed variables. A majority rule can be applied on the noises and a corresponding value can be assigned to the linear part. Thus a linear equation is obtained and by exploring many such linear equations, the secret could be recovered by solving the derived linear system.

An improved version is to build enough noisy equations with two linear terms of the form $x_{\sigma^i} + x_{\sigma^j}(+e_i + e_j) = y_i + y_j$, by XORing a pair of equations sharing the other linear variables, and then apply some known algorithms, e.g., semidefinite programming [App16], to get a solution $\hat{x}$ satisfying a large fraction of the equations. This solution would be highly correlated with the true one and the system could be inverted with high probability based on it [BQ09], or at least a distinguishing attack is achieved.

Linear tests can be viewed as one special type of linear cryptanalysis, in which the adversary considers linear combinations of several output bits and investigates the bias of these combinations. It can result into a distinguishing attack if the bias is nonnegligible, which indicates that the local PRG is not fully random.

1.1.3 Algebraic Attacks

In an algebraic attack, the equation system is extended by, e.g., multiplying some equations with lower-degree terms until a solution is possible to be found by, e.g., linearization, Gaussian elimination or by computing a Gröbner basis of the expanded system. In [AL18], Applebaum and Lovett analyzed how the underlying predicate affects pseudorandomness using algebraic attacks and proved that besides high *resiliency* and high *algebraic degree*, the predicate must have high *rational degree* as well. The requirement is relevant to the criterion of high *algebraic immunity* on Boolean functions in stream ciphers to resist algebraic attacks. The paper also gave some advice on the choices of predicates in terms of these properties. Algebraic attacks based on linearization and Gröbner basis algorithms were further considered in [CDM+18], and some results on concrete choices of parameters were given.

1.2 Contributions

In this paper, we present new attacks which significantly improve the complexity of inverting the local PRGs instantiated on the P_5 predicate.

- Our first result is a novel guess-and-determine-style attack with much lower complexity than the results presented in [CDM+18]. We develop theoretical and also numerical analysis about the number of required guesses for various (n, s) parameters, where n and s denote the seed size and stretch, respectively, and experimentally verify the analysis for some small parameters.

This approach is basically a greedy method. We classify the equations occurring during the guess-and-determine process into three different classes, by the number of included monomials. The class of equations with two monomials are desired, as guessing variables occurring in these equations could introduce some "free" determined variables. We then design our guessing criterion over the equation classes, to generate as many "free" variables as possible when assuming for a limited number of guesses.

Our guess-and-determine attack reduces the asymptotic complexity from $\tilde{O}(2^{\frac{1}{2}n^{2-s}})$ in [CDM+18] to $\tilde{O}(2^{\frac{73}{288}n^{2-s}})$, thereby achieving a slightly less than square-root speed-up. Regarding the concrete complexity, we could solve all the challenge parameters suggested in [CDM+18] with complexity much below their claimed security levels. As shown in Table 1, our complexity gains (measured by the ratio of the two complexity numbers) range from 2^{23} to 2^{47} for parameters aiming for 80 bits of security, and from 2^{40} to 2^{77} for those aiming for 128 bits.

– Based on the guessing strategies proposed in the first attack, we further exploit the extremely sparse structure of P_5 and combine ideas from iterative decoding for solving a linear system. Our method differs from the classic iterative decoding or belief propagation, since the system in our case is quadratic and no a-priori information is available. We design new belief propagation rules for this specific setting. This is a novel method for solving a system of non-linear equations and we call it the *guess-and-decode* approach. The gain of it compared to the first attack comes from: 1) the better information extraction from the equations with a quadratic term (the guess-and-determine approach only exploits the generated linear equations); and 2) the soft probabilities used in iterative decoding (inverting a linear matrix in the guess-and-determine approach can be regarded as using "hard" (binary) values).

  Experimental results show that, with a smaller number of guesses, the resulting system has a good chance to be determined and then the secret could be fully recovered. As shown in Table 1, the new guess-and-decode approach could further significantly improve the guess-and-determine approach for all the challenge parameters proposed in [CDM+18]. For instance, the improvement factor is as large as 2^{22} for the parameter set $(4096, 1.295)$, and could be even larger for a parameter with a larger n value in our prediction. With this new method, the challenge parameter sets proposed in [CDM+18] for 128-bit security are insufficient even for providing security of 80 bits.

  We also suggest new challenge parameters to achieve 80-bit and 128-bit security levels for various seed sizes.

– Lastly, we extend the attacks to other promising predicates of the type of XOR-AND and XOR-MAJ, which are the two main types of predicates suggested for

Security Level	(n, s)	[CDM+18]	Sec. 3	Sec. 4
	Asymptotic complexity	$O(n^2 2^{\frac{1}{2}n^{2-s}})$	$O(n^2 2^{\frac{73}{288}n^{2-s}})$	$O(n^s 2^{\frac{73}{288}n^{2-s}})$
80 bits	$(512, 1.120)$	2^{91}	2^{44}	2^{52}
	$(1024, 1.215)$	2^{90}	2^{53}	2^{53}
	$(2048, 1.296)$	2^{91}	2^{68}	2^{57}
	$(4096, 1.361)$	2^{91}	2^{68}	2^{58}
128 bits	$(512, 1.048)$	2^{140}	2^{63}	2^{68}
	$(1024, 1.135)$	2^{140}	2^{75}	2^{72}
	$(2048, 1.222)$	2^{139}	2^{98}	2^{77}
	$(4096, 1.295)$	2^{140}	2^{100}	2^{78}

The column "Sec. 3' shows the complexity of the guess-and-determine attack and the column "Sec. 4" shows the complexity of the guess-and-decode attack.

Table 1: The Complexity Comparison of the Algorithms for Solving the Challenge Parameters Proposed in [CDM+18].

constructing local PRGs. We investigate their resistance against our attacks and give some initial sights into their possibly safe stretches.

Comparison of the Two Attacks. Both attacks follow a similar guessing phase which targets to obtain as many "free" variables as possible, but with slightly different goals and purposes: the first (guess-and-determine) attack targets to derive as many linear equations as possible and further recovers the secret by solving a linear sub-system of equations; while the second (guess-and-decode) aims to derive as many low-weight equations, no matter linear or nonlinear, and as much a-priori information as possible to be used for iterative decoding to recover the secret.

When applied to P_5, though it might be uncertain to tell which attack applies better for one specific system, the second attack is generally more powerful against stricter systems, e.g., systems with larger secret sizes n and smaller stretches s, since it can make use of more soft information. Thus, when choosing secure parameters for a local PRG over P_5, the second attack should always be considered. When applied to more general predicates, the first attack has more advantages when deriving a linear equation is easier, e.g., an XOR-AND predicate with a large locality; while the second attack performs better when the predicate has more complex forms of non-linear terms but with a relatively low weight, e.g., a low-weight XOR-MAJ predicate.

Discussions It is pointed out in [BCG+19] that "...it (building cryptographic capsule upon the group-based homomorphic secret sharing together with Goldreich's low-degree PRG in [BCG+17]) is entirely impractical: Goldreich's PRG requires very large seeds...", and the best-known instantiations of some recent attractive applications such as pseudorandom correlation generators [BCG+19, BCG+20] are based on computational assumptions such as LPN (learning parity with noise) and Ring-LPN, rather than on Goldreich's low-weight PRGs. Our novel attacks further diminish their practicality, though

their significance in theoretical cryptography is unaffected. We strongly recommend to instantiate a more complex local PRG with higher locality and/or more non-linear terms, if relatively high stretches and high security are required. Then, the research on the concrete security of the new PRGs should be renewed. However, in some special applications where drastic limits on the number of output bits are not an issue, while the significant advantages from the extremely low locality, multiplicative and overall complexity for deriving output bits are more desired, local PRGs instantiated on P_5 would probably still be considered. Our attacks provide better understanding of the concrete security of Goldreich's PRGs in a more fine-grained manner, and shed light on producing better cryptanalytical works on the MPC-friendly primitives with similar inherent structures.

1.3 Organization

The rest of this paper is organized as follows. We first give some preliminaries and briefly describe the guess-and-determine attack in [CDM+18] in Section 2. We then present our improved guess-and-determine attack in Section 3, describing how it works and providing theoretical analysis and experimental verification. In Section 4, we thoroughly describe the guess-and-decode attack and verify it by extensive experimental results. We extend the attacks to other promising predicates in Section 5 and lastly conclude the paper in Section 6.

2 Preliminaries

2.1 Notations

Let $GF(2)$ denote the finite field with two elements. Thus, the operation of addition "+" is equivalent to the XOR "$\oplus$" operation. Our paper focuses on Goldreich's PRGs instantiated on the P_5 predicate. Throughout the paper, we use $x \in \{0,1\}^n$ to denote the secret seed of size n and x_i $(1 \leq i \leq n)$ to denote the i-th bit of x. We use $y \in \{0,1\}^m$ to denote the output string of size m and $m = n^s$, where s is the expansion stretch. Each bit of y, denoted as y_i for $1 \leq i \leq m$, is derived by applying P_5 on a 5-tuple subset of the seed variables. These seed variables are indexed by a publicly known index subset $\sigma^i = [\sigma_1^i, \sigma_2^i, \sigma_3^i, \sigma_4^i, \sigma_5^i]$, where indices are distinct and randomly chosen. Thus, y_i is generated as below:

$$x_{\sigma_1^i} + x_{\sigma_2^i} + x_{\sigma_3^i} + x_{\sigma_4^i} x_{\sigma_5^i} = y_i.$$

We call such a relation an equation.

For a binary variable u, we use p_u^y or $P(u = y)$ to denote the probabilities of u being the value of y, where $y \in \{0,1\}$. The Log-Likelihood Ratio (LLR) value of u is defined as

$$L_u = \log \frac{P(u=0)}{P(u=1)}, \tag{1}$$

i.e., the logarithmic value of the ratio of probabilities of u being 0 and being 1. All the secret bits are assumed to be uniformly random distributed, thus the initial LLR values for them are all zero.

2.2 The Guess-and-Determine Attack in [CDM+18]

In a guess-and-determine attack, some well-chosen variables of a secret are guessed and more other variables could be correspondingly determined according to predefined relationships connected to these guessed variables. If the partial guessed variables are guessed correctly, the further determined variables would be correct as well. Thus, one hopes that by guessing a smaller number than the expected security level of variables, all the other variables can be determined directly or through some more complex method.

A guess-and-determine attack on Goldreich's PRGs is given in [CDM+18]. The basic idea is to guess enough secret variables and derive a system of linear equations involving the remaining variables and by solving this linear system, the secret is expected to be recovered.

When guessing a variable, those equations involving this guessed variable in the quadratic terms would become linear. The guessing strategy in [CDM+18] is then to always guess the variable which appears most often in the quadratic terms of the remaining quadratic equations, thus to obtain a locally optimal number of linear equations for each guess. This process is iteratively performed until enough linear equations are derived.

Besides, some "free" linear equations can be obtained before the guessing phase by XORing two equations sharing a same quadratic term, which is called a "*collision*". The average number of linear equations c derived in this way has been computed in [CDM+18] as below:

$$c = n^s - \binom{n}{2} + \binom{n}{2}\left(\frac{\binom{n}{2}-1}{\binom{n}{2}}\right)^{n^s}, \tag{2}$$

where the symbol $\binom{n}{i}, 0 \leq i \leq n$, denotes the binomial coefficient. The guessing process can be stopped once $n - c$ linear equations are obtained (with the number of guesses included). Suppose that after guessing ℓ variables, the stopping condition is satisfied. The algorithm will enumerate over all 2^ℓ possible assignments for these guessed variables, and for each assignment e, a distinct system of linear equations would be derived. Let A_e denote the matrix of this linear system whose rank could be equal to or smaller than n. The paper shows that when the rank of A_e is smaller than n, an invertible submatrix with fewer variables involved can almost always be extracted and thus a fraction of variables can always be recovered. The remaining variables can be easily obtained by injecting those already recovered ones.

In [CDM+18], an upper bound of the expected number of guesses, denoted as ℓ, is given as $\ell \leq \lfloor \frac{n^2}{2m} + 1 \rfloor$, and the asymptotic complexity is $O(n^2 2^{\frac{1}{2}n^{2-s}})$, where $O(n^2)$ is the asymptotic complexity for inverting a sparse matrix. The attack is experimentally verified and some challenge parameters under which the systems are resistant against the attack are suggested.

3 New Guess-and-Determine Cryptanalysis of Goldreich's PRGs with P_5

In this section, we give a new guess-and-determine attack on Goldreich's pseudorandom generators. The fundamental difference with the guess-and-determine attack in [CDM+18] is that the guessing process in our attack is "dynamic", by which we mean that the choice of a new variable to guess depends not only on previously guessed variables but also on their guessed values. The main observation for our attack is that some variables could be determined for free after having guessed some variables, and the goal of our attack is to exploit as many such "free" variables as possible. To achieve so, we first define three different equation classes, **Class I, II** and **III**, which include different forms of equations, and further design guessing strategies based on them.

3.1 Equation Classes and "Free" Variables

We categorize equations generated during the guessing process into three classes according to the number of terms.

Class I includes equations having no less than four terms, either quadratic or linear, with forms as below:

$$x_{\sigma_1^i} + x_{\sigma_2^i} + x_{\sigma_3^i} + x_{\sigma_4^i} x_{\sigma_5^i} = y_i, \tag{3}$$

$$x_{\sigma_1^j} + x_{\sigma_2^j} + x_{\sigma_3^j} + x_{\sigma_4^j} = y_j. \tag{4}$$

Equations with more than four terms, which can be generated from, e.g., XORing two equations sharing a same quadratic term, are also categorized into Class I. The equations with the form in (3) are the initially generated equations. If one variable in the quadratic term of such an equation, e.g., $x_{\sigma_4^i}$ or $x_{\sigma_5^i}$, is guessed as 1, the equation would be transformed to an equation in (4). While if the guessed value is 0 or a variable in the linear terms is guessed, we would get a different equation which is categorized into another class, Class II.

Class II includes those equations having three terms, either quadratic or linear, with forms as below:

$$x_{\sigma_1^i} + x_{\sigma_2^i} + x_{\sigma_3^i} x_{\sigma_4^i} = y_i, \tag{5}$$

$$x_{\sigma_1^j} + x_{\sigma_2^j} + x_{\sigma_3^j} = y_j. \tag{6}$$

As mentioned before, an equation with the form in (6) could be obtained if one variable in the quadratic term of an equation in (3) is guessed as 0. It can also be derived when one arbitrary variable in an equation in (4) is guessed (either 1 or 0), or if one variable in the quadratic term of an equation in (5) is guessed as 1. Equations with the form in

(5) can be derived by guessing one linear variable of an equation in (3).

Class III includes those equations having only two terms, either quadratic or linear, with forms as below:

$$x_{\sigma_1^i} + x_{\sigma_2^i} x_{\sigma_3^i} = y_i, \tag{7}$$

$$x_{\sigma_1^j} + x_{\sigma_2^j} = y_j. \tag{8}$$

If one linear term of an equation in (5) is guessed, it will be transformed to an equation with the form of (7). An equation with the form of (8) can be derived from: guessing one variable in the quadratic term in (5) as 0; or guessing one variable in (6) (either 1 or 0); or guessing one variable in the quadratic term in (7) as 1.

If we guess a variable which is involved in an equation in Class III, some "free" information could be obtained, which can happen in the following cases:

(1) For a linear equation $x_{\sigma_1^i} + x_{\sigma_2^i} = y_i$, when one variable, e.g., $x_{\sigma_1^i}$, is guessed, the other variable, $x_{\sigma_2^i}$ in this case, can be directly derived as $x_{\sigma_2^i} = x_{\sigma_1^i} + y_i$ for free;

(2) For a quadratic equation $x_{\sigma_1^i} + x_{\sigma_2^i} x_{\sigma_3^i} = y_i$, if the linear variable $x_{\sigma_1^i}$ is guessed to be different from y_i, $x_{\sigma_2^i}$ and $x_{\sigma_3^i}$ could be easily derived as $x_{\sigma_2^i} = 1, x_{\sigma_3^i} = 1$, thus two "free" variables are obtained;

(3) If any of the variables in the quadratic term is guessed as 0 in the above quadratic equation, the linear term is derived to be $x_{\sigma_1^i} = y_i$ for free;

(4) If $x_{\sigma_1^i}$ is guessed as the value of y_i in the above quadratic equation, an equation with only one quadratic term would be obtained, i.e., $x_{\sigma_2^i} x_{\sigma_3^i} = 0$. There are no "free" variables obtained at the moment. If at a future stage any one of the two variables, $x_{\sigma_2^i}$ or $x_{\sigma_3^i}$, is guessed to be 1, the other one can be derived as 0 for free.

