Analysis of Nonlinear Sequences and Stream Ciphers

by

Sui-Guan Teo

Bachelor of Information Technology with Distinction *(QUT)* – 2007
Bachelor of Information Technology (First Class Honours) *(QUT)* – 2008

Thesis submitted in accordance with the regulations for the Degree of Doctor of Philosophy

Institute for Future Environments
Science and Engineering Faculty
Queensland University of Technology

7th March 2013

Keywords

Stream ciphers, keystream generators, linear feedback shift register (LFSR), nonlinear feedback shift register (NLFSR), clock-control, Boolean functions, state-update functions, output functions, keystream sequence properties, nonlinear filter generator, linearly filtered NLFSR, slid pairs, A5/1, Trivium, Mixer, summation generator, state convergence, cryptanalysis, time-memory-data tradeoff attacks, algebraic attacks, F4 algorithm, Gröbner basis

Abstract

Stream ciphers are common cryptographic algorithms used to protect the confidentiality of frame-based communications like mobile phone conversations and Internet traffic. Stream ciphers are ideal cryptographic algorithms to encrypt these types of traffic as they have the potential to encrypt them quickly and securely, and have low error propagation.

The main objective of this thesis is to determine whether structural features of keystream generators affect the security provided by stream ciphers. These structural features pertain to the state-update and output functions used in keystream generators. Using linear sequences as keystream to encrypt messages is known to be insecure. Modern keystream generators use nonlinear sequences as keystream. The nonlinearity can be introduced through a keystream generator's state-update function, output function, or both.

The first contribution of this thesis relates to nonlinear sequences produced by the well-known Trivium stream cipher. Trivium is one of the stream ciphers selected in a final portfolio resulting from a multi-year project in Europe called the ECRYPT project. Trivium's structural simplicity makes it a popular cipher to cryptanalyse, but to date, there are no attacks in the public literature which are faster than exhaustive keysearch. Algebraic analyses are performed on the Trivium stream cipher, which uses a nonlinear state-update and linear output function to produce keystream. Two algebraic investigations are performed: an examination of the sliding property in the initialisation process and algebraic analyses of Trivium-like stream ciphers using a combination of the algebraic techniques previously applied separately by Berbain et al. and Raddum. For certain iterations of Trivium's state-update function, we examine the sets of slid pairs, looking particularly to form chains of slid pairs. No chains exist for a small number of iterations. This has implications for the period of keystreams produced by Trivium. Secondly, using our combination of the methods of Berbain et al. and Raddum, we analysed Trivium-like ciphers and improved on previous on previous analysis with regards to forming systems of equations on these ciphers. Using these new systems of

equations, we were able to successfully recover the initial state of Bivium-A. The attack complexity for Bivium-B and Trivium were, however, worse than exhaustive keysearch. We also show that the selection of stages which are used as input to the output function and the size of registers which are used in the construction of the system of equations affect the success of the attack.

The second contribution of this thesis is the examination of state convergence. State convergence is an undesirable characteristic in keystream generators for stream ciphers, as it implies that the effective session key size of the stream cipher is smaller than the designers intended. We identify methods which can be used to detect state convergence. As a case study, the Mixer stream cipher, which uses nonlinear state-update and output functions to produce keystream, is analysed. Mixer is found to suffer from state convergence as the state-update function used in its initialisation process is not one-to-one. A discussion of several other stream ciphers which are known to suffer from state convergence is given. From our analysis of these stream ciphers, three mechanisms which can cause state convergence are identified. The effect state convergence can have on stream cipher cryptanalysis is examined. We show that state convergence can have a positive effect if the goal of the attacker is to recover the initial state of the keystream generator.

The third contribution of this thesis is the examination of the distributions of bit patterns in the sequences produced by nonlinear filter generators (NLFGs) and linearly filtered nonlinear feedback shift registers. We show that the selection of stages used as input to a keystream generator's output function can affect the distribution of bit patterns in sequences produced by these keystream generators, and that the effect differs for nonlinear filter generators and linearly filtered nonlinear feedback shift registers. In the case of NLFGs, the keystream sequences produced when the output functions take inputs from consecutive register stages are less uniform than sequences produced by NLFGs whose output functions take inputs from unevenly spaced register stages. The opposite is true for keystream sequences produced by linearly filtered nonlinear feedback shift registers.

For my parents

Contents

List of Figures

List of Tables

List of Acronyms

AE	Authenticated Encryption
AES	Advanced Encryption Standard
AND	Multiplication Modulo 2
ANF	Algebraic Normal Form
CONS	Consecutive
DK	Dunkelman and Keller
FCSR	Feedback with Carry Shift Register
FPDS	Full Positive Difference Set
HPC	High Performance Computing
HS	Hong and Sarkar
IV	Initialisation Vector
LC	Linear Complexity
LFSR	Linear Feedback Shift Register
MAC	Message Authentication Code
NIST	National Institute of Standards and Technology
NLFG	Nonlinear Filter Generator
NLFSR	Nonlinear Feedback Shift Register
QUT	Queensland University of Technology
RAM	Random Access Memory
S.D	Standard Deviation
TMDT	Time-memory-data-tradeoff
XOR	Addtional Modulo 2

Declaration

The work contained in this thesis has not been previously submitted for a degree or diploma at any higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Signed: Tee Sullivan ..... **Date:** 4/3/2013 .....

Previously Published Material

The following papers have been published or presented, and contain material based on the content of this thesis.

[1] Sui-Guan Teo, Kenneth Koon-Ho Wong, Ed Dawson, and Leonie Simpson. State convergence and keyspace reduction of the Mixer stream cipher. *Journal of Discrete Mathematical Sciences & Cryptography*, 15(1):89–104, 2012.

[2] Sui-Guan Teo, Leonie Simpson, Kenneth Koon-Ho Wong, and Ed Dawson. State Convergence and the effectiveness of Time-Memory-Data Tradeoffs. In Ajith Abraham, Daniel Zheng, Dharma Agrawal, Mohd Faizal Abdollah, Emilio Corchado, Valentina Casola, and Choo Yun Choy, editors, *Proceedings of the 7th International Conference on Information Assurance and Security (IAS 2011)*, pages 92–97. IEEE, 2011. Updated version available from `http://eprints.qut.edu.au/47843/`.

[3] Sui-Guan Teo, Ali Al-Hamdan, Harry Bartlett, Leonie Simpson, Kenneth Koon-Ho Wong, and Ed Dawson. State Convergence in the Initialisation of Stream Ciphers. In Udaya Parampalli and Phillip Hawkes, editors, *Information Security and Privacy (ACISP 2011)*, volume 6812 of *Lecture Notes in Computer Science*, pages 75–88. Springer, 2011.

[4] Sui-Guan Teo, Leonie Simpson, and Ed Dawson. Bias in the Nonlinear Filter Generator Output Sequence. In Muhammad Rezal Kamel Ariffin, Rabiah Ahmad, Mohamad Rushdan Md. Said, Bok Min Goi, Swee Huay Heng, Nor Azman Abu, and Mohd Zaki Mas'ud, editors, *Proceeding of Cryptology 2010; The Second International Cryptology Conference*, pages 40–46, 2010.

Acknowledgements

The byline of this thesis contains the name of one individual. Mine. I wish I could have included, in the same byline, all the names of all the individuals who have supported and accompanied me on a journey which has lasted almost $4\frac{1}{2}$ years, but I think the university would have none of it. Therefore the only way I can acknowledge their help is in this section. Even then, I am not sure the words written here fully express my gratitude to them, but here, nevertheless, is my humble attempt.

My interest in information security was first seeded when I enrolled in the Information Security Fundamentals, and Network Security units taught by Dr. Greg Maitland. If not for Greg, my interest in information security would not have kindled, and I would not have started writing this thesis, let alone complete it.

I will be forever grateful to my supervisory team: Dr. Leonie Simpson, Professor Emeritus Ed Dawson, and Dr. Kenneth Wong. Leonie is one of the best academic wordsmith I have had the privilege of working with, and I value her comments which have vastly improved the readability of my papers and this thesis. Ed has also provided excellent guidance during the course of this thesis and suggested the change in research topic when my research direction did not seem to fit my original research plans. Kenneth joined my supervisory team in the middle of my PhD, when my thesis seemed destined to have an algebraic flair to it. I am grateful to him for his tips for the Magma mathematical software. If not for his help, the results in Sections 4.2 and 4.3 will not have been possible. I am indebted to the innumerable suggestions they made over the years. I particularly want to thank them for their support during the difficult month of April 2012.

I would like to thank Dr. Harry Bartlett, Dr. Ernest Foo, Associate Professor Diane Donovan and Dr. Udaya Parampalli for taking part in the examination reading committee and providing many useful suggestions for improving the quality of the thesis, particually to Harry for his suggestions to improve the contents of Section 4.3 and Chapters 5–6.