Thus, the criterion for the guessing is always guessing variables in equations from Class III if it is not empty and thus obtaining "free" variables. Every time after guessing one variable, some equations in Class I and Class II could be transformed to equations in Class II or Class III. So it is highly likely that there always exist equations in Class III to explore for "free" variables. Furthermore, after plugging in the "free" variables, it could happen that more "free" variables would be derived. Thus, the required number of guesses can be largely reduced. That is the motivation of categorizing equations into three different classes and we next describe the new guess-and-determine attack designed over the defined equation classes.

3.2 Algorithm for the New Guess-and-Determine Attack

Algorithm 1 presents the general process of the proposed guess-and-determine attack. For convenience of description, we use the term *equation reduction* to denote the process of plugging in the value of a variable, either guessed or freely derived, and transforming

Algorithm 1 The new guess-and-determine attack

Input m quadratic equations derived from a length-n secret according to the P_5 predicate

Output the secret

1: Explore linear equations obtained through collisions
2: Set $t = 0$, back up the current system, mark all variables as "non-reversed"
3: Guess a variable and perform equation reduction following Algorithm 2, $t = t + 1$, back up the derived system
4: **if** no less than n linear equations are derived **then**
5: solve the linear system
6: **if** the recovered secret is correct **then return** the recovered secret
7: **else**
8: $t =$ `backtracking`() following Algorithm 3
9: go to step 4
10: **else**
11: go to step 3

the equations with the variable involved to ones with lower localities. Below we give the details of the attack.

Before guessing, as done in [CDM$^+$18], we first explore linear equations obtained through XORing two equations sharing a same quadratic term. The quadratic terms would be canceled out and only the linear terms of the two equations remain. If these two equations further share one or two linear terms, the shared variables should be canceled out as well. Thus, linear equations derived in this way could have two, four, or six variables, which are put into Class III, Class I and Class I, respectively. Note that when any two equations share the same variables in the linear and quadratic part, respectively, it already leads to a distinguishing attack. The expected number of linear equations derived in this way has been given in [CDM$^+$18].

After finding out the linear equations derived from collisions in the quadratic terms, we back up the current system and start the guessing process. We follow the rules in Algorithm 2 to choose variables for guessing. Specifically, we always choose a variable in an equation in Class III, since it is more likely to obtain "free" variables. Here we set the rule that when guessing a quadratic equation in Class II, we always guess the linear term as it can introduce more "free" variables on average. If Class III is empty, we guess a variable in an equation from a less attractive class, Class II, or Class I if Class II is also empty. As for choosing which variable to guess, we use the following criterion: we always choose the variable appearing the most number of times in the local class, e.g., Class III if a variable in an equation in Class III is guessed, since it could introduce more "free" variables or transform more equations; if all variables have the same occurrences locally, choose the variable which appears most often in the global system of equations, thus more equations would be transformed.

Every time when one variable is guessed, an equation involving this guessed variable would be transformed to either a linear equation in the same class or an equation (could be quadratic or linear) in the next class. The goal is to transform as many equations as possible to Class III such that when guessing one variable in Class III, it is more likely

Algorithm 2 Algorithm for guessing a variable

Input A system of equations

1: **if Class III** is not empty **then**
2: guess the variable appearing most often in **Class III**
3: plug in the guessed variable and perform equation reduction
4: continually plug in "free" variables and perform equation reduction, if there are any, until there are no more "free" variables
5: **else if Class II** is not empty **then**
6: guess the variable appearing most often in **Class II**
7: plug in the guessed variable and perform equation reduction
8: **else**
9: guess the variable appearing most often in **Class I**
10: plug in the guessed variable and perform equation reduction

Algorithm 3 Algorithm for `backtracking`

Input The whole systems of equations, an index t
Output an updated index t

1: delete the t-th back-up system
2: **if** the t-th guessed variable has been marked as "reversed" **then**
3: recover the variable as "non-reversed"
4: $t = t - 1$, go to step 1
5: **else**
6: reverse the guessed value of the t-th guess and mark the variable as "reversed"
7: perform equation reduction over the $(t-1)$-th back-up system with the reversed t-th guess, back up the derived system
8: return t

to get some "free" variables. After each guess and corresponding equation reduction, we should also plug in the "free" variables, if there are any, and perform equation reduction. This step might introduce more "free" variables and we continue this process until there are no further "free" variables.

Every time after performing equation reduction for a guessed and corresponding freely determined variables, we always back up the derived system, in case the guesses are not correct and at some future stage, tracing back is needed. If there are enough linear equations derived after some guesses, we would stop guessing and solve the linear system. By "enough" we use the condition in [CDM+18] for key recovery, i.e., the number of linear equations (including the guessed and freely determined variables and linear equations derived by finding collisions in quadratic terms) is not smaller than n. The rank of the matrix for the linear system does not have to be n, as it shows in [CDM+18] that an invertible subsystem with fewer variables involved can almost always be extracted and solved. The remaining small fraction of variables can be easily recovered by injecting those already recovered ones.

If the recovered secret is correct, which can be verified by checking if it can produce the same output sequence, the attack succeeds and stops. While if not, we need to trace

back following the `backtracking` algorithm given in Algorithm 3: either 1) reversing the guessed value of the last guessed variable if it has not been reversed before and perform equation reduction based on the last back-up system; or 2) tracing back more steps until to a guessed variable which is not reversed before. Since we have backed up in each stage, we can retrieve the desired system of equations, perform equation reduction over that system, and delete all subsequent back-ups when we trace back. If some conflicts happen during the reduction, for example, we have guessed a variable as 1 while some other equations give the information that this variable should be 0, we stop going into deeper and immediately trace back.

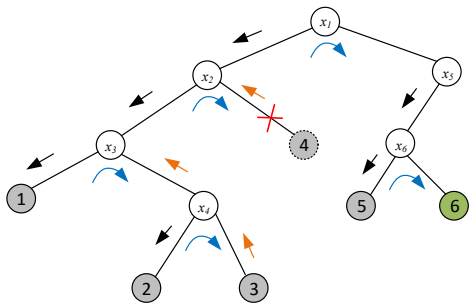

Figure 1: A simple illustration of the guessing process.

The path of the guessing is a binary tree while the tree is highly likely to be irregular, i.e., the depths for different paths could be different, but with small gaps. Figure 1 shows a simple illustration of the guessing process. The leaf nodes, which are color-filled, represent the process to solve a system of linear equations when enough linear equations are derived. The gray-filled ones indicate the wrong paths, i.e., some guesses are not correct, while the green-filled one is the correct one and the guessing stops at this node. Black arrows indicate normal guessing steps, while blue arrows indicate the processes of reversing the guessed values of the last guessed variables and orange arrows indicate the processes of tracing back. At the leaf node with index 4, some conflicts happened during the equation reduction, which indicates that some previous guessed values are not correct, e.g., x_1 is not correctly guessed in this case. So we need to reverse the guessed value of it and continue guessing. In this simple example, we have visited six nodes, meaning that we have guessed six variables, and solved four systems of linear equations until we find the correct path. We could then get the correct values of the variables x_1, x_5, x_6 as (1, 0, 1) (we denote the left direction as guessing 0 and right direction as guessing 1), and other variables can be recovered by solving the derived linear system at the leaf node with index 6.

In practice, guessing 0 or 1 has a negligible impact on the attack complexity for the wrong paths, as both values would be tried. However, for the correct path, for an equation in Class III, e.g., $x_{\sigma_1^i} + x_{\sigma_2^i} x_{\sigma_3^i} = y_i$, $x_{\sigma_1^i}$ is more likely to equal y_i than the complement. Thus in practice when guessing the linear term in a quadratic equation in Class III, one can first guess it to be equal to the value of the equation.

Avalanche Effect. After guessing a certain large number of variables, an avalanche

effect will happen, namely that "free" variables recursively introduce more "free" variables and all variables are immediately determined. It happens at different stages, i.e., different numbers of variables are guessed, for different systems, but generally earlier for systems with smaller n while relatively larger s.

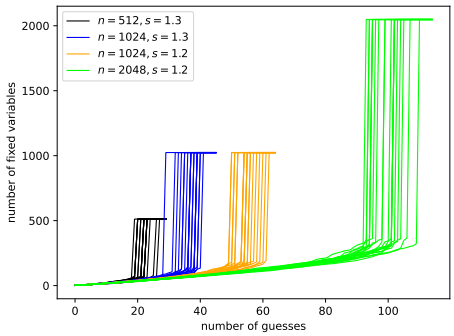

Figure 2: The number of fixed variables under different numbers of guesses.

Figure 2 illustrates the experimental results for the number of variables that will be known under different numbers of guesses for four different (n, s) pairs, each with 20 independent instances. One can see that for each system, the avalanche effect happens at some certain point and all variables are immediately determined. In this case, solving a linear system of equations is not needed and the complexity is denoted as 2^{ℓ_A}, where ℓ_A is the number of guesses under which the avalanche effect appears.

We will not consider the avalanche effect when deriving the asymptotic complexity, as for relatively safer systems, guessing and solving linear systems has more advantages, i.e., lower complexity. However, when experimentally breaking the challenging systems suggested in [CDM+18] in Section 3.4, the avalanche effect is also considered.

3.3 Theoretical analysis

In this subsection, we perform theoretical analysis of the attack. We will first present some propositions which help to gradually derive the required number of guesses and attack complexity. Note that we are considering the stretch regime $1 < s < 1.5$.

For convenience, we further categorize the intermediate equations into smaller subclasses and denote them with some notations. Specifically, we use C_{11}, C_{12} to denote the quadratic (i.e., equations in (3)) and linear equations (i.e., equations in (4)) in Class I, and similarly use C_{21}, C_{22} (and C_{31}, C_{32}) to denote the quadratic and linear equations in Class II (and Class III), respectively. Besides, we use C_{33} to denote the class of equations of the form $x_{\sigma_1^i} x_{\sigma_2^i} = 0$ and C_{null} to denote the equations which become empty, i.e., in which all terms are known (either guessed, determined or canceled).

Suppose that γ out of the n secret variables have been known, either from guessing or determining, which indicates that γ times of equation reduction have been performed. The equations are transformed into other classes with certain probabilities independently, which can be approximated as Bernoulli processes. Denote $\xi = n - \gamma$. We first

present some propositions about how the equation reduction develops.

Proposition 1. For an intermediate equation E generated in the proposed attack with $k \in \{0,1,2,3\}$ linear variables and $q \in \{0,2\}$ quadratic variables sampled from ξ secret variables, the number of different possible forms is $\binom{\xi}{k+q} \cdot \binom{k+q}{k}$.

Proof. We can first choose $k+q$ variables from the ξ variables, which has $\binom{\xi}{k+q}$ different combinations. From the $k+q$ chosen variables we select k of them for the linear terms and the rest for the quadratic term. As the order of the linear variables (and similarly of the quadratic variables) does not matter, we have $\binom{k+q}{k}$ possible combinations. Thus, in total we will have $\binom{\xi}{k+q} \cdot \binom{k+q}{k}$ different possible combinations for E. □

Proposition 2. After γ variables have been fixed, the average numbers of equations in C_{11}, C_{21}, C_{31}, C_{33} can be approximated as $\frac{(n-\gamma)^5}{n^{5-s}}$, $\frac{3\gamma(n-\gamma)^4}{n^{5-s}}$, $\frac{3\gamma^2(n-\gamma)^3}{n^{5-s}}$, $\frac{\gamma^3(n-\gamma)^2}{2n^{5-s}}$, respectively.

Proof. We take the number of equations in C_{21} as an example. An equation is transformed into C_{21} only when (any) one linear variable in this equation is fixed, i.e., among the γ known variables, and the remaining four are not fixed, i.e., among the $n-\gamma$ unknown variables. According to Proposition 1, the probability of such event, denoted p_{21}^{γ}, can be computed as:

$$\begin{aligned} p_{21}^{\gamma} &= \frac{\binom{\gamma}{1} \cdot \binom{n-\gamma}{4} \cdot \binom{4}{2}}{\binom{n}{5} \cdot \binom{5}{3}} \\ &= \frac{3\gamma(n-\gamma)(n-\gamma-1)(n-\gamma-2)(n-\gamma-3)}{n(n-1)(n-2)(n-3)(n-4)} \\ &\approx \frac{3\gamma(n-\gamma)^4}{n^5}. \end{aligned}$$

The approximation has a small error in $o(\frac{1}{n})$, e.g., when $n = 2048, \gamma = 100$, the approximation error is 0.0002. When n goes larger, the approximation error will be even smaller. The average number of equations that will be transformed into C_{21} is then computed as $p_{21}^{\gamma} \cdot m = \frac{3\gamma(n-\gamma)^4}{n^{5-s}}$. Similarly, the probabilities of an equation staying in C_{11} and being transformed into C_{31} are computed as $p_{11}^{\gamma} = \frac{\binom{n-\gamma}{5}}{\binom{n}{5}} \approx \frac{(n-\gamma)^5}{n^5}$ and $p_{31}^{\gamma} = \frac{\binom{\gamma}{2} \cdot \binom{n-\gamma}{3} \cdot \binom{3}{1}}{\binom{n}{5} \cdot \binom{5}{3}} \approx \frac{3\gamma^2(n-\gamma)^3}{n^5}$ with approximation errors in $o(1)$ and $o(\frac{1}{n^2})$, respectively. For p_{33}^{γ}, we should additionally multiply with 0.5, since the XOR sum of the three guessed linear variables should be constrained to be equal to the value of the equation, thus the probability is computed as $p_{33}^{\gamma} = \frac{0.5 \cdot \binom{\gamma}{3} \cdot \binom{n-\gamma}{2}}{\binom{n}{5} \cdot \binom{5}{3}} \approx \frac{\gamma^3(n-\gamma)^2}{2n^5}$ with an approximation error in $o(\frac{1}{n^3})$. □

Actually, we should multiply the probabilities with $m - N_{\text{null}}^{\gamma}$ to derive the average numbers of equations in Proposition 2. However, N_{null}^{γ} is small, which we will show below, and the results of these probabilities multiplying N_{null}^{γ} will be much smaller than 1 thus can be ignored.

Every time when guessing a variable and deriving a "free" variable in Class III, the equation will become empty. On the other hand, an arbitrary equation turns empty with a very small probability, similarly to p_{33}^{γ}. Thus we have the following conjecture.

Conjecture 1. The number of empty equations before the avalanche effect happens is close to, maybe slightly more than, the number of guessed variables.

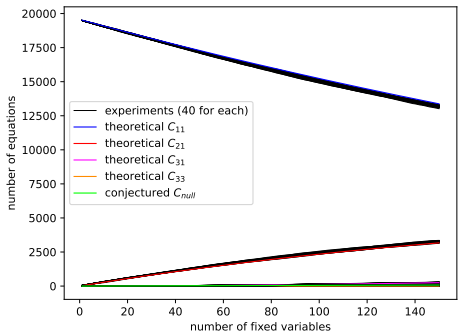

Figure 3: The number of different types of equations corresponding to different numbers of fixed variables ($n = 2048, s = 1.296$).

Figure 3 illustrates some experimental results of the numbers of different types of equations under different numbers of fixed variables. One can see that the theoretical analysis matches well with the experimental results. Figure 4 presents a clearer illustration for C_{31}, C_{33}, and C_{null}.

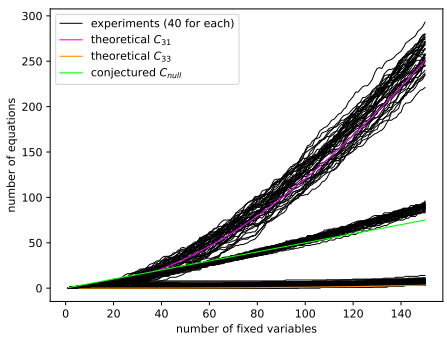

Figure 4: A clearer illustration of C_{31}, C_{33}, and C_{null} ($n = 2048, s = 1.296$).

Actually, we can also use the similar method as in Proposition 2 to derive the average number of linear equations in each class. For example, an equation will be transformed into a linear equation in Class III if: 1) one variable in the quadratic term is fixed as 0, and one more linear variable is fixed; 2) two variables in the quadratic and one

more linear variable are fixed; and 3) one variable in the quadratic term is fixed as 1 and two more linear variables are fixed. Thus the probability that an equation will be transformed into a linear equation in Class III is computed as:

$$p_{32}^{\gamma} = \frac{3\gamma^2(n-\gamma)^2(n+\gamma)}{n^5}. \tag{9}$$

One can see that one more assumption is needed when computing the numbers of linear equations: a variable in a quadratic term is guessed as 0 or 1 with probability 0.5. This requires more data in experiments to get the same statistical results. Besides, as we always first guess a linear equation in Class III, the number of linear equation in Class III gets more complex. Thus we will use the numbers of quadratic equations to derive the average number of linear equations and the attack complexity.

Proposition 3. After γ ($\gamma \ll n$) variables have been fixed, fixing one more variable (either from guessing or determining) will introduce roughly $\frac{2n^s}{n-\gamma}$ more linear equations.

Proof. We use $N_{\text{class}}^{\gamma}$ to denote the average number of equations in class $\in \{\text{null}, 11, 12, 21, 22, 31, 32, 33\}$ after knowing γ variables. One can easily get that $\sum_{\text{class}} N_{\text{class}}^{\gamma} = n^s$ always holds for any γ. We also use $N_{\text{linear}}^{\gamma}$ to denote the total number of linear equations after knowing γ variables, then $N_{\text{linear}}^{\gamma} = N_{12}^{\gamma} + N_{22}^{\gamma} + N_{32}^{\gamma} + \gamma$.

Now suppose a $(\gamma+1)$-th variable, say x_σ, will be fixed, either by guessing or determining. We assume that x_σ is fixed as 0 or 1 with probability 0.5. We below consider different cases that may happen to equations in each class, and Proposition 1 will be frequently used.

Let us first consider equations in C_{11}. For an equation E_{11} in C_{11}, the variables in this equation can choose from $n - \gamma = \xi$ variables. Then the number of possible combinations of E_{11} is computed as $\binom{\xi}{5} \cdot \binom{5}{3}$. There are several different cases that may happen to E_{11} as below.

- x_σ does not appear in E_{11} and E_{11} will still stay in C_{11}. E_{11} could have $\binom{\xi-1}{5} \cdot \binom{5}{3}$ different possible forms as the five variables are chosen from the remaining $\xi - 1$ variables. Thus, the probability of such case is computed as $\frac{\binom{\xi-1}{5} \cdot \binom{5}{3}}{\binom{\xi}{5} \cdot \binom{5}{3}} = \frac{\xi-5}{\xi}$.
- x_σ appears as one linear term in E_{11} and E_{11} will be transformed into C_{21}. The other four variables are chosen from the remaining $\xi-1$ variables and have $\binom{\xi-1}{4} \cdot \binom{4}{2}$ different possible combinations. Thus the probability of such case is computed as $\frac{\binom{\xi-1}{4} \cdot \binom{4}{2}}{\binom{\xi}{5} \cdot \binom{5}{3}} = \frac{3}{\xi}$
- x_σ appears as one quadratic term and two further cases can happen.
  - x_σ is fixed as 0 and E_{11} will be transformed into C_{22}. The number of possible combinations of E_{11} is $0.5 \cdot \binom{\xi-1}{4} \cdot \binom{4}{3}$. The probability of such case is then computed as $\frac{0.5 \cdot \binom{\xi-1}{4} \cdot \binom{4}{3}}{\binom{\xi}{5} \cdot \binom{5}{3}} = \frac{1}{\xi}$;
  - x_σ is fixed as 1 and E_{11} will be transformed into C_{12}. The probability can be similarly computed as $\frac{1}{\xi}$ as the previous case.

One can see that the probabilities of these cases sum to be one. Similarly, we can derive the possible transforms of an equation in other classes when x_σ is fixed. Figure 5 illustrates how the equations can be transformed and the corresponding occurring probabilities. We mention that when a "free" variable is derived from an equation in class C_{31}, C_{32} or C_{33}, this equation will become empty and be categorized into C_{null}.

Thus we can get the average number of equations in each subclass after fixing $\gamma+1$ variables as below:

$$
\begin{aligned}
N_{11}^{\gamma+1} &= N_{11}^{\gamma} \cdot \frac{(\xi-5)}{\xi}, \\
N_{12}^{\gamma+1} &= N_{11}^{\gamma} \cdot \frac{1}{\xi} + N_{12}^{\gamma} \cdot \frac{(\xi-4)}{\xi}, \\
N_{21}^{\gamma+1} &= N_{11}^{\gamma} \cdot \frac{3}{\xi} + N_{21}^{\gamma} \cdot \frac{(\xi-4)}{\xi}, \\
N_{22}^{\gamma+1} &= N_{11}^{\gamma} \cdot \frac{1}{\xi} + N_{12}^{\gamma} \cdot \frac{4}{\xi} + N_{21}^{\gamma} \cdot \frac{1}{\xi} + N_{22}^{\gamma} \cdot \frac{(\xi-3)}{\xi}, \\
N_{31}^{\gamma+1} &= N_{21}^{\gamma} \cdot \frac{2}{\xi} + N_{31}^{\gamma} \cdot \frac{(\xi-3)}{\xi}, \\
N_{32}^{\gamma+1} &= N_{21}^{\gamma} \cdot \frac{1}{\xi} + N_{22}^{\gamma} \cdot \frac{3}{\xi} + N_{31}^{\gamma} \cdot \frac{1}{\xi} + N_{32}^{\gamma} \cdot \frac{\xi-2}{\xi}, \\
N_{33}^{\gamma+1} &= N_{31}^{\gamma} \cdot \frac{1}{2\xi} + N_{33}^{\gamma} \cdot \frac{\xi-2}{\xi}, \\
N_{\text{null}}^{\gamma+1} &= N_{\text{null}}^{\gamma} + N_{31}^{\gamma} \cdot \frac{3}{2\xi} + N_{32}^{\gamma} \cdot \frac{2}{\xi} + N_{33}^{\gamma} \cdot \frac{2}{\xi}, \\
N_{\text{free}}^{\gamma+1} &= N_{31}^{\gamma} \cdot \frac{2}{\xi} + N_{32}^{\gamma} \cdot \frac{2}{\xi} + N_{33}^{\gamma} \cdot \frac{1}{\xi},
\end{aligned}
$$

where $N_{\text{free}}^{\gamma+1}$ is the number of additional "free" variables we can get when fixing a $(\gamma+1)$-th variable (not the total number of all "free" variables). These "free" variables should be recursively plugged into the system for equation reduction, but here we leave them as they are, since our focus is to derive the increased number of linear equations when one more variable is fixed. Thus after fixing $\gamma+1$ variables, the number of linear equations can be derived as below:

$$
\begin{aligned}
N_{\text{linear}}^{\gamma+1} &= N_{12}^{\gamma+1} + N_{22}^{\gamma+1} + N_{32}^{\gamma+1} + N_{\text{free}}^{\gamma+1} + \gamma + 1 \\
&= N_{11}^{\gamma} \cdot \frac{2}{\xi} + N_{21}^{\gamma} \cdot \frac{2}{\xi} + N_{31}^{\gamma} \cdot \frac{3}{\xi} + N_{33}^{\gamma} \cdot \frac{1}{\xi} + N_{12}^{\gamma} + N_{22}^{\gamma} + N_{32}^{\gamma} + \gamma + 1.
\end{aligned}
$$

Since $\sum_{\text{class}} N_{\text{class}}^{\gamma} = n^s$, we can get $N_{11}^{\gamma} + N_{21}^{\gamma} + N_{31}^{\gamma} + N_{33}^{\gamma} = n^s + \gamma - N_{\text{linear}}^{\gamma} - N_{\text{null}}^{\gamma}$. Then,

$$
\begin{aligned}
N_{\text{linear}}^{\gamma+1} &= \frac{2}{\xi} \cdot (n^s + \gamma - N_{\text{linear}}^{\gamma} - N_{\text{null}}^{\gamma}) + \frac{1}{\xi} \cdot (N_{31}^{\gamma} - N_{33}^{\gamma}) + N_{\text{linear}}^{\gamma} + 1 \\
&= \frac{2n^s}{\xi} + N_{\text{linear}}^{\gamma} + \frac{2\gamma}{\xi} - \frac{2}{\xi} N_{\text{linear}}^{\gamma} - \frac{2}{\xi} N_{\text{null}}^{\gamma} - \frac{1}{\xi} N_{33}^{\gamma} + \frac{1}{\xi} N_{31}^{\gamma} + 1.
\end{aligned}
$$

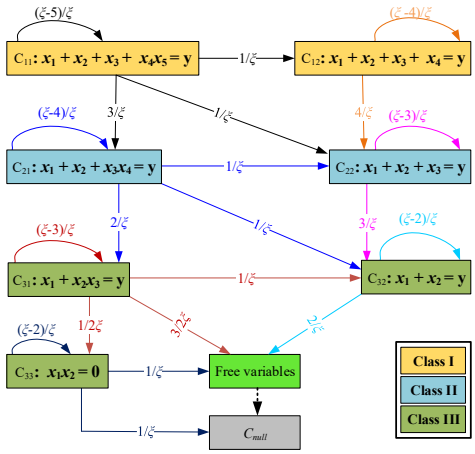

Figure 5: The equation reduction and occurring probabilities when fixing one variable.

Since γ is much smaller than n, we have $1 \geq \lfloor 1 + \frac{2\gamma}{\xi} - \frac{2}{\xi} \cdot N^{\gamma}_{\text{linear}} \rfloor \geq -1$. Besides, the last parts of $N^{\gamma+1}_{\text{linear}}$, i.e., $-\frac{2}{\xi}N^{\gamma}_{\text{null}} - \frac{1}{\xi}N^{\gamma}_{33} + \frac{1}{\xi}N^{\gamma}_{31}$, is in $O(\frac{1}{n})$ according to Proposition 2 and Conjecture 1 and can be neglected. Thus we have:

$$N^{\gamma+1}_{\text{linear}} \approx \frac{2n^s}{n-\gamma} + N^{\gamma}_{\text{linear}},$$

and $\frac{2n^s}{n-\gamma}$ more linear equations are introduced. □

Actually, Figure 5 represents a Markov chain and one can also use its transition matrix to derive the same results.

Proposition 4. After γ variables have been fixed, the average number of linear equations is around $2m \cdot \ln \frac{n}{n-\gamma}$.

Proof. According to Proposition 3, we know that the number of linear equations after γ variables having been fixed can be computed as

$$\begin{aligned} N^{\gamma}_{\text{linear}} &= \frac{2m}{n-\gamma+1} + N^{\gamma-1}_{\text{linear}} \\ &= \frac{2m}{n-\gamma+1} + \frac{2m}{n-\gamma+2} + \cdots + \frac{2m}{n-1} + \frac{2m}{n} \\ &= 2m \cdot (\frac{1}{n-\gamma+1} + \frac{1}{n-\gamma+2} + \cdots + \frac{1}{n-1} + \frac{1}{n}). \end{aligned}$$

We can use the approximation of harmonic series to help derive the result. The harmonic series $H_z = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{z}$ can be approximated using the following expression:

$$H_z \approx \ln(z) + \gamma + \frac{1}{2z} - \frac{1}{12z^2},$$

where $\gamma = 0.5772156649$ is known as the Euler-Mascheroni constant. The approximation error is small, e.g., when $z = 15$, the approximation error is 0.00000018, and when z is larger, the error is even smaller.

Thus $N^{\gamma}_{\text{linear}}$ can be further derived as:

$$\begin{aligned} N^{\gamma}_{\text{linear}} &= 2m \cdot (H_n - H_{n-\gamma}) \\ &= 2m \cdot \left(\ln \frac{n}{n-\gamma} - \frac{\gamma}{2n(n-\gamma)} - \frac{\gamma(2n-\gamma)}{12n^2(n-\gamma)^2} \right). \end{aligned}$$

We can approximate $N^{\gamma}_{\text{linear}}$ as below with an error in $O(\frac{1}{n^{2-s}})$:

$$N^{\gamma}_{\text{linear}} \approx 2m \cdot \ln \frac{n}{n-\gamma}. \tag{10}$$

□

However, we cannot get a closed form of the required number of guesses to achieve n linear equations according to (10), thus we further approximate $\frac{2n^s}{n-\gamma}$ in Proposition 4 as $\frac{2n^s}{n} = 2n^{s-1}$ with an approximation error in $O(\frac{1}{n^{2-s}})$ and use it to derive the average required number of guesses and asymptotic complexity. The approximation will introduce a larger error, however, when n is large, the error can still be ignored. For example, when $n = 2048, \gamma = 100, s = 1.296$, the difference of the two approximations $|\frac{2n^s}{n-\gamma} - 2n^{s-1}|$ is 0.9, less than one equation. We call this approximation as *Approximation 2* and the approximation in Proposition 3 as *Approximation 1*.

Figure 6 and Figure 7 present the number of linear equations one can get for each fixed variable and in total, respectively. Our two theoretical approximations match well with the experimental results. When the number of fixed variables goes larger, the total number of linear equations under *Approximation 1* becomes slightly higher than the experimental results, since each guess in C_{32} will eliminate one linear equation in the experiments. Next we will use *Approximation 2* to derive the average number of required guesses.

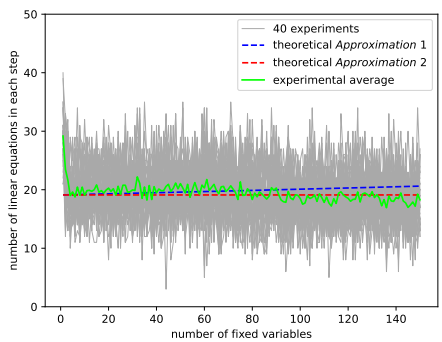

Figure 6: The number of linear equations derived for each fixed variable ($n = 2048, s = 1.296$).

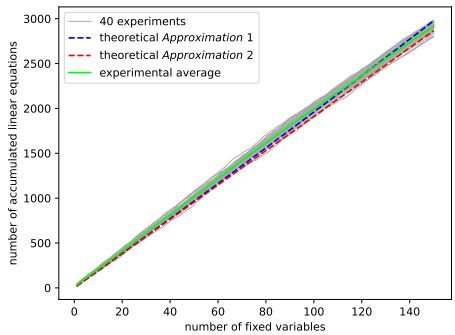

Figure 7: The number of accumulated linear equations under different number of fixed variables ($n = 2048, s = 1.296$).

Proposition 5. When guessing one variable from an equation in Class III, one can get one "free" variable on average.

Proof. There are two different cases in terms of guessing and obtaining "free" variables in Class III:

(1) **Guessing a variable in a linear equation.** For a linear equation, if one variable is guessed, the other variable could be immediately determined for free. Thus one "free" variable can always be obtained.

(2) **Guessing the linear variable in a quadratic equation.** For a quadratic equation $x_{\sigma_1^i} + x_{\sigma_2^i} x_{\sigma_3^i} = y_i$, if $x_{\sigma_1^i}$ is guessed as $x_{\sigma_1^i} = y_i \oplus 1$, we could get two "free" variables, i.e., $x_{\sigma_2^i} = 1, x_{\sigma_3^i} = 1$; while if it is guessed as y_i, no "free" variables could be obtained from the quadratic equation $x_{\sigma_2^i} x_{\sigma_3^i} = 0$ for the moment. If we assume $x_{\sigma_1^i}$ is guessed randomly, one "free" variable on average can be obtained.

We mention that in the second case, $x_{\sigma_1^i}$ is more likely to equal y_i for the correct path: with probability of 0.75, instead of 0.5, and we would get less than one "free" variable. However, the correct path only happens once and it does not affect the average complexity. Thus we can get one "free" variable on average for guessing one variable in Class III. □

Proposition 6. When guessing ℓ variables, where ℓ is relatively large but below the number of guesses under which the avalanche effect happens, $2\ell - \delta(n, s)$ variables on average will be known, where $\delta(n, s)$ is a function of n and s expressed as $\delta(n, s) = \frac{n^{2-s}}{144} + \frac{4}{3}$.

Proof. We assume that Class II and Class III are empty when guessing the first variable and if they are not, we have more advantages. There are three cases of the guess-and-determine process:

- *Case I*: for the first guess, it is impossible to get any free variable, since one equation can be at most transformed into Class II with only one guess;

- *Case II*: Class III is empty and one has to guess one variable from an equation in Class II, and gets one equation in Class III. One "free" variable will be introduced on average if one more variable in this equation is guessed according to Proposition 5. Thus, in this case, by guessing two variables, three variables will be fixed on average.

- *Case III*: there exist equations in Class III and one gets one "free" variable on average for each guess according to Proposition 5.