I am extremely grateful for the financial support invested in me and this work. I sincerely thank the now-defunct Faculty of Information Technology and the (also defunct) Information Security Institute (ISI) for awarding me the Faculty of Information Technology Postgraduate Scholarship, which allowed me to put food on my (work) table and a roof over my head. I would also like to thank the Queensland University of Technology (QUT) for their fee-waiver scholarship, and travel grants which have allowed me to attend conferences overseas and in Australia.

The execution of some of the computer experiments in this thesis: the experiments in Chapter 5, and especially the experiments which required the Magma Computational Algebra System in Sections 4.2–4.3, would not have been possible without the computational facilities provided by the QUT's High Performance Computing & Research Support Centre (HPC). I would particularly like to extend my gratitude to Mr. Mark Barry and the staff from QUT HPC for their technical help in setting up Magma on the LYRA supercomputer.

I would like to acknowledge some of the work done in this thesis. The work on the estimation of the initial state space in the original proposal of A5/1 in Section 6.2.1 is solely the work of a fellow PhD student, Ali Alhamdan, and is included in this thesis as part of a discussion on state convergence.

Many thanks goes to my friends and colleagues at the ISI and IFE. Fellow PhD students like Choudary, Kaleb, Ken, Kush, Sajal and Vik Tor; staff like Andrew Clark, Edward, Gleb, Jason Smith and Juanma are just some of the people who shared the same office as me.

Marianne Hirschbichler occupied the desk just behind mine for a couple of years. I will remember Marianne for her advice when things were not that rosy, and for talking very excitedly in German in her almost-daily calls back to her home country, Austria; and James Birkett for his (sometimes) irreverent humour.

Hüseyin Hışıl and Georg Lippold were instrumental on convincing me to switch to using Linux for my research; I shudder to think about what might have been if I had kept on using a Windows machine to do my research. Georg, Hüseyin and Muhammad Reza Z'aba graciously helped me with any programming problems I encountered; I thank them for their assistance.

Mark Branagan and Farzad Salim both endured my complaints about how bad I thought my research progress was with good grace, and continuously encouraged me. An honourable mention goes to Mark for his sage-like advice that a thesis has to include: (1) Equations, (2) Tables, (3) Figures, (4) Graphs, (5) References, and (6) Footnotes.

Special thanks goes to Verona, who without complaint, brewed me a cup of cafè latte regardless of whether it was 1 a.m. or 1 p.m., 365 days a year.

Friends outside of my office were also instrumental in helping me maintain my sanity. The QUT Singapore Students Association (SSA) has over the years, provided an excellent support base for Singaporean students in Queensland and has facilitated the forging of new (and hopefully, life-long) friendships. I acknowledge the hard work put in by the QUT SSA committees, both past and present, for organising activities for its members, activities which formed part of my social life. The list of friends I have made (directly or indirectly) through QUT SSA is too long to mention, so here is a SHA-3 (Keccak-256) [12] hash value of the said list:

`0x72ea05727925e4baa4445b6783883b9c630c65f0dd7b3379e8f1fa4c11b08679.`

Thanks to all those friends who had encouraged me and had the irrepressible belief that the light at the end of the tunnel was not that of an oncoming train. I am grateful to Samuel and Steve for their help during the Riverfire weekend of 2010. To the lunch/dinner kakis[1], which include but are not limited to: Anna, Desmond, Eunice, Gareth, James Chew, Johnny, Kai Hui, Leon Ng, Natalie, Simon, Steve, Terrix, and Vanda — Thank you making lunch/dinner a less lonely affair.

I want to acknowledge the friendships of Dilys, Hermann, and Yinghui. It really means a lot to me.

The Germans have a saying: "*Blut ist dicker als Wasser*", or as more commonly known in English-speaking countries: "*Blood is thicker than water*". Truer words have never been spoken. My aunts, uncles, and cousins have unfailingly encouraged me over the years and I thank them for their support over the years.

Last, but not least, I thank my parents and sister for their love and unending support through the years, especially to my late father who did not live long enough to see me through to the end of this study.

[1]Kakis (its pronunciation is very similar to *car keys*) is a Singaporean colloquialism for companions. Incidentally, this is the only footnote in the entire thesis.

Chapter 1

Introduction

In the age of digital communications, stream ciphers play an important role to protect transmitted data. Examples of these digital telecommunications include mobile phone communication and Internet traffic. Users of these telecommunications require the confidential transmission of data. For example, two parties calling each other using mobile phones require that no one can eavesdrop on their conversations. A user buying items over the Internet may be required to send their credit card details to an online store. In this case, the user needs an assurance that their credit card details are not read by a third party, who could then use the credit card to make unauthorised purchases.