After a certain number of guesses, there will always exist equations in Class III such that one can keep guessing variables from this class. Suppose that *Case II* happened t times before such situation happens, then $l_0 = 2t$ variables were guessed and around $\gamma_0 = 1.5l_0$ variables would be fixed from *Case II*. *Case III* will then happen $l - l_0 - 1$ times and $2(l - l_0 - 1)$ variables will be fixed on average. Thus, the total number of variables that will be fixed is computed as $2(l - l_0 - 1) + \gamma_0 + 1 = 2\ell - \frac{1}{3}\gamma_0 - 1$.

Next, we will show when there will always exist equations in Class III. The average number of equations in Class III after γ variables are fixed can be computed according to Proposition 2 and (9) as below:

$$N_3^\gamma = (p_{31}^\gamma + p_{32}^\gamma) \cdot m = \frac{6\gamma^2(n-\gamma)^2}{n^{4-s}}. \tag{11}$$

To be more accurate, $N_3^\gamma = (p_{31}^\gamma + p_{32}^\gamma) \cdot (m - N_{\text{null}}^\gamma)$. At this early stage, the empty equations are mainly from guessing and determining variables in Class III, while the probability that an arbitrary equation turns empty can be neglected. Thus N_{null}^γ will be smaller than $\frac{\gamma}{2}$, and $(p_{31}^\gamma + p_{32}^\gamma) \cdot N_{\text{null}}^\gamma$ is much smaller than 1 and can be ignored. We define a function as below,

$$\begin{aligned} f(\gamma) &= N_3^\gamma - \frac{\gamma - \gamma_0 - 1}{2} - 1 \\ &= \frac{6\gamma^2(n-\gamma)^2}{n^{4-s}} - \frac{\gamma - \gamma_0 - 1}{2} - 1. \end{aligned}$$

We hope that the function keeps being non-negative for $\gamma, \gamma + 2, \gamma + 4...$, so that in Class III, the generated equations are always at least one more than the disappeared equations. Then there will always exist equations in Class III. First we note that when about $\gamma \geq \frac{n^{2-s}}{24}$, $f(\gamma)$ would be monotone increasing based on its derivative. If $f(\gamma) \geq 0$ when $\gamma = \frac{n^{2-s}}{24}$, the function will keep being non-negative for $\gamma > \frac{n^{2-s}}{24}$. Thus, we plug $\gamma = \frac{n^{2-s}}{24}$ into $f(\gamma)$ and keep it be non-negative, and we will get:

$$\gamma_0 \geq \frac{n^{2-s}}{48} + 1. \tag{12}$$

If we set $\gamma_0 = \frac{n^{2-s}}{48} + 1$, the total variables that can be fixed is $2\ell - \frac{1}{3}\gamma_0 - 1 \approx 2l - \frac{n^{2-s}}{144} - \frac{4}{3}$. □

Figure 8 illustrates the number of variables that will be known under different numbers of guesses before the avalanche effect happens. We can see that despite some small

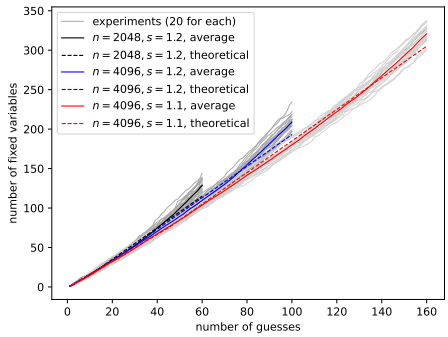

Figure 8: The number of fixed variables corresponding to different number of guesses before the avalanche effect happens.

deviations, the theoretical analysis matches well with the experimental results, especially in the initial phase of the guessing. When guessing slightly more variables, the experimental results get better than the theoretical analysis, since more than one "free" variable can be introduced when there are many equations in Class III.

Proposition 7. (**Number of guesses**). For a local PRG instance with n variables, n^s equations and c collisions, the average number of guesses required to build n linear equations can be approximated as:

$$\ell = \left\lceil \frac{n-c}{4n^{s-1}+2} + \frac{n^{2-s}}{288} + \frac{2}{3} \right\rceil. \tag{13}$$

Proof. If ℓ variables are guessed, $2\ell - \delta(n,s)$ variables will be fixed according to Proposition 6. Thus the number of linear equations that can be derived is computed as:

$$\big(2\ell - \delta(n,s)\big) \cdot 2n^{s-1} + c + 2\ell - \delta(n,s). \tag{14}$$

We use the same stop condition as that in [CDM+18], i.e., the number of linear equations is not smaller than n. Suppose that the condition is satisfied for the first time after guessing ℓ variables, we will then get the value of $\ell = \left\lceil \frac{n-c}{4n^{s-1}+2} + \frac{n^{2-s}}{288} + \frac{2}{3} \right\rceil$. □

The required number of guesses would actually be smaller than what is given in Proposition 7, as more than one "free" variable on average can be introduced in the later phase of the guessing. Now we can derive the complexity as below.

Storage Complexity. We need to back up several intermediate systems generated during the guessing process. The number of nodes with back-up is the depth of a guessing path, i.e., the number of guesses ℓ. Thus the storage complexity is $O(\ell \cdot m)$. As $\ell = \left\lceil \frac{n-c}{4n^{s-1}+2} + \frac{n^{2-s}}{288} + \frac{2}{3} \right\rceil$ according to Proposition 7, and $c \ll n$, the storage complexity is further derived as $O(n^2)$.

Time Complexity. When selecting a variable to guess, we need to go through all the equations in a local class, e.g., Class III, or Class II if Class III is empty, and choose the variable which appears most often. If these variables have the same occurrences in the local classes, we choose the one among them which appears most often in the global system. We at most need to go through all the equations and the complexity is upper bounded by $O(m)$. We could further reduce the complexity by keeping a global list recording the occurrence of each variable and updating it when performing equation reduction. Thus, the global maximum value can be found in complexity $O(n)$. The total number of selections is $O(2^\ell)$ and thus the complexity for selecting variables is $O(2^\ell n)$, i.e., $O(n2^{\left\lceil \frac{n-c}{4n^{s-1}+2}+\frac{n^{2-s}}{288}+\frac{2}{3}\right\rceil})$ according to Proposition 7. The asymptotic complexity of selection can be further expressed as $O(n2^{\frac{73}{288}n^{2-s}})$.

The largest computation overhead lies in solving linear systems, for which the cost is dominated by inverting a matrix. We use the same estimation of time complexity as that in [CDM+18] for inverting a sparse matrix, which is $O(n^2)$. Thus the total time complexity is dominated by $O(2^\ell \cdot n^2)$, i.e., $O(n^2 2^{\frac{73}{288}n^{2-s}})$ according to (13), which is much improved than the time complexity $O(n^2 2^{\frac{1}{2}n^{2-s}})$ in [CDM+18].

We mention that we have computed the complexity using the average number of required guesses, i.e., $O(n^2 \cdot 2^{E(\ell)})$ where $E()$ denotes the average operation, and 50% of the cases have complexities smaller than that. The complexity in average of the full algorithm, i.e., $O(E(n^2 \cdot 2^\ell))$, could be a more relevant metric on the security argument. Actually, we have verified these two metrics with our experimental results and observed very small gaps, but the theoretical result of it can still be interesting. We leave that for future work.

Lemma 1. The asymptotic complexity of the proposed guess-and-determine attack is

$$O(n^2 2^{\frac{73}{288}n^{2-s}}).$$

3.4 Experimental Verification

In this section, we first experimentally verify the attack and the theoretical analysis, then break the candidate non-vulnerable parameters suggested in [CDM+18]. We first use a simpler way with much less complexity to verify the attack and later show that the results derived in this way match well with those for practically recovering the secret. In this simple verification, we test the required number of guesses to collect enough linear equations, i.e., the number of linear equations is not smaller than n. We iteratively select a variable using the criterion described above and guess it randomly until the condition is satisfied. We run 1040 instances for each (n, s) pair and get the average number of guesses, whose distribution shows a low variance. The theoretical results computed according to Proposition 7 and experimental results of the average required number of guesses are shown in Table 2. The experimental results are better than the theoretical ones, just as happened in [CDM+18]. Compared to the experimental results in [CDM+18], the proposed attack requires fewer guesses.

We further implement the proposed attack to actually recover the secret as described in Algorithm 1. The algorithm will not terminate until the guessing values for the guessed variables are correct when enough linear equations are derived. We record the number

n	256			512			1024			2048			4096		
	theo.	exp.	[CDM+18]	theo.	exp.	[CDM+18]	theo.	exp.	[CDM+18]	theo.	exp.	[CDM+18]	theo.	exp.	[CDM+18]
$s = 1.45$	3	3.6	4	5	4.9	6	7	7.0	9	10	10.1	14	15	14.9	21
$s = 1.4$	6	5.7	6	9	8.4	11	13	12.7	17	20	19.3	27	31	30.0	44
$s = 1.3$	11	10.4	20	18	16.7	23	30	27.1	39	49	44.3	65	81	72.6	110

Columns of theo. and exp. denote the theoretical and experimental results, respectively.

Table 2: The Average Number of Guesses Required to Achieve n Linear Equations.

of guesses for the correct path, the number of nodes we have visited, and the number of times of solving linear systems. Table 3 shows the results under some (n, s) pairs, each with 40 instances.

(n, s)	The Proposed Attack			[CDM+18]
	#1	#2	#3	
$(256, 1.45)$	3.7	7.4	8.1	4
$(256, 1.4)$	6.0	29.0	31.0	6
$(256, 1.3)$	11.1	1038.0	1051.0	13
$(512, 1.45)$	5.1	18.5	20.0	6
$(512, 1.4)$	8.8	191.3	195.0	11
$(512, 1.3)$	17.3	81562.5	81887.3	23
$(1024, 1.45)$	6.9	70.9	73.1	9
$(1024, 1.4)$	13.5	4366.3	4373.0	17
$(2048, 1.45)$	10.4	663.9	668.3	14
$(4096, 1.45)$	15.4	19357.1	19364.7	21

The column #1 and the last column denote the required number of guesses, columns #2 and #3 denote the number of times of solving linear systems, and the number of visited nodes, respectively.

Table 3: The Results for Practical Key Recovery.

One can see that the number of visited nodes is slightly larger than the number of times the algorithm is solving linear equation systems, which makes sense since we always immediately trace back whenever a conflict occurs without going into solving a linear system. The available results match well with the simplified verification results in Table 2, from which we could estimate the required numbers of guesses for larger parameters.

The results are better than those in [CDM+18], particularly when more guesses are required. For example, when $n = 4096, s = 1.3$, 110 guesses are required in [CDM+18], while our attack only needs around 73. This is because when more guesses are needed, we perform more guesses in Class III and get more "free" variables, thus the advantage from exploiting "free" variables is more obvious.

If a system is expected to achieve a security level of r bits, the complexity for inverting the system should be larger than 2^r. We use the same estimation of time complexity[1] as that in [CDM+18], which is $2^\ell \cdot n^2$, where ℓ and n denote the number of guesses and seed size, respectively. Actually, there should be some constant factor for $2^\ell \cdot n^2$ when deriving the actual complexity from the asymptotic complexity $O(2^\ell \cdot n^2)$. However, for a fair comparison, we take the constant factor as 1 as done in [CDM+18]. Thus, to achieve the r-bit security, ℓ should satisfy $2^\ell n^2 > 2^r$, i.e., $\ell > r - 2\log_2 n$. Let us set $\ell = r - 2\log_2 n$ and try to derive the limit of stretch s above which the parameters are susceptible to our attack. We can substitute ℓ in (14) with $r - 2\log_2 n$ and get the number of linear equations. For an unsecure stretch s, the number is not less than n, i.e.,

$$\big(2(r - 2\log_2 n) - \delta(n,s)\big) \cdot 2n^{s-1} + c + \big(2(r - 2\log_2 n) - \delta(n,s)\big) \geq n. \tag{15}$$

It is difficult to get a closed-form expression for the stretch limit s as the average value of c involves a complicated expression in s as shown in (2). Thus, we derive the theoretical limits by starting from a relatively large value of s and gradually decreasing it with a small interval, until we find the maximal s that does not satisfy (15), for given n and r. We can use the same way to derive the theoretical limits for the guess-and-determine attack in [CDM+18], which were not given there. It is proved in [CDM+18] that the number of linear equations for guessing one variable is larger than $2\frac{m}{n}$, thus after guessing ℓ variables, the number of linear equations is larger than $2\ell\frac{m}{n} + c + \ell$. The unsecure stretch s in [CDM+18] would satisfy the condition below:

$$2(r - 2\log_2 n)n^{s-1} + c + r - 2\log_2 n \geq n. \tag{16}$$

Note that the stretch limits in [CDM+18] would be looser than ours, as the theoretical analysis there is based on the worst case, while we consider the average case.

We also get experimental limits by running extensive instances. For a given seed size n, we start with a high enough s value and decrease it by a 0.001 interval each time. For each (n, s) pair, we implement 400 independent instances[2], and check whether the complexity is larger than 2^{80} or 2^{128} under our guess-and-determine attack. We have shown in Table 2 and Table 3 that the required numbers of guesses to obtain enough linear equations match well with those to actually recover the secret, and we use the former to represent the latter, since actually solving the linear equation systems consumes large computation overhead.

Figure 9 shows the stretch limits for different seed sizes under the 80-bit and 128-bit security levels. The dashed lines denote the theoretical limits for our attack and the attack in [CDM+18], which are computed according to (15) and (16), respectively. The solid lines denote the experimental results (the experimental stretch limit for 128-bit security is not given in [CDM+18]). The zone above the lines represent the insecure choices of (n, s) parameters. From the results, one can easily see that the stretch limits under our attack are stricter than the ones in [CDM+18]. Thus, some systems which are secure under the attack in [CDM+18] cannot resist against our attack given a security

[1] The accurate estimation formula can be found in the proof of concept implementation of [CDM+18]. See `https://github.com/LuMopY/SecurityGoldreichPRG`.

[2] We tested a relatively smaller number of instances here, since the cases for $n = 8192$ require much more computation overhead. We found in the experiment that the results averaged on 400 instances and 1000 instances are almost the same, due to the small variances.

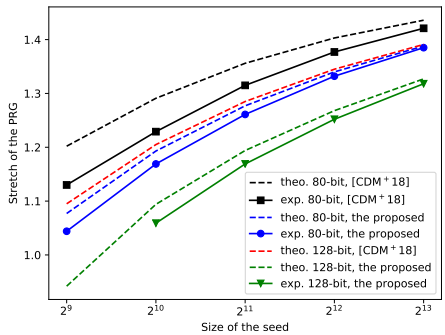

Figure 9: The limits of stretchs for vulnerable instances (theo. and exp. denote the theoretical and experimental results, respectively). The zones above the lines denote the insecure choices of parameters.

Security Level	(n, s)	New	[CDM$^+$18]
80 bits	$(512, 1.120)$	2^{44^*}	2^{91}
	$(1024, 1.215)$	2^{53^*}	2^{90}
	$(2048, 1.296)$	2^{68}	2^{91}
	$(4096, 1.361)$	2^{68}	2^{91}
128 bits	$(512, 1.048)$	2^{63^*}	2^{140}
	$(1024, 1.135)$	2^{75^*}	2^{140}
	$(2048, 1.222)$	2^{98}	2^{139}
	$(4096, 1.295)$	2^{100}	2^{140}

* Complexities are from the avalanche effect.

Table 4: Complexity for Solving the Challenge Parameters Proposed in [CDM$^+$18].

level. Particularly, our stretch limits for 80-bit security are even stricter than the theoretical limit for 128-bit in [CDM$^+$18], thought the comparison to the experimental limit is unclear. We will later show that we can attack some parameter sets suggested for 128-bit security in [CDM$^+$18] with complexity lower than 2^{80}. Besides, our theoretical and experimental limits have smaller gaps than the ones in [CDM$^+$18], especially when n goes larger, which indicate that with our theoretical analysis, one can have better predication of the security for a system when the parameters go larger.

Breaking Challenge Parameters. The authors in [CDM$^+$18] suggested some challenge parameters for achieving 80 bits and 128 bits of security. Table 4 compares the time complexities for attacking systems under these parameters. For a fair comparison, we again use the same estimation as that in [CDM$^+$18] where the time complexity is estimated as $2^\ell \cdot n^2$, with ℓ being the required number of guesses under which the

number of linear equations is not less than n. We also considered the avalanche effect for each system and the complexity is computed as 2^{ℓ_A}, where ℓ_A is the number of guesses under which the avalanche effect appears. We choose the smaller one between the two complexities. Note that the authors of [CDM+18] took a margin of 10% when selecting these security parameters. For each (n, s) parameter, we implemented 1040 instances. We found that the number of required guesses shows a small variance (but larger than the ones in [CDM+18]). Figure 10 presents the distribution of required number of guesses for the systems targeting 80-bit and 128-bit security levels when $n = 4096$.

We see that all the proposed challenge parameters fail to achieve the claimed security levels and the improvement factor becomes larger when n is smaller. When the seed size is smaller, the system is more susceptible to the avalanche effect, e.g., when $n = 512$ and 1024. For $(n, s) = (512, 1.048)$, the time complexity of the improved algorithm is smaller by a factor of about 2^{77}; this parameter set is even insufficient for providing 80 bits of security, though it is originally suggested for 128 bits.

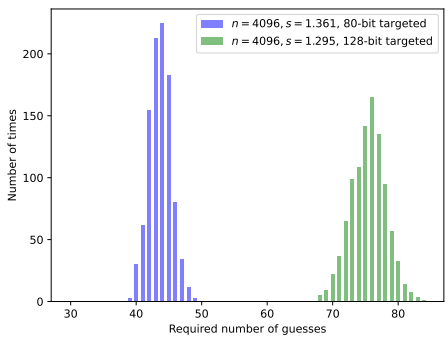

Figure 10: The distribution of required number of guesses ($n = 4096$).

4 Guess-and-Decode: A New Iterative Decoding Approach for Cryptanalysis on Goldreich's PRGs

In this section, we present a new attack on Goldreich's PRGs with even lower complexities. We combine the guessing strategies described in Section 3 and iterative decoding to invert a quadratic system and recover the secret, which we call guess-and-decode attack. Specifically, we first guess some variables using similar strategies described in our guess-and-determine attack; then instead of inverting a linear system, we apply probabilistic iterative decoding on the derived quadratic system to recover the secret, for which the complexity is smaller. Besides this gain in reducing the complexity, we also expect that fewer guesses are required before a system can be correctly inverted.

We first give a recap on the classical iterative decoding showing how it works on linear checks; then describe how we modify it to invert a quadratic system. We experimentally

verify the attack, showing that with soft decoding we can further reduce the complexities for attacking the challenge parameters given in [CDM+18], and finally suggest some new challenge parameters which appear to resist against our attacks for further investigation.

4.1 Recap on Iterative Decoding

Iterative decoding has been commonly used in information theory, e.g., most notably used for decoding LDPC (low-density parity-check) codes. The basic idea of iterative decoding is to break up the decoding problem into a sequence of iterations of information exchange, and after some iterations the system is expected to converge and give a result (or the process is halted). It can provide sub-optimal performance with a much reduced complexity compared to a maximum likelihood decoder. Below we give a short introduction to LDPC codes and describe how iterative decoding works. For more details about iterative decoding, we refer to [RL09, HOP96].

A binary (n, k) LDPC code, where k and n respectively denote the lengths of the information block and the codeword, is a linear block code which is usually defined as the null space of a *parity-check matrix* $\mathbf{H}$ of size $(n-k) \times n$ whose entries are either 1 or 0. For every valid codeword $\mathbf{v}$, $\mathbf{v}\mathbf{H}^T = 0$ is always satisfied. Each row of $\mathbf{H}$ denotes one parity check and the number of rows indicates the number of checks every codeword should satisfy, while each column denotes one code symbol. If the code symbol j is involved in a check i, the entry (i, j) of $\mathbf{H}$, denoted h_{ij}, would be 1; otherwise 0. As follows from the constraint $\mathbf{v}\mathbf{H}^T = 0$, the symbols involved in one same parity check must sum to zero (modulo 2). Usually, the density of 1's in $\mathbf{H}$ should be sufficiently low to allow for iterative decoding, thus getting the name LDPC.

A *Tanner graph* is usually used to represent a code and help to describe iterative decoding. It is a bipartite graph with one group of nodes being the variable nodes (VNs), i.e., the code symbols, and the other group being the check nodes (CNs). If the variable j is involved in the check i, an edge between CN i and VN j is established.

Example. The following parity-check matrix $\mathbf{H}$ expresses a $(6, 2)$ linear block code. The code has six symbols, say $\{x_1, x_2, \ldots, x_6\}$, each involved in two parity checks, and four parity checks, say $\{c_1, c_2, c_3, c_4\}$, each involving three symbols. For example, the first parity check c_1, i.e., the first row of $\mathbf{H}$, can be expressed as $x_1 + x_2 + x_3 = 0$.

$$\mathbf{H} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}. \tag{17}$$

The Tanner graph corresponding to $\mathbf{H}$ in (17) is depicted in Figure 11. The edges connecting variable nodes and check nodes correspond to the 1's in $\mathbf{H}$. For example, VNs x_1, x_2, x_3 are connected to CN c_1 in accordance with the fact that in the first row of $\mathbf{H}$, $h_{00} = h_{01} = h_{02} = 1$, while all others in this row are zero.

The Tanner graph of a code acts as a blueprint for an iterative decoder. The VNs and CNs exchange information along the edges, and the process is referred to as *message passing*. Each node (either a variable node or a check node) acts as a local computing processor, having access only to the messages over the edges connected to it. Based on

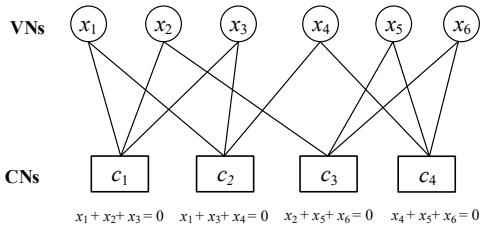

Figure 11: An illustration of a simple Tanner graph.

the incoming messages over the connected edges, a node would compute new messages and send them out. In some algorithms, e.g., *bit-flipping* decoding, the exchanged messages are binary (hard) values, while in others, such as *belief-propagation* or *sum-product* decoding, the messages are soft probabilities, which represent a level of belief about the values of the code symbols. The probabilities can be implemented in the logarithm domain, i.e., log-likelihood ratios (LLRs), which makes the iterative decoding more stable. We use the sum-product decoding based on LLRs in our attack.

The sum-product algorithm accepts *a-priori* probabilities for the information symbols, which are usually received from the channel and known in advance before the decoding, and outputs *a posteriori* values for all symbols after some iterations of message passing. Below we give the general steps of the sum-product algorithm.

Step 1 Initialization. For every variable v, initialize its LLR value according to its a-priori LLR value $L_a(v)$, i.e., $L_v^{(0)} = L_a(v)$. For every edge connected to v, initialize the conveyed LLR values as $L_v^{(0)}$.

After the initialization, the algorithm starts iterations for belief propagation, in which **Step 2 - Step 4** below are iteratively performed until the recovered secret is correct or reaching the maximum allowed number of iterations. In each iteration, every node (either variable node or check node) updates the outgoing LLR value over every edge based on the incoming LLR values over all the *other* edges, which are called "extrinsic" information; while ignoring the incoming LLR value over this specific edge, which is called "intrinsic" information. It is always the "extrinsic" information that should be only used to update the LLR values.

Step 2 Check node update. In each iteration i, every check node c computes an outgoing LLR value over each of its edges e_k, based on the incoming LLR values updated in the $(i-1)$-th iteration from every *other* edge e'_k connected to c. The computation is as below, and we refer to [RL09, HOP96] for more detailed derivation:

$$\begin{aligned} L_c^{(i)}(e_k) &= \boxplus_{k' \neq k} L_v^{(i-1)}(e_{k'}) \\ &= 2\tanh^{-1}\left(\prod_{k' \neq k} \tanh\left(1/2 L_v^{(i-1)}(e'_k)\right)\right), \end{aligned} \tag{18}$$

where the "box-plus" operator $\boxplus$ is used to denote the computation of the LLR value of the XOR sum of two variables, i.e., for $a = a_1 \oplus a_2$, $L(a) = L_{a_1} \boxplus L_{a_1} = \log(\frac{1+e^{L_{a_1}+L_{a_2}}}{e^{L_{a_1}}+e^{L_{a_2}}})$;

tanh() corresponds to the hyperbolic tangent function, and appears in the expression due to an easily proven relation $\tanh(u/2) = (e^u - 1)/(e^u + 1)$.

Step 3 Variable node update. In each iteration i, every variable node v computes an outgoing LLR value over each of its edges e_j, based on the a-priori information, and LLR values updated in the i-th iteration from every *other* edge e'_j connected to v, as below:

$$L_v^{(i)}(e_j) = L_a(v) + \sum_{j' \neq j} L_c^{(i)}(e_{j'}). \tag{19}$$

Step 4 Distribution update. After each iteration, update the LLR value of every variable v using (19), but with every edge included, i.e.,

$$L_v^{(i)} = L_a(v) + \sum_{j \in N(v)} L_c^{(i)}(e_j), \tag{20}$$

where $N(v)$ is the set of edges connected to v. Set

$$\hat{v} = \begin{cases} 1, & \text{if } L_v^{(i)} < 0, \\ 0, & \text{otherwise,} \end{cases} \tag{21}$$

to recover an intermediate value for every variable v. The algorithm immediately stops whenever $\hat{\mathbf{v}}\mathbf{H}^{\mathbf{T}} = \mathbf{0}$ is satisfied or the number of iterations reaches the maximum limit; otherwise, it continues with a new iteration.

4.2 Algorithm for the Guess-and-Decode Attack

In this subsection, we show how we combine guessing and iterative decoding to invert a Goldreich's PRG and recover the secret. We modify the classical iterative decoding to accommodate our use case, and the differences are listed below:

1. The most important difference lies in that the system in classical iterative decoding is linear, while quadratic in our application. We have designed special belief propagation techniques for quadratic equations.

2. In classical iterative decoding, a-priori information, either from a non-uniform source or from the channel output, is required and plays an important role for the convergence of the belief propagation. However, in our case, there is no available a-priori information and all the variables are assumed to be uniformly random distributed. The system is expected to achieve self-convergence.

3. In classical iterative decoding, the check values are always zero; while in our case, a check value could be one, and special belief propagation techniques are designed for it.

Algorithm 4 shows the general process of the guess-and-decode attack. It basically consists of two phases: guessing phase, during which guessing strategies similar to what have been described in Section 3 are applied to guess and derive "free" variables; and decoding phase, during which the modified iterative decoding is applied on the resulting quadratic system to recover the remaining secret bits.

Algorithm 4 The guess-and-decode attack

Input A Goldreich's PRG system instantiated on P_5, the maximum allowed number of iterations $iterMax$

Output A recovered secret $\hat{x}$

1: Guess some variables using strategies similar to what have been described in Section 3
2: **if** the secret can already be correctly recovered **then**
3: return the recovered secret
4: **else**
5: Build an iterative decoding model for the resulting system of equations, initialization, set $it = 0$
6: **while** $it < iterMax$ **do**
7: $it = it + 1$
8: Update check nodes
9: Update variable nodes
10: Update the distributions of variables and get an intermediate recovered secret $\hat{x}$, return $\hat{x}$ if it is correct.
11: If $it == iterMax$, trace back following Algorithm 3 and go to step 2

4.2.1 Guessing Phase

The guessing process generally follows the strategies in the guess-and-determine attack described in Section 3, but with some modifications. Specifically, in Class III, we only guess variables from the linear equations, and if there are no linear equations, we guess a variable in equations in Class II (or further Class I if Class II is empty), instead of guessing a variable in a quadratic equation in Class III. This is because we want to keep the quadratic equations and get biased information for some involved variables. For example, for an equation $x_{\sigma_1} + x_{\sigma_2}x_{\sigma_3} = y$, after the first iteration of belief propagation, we would get a biased distribution of x_{σ_1}, i.e., $P(x_{\sigma_1} = 1) = 0.25$ and $P(x_{\sigma_1} = 0) = 0.75$. Such biased information could propagate through the graph and help to make the system converge. This is how the system achieves self-convergence without any a-priori information.

Similarly, after guessing a relatively large number of variables, it could happen that all the remaining variables can be directly recovered without going into iterative decoding. This happens in two cases:

(1) all other variables are determined for free because of the avalanche effect;

(2) besides the guessed and freely determined variables, every remaining variable is involved as the linear term in a quadratic equation in Class III and can be recovered correctly with high probability. For example, for an equation $x_{\sigma_1} + x_{\sigma_2}x_{\sigma_3} = y$, we can recover x_{σ_1} as $x_{\sigma_1} = y$. This happens more often when more variables are guessed. The experimental results would verify this later.

Thus, after the guessing phase, we always check if all the variables can already be correctly recovered; and if so, the iterative decoding can be omitted.

4.2.2 Decoding Phase

When the guessing phase is done, we build an iterative decoding model for the derived system of equations and start the belief propagation.

Iterative decoding model. The remaining unknown secret variables (neither guessed nor freely determined) are modeled as the variable nodes, while all the remaining valid equations, either quadratic or linear, serve as the checks. Recall that we could have six different forms of equations in the derived system which we have categorized into three classes in Section 3.1, along with one more form of those which only have a quadratic term, e.g., $x_{\sigma_1^i} x_{\sigma_2^i} = 0$. A check node and the variable nodes which are involved in this given check would be connected through edges in the Tanner graph. We define two different types of edges: *type 1* edge, which connects a check node and a variable node that is involved as a linear term of this check; and *type 2* edge, which connects a check and a variable that is involved in the quadratic term of this check. For these two different types of edges, different belief propagation techniques are applied, which we would elaborate below. Besides, the check value of a check could be either 1 or 0, which does not happen in the classical iterative decoding, and we would describe how we deal with it.

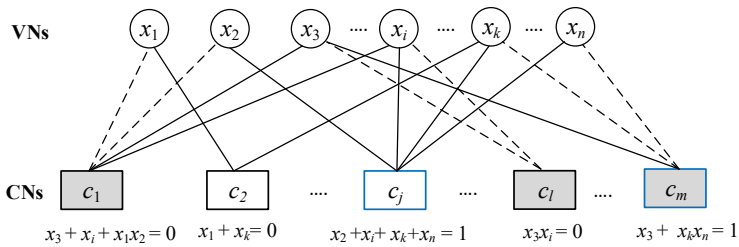

Figure 12: Illustration of an example of the iterative decoding model.

Figure 12 shows a simple illustration of our iterative decoding model, where circles denote the variable nodes while rectangles denote the check nodes. The gray-filled checks have quadratic terms where the solid lines denote *type 1* edges while the dashed lines denote *type 2* edges. The blue-bordered checks are those with check values being one.

The decoding phase generally follows the routine of classical iterative decoding described in Section 4.1, but we have some novel modifications for, e.g., dealing with checks with quadratic terms or with check values being one. We next describe these details in each step.

Step 1 Initialization. Since there is no a-priori information for the variables, the LLR values of all the variables are initialized to be zero. All the outgoing messages, i.e., LLR values, over the edges of variables or checks are set to be zero as well.

After the initialization, the algorithm starts iterations for belief propagation. The update of variable nodes follows the classical iterative decoding, while for updating the check nodes, we have special belief propagation techniques, which are described in details below.

Step 2 Check node update. The update of a linear check just follows the classical way, with a small modification when the check value is one. We below give a simple example to show how it affects the belief propagation.

Example. Given a weight-2 check $x_{\sigma_1} + x_{\sigma_2} = y$, assume the incoming LLR value in a certain iteration for x_{σ_2} is $L_{x_{\sigma_2}}$.

– If $y = 0$, the combinations of $(x_{\sigma_1}, x_{\sigma_2})$ to validate the check is $(0,0),(1,1)$. Thus,

$$\begin{aligned} &P(x_{\sigma_1} = 0) \\ &= P(x_{\sigma_1} = 0, x_{\sigma_2} = 0) + P(x_{\sigma_1} = 0, x_{\sigma_2} = 1) \\ &= P(x_{\sigma_1} = 0 | x_{\sigma_2} = 0) P(x_{\sigma_2} = 0) + 0 \\ &= P(x_{\sigma_2} = 0). \end{aligned}$$

Similarly, we can get $P(x_{\sigma_1} = 1) = P(x_{\sigma_2} = 1)$. Thus, the outgoing LLR value of x_{σ_1} value is computed as $L_{x_{\sigma_1}} = \log \frac{P(x_{\sigma_2}=0)}{P(x_{\sigma_2}=1)} = L_{x_{\sigma_2}}$.

– While if $y = 1$, the combinations of $(x_{\sigma_1}, x_{\sigma_2})$ to validate the check is $(0,1),(1,0)$. Similarly we would get $P(x_{\sigma_1} = 0) = P(x_{\sigma_2} = 1)$ and $P(x_{\sigma_1} = 1) = P(x_{\sigma_2} = 0)$. The outgoing LLR value of x_{σ_1} is then computed as $L_{x_{\sigma_1}} = \log \frac{P(x_{\sigma_2}=1)}{P(x_{\sigma_2}=0)} = -L_{x_{\sigma_2}}$.