These forms of telecommunications typically consist of a series of frames which are sent between the two communicating parties. Each frame is encrypted using a secret key which has been pre-established between two parties prior to the transmission of these frames and a publicly known value called the initialisation vector (IV). In frame-based applications, the frame number is typically used as the IV. The stream cipher uses this key-IV pair to produce keystream, which is combined with the frame (plaintext) to produce an encrypted frame, also known as a ciphertext. This key should be long enough to preclude exhaustive keysearch. The process of producing a ciphertext is called encryption, while the reverse operation of producing a plaintext given a ciphertext, is called decryption.

The frame-based nature of modern telecommunications make stream ciphers ideal cryptographic algorithms as they are generally able to encrypt frame-based traffic faster than block ciphers. The requirement for stream ciphers to be faster than block ciphers is evident during the eSTREAM project [44], a multi-year project which identified stream

ciphers suitable for widespread adoption. One of the requirements in eSTREAM is that any proposed stream cipher must be demonstrably superior to the AES (when used in some appropriate mode, like counter mode) in at least one significant aspect [44]. However, the speed in which stream ciphers can encrypt frame-based traffic is not the only criterion for selecting a stream cipher. The stream cipher's security properties also needs to be taken into consideration. For example, early keystream generators for stream ciphers produced sequences which had linear mathematical relationships between sequence bits. This could be exploited in attacks [82]. Using functions which are not linear makes these attacks more difficult.

Due to the insecurity of encrypting messages using linear keystream sequences, keystreams produced by modern keystream generators have to be nonlinear. However, the properties of sequences produced by modern stream ciphers are generally not known. For example, the period of sequences produced by modern stream ciphers may not have a fixed value, and designers of stream ciphers usually give an estimate of what they believe this period may be. Another property of keystream sequences produced by keystream generators which are not well-understood are the distribution of patterns in the keystream sequences. In a truly random sequence, these patterns have a uniform distribution, but in keystream sequences produced by keystream generators, this may not be the case.

1.1 Aims and objectives

The main objective of this thesis is to determine whether structural features of keystream generators affect the security provided by stream ciphers. Modern keystream generators produce nonlinear sequences, as they are considered more secure than linear sequences. This nonlinearity can be introduced through a stream cipher's state-update function, output function, or both. To achieve this aim the research is split into the following tasks:

1. The determination of security implications of nonlinear sequence properties for Trivium [26]. Trivium is one of the stream ciphers selected in a final portfolio resulting from the ECRYPT project. Trivium's simplicity makes it a popular cipher to cryptanalyse, but to date, there are no attacks in the public literature which are faster than exhaustive keysearch. In this thesis, we perform algebraic analyses on Trivium's state-update function and analyse its resistance to certain algebraic attacks.

2. The investigation of the state convergence problem in stream ciphers in detail. The research on state convergence in stream ciphers has three objectives:

   - The identification of methods which can be used to detect state convergence.
   - The identification of mechanisms used in stream ciphers which can cause state convergence.
   - An investigation into the effect state convergence can have on stream cipher cryptanalysis.

   The presence of state convergence in a stream cipher indicates a potential weakness which may be exploited in a key-recovery attack, as was demonstrated in attacks on the stream ciphers Py, Pypy [73, 112, 113] and ZUC [110].

3. An investigation into the distribution of bit patterns in keystream generators produced by:

   - nonlinear filter generators, and
   - (the complementary concept) linearly filtered nonlinear feedback shift registers.

   We analyse how the selection of different:

   - stages used as input to the output function, or
   - feedback functions used in the keystream generator

   can affect this bit pattern distribution. The presence of significant bit pattern biases in the output sequence may be exploited in attacks ranging from distinguishing attacks [80] to ciphertext-alone attacks [37].

1.2 Results

This thesis contains the following contributions to knowledge regarding stream cipher analyses.