Thus, for a linear check, when the check value is zero, the outgoing LLR values are computed with the standard way in (18); while when it is one, the negative versions of values computed using (18) are sent.

When updating the quadratic check nodes, different belief propagation techniques are applied for *type 1* and *type 2* edges. The update for *type 1* edges follows the classical way using (18), while including the equivalent incoming LLR value from the quadratic term as well. For *type 2* edges, we should deal with them carefully by distinguishing "intrinsic" and "extrinsic" information. Below we show how to update a quadratic check node.

For a quadratic check, e.g., $x_{\sigma_1} + x_{\sigma_2} + x_{\sigma_3} + x_{\sigma_4} x_{\sigma_5} = y$, assume the incoming LLR values over the edges are $L_{x_{\sigma_1}}, L_{x_{\sigma_2}}, L_{x_{\sigma_3}}, L_{x_{\sigma_4}}, L_{x_{\sigma_5}}$, respectively, and we want to compute the outgoing LLR value over each edge. We denote the linear part and quadratic part as x_l and x_q, respectively, i.e., $x_l = x_{\sigma_1} + x_{\sigma_2} + x_{\sigma_3}$, $x_q = x_{\sigma_4} x_{\sigma_5}$. We could compute the equivalent LLR values for x_l and x_q, denoted as L_{x_l} and L_{x_q}, respectively. For x_l, we could easily get $L_{x_l} = L_{x_{\sigma_1}} \boxplus L_{x_{\sigma_2}} \boxplus L_{x_{\sigma_3}}$. While for x_q, we first get

$$\begin{aligned} p^1_{x_q} &= p^1_{x_{\sigma_4}} p^1_{x_{\sigma_5}} = \frac{1}{(e^{L_{x_{\sigma_4}}} + 1)(e^{L_{x_{\sigma_5}}} + 1)}, \\ p^0_{x_q} &= 1 - p^1_{x_{\sigma_4}} p^1_{x_{\sigma_5}} = \frac{(e^{L_{x_{\sigma_4}}} + 1)(e^{L_{x_{\sigma_5}}} + 1) - 1}{(e^{L_{x_{\sigma_4}}} + 1)(e^{L_{x_{\sigma_5}}} + 1)}. \end{aligned} \tag{22}$$

Thus the equivalent incoming LLR value for x_q can be computed as:

$$L_{x_q} = \log \frac{p^0_{x_q}}{p^1_{x_q}} = \log \left((e^{L_{x_{\sigma_4}}} + 1)(e^{L_{x_{\sigma_5}}} + 1) - 1 \right).$$

If $y = 0$, the outgoing LLR value of a linear variable, say x_{σ_1}, can be computed as:

$$L_{x_{\sigma_1}} = L_{x_{\sigma_2}} \boxplus L_{x_{\sigma_3}} \boxplus L_{x_q}. \tag{23}$$

While if $y = 1$, the negative version, i.e., $-L_{x_{\sigma_1}}$, should be sent, just like the update of a linear check when its check value is one. The outgoing LLR values for x_{σ_2} and x_{σ_3} can be computed in the same way.

Next we show how to compute the outgoing LLR values for the variables in the quadratic term, e.g., x_{σ_5}. For combinations of $(x_l, x_{\sigma_4}, x_{\sigma_5})$, when $y = 0$, the possible values to validate the check are $(0,0,0)$, $(0,0,1)$, $(0,1,0)$, $(1,1,1)$. Thus the updated LLR value for x_{σ_5} is

$$L_{x_{\sigma_5}} = \log \frac{p^0_{x_l} p^0_{x_{\sigma_4}} p^0_{x_{\sigma_5}} + p^0_{x_l} p^1_{x_{\sigma_4}} p^0_{x_{\sigma_5}}}{p^0_{x_l} p^0_{x_{\sigma_4}} p^1_{x_{\sigma_5}} + p^1_{x_l} p^1_{x_{\sigma_4}} p^1_{x_{\sigma_5}}} \tag{24}$$

$$= \log \frac{p^0_{x_{\sigma_5}}}{p^1_{x_{\sigma_5}}} + \log \frac{p^0_{x_l}}{p^0_{x_l} p^0_{x_{\sigma_4}} + p^1_{x_l} p^1_{x_{\sigma_4}}}. \tag{25}$$

In (25), the first term is regarded as the "intrinsic" information while the second term is the "extrinsic" information which should be propagated. Thus the outgoing LLR value for x_{σ_5} is

$$L_{x_{\sigma_5}} = \log \frac{p^0_{x_l}}{p^0_{x_l} p^0_{x_{\sigma_4}} + p^1_{x_l} p^1_{x_{\sigma_4}}}. \tag{26}$$

While if $y = 1$, the possible values of $(x_l, x_{\sigma_4}, x_{\sigma_5})$ to validate the check are $(1,0,0)$, $(1,0,1)$, $(1,1,0)$, $(0,1,1)$. $L_{x_{\sigma_5}}$ can still be computed using (26), but with the values of $p^0_{x_l}$ and $p^1_{x_l}$ being exchanged. We could understand it as that L_{x_l} now has the negative version. The outgoing LLR value for x_{σ_4} can be derived in the same way.

The update techniques apply to all types of checks: a linear check and a check with only a quadratic term can be regarded as special cases without x_q and x_l, respectively. For the latter case, e.g., $x_{\sigma_1} x_{\sigma_2} = 0$, the possible combinations of $(x_{\sigma_1}, x_{\sigma_2})$ to validate the check is $(0,0), (0,1), (1,0)$, thus the LLR value of x_{σ_1} can be updated as $L_{x_{\sigma_1}} = \log(1/p^0_{x_{\sigma_2}}) = \log(1 + e^{-L_{x_{\sigma_2}}})$. The LLR value of x_{σ_2} can be updated in the same way.

Important Notes. We mentioned above that no a-priori information is required for the convergence in our iterative decoding model. Instead, after the first iteration of updating the check nodes, some biased information for some variables would be obtained. The biased information mainly comes from quadratic checks in Class III and quadratic checks without any linear terms. As we mentioned, for a quadratic check in Class III $x_{\sigma_1} + x_{\sigma_2} x_{\sigma_3} = 0$, x_{σ_1} can become highly biased soon: i.e., $P(x_{\sigma_1} = 0) = 0.75$ and $P(x_{\sigma_1} = 1) = 0.25$, and the outgoing LLR value over the edge would become $\log 3$ instead of zero. Similarly, for a check $x_{\sigma_1} x_{\sigma_2} = 0$, the outgoing LLR values for x_{σ_1} and x_{σ_2} would become $\log 2$. Such biased information can then be propagated and spread to other nodes during the iterations, which plays an important role for the convergence and correctness of the iterative decoding. Obviously, the more accurate biased information

is obtained, the more likely the system would be to correctly converge, which explains why we try to avoid guessing a variable in a quadratic equation in Class III during the guessing phase.

Steps 3, **4** just follow the details described in **Steps 3**, **4** in Section 4.1.

4.3 Theoretical Analysis

A deeper theoretical investigation into the complexity is hard, as we have much more complicated model compared to classical iterative decoding: we have irregular checks which can be linear or quadratic, and have different localities; besides, there is no available external a-priori information. Instead, we derive a very rough estimation for the asymptotic complexity. Under the same amount of guesses as that in the guess-and-determine attack, the guess-and-decode attack would succeed with a large probability. For example, we experimentally checked which attack succeeds first when guessing from a very small and gradually increasing number of guesses. We tested 5000 independent instances for each parameter set, and the table below shows some probabilities of the systems being solved first by the guess-and-decode attack.

Seed Size	2048		4096	
Stretch	1.296	1.222	1.361	1.295
Probability	77.8%	80.7%	90.8%	96.4%

Table 5: Probability of systems being solved first by the guess-and-decode attack.

When one guessing path results in a failure, several other paths can be tried and it is highly likely that one would succeed. We allow for a constant number of iterations, and in each iteration, we need to go through $O(n^s)$ nodes. Thus a rough asymptotic complexity can be expressed as $O(n^s 2^{\frac{73}{288}n^{2-s}})$.

4.4 Experimental Verification

We have done extensive experiments to verify the attack. We first test the success probabilities of correctly recovering a secret under different numbers of guesses ℓ for the challenge parameters suggested in [CDM+18], averaged over 5000 independent instances for each (n, s, ℓ) parameter set.

Figure 13 illustrates the results of success probabilities when maximally 100 iterations are allowed, where the left and right sub-figures are for the parameters suggested for 80-bit and 128-bit security levels in [CDM+18], respectively. One can see that the success probabilities increase with the number of guesses, though sometimes with very small fluctuations. We found in the results that under different numbers of guesses, the required numbers of iterations for the convergence of a system vary: basically, when more variables are guessed, fewer iterations are required. Particularly, when guessing more than some certain number of variables, a secret could be recovered without going into iterations of belief propagation in some instances, which could happen with two cases as we mentioned before.

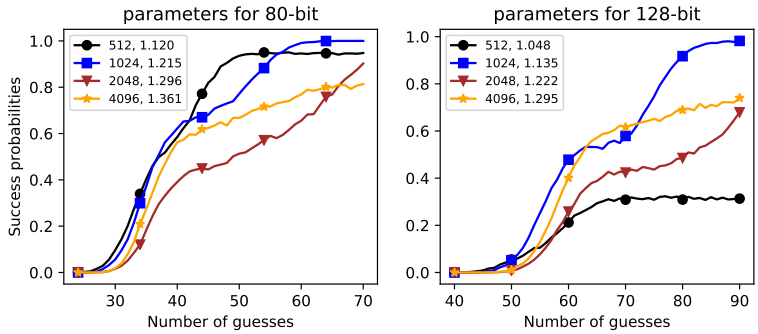

Figure 13: Success probabilities under different number of guesses.

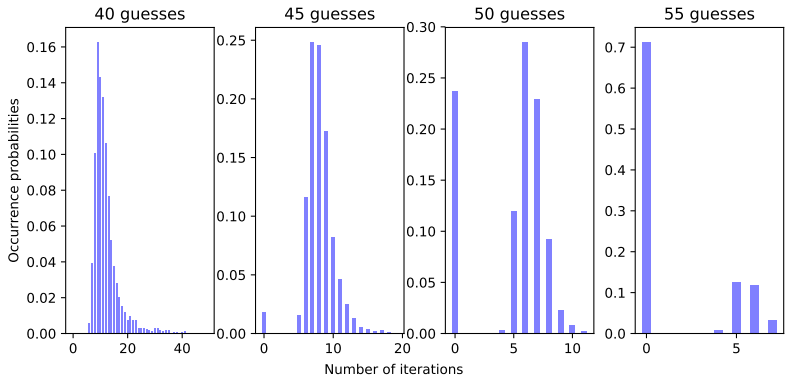

Figure 14: Distributions of number of iterations for successful instances ($n = 1024, s = 1.215$).

Figure 14 shows the distributions of the required numbers of iterations for convergence when guessing 40, 45, 50 and 55 variables, respectively, when $n = 1024, s = 1.215$. The instances with zero iterations denote those which can be solved without iterative decoding. We can see that with the increase in the number of guesses, the proportion of these instances is increasing: when guessing 40 variables, all the successful recoveries are solved through iterative decoding; while when guessing 55 variables, 71.0% of the successful instances can be solved without going into iterative decoding.

We also see from Figure 14 that the required numbers of iterations for the vast majority of instances are below 100, and mostly below 40, and the average required number of iterations decreases when the number of guesses increases. This applies to all instances under different parameter sets. In Figure 15 we give an example to further illustrate this observation. The left sub-figure shows the distribution of the required number of iterations for 97259 independent successfully inverted instances when guessing

26 to 51 variables for $n = 1024, s = 1.215$ and 44 to 69 variables for $n = 1024, s = 1.215$[3]. The right sub-figure shows the average required number of iterations under different numbers of guesses. Basically, the required number of iterations decreases when the number of guesses increases (with some exceptions in the beginning due to the small number of samples), which is because more equations with lower localities could be obtained when more variables are guessed, enabling faster convergence of the iterative decoding. Specifically, when guessing more than 53 and 75 variables when $s = 1.215$ and $s = 1.135$, respectively, the required numbers of iterations for most instances are zero, meaning that we can recover the secret directly without iterative decoding.

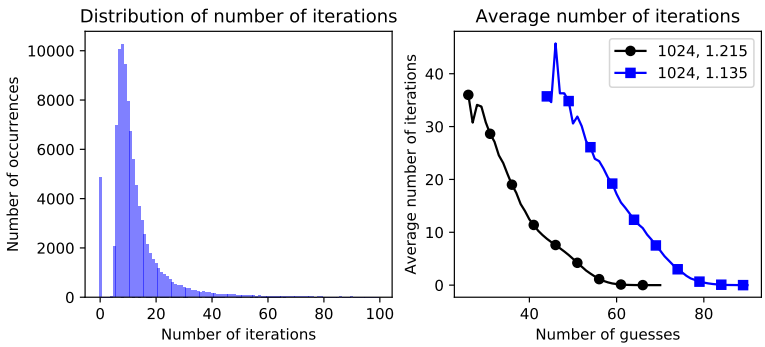

Figure 15: More results about the required number of iterations.

We also tested the distribution of number of guesses required to correctly recover a secret. We ran 5000 independent instances for parameter sets $n = 1024, s = 1.215$, and $n = 1024, s = 1.135$, which respectively aim for 80-bit and 128-bit security levels in [CDM+18]. We start from guessing a small number of variables, and continue with guessing one more variable if the recovery fails, until the secret is correctly recovered. The results are shown in Figure 16. We can obviously see from the figure that instances aiming for 80-bit security level require fewer guesses than those for 128-bit security level, for which the peak values are respectively achieved around 36 and 59 guesses.

4.5 Complexity

In our guessing strategies, we always prefer to guess the variable which appears most often in a local class or in the global system, which is a greedy method. Actually, if we choose to guess the sub-optimal ones, the success probabilities do not have big differences, as the main gain comes from exploiting "free" variables. This enables us to explore other guessing paths if the current one fails for iterative decoding.

If the success probability of the guess-and-decode attack when guessing ℓ variables for a given (n, s) parameter set is p, the required number of times to perform iterative decoding could be derived by multiplying a factor of $1/p$. This can be linked to two

[3]We did not consider the required numbers of iterations when guessing more variables since most would be zero and here we only want to show that the required numbers of iterations are mostly below 100.

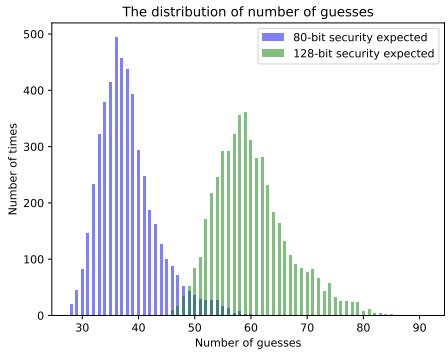

Figure 16: Distribution of number of guesses for $n = 1024$.

practical attacking scenarios: 1) $1/p$ many independent systems generated by different secrets are collected and there should be one system that can be recovered; 2) or given a specific system, it is possible to recover it if we try $1/p$ independent guessing paths. We have experimentally verified this, and will be shown later.

We have shown in Figure 15 that the vast majority of instances require less than 40 iterations for convergence, thus we set the maximum allowed number of iterations as 40. For each iteration, we need to update each variable node and check node, which requires complexity $O(n^s)$. The update of each variable node is simple, which only costs several additions, as can be seen from (19). For a check node, let us consider a worst case of complexity for updating, i.e., considering a check with most edges, e.g., $x_{\sigma_1} + x_{\sigma_2} + x_{\sigma_3} + x_{\sigma_4} x_{\sigma_5} = y$. For updating such a check, it requires 3 exponentiation, 5 division, 4 logarithm, 13 multiplication, 4 tangent and 4 inverse tangent functions. These functions roughly take 1300 clock cycles if we refer to the instruction manuals for mainstream CPUs. The actual cost would be less, as there are many simpler checks which require fewer clock cycles. For example, the update of an equation check of the form $x_{\sigma_1} + x_{\sigma_2} = y$ is almost for free. We have experimentally verified the cost with our non-optimized code. For example, one iteration for a system of parameters $n = 512, s = 1.120$ consumes 0.46 milliseconds[4], running on a CPU with maximum supported clock speed 3000 MHz, corresponding to 1275 clock cycles for updating one node. There are many optimization techniques both for the iterative decoding algorithm and for the implementation details (e.g., using parallelization and look-up tables), which would largely reduce the required number of clock cycles. We believe that the constant factor is not larger than that in $O(n^2)$ for inverting a sparse matrix. Thus, the total complexity for inverting an instance of parameter set (n, s, ℓ) can be derived as below, up to some constant factor:

$$C = \frac{1}{p} 2^{\ell} \cdot 40 \cdot n^s. \tag{27}$$

Having made this clear, we could get the complexities for attacking the systems under the challenge parameters suggested in [CDM+18]. As ℓ and p vary with the given

[4]This includes all the clock cycles for one iteration, i.e., updating the check nodes and variable nodes, updating the distributions of variables, recovering an intermediate secret, etc.

systems, we experimentally derive them for each parameter set based on a large number of independent instances. For every (n, s) parameter set, we generate 5000 independent instances[5] and run the attack with maximally 40 iterations allowed when guessing ℓ variables for different ℓ values. We compute the complexities using (27) according to the experimental success probabilities for different ℓ, and choose the smallest one as the complexity for breaking the given (n, s) pair. For $n = 512$, we have some additional results: we see from Figure 13 that the success probabilities are consistently increasing and when we guess a large enough number of guesses ℓ', most instances can be recovered without iterative decoding. If we denote the success probability of such case as p', the complexity would be $1/p' \cdot 2^{\ell'} \cdot n$ (with n included since we need to go through every secret variable)[6] .

Security Level	(n, s)	Number of Guesses ℓ	Success (probabilities p)	Complexity C
80 bits	$(512, 1.120)^*$	40	718 (0.1446)	2^{52}
	$(1024, 1.215)$	32	613 (0.1226)	2^{53}
	$(2048, 1.296)$	32	210 (0.042)	2^{57}
	$(4096, 1.361)$	32	285 (0.057)	2^{58}
128 bits	$(512, 1.048)^*$	53	86 (0.0172)	2^{68}
	$(1024, 1.135)$	50	209 (0.0418)	2^{72}
	$(2048, 1.222)$	52	82 (0.0164)	2^{77}
	$(4096, 1.295)$	51	85 (0.0166)	2^{78}

The complexities for parameters with * are derived based on the instances for which iterative decoding is not needed.

Table 6: The Simulated Complexities for the Challenge Parameters Proposed in [CDM+18].

Table 6 shows the experimental results in terms of the optimum choices of the number of guesses and corresponding success probabilities, from which we compute the complexities according to (27). By "optimum" we mean that the complexity is the lowest by guessing the chosen number of variables, and the complexity can be different if we guess a different number of variables. However, we found from the experimental results that the complexities vary slowly with ℓ, and there are several ℓ values under which the complexities are the same to the best ones shown in the table. We can see that the complexities are much lower than the claimed ones in [CDM+18]. Particularly, the parameter sets suggested for 128 bits of security in [CDM+18] cannot even provide 80 bits of security under our attack.

As we mentioned, the attack can be applied to one specific system, with roughly $1/p$ attempts, and we have experimentally verified this. In our implementation, we guess the numbers presented in Table 6 of variables and check how many different guessing paths that we need to try before we finally recover the secret. Specifically, for each system,

[5]We generate 10 times more instances when $n = 512$ and 1024, and the success probabilities do not have big differences as those computed from 5000 instances. Thus we conjecture that 5000 instances are large enough for computing a stable success probability.

[6]We also checked the complexities using this way for other parameter sets, but they are all higher than the ones derived through iterative decoding.

we first follow the steps described above and perform iterative decoding, and if it fails, we try different paths: in the i-th attempt, we choose the i-th most often appeared variable as the first guess and the subsequent steps are just the same as before, i.e., roughly following Algorithm 2. We allow for maximally 250 attempts for each system. It is tricky to prove that these guessing paths are independent, but we observed that a different starting guess can affect the whole system a lot. Table 7 shows the average number of guessing paths that we need to try and the success probabilities, where ℓ and p are from Table 6. For each parameter set, we generated 5000 independent instances.

From the table, one can see that a specific system can be attacked in around $1/p$ attempts with a high success probability. When allowing for more attempts, the success probabilities will be even higher.

Security Level	(n, s)	ℓ	$\frac{1}{p}$	#attempts	success
80 bits	$(1024, 1.215)$	32	8.2	9.2	100%
	$(2048, 1.296)$	32	23.8	24.6	94.8%
128 bits	$(1024, 1.135)$	50	23.9	32.8	87.0%
	$(2048, 1.222)$	52	61.0	68.9	88.2%

Table 7: Results for multiple attempts.

4.6 New Challenge Parameters

We also suggest some practical range of parameter sets which appear to be resistant against both the proposed guess-and-determine attack and the guess-and-decode attack for 80 and 128 bits of security. As done in [CDM+18], we take a 10% margin to select the security parameters. Under a certain seed size n, we vary s from a high enough value to lower ones with a relatively larger interval 0.01 (for instances of $n = 512$, we take a smaller interval 0.001), since we need to run the iterative decoding process and recover the secret in our implementation, which requires more running time. We choose the maximum s value for which the system is not vulnerable to our attacks as the conjectured stretch limit.

Given each (n, s) pair, we vary the number of guesses ℓ from a low enough value to a high enough value and run 5000 instances for each (n, s, ℓ) parameter set. We record the success probability for each ℓ and further compute the corresponding complexity using (27). For $n = 512$, we further compute the complexity for recovering the secret without using iterative decoding. We then chose the minimum one as the complexity for the (n, s) pair and check if it is larger than 2^{80} or 2^{128}. We chose the maximum s for which the condition is satisfied as the conjectured stretch limit for challenge parameters.

Table 8 shows the experimental results under different seed sizes. Further study is required to guarantee confidence in the security levels given by these parameters. One can see that the stretch limits are further narrowed down with a large gap from the ones in [CDM+18]. For example, when $n = 1024$, the results in [CDM+18] suggest that the systems with stretches smaller than 1.215 and 1.135 can provide 80 and 128 bits of security, respectively, while our results show that the stretches should be smaller than

1.08 and 1.02, respectively. Besides, our results show that systems with seed sizes of 512 are not suitable for constructing local PRGs: they cannot even provide 80 bits of security.

security level	512	1024	2048	4096
80	-	1.08	1.18	1.26
128	-	1.02	1.10	1.19

Table 8: Challenge Parameters for Seed Recovery Attack.

5 Extension to Other Predicates

It is interesting and of importance to know if the proposed attacks apply to other predicates and which ones are susceptible to or resistant against them. In this section, we investigate this question and focus on the two main types of predicates suggested for constructing local PRGs, i.e., XOR-AND and XOR-THR (threshold) predicates, which are defined as below:

$$\text{XOR}_k - \text{AND}_q : x_1 + \cdots + x_k + x_{k+1}x_{k+2}\cdots x_{k+q},$$
$$\text{XOR}_k - \text{THR}_{d,q} : x_1 + \cdots + x_k + \text{THR}_{d,q}(x_{k+1}, \ldots, x_{k+q}),$$

where $\text{THR}_{d,q}(x_{k+1}, \ldots, x_{k+q})$ is a threshold function of which the value would be one only when the number of one's in $(x_{k+1}, \ldots, x_{k+q})$ is not less than d; otherwise, the value is zero. P_5 can be regarded as a special case of XOR-AND and XOR-THR predicates. These predicates have been investigated in [CDM+18, App16, AL18, MCJS19].

We applied both attacks against these predicates under some suggested challenge parameters, and observed different resistance of these predicates generalized as below.

- The considered XOR-AND predicates are more vulnerable to the guess-and-determine attack, especially when the locality gets larger. The reason is that the iterative decoding has worse performance when the parity checks have higher weights, while obtaining a linear equation could be cheaper: as long as one variable in the AND term is fixed as zero, or all variables inside are known.

- For the considered XOR-MAJ predicates, the guess-and-decode attack applies better, mainly due to the high overhead to obtain a linear equation: only when more than half of the variables in the MAJ term are known to be the same, the non-linear term can be eliminated. On the other hand, the guess-and-decode attack can still exploit some soft information useful for iterative decoding from the MAJ term even if its value is unknown. For example, if one variable in the MAJ term is fixed as 1 (0), its value will be more likely to be 1 (0) than 0 (1).

We mention here that the asymptotic complexities against these predicates are more difficult to get, due to the much more diversified intermediate equations and complicated equation reduction. For example, for an XOR_k-AND_q predicate, there are $(k+1)q - 1$

different types of intermediate equations, e.g., when $k = 6, q = 3$, the number is 20. This not only makes it difficult to derive the numbers of "free" variables and linear equations, but also makes the performance of iterative decoding (if it is used) much more obscure. Actually, the performance of iterative decoding for linear checks of various different weights, e.g., in irregular LDPC code, is not so easy to analyze, let alone a system of equations with non-linear terms of various degrees. However, knowing the asymptotic complexity will be definitely very helpful for understanding the security of local PRGs built on these predicates, and we leave that for future work.

5.1 Extensions to Other XOR-AND Predicates

In the survey paper [App16], the authors asked the question "*Is it possible to efficiently invert the collection $\mathcal{F}_{P,n,m}$ for every predicate P and some $m = n^{\frac{1}{2}\lfloor 2d/3 \rfloor - \epsilon}$ for some $\epsilon > 0$?* " and gave "a more concrete challenge": XOR_k-AND_q predicates with $k = 2q$. We consider these challenging predicates and investigate their resistance against our attacks.

The guessing phase generally follows the strategies described in Subsection 3.2, but involves more classes of equations and more complex equation reduction. The equations that can possibly introduce "free" variables are of the forms: $x_{i_0} + x_{i_1}x_{i_2}\cdots x_{i_h} = y_i$ $(2 \le h \le q), x_{j_1} + x_{j_2} = y_j$ and $x_{u_1}x_{u_2} = 0$. One can see that more times of equation reduction are required to derive such equations. The way to derive the "free" variables are the same, with a small difference when $x_{i_0} = y_i + 1$ in the first case: all the variables in the high-degree term are determined as 1. However, such equations do not always exist during the guessing phase, thus making the number of "free" variables not so clear.

For the guess-and-decode attack, we keep the equations $x_{i_0} + x_{i_1}x_{i_2}\cdots x_{i_h} = y_i$ $(2 \le h \le q)$ as they are the source of biased information. The quality of such biased information is better than that in P_5, as the value of x_{i_0} is more biased: equals to y_i with probability $1 - 0.5^h$. However, the large weights of the equations lead to bad performance of iterative decoding. Actually, the guess-and-decode attack does not apply as good as the guess-and-determine attack. We below still provide the belief propagation techniques for a general XOR_k-AND_q predicate, as the techniques may apply to some other predicates which involve a higher-degree AND term.

For an XOR_k-AND_q equation $x_1 + x_2 + \cdots + x_k + x_{k+1}x_{k+2}\cdots x_{k+q} = y$, suppose the incoming LLR values for $x_1, x_2, ..., x_{k+q}$ are $L_{x_1}, L_{x_2}, \ldots, L_{x_{k+q}}$, respectively. Let x_h denote the AND term, i.e., $x_h = x_{k+1}x_{k+2}\cdots x_{k+q}$, such that we get $P(x_h^1) = p^1_{x_{k+1}}p^1_{x_{k+2}}\cdots p^1_{x_{k+q}}$, $P(x_h^0) = 1 - P(x_h^1)$. The equivalent incoming LLR value of x_h, denoted as L_{x_h}, can be computed as:

$$\begin{aligned} L_{x_h} &= \log \frac{1 - p^1_{x_{k+1}}p^1_{x_{k+2}}\cdots p^1_{x_{k+q}}}{p^1_{x_{k+1}}p^1_{x_{k+2}}\cdots p^1_{x_{k+q}}} \\ &= \log\left((1 + e^{L_{x_{k+1}}})(1 + e^{L_{x_{k+2}}})\cdots(1 + e^{L_{x_{k+q}}}) - 1\right). \end{aligned} \tag{28}$$

The outgoing LLR value of a linear term, say x_1, can thus be computed as:

$$L_{x_1} = L_{x_2} \boxplus L_{x_3} \boxplus \cdots \boxplus L_{x_h}. \tag{29}$$

Similarly, if $y = 1$, L_{x_1} should be the negative value. The outgoing LLR values for other linear terms can be derived in the same way. We next show how to update the LLR value of a variable involved in the AND term, say x_{k+1} without loss of generality. We denote the LLR value of the linear part $x_l = x_1 + x_2 + \cdots + x_k$ as L_{x_l}, which can be computed as $L_{x_l} = L_{x_1} \boxplus L_{x_2} \boxplus \cdots \boxplus L_{x_k}$.

Denote the product of other variables in the AND term excluding x_{k+1} as x'_h, i.e., $x'_h = x_{k+2}x_{k+3}\cdots x_{k+q}$. For the combinations (x_l, x_{k+1}, x'_h), the possible values to validate the check are $(y, 0, 0)$, $(y, 0, 1)$, $(y, 1, 0)$, $(y+1, 1, 1)$. The updated LLR value for x_{k+1} can then be computed as:

$$\begin{aligned} L_{x_{k+1}} &= \log \frac{p^y_{x_l} p^0_{x_{k+1}} p^0_{x'_h} + p^y_{x_l} p^0_{x_{k+1}} p^1_{x'_h}}{p^y_{x_l} p^1_{x_{k+1}} p^0_{x'_h} + p^{y+1}_{x_l} p^1_{x_{k+1}} p^1_{x'_h}} \\ &= \log \frac{p^0_{x_{k+1}}}{p^1_{x_{k+1}}} + \log \frac{p^y_{x_l}}{p^y_{x_l} p^0_{x'_h} + p^{y+1}_{x_l} p^1_{x'_h}}. \end{aligned} \tag{30}$$

With the "intrinsic" part excluded, i.e., the first term in the second line of (30), the outgoing LLR value of x_{k+1} can be computed as

$$L_{x_{k+1}} = \log \frac{p^y_{x_l}}{p^y_{x_l} p^0_{x'_h} + p^{y+1}_{x_l} p^1_{x'_h}}, \tag{31}$$

where $p^1_{x_{h'}} = p^1_{x_{k+2}} p^1_{x_{k+3}} ... p^1_{x_{k+q}}$ and $p^0_{x_{h'}} = 1 - p^1_{x_{h'}}$. For an equation having only the higher-degree term, e.g., $x_{k+1}x_{k+2}\ldots x_{k+q} = 0$, we get $L_{x_{k+1}} = \log \frac{1}{p^0_{x'_h}}$.

Experimental results We investigate several challenging XOR$_k$-AND$_q$ predicates with $k = 2q$, but for relatively small localities, since iterative decoding typically applies to sparse systems. Specifically, we investigate the following three concrete predicates:

$$\text{XOR}_4 - \text{AND}_2 : x_1 + x_2 + x_3 + x_4 + x_5x_6, \tag{32}$$

$$\text{XOR}_6 - \text{AND}_3 : x_1 + x_2 + \cdots + x_6 + x_7x_8x_9, \tag{33}$$

$$\text{XOR}_8 - \text{AND}_4 : x_1 + x_2 + \cdots + x_8 + x_9x_{10}x_{11}x_{12}. \tag{34}$$

Figure 17 shows the success probabilities of the guess-and-decode attack applied on these three predicates under certain stretches. Basically, the attack works better when the locality is lower. For example, the attack applies to XOR$_4$-AND$_2$ well, while not so good to XOR$_6$-AND$_3$ and XOR$_8$-AND$_4$ predicates. We could get the attacking complexities for these parameters using (27), and results show that XOR$_4$-AND$_2$ under parameter sets $(512, 1.3)$ and $(1024, 1.3)$ cannot provide security levels of 80 bits and 128 bits, respectively; while XOR$_6$-AND$_3$ under $(512, 1.4)$, and XOR$_8$-AND$_4$ under $(512, 1.45)$ cannot achieve 128 bits of security.