1.2.1 Contributions of Chapter 3

Some of the properties of sequences generated by keystream generators which employ linear state-update and nonlinear output functions are investigated. We specifically investigate the effect different stages used as input to the output function has the

distribution of m-bit patterns of keystreams produced by nonlinear filter generators (NLFGs). The almost uniform distributions of m-bit patterns for $m = 1$ was proved by Simpson [105], while the non-uniform distribution of for some m-bit patterns was demonstrated by Anderson [4]. However, factors influencing the non-uniform distribution observed by Anderson in m-bit patterns for keystream sequences produced by NLFGs are not known. We show that the m-bit patterns, for $m \geq 2$ of NLFGs are biased, regardless of the type of tap settings used, although the bias is generally greater when the tap settings to the filter function are consecutive. In some cases, there are some m-bit patterns which do not occur at all in the outputs. This happens for smaller values of m when the NLFGs use consecutive tap settings than when uneven tap settings are used, from $m \geq 8$ for consecutive and $m \geq 11$ for uneven tap settings, respectively. The experiments also show that the frequency distributions of m-bit patterns for NLFGs using consecutive tap settings are similar regardless of the size of the LFSR, but was not the same for NLFGs using uneven tap settings.
The contents of this chapter have appeared in the following publication:

- Sui-Guan Teo, Leonie Simpson, and Ed Dawson. Bias in the Nonlinear Filter Generator Output Sequence. In Muhammad Rezal Kamel Ariffin, Rabiah Ahmad, Mohamad Rushdan Md. Said, Bok Min Goi, Swee Huay Heng, Nor Azman Abu, and Mohd Zaki Mas'ud, editors, *Proceeding of Cryptology 2010; The Second International Cryptology Conference*, pages 40–46, 2010.

1.2.2 Contributions of Chapter 4

We analyse keystream generators which use a nonlinear state-update function to update the internal state and a linear output function to generate keystream. In addition to examining the distribution of m-bit patterns produced by these keystream generators, we analyse the well-known Trivium stream cipher [26] and variants with a similar structure. Two investigations are performed: the sliding property of its initialisation process is examined and algebraic analyses of Trivium-like stream ciphers is performed.

Contributions of Section 4.1

In Section 4.1, the distribution of m-bit patterns in keystream sequences formed by linearly filtered nonlinear feedback shift registers (NLFSRs) is examined. We specifically investigate the effect different stages used as input to the output function has the distribution of m-bit patterns for keystreams produced by linearly filtered nonlinear feedback

shift registers. Our findings indicate that the keystream formed by these generators can have a non-uniform distribution if the linear output function takes inputs from more than three stages in the nonlinear feedback shift register. Non-occurring m-bit patterns were also observed for all keystream generators used in our experiments.

Similar to the distributions of m-bit patterns in sequences produced by NLFGs, the distributions of m-bit patterns in keystream sequences produced by linearly filtered NLFSRs are influenced by tap settings to the linear output function. However, unlike NLFGs, the distribution of m-bit patterns of keystreams produced by linearly filtered NLFSRs, where the linear function takes as input, stages which form a Full Positive Difference Set (FPDS) are generally less uniformly distributed as compared to the distribution of m-bit patterns in the keystream produced by linear functions which take as input, taps which are consecutive. This is in contrast to the distribution of m-bit patterns of NLFGs.

Contributions of Section 4.2

In Section 4.2, we search for slid pairs in Trivium. We extend the work of Priemuth-Schmid and Biryukov [92] and Zeng and Qi [115] and search for particular types of slid pairs. We show that by forming a new system of equations, the size of the search space for these types of slid pairs can be significantly reduced. This reduces the time and memory requirements needed compared to searching for the same type of special slid pairs using Priemuth-Schmid and Biryukov's system of equations. We also show that particular groups of slid pairs in Trivium do not exist.

Contributions of Section 4.3

In Section 4.3, we perform algebraic analyses on Trivium-like ciphers using the combination of the techniques introduced by Raddum [93], and Berbain et al. [10]. We answer Berbain et al.'s open question regarding whether it is possible to extend their algebraic attack to ciphers which update q internal state bits at each iteration and only output q', where $q' < q$, linear combinations of state bits at each iteration. Our attack on Bivium-A is, to the best of our knowledge, the fastest initial state recovery attack using the F4 algorithm which uses the least amount of keystream. We show that the success of performing an algebraic divide-and-conquer attack on Trivium-like ciphers depends on the relationship between the number of registers in the Trivium-like cipher, and the distance between the stages used as inputs to the output function which generate keystream. By changing the taps positions to the output functions, attacks on Trivium-like

ciphers using the combined techniques by Raddum and Berbain et al. may be prevented. The influence of tap settings and the state size of registers whose stages are used in the construction of the system of equations are factors which affect the number of solutions obtained when the system of equations is solved. These factors are discussed in detail in Section 4.3.8.

1.2.3 Contributions of Chapter 5

The state-update functions of the Mixer stream cipher [79] are analysed. We show that it is possible for two or more distinct key-IV pairs to generate the same Mixer keystream due to state convergence during the initialisation process. As a consequence, the effective session key size of Mixer is reduced. We estimate that this effective key size, after 200 initialisation rounds, is between 2^{109} and 2^{191}. This reduction in effective key size continues during Mixer's keystream generation due to Mixer using a shrinking generator-like mechanism to generate keystream.
The contents of this chapter have appeared in the following publications:

- Sui-Guan Teo, Kenneth Koon-Ho Wong, Ed Dawson, and Leonie Simpson. State convergence and keyspace reduction of the Mixer stream cipher. *Journal of Discrete Mathematical Sciences & Cryptography*, 15(1):89–104, 2012.

- Sui-Guan Teo, Ali Al-Hamdan, Harry Bartlett, Leonie Simpson, Kenneth Koon-Ho Wong, and Ed Dawson. State Convergence in the Initialisation of Stream Ciphers. In Udaya Parampalli and Phillip Hawkes, editors, *Information Security and Privacy (ACISP 2011)*, volume 6812 of *Lecture Notes in Computer Science*, pages 75–88. Springer, 2011.

1.2.4 Contributions of Chapter 6

We study the problem of state convergence in greater detail. In Section 6.1, we identify techniques which can be used to detect state convergence in keystream generators. In Section 6.2, we review irregularly-clocked and regularly-clocked stream ciphers which experience state convergence during initialisation. We perform an investigation into a modified version of A5/1, and show why state convergence occurs in the cipher. We also provide counter-arguments to the claim made by the designers of Mickey-v2 that increasing the state size of its Mickey-v2's registers reduces the degree of state convergence which occurs in Mickey-v2.

In Section 6.3, we analyse regularly clocked stream ciphers which experience state convergence. We show how the analysis on the state-update function which causes state convergence in the F-FCSR stream cipher [77] can be also be applied to analyse the state-update function used in the summation generator in Section 6.3.3. As a result, we show how the summation generator suffers from state convergence.

Using the case studies and analyses in Section 6.2 and Section 6.3, we identify three possible mechanisms which may cause state convergence in Section 6.4. These are mutual-clock control, self-update mechanisms and addition-with-carry mechanisms. In particular, we show why mutual clock-control, an amalgamation of mutual-update mechanisms and clock-control mechanisms, causes state convergence when the individual mechanisms may not cause state convergence.

In Section 6.5, we analyse the effectiveness of state convergence on stream cipher cryptanalysis with regards to some common techniques applied to bit-based stream ciphers. These include time-memory-data tradeoff attacks, correlation attacks, algebraic attacks and differential attacks.
The investigation on the effect of state convergence on time-memory-data tradeoff attacks has appeared in the following publication:

- Sui-Guan Teo, Leonie Simpson, Kenneth Koon-Ho Wong, and Ed Dawson. State Convergence and the effectiveness of Time-Memory-Data Tradeoffs. In Ajith Abraham, Daniel Zheng, Dharma Agrawal, Mohd Faizal Abdollah, Emilio Corchado, Valentina Casola, and Choo Yun Choy, editors, *Proceedings of the 7th International Conference on Information Assurance and Security (IAS 2011)*, pages 92–97. IEEE, 2011. Updated version available from `http://eprints.qut.edu.au/47843/`.

1.3 Organisation of thesis

This thesis is organised as follows: In Chapter 2, we review concepts relevant to the understanding of contents in this thesis. In Chapter 3, we analyse the distribution of m-bit patterns in keystream sequences produced by nonlinear filter generators. In Chapter 4, we analyse the distribution of m-bit patterns in keystream sequences produced by linearly filtered nonlinear feedback shift registers. Algebraic analyses are also performed on Trivium ciphers to determine if particular types of slid pairs exist in Trivium. Algebraic analyses which use the combined methods of Berbain et al. and Raddum are also performed in this chapter. In Chapter 5, the Mixer stream cipher is analysed and it is shown why state convergence occurs during its initialisation process. In Chapter 6, the

state convergence problem in stream cipher is analysed in more detail. Chapter 7, we summarise the research results in this thesis, and suggest directions for future research.

Chapter 2

Background

This chapter presents a review of the theory relevant to the analysis and design of stream ciphers. We review stream ciphers, keystream generators and the various phases involved in the initialisation and keystream generation processes in Section 2.1. Following this, common components of a keystream generator are introduced in Section 2.2. The components reviewed in this section include Boolean functions, Linear Feedback Shift Registers (LFSRs), Nonlinear Feedback Shift Registers (NLFSRs), clock-control mechanisms, and output functions. Three combinations of linear and nonlinear components which can be used to generate keystream are described in Section 2.3. These different combinations form a framework for the material presented in Chapters 3–5. Techniques for cryptanalysing stream ciphers are reviewed in Section 2.4.

2.1 Stream ciphers and keystream generators

A symmetric cipher algorithm transforms a cleartext message (plaintext) to an unreadable format called ciphertext, and vice-versa using the same secret key. The transformation of the plaintext P to ciphertext C is called encryption and the reverse operation, the transformation from ciphertext to plaintext, is called decryption. If E_K denotes the symmetric encryption operation using the secret key K, then the encryption and decryption functions can be described as follows:

$$E_K(P) = C, E_K^{-1}(C) = P$$

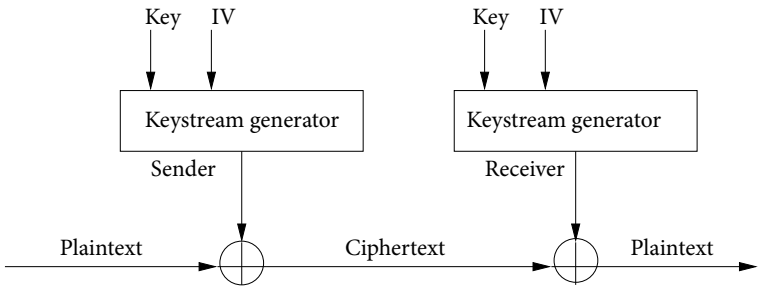


Figure 2.1: Stream cipher operation

In a typical communication two parties, the sender and receiver, first establish a secret key. This secret key is established out-of-band over a secure channel using either key transport or key agreement protocols. These protocols are beyond the scope of this thesis and are not discussed any further in this thesis. The sender creates the ciphertext using a given algorithm and a secret key. This ciphertext is sent over an insecure channel to the receiver. The receiver then uses the same algorithm and secret key to decrypt the ciphertext and recover the plaintext. An example of a symmetric key algorithm is the stream cipher.

Stream ciphers use a keystream generator to generate keystream, which is combined with a message to encrypt/decrypt a frame. The most common encryption and decryption function used is binary addition modulo 2, also known as the XOR operation. The XOR function is used as it is fast and easy to implement in both hardware and software. Furthermore, due to XOR's commutative properties, the same device can be used to perform both encryption and decryption functions. A stream cipher which uses the XOR function for encryption and decryption is called a binary-additive stream cipher. The keystream and message are then combined using the XOR operation to produce the encrypted frame or ciphertext. To decrypt the message, the receiver must use the same key and IV to initialise the keystream generator and produce the same keystream. The ciphertext and keystream are then combined using XOR operations to recover the original frame. The operation of a stream cipher is shown in Figure 2.1.

There are two types of stream ciphers: synchronous stream ciphers and self-synchronous stream ciphers. For synchronous stream ciphers, the keystream is generated calculated as some function of the internal state. For self-synchronous stream ciphers, the keystream is generated as some function of the internal state and the ciphertext. There has been evidence to suggest that self-synchronous stream ciphers are less secure than synchronous stream ciphers [35, 90]. Thus, we do not analyse self-synchronous stream ciphers in this thesis.

Keystream generators for stream ciphers operate by maintaining an internal state and applying update and output functions to the state. The state is generally stored in $h \geq 1$ registers. We use the notation $R_i(t)$ to denote the contents of stage i of register R at time t where $i = 0, 1, \ldots, r-1$, for an r-stage register; we also denote the register's state-update function by $R(x)$. If $R(x)$ is a linear function, the shift register is known as a Linear Feedback Shift Register (LFSR). If $R(x)$ is nonlinear, the shift register is a Nonlinear Feedback Shift Register (NLFSR). The size ω of each stage can be one bit for a binary shift registers, or more than one bit (usually a multiple of eight bits) for word-based shift registers. The state size of register R is $r \times \omega$. The state S of the keystream generator is of size s bits and is calculated by summing up the sizes of all the registers in the keystream generator. This thesis focuses on the analysis of stream ciphers based on binary shift registers. That is, each stage in a register contains $\omega = 1$ bits.

Stream ciphers are typically used to encrypt data for frame based, real-time applications, like pay-TV signals and mobile phone communications, as they are generally faster than block ciphers, the other symmetric cipher algorithm. A single communication in one of these applications may consist of multiple frames. To encrypt a frame-based communication, a single secret key K of size l bits (usually 80, 128, or 256 bits) is used for the entire communication. Each frame in this communication will use an initialisation vector or IV, V, of size j bits in combination with K. This key-IV pair is known as the session key which will be used in the encryption and decryption of this frame. Let $k_0, k_1, \ldots k_{l-1}$ represent the l-bit key and $v_0, v_1, \ldots v_{j-1}$ represent the j-bit IV used for a particular frame. The l-bit key and j-bit IV are used as inputs in a keystream generator. The operation of a keystream generator has two phases: an initialisation phase and a keystream generation phase.

2.1.1 Initialisation phase

Prior to encrypting each frame in the communication, the keystream generator needs to undergo an initialisation process, where the key-IV pair for that frame is used to form the initial internal state of the keystream generator. The goal of the initialisation process is to diffuse this key-IV pair across the entire state and make mathematical relationships between the key-IV pair and the keystream hard to establish. The initialisation process is usually performed in two phases: key and IV loading phase, and diffusion phase.

Key and IV loading phase

In the loading phase, the key and IV are loaded into the internal state of the cipher. This phase can either be linear or nonlinear. For some keystream generators, the key-loading and IV-loading phase are conducted simultaneously. That is, the secret key and IV are transferred to the stream cipher's state at the same time. At the end of this loading phase, the keystream generator is in a *loaded* state. If the loading phase is well-designed and the state size is greater or equal to $l + j$, one would expect that the number of possible loaded states will be 2^{l+j}.

Diffusion phase

The diffusion phase consists of a number of iterations, denoted α, of the initialisation state-update function. The value of α requires careful consideration. A small number can be performed quickly, which is desirable in real-time applications where rekeying is frequent. However, an initialisation process with few iterations may not provide sufficient diffusion and could leave the cipher vulnerable to attacks such as algebraic attacks [34] or linear cryptanalysis [83]. For example, Dinur and Shamir [39] claimed that they were able to mount a key-recovery attack on Trivium [26] whose initial state was produced using 767 iterations of the Trivium's diffusion process, compared to the 1152 iterations recommended by Trivium's designers. Similarly, Turan and Kara [109] claim that if the Trivium diffusion process was only performed for 288 iterations, Trivium's keystream can be approximated with a bias of 2^{-31}.

After the initialisation process is complete, the keystream generator is said to be in its *initial state*. Following this, the keystream generation phase begins. If the diffusion process is well-designed, the recovery of an initial state should not reveal any information about the secret key which generated it without any significant computational effort. A layered diagram showing the interaction between the various phases involved during initialisation is shown in Figure 2.2.

2.1.2 Keystream generation

During keystream generation, the internal state is updated using a state-update function. This state-update function may be the same state-update function which was used during the diffusion phase, or a different one. After the state-update function is applied once, a keystream bit is generated from the initial state, typically by applying an *output function* to the internal state's contents. This entire process continues until sufficient keystream

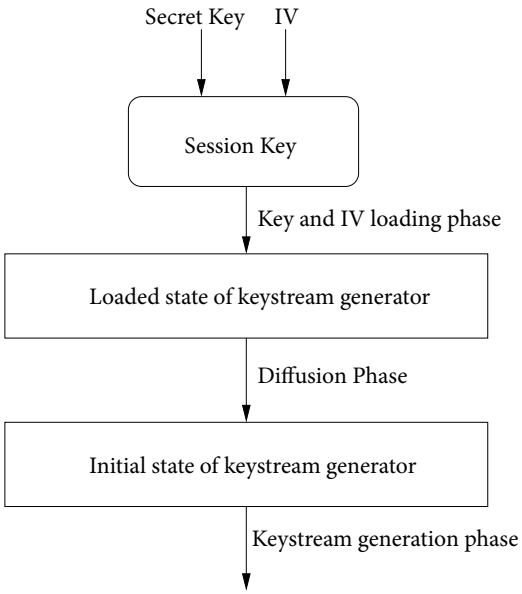


Figure 2.2: Layered diagram of initialisation process

Table 2.1: 3-tuple distribution table for Z

m-tuple	000	001	010	011	100	101	110	111
Occurrences	1	2	2	1	1	1	0	0

has been generated to encrypt the frame.

Keystream sequences produced by keystream generators are commonly viewed as a long sequence of binary digits. For example, let a 10-bit keystream sequence Z be: Z = 0001010011. This 10 bit sequence can be divided into a series of overlapping m-bit patterns, or, m-tuples. This thesis