Challenge Parameters. We further give some challenge parameters for providing 128-bit security following the way in Section 4.6. We tried both attacks and the guess-and-determine attack requires stricter limits as expected. For example, when $n = 1024$, the challenge limit for XOR$_6$-AND$_3$ is 1.17, while 1.51 under the guess-and-decode attack. Table 9 presents more results and all of them are from applying the guess-and-determine

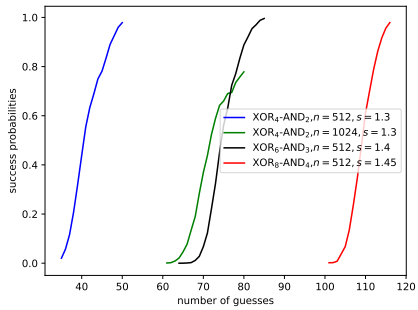

Figure 17: Success probabilities for XOR-AND predicates under different parameters.

attack, over 5000 independent instances for each (n, s, ℓ) set. The possibly safe stretches are higher than those for P_5, which indicate that these predicates have better resistance against our attacks, especially against the guess-and-decode one. Further study is encouraged to gain more confidence in these parameters.

predicate	512	1024	2048	4096
$\text{XOR}_4\text{-AND}_2$	-	1.10	1.20	1.28
$\text{XOR}_6\text{-AND}_3$	1.04	1.17	1.27	1.34
$\text{XOR}_8\text{-AND}_4$	1.05	1.16	1.25	1.32

Table 9: Challenge Parameters for XOR-AND Predicates

5.2 Extensions to XOR-THR Predicates

In [AL18], the authors show that predicates with high resiliency and high degrees are not sufficient for constructing local PRGs. They proposed a new criterion called rational degree against algebraic attacks and suggested XOR-MAJ (majority) predicates, which can be generalized to the XOR-THR predicates. In [MCJS19], the properties of XOR-THR predicates are investigated and the special case XOR-MAJ predicates are used to build the new version of FLIP, called FiLIP, which is a construction designed for homomorphic encryption. It is mentioned in the paper that "no attack is known relatively to the functions $\text{XOR}_k\text{-THR}_{d,2d}$ or $\text{XOR}_k\text{-THR}_{d,2d-1}$ since $k \geq 2s$ and $d \geq s$". These predicates are actually $\text{XOR}_k\text{-MAJ}_d$ predicates.

For the guessing phase, the cases how free variables are obtained differs a bit compared to the XOR-AND predicates. Specifically, there are two cases a free variable could be obtained:

(1) For an equation with one linear term and a THR term, if the value of the THR term is known, for example, there are already not less than d one's or more than $q - d$ zero's in the THR term, the linear term can be freely derived;

(2) For an equation which only has a THR term, if there already exist $d-1$ one's in the THR term while the value for the equation is zero, every other variable could be derived as zero for free; on the other hand, if there are $q-d$ zero's while the value of the equation is one, all the other variables could be derived as one for free.

Obviously, it gets more complex for both deriving a "free" variable and linear equation for an XOR-THR predicate, as eliminating a THR term involves more steps. We tested the guess-and-determine attack against XOR-MAJ predicates, but the performance is not satisfying as expected. Thus we focus on the guess-and-decode attack against XOR-MAJ predicates below.

After the guessing phase, the iterative decoding is performed. For an equation $x_1 + \cdots + x_k + \mathrm{THR}(x_{k+1}, \ldots, x_{k+q}) = y$, suppose the incoming LLR values of the variables are $L_{x_1}, L_{x_2}, \ldots, L_{x_{k+q}}$, respectively. When computing the outgoing LLR values for the linear terms, say x_1 without loss of generality, we need to first compute the equivalent incoming LLR value of the THR term. We denote the THR term as x_t and its LLR value as L_{x_t}. The number of the combinations of $(x_{k+1}, \ldots, x_{x+q})$ that make $\mathrm{THR}(x_{k+1}, \ldots, x_{x+q})$ one is $\sum_{w=d}^{q} \binom{q}{w}$. If we denote the set of these combinations as $\mathcal{S}$, we can get

$$P(x_t^1) = \sum_{(c_{k+1}, \ldots, c_{x+q}) \in \mathcal{S}} \prod_{i=1}^{q} p_{x_{k+i}}^{c_{k+i}}, \tag{35}$$

where $(c_{k+1}, \ldots, c_{x+q})$ are the possible values in $\mathcal{S}$ and every value is either one or zero. $P(x_t^0) = 1 - P(x_t^1)$ and thus $L_{x_t} = \log \frac{P(x_t^0)}{P(x_t^1)}$. The LLR value of x_1 can be computed as $L_{x_1} = L_{x_2} \boxplus \cdots \boxplus L_{x_k} \boxplus L_{x_t}$.

If there are some variables in the $\mathrm{THR}(x_{k+1}, \ldots, x_{x+q})$ term being fixed, e.g., guessed or determined during the guessing phase, we can just fix its probability of being the fixed value as one, while zero for the complement value.

We next show how to compute the outgoing LLR value of a variable in the $\mathrm{THR}(x_{k+1}, \ldots, x_{x+q})$ term, say x_{k+1} without loss of generality. Let x_l denote the linear part, i.e., $x_l = x_1 + \cdots + x_k$, then the LLR value of it, denoted L_{x_l}, is derived as $L_{x_l} = L_{x_1} \boxplus \cdots \boxplus L_{x_k}$. Denote the set of combinations of $(x_{k+2}, x_{k+3}, \ldots, x_{x+q})$ that have no less than $d-1$ one's as $\mathcal{W}$, and $\bar{\mathcal{W}}$ for the complement set. $P(W), P(\bar{W})$ can be computed using the same way as in (35). Then we would get:

$$\begin{aligned} &P(x_{k+1} = 1) \\ &= p(x_{k+1} = 1, x_l = y+1, \mathcal{W}) + p(x_{k+1} = 1, x_l = y, \bar{\mathcal{W}}) \\ &= p_{x_{k+1}}^1 \cdot p_{x_l}^{y+1} \cdot P(\mathcal{W}) + p_{x_{k+1}}^1 \cdot p_{x_l}^{y} \cdot P(\bar{\mathcal{W}}) \end{aligned} \tag{36}$$

Denote the set of combinations of $(x_{k+2}, x_{k+3}, \ldots, x_{x+q})$ that have no less than d one's as $\mathcal{V}$, and $\bar{\mathcal{V}}$ for the complement set. We can get the probability of $p(x_{k+1} = 0)$ using the same way as below:

$$\begin{aligned} &P(x_{k+1} = 0) \\ &= p(x_{k+1} = 0, x_l = y+1, \mathcal{V}) + p(x_{k+1} = 0, x_l = y, \bar{\mathcal{V}}) \\ &= p_{x_{k+1}}^0 \cdot p_{x_l}^{y+1} \cdot P(\mathcal{V}) + p_{x_{k+1}}^0 \cdot p_{x_l}^{y} \cdot P(\bar{\mathcal{V}}) \end{aligned} \tag{37}$$

With the "intrinsic" part being excluded, the outgoing LLR value of x_{k+1} can be computed as

$$L_{x_{k+1}} = \frac{p_{x_l}^{y+1} \cdot P(\mathcal{V}) + p_{x_l}^{y} \cdot P(\bar{\mathcal{V}})}{p_{x_l}^{y+1} \cdot P(\mathcal{S}) + p_{x_l}^{y} \cdot P(\bar{\mathcal{S}})}. \tag{38}$$

The LLR values of other variables in the THR term can be derived using the same way. One can see that when q is large, computing the combinations would require large overhead, introducing better resistance against our attack. Thus we only consider four concrete XOR-THR predicates, which are actually XOR-MAJ predicates: XOR_3-MAJ_3, XOR_3-MAJ_4, XOR_4-MAJ_3, and XOR_4-MAJ_4.

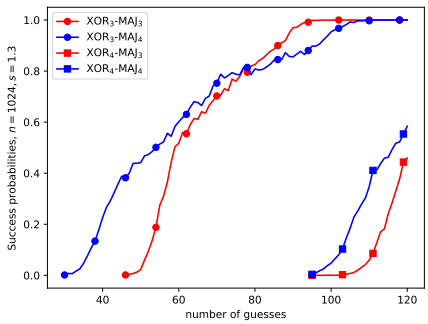

Figure 18: Success probabilities for XOR-MAJ predicates when $n = 1024, s = 1.3$.

Figure 18 shows the success probabilities under different numbers of guesses when $n = 1024, s = 1.3$. We can see that our guess-and-decode attack applies to these predicates well. Using (27), we can get that when $n = 1024, s = 1.3$, local PRGs instantiated on XOR_4-MAJ_3, XOR_4-MAJ_4 predicates cannot achieve 128 bits of security, while XOR_3-MAJ_3, XOR_3-MAJ_4 cannot even achieve 80 bits of security.

Discussion. One important observation from Figure 18 is that the number of linear terms matters more than the number of variables in the MAJ term. For example, the differences of success probabilities between predicates with a same number of linear terms, e.g., XOR_4-MAJ_3 and XOR_4-MAJ_4, are much smaller than those between predicates with a same number of variables in the MAJ term, e.g., XOR_3-MAJ_3 and XOR_4-MAJ_3. This makes sense since iterative decoding applies well to sparse systems, and it is more sensitive to the number of terms in a check: the MAJ term can be regarded as one special term. Another interesting observation is that predicates XOR_i-MAJ_3 are more resistant against our attack compared to XOR_i-MAJ_4, $i \in [3, 4]$. This could be because that the predicates are balanced when $q = 2d - 1$, while not so when $q = 2d$. It is pointed out in [MCJS19] that the *resiliency* of a XOR_k-$\text{THR}_{d,q}$ is k if $q = 2d-1$, while $k - 1$, otherwise. The predicates used in the suggested FiLIP instances in [MCJS19] are all of the type XOR_k-$\text{THR}_{d,2d-1}$. Our results match well with the analysis in [MCJS19] and serve as a direct illustration of how the *resiliency* of a predicate would affect its security.

Challenge Parameters. We also ran extensive experiments to get some challenge parameters following the way in Section 4.6 for providing 128-bit security. Table 10 shows the results and all of them are from applying the guess-and-decode attack. Size 512 is not suitable for constructing efficient local PRGs instantiated on these given predicates, as the safe stretches would be smaller than 1. For each (n, s, ℓ) set, we only run 2000 instances as the iterative decoding takes some time, and more cryptanalysis are encouraged.

n	XOR_3-MAJ_3	XOR_3-MAJ_4	XOR_4-MAJ_3	XOR_4-MAJ_4
1024	1.07	1.04	1.25	1.22
2048	1.23	1.16	1.48	1.42

Table 10: Challenge Parameters for Different XOR-MAJ Predicates.

6 Concluding Remarks

We have presented a novel guess-and-determine attack and a guess-and-decode attack on Goldreich's pseudorandom generators instantiated on the P_5 predicate, greatly improving the attack proposed in [CDM+18]. Both attacks work based on similar guessing strategies: we try to explore as many variables which can be determined for free as possible. In the guess-and-decode attack, we use a modified iterative decoding method to solve the resulting system after guessing a certain number of variables, which provides a new idea to solve a sparse quadratic system. We broke the candidate non-vulnerable parameters given in [CDM+18] with a large gap and suggested some new challenge parameters which could be targets for future investigation.

The attacks further narrow the concrete stretch regime of Goldreich's pseudorandom generators instantiated on the P_5 predicate and largely shake the confidence in their efficiency when the seed sizes are small.

We further extend the attacks to investigate some other predicates of the XOR-AND and XOR-MAJ type, which are suggested as research target for constructing local PRGs. Generally, local PRGs instantiated over predicates with low localities show susceptibility to our attacks. It is safer to have more terms if large stretches are desired. For the nonlinear part of a predicate, it is better to be balanced than being unbalanced to resist against our attacks. If in some extreme cases local PRGs instantiated on predicates with low localities are required, our attacks could be helpful for choosing a safe stretch. The attacks might apply to other predicates with similar structures as well.

Acknowledgment

The authors would like to thank the reviewers and the associate editor Anne Canteaut for their valuable comments and instructive suggestions that made it possible to improve this paper to a great extent. This work was in part financially supported by the Swedish Foundation for Strategic Research, grant RIT17-0005 and the ELLIIT program. Jing

Yang is also supported by the scholarship from the National Digital Switching System Engineering and Technological Research Center, China.

References

[ABR16] Benny Applebaum, Andrej Bogdanov, and Alon Rosen. A dichotomy for local small-bias generators. *Journal of Cryptology*, 29(3):577–596, 2016.

[ADI+17] Benny Applebaum, Ivan Damgård, Yuval Ishai, Michael Nielsen, and Lior Zichron. Secure arithmetic computation with constant computational overhead. In *Annual International Cryptology Conference*, pages 223–254. Springer, 2017.

[AHI06] Michael Alekhnovich, Edward A Hirsch, and Dmitry Itsykson. Exponential lower bounds for the running time of DPLL algorithms on satisfiable formulas. In *SAT 2005*, pages 51–72. Springer, 2006.

[AIK06] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Cryptography in NC^0. *SIAM Journal on Computing*, 36(4):845–888, 2006.

[AIK08] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. On pseudorandom generators with linear stretch in NC^0. *Computational Complexity*, 17(1):38–69, 2008.

[AL18] Benny Applebaum and Shachar Lovett. Algebraic attacks against random local functions and their countermeasures. *SIAM Journal on Computing*, 47(1):52–79, 2018.

[App13] Benny Applebaum. Pseudorandom generators with long stretch and low locality from random local one-way functions. *SIAM Journal on Computing*, 42(5):2008–2037, 2013.

[App16] Benny Applebaum. Cryptographic hardness of random local functions. *Computational complexity*, 25(3):667–722, 2016.

[ARS+15] Martin R Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 430–454. Springer, 2015.

[BCG+17] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, and Michele Orrù. Homomorphic secret sharing: optimizations and applications. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2105–2122, 2017.

[BCG+19] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In *Annual International Cryptology Conference*, pages 489–518. Springer, 2019.

[BCG+20] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from Ring-LPN. In *Annual International Cryptology Conference*, pages 387–416. Springer, 2020.

[BQ09] Andrej Bogdanov and Youming Qiao. On the security of Goldreich's one-way function. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 392–405. Springer, 2009.

[CCF+18] Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrède Lepoint, María Naya-Plasencia, Pascal Paillier, and Renaud Sirdey. Stream ciphers: A practical solution for efficient homomorphic-ciphertext compression. *Journal of Cryptology*, 31(3):885–916, 2018.

[CDM+18] Geoffroy Couteau, Aurélien Dupin, Pierrick Méaux, Mélissa Rossi, and Yann Rotella. On the concrete security of Goldreich's pseudorandom generator. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 96–124. Springer, 2018.

[CM01] Mary Cryan and Peter Bro Miltersen. On pseudorandom generators in NC^0. In *International Symposium on Mathematical Foundations of Computer Science*, pages 272–284. Springer, 2001.

[GJLS20] Romain Gay, Aayush Jain, Huijia Lin, and Amit Sahai. Indistinguishability obfuscation from simple-to-state hard problems: New assumptions, new techniques, and simplification. *IACR Cryptol. ePrint Archive, Report 2020/764*, 2020.

[Gol00] Oded Goldreich. Candidate one-way functions based on expander graphs. *IACR Cryptology ePrint Archive, Report 2000/63*, 2000.

[GRR+16] Lorenzo Grassi, Christian Rechberger, Dragos Rotaru, Peter Scholl, and Nigel P Smart. MPC-friendly symmetric key primitives. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 430–443, 2016.

[HOP96] Joachim Hagenauer, Elke Offer, and Lutz Papke. Iterative decoding of binary block and convolutional codes. *IEEE Transactions on information theory*, 42(2):429–445, 1996.

[IKOS08] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Cryptography with constant computational overhead. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 433–442, 2008.

[LT17] Huijia Lin and Stefano Tessaro. Indistinguishability obfuscation from trilinear maps and block-wise local PRGs. In *Annual International Cryptology Conference*, pages 630–660. Springer, 2017.

[MCJS19] Pierrick Méaux, Claude Carlet, Anthony Journault, and François-Xavier Standaert. Improved filter permutators for efficient FHE: better instances

and implementations. In *International Conference on Cryptology in India*, pages 68–91. Springer, 2019.

[MJSC16] Pierrick Méaux, Anthony Journault, François-Xavier Standaert, and Claude Carlet. Towards stream ciphers for efficient FHE with low-noise ciphertexts. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 311–343. Springer, 2016.

[MST03] Elchanan Mossel, Amir Shpilka, and Luca Trevisan. On ϵ-biased generators in NC^0. In *Annual Symposium on Foundations of Computer Science*, volume 44, pages 136–145. Citeseer, 2003.

[OW14] Ryan ODonnell and David Witmer. Goldreich's PRG: evidence for near-optimal polynomial stretch. In *2014 IEEE 29th Conference on Computational Complexity (CCC)*, pages 1–12. IEEE, 2014.

[RL09] William Ryan and Shu Lin. *Channel codes: classical and modern*. Cambridge university press, 2009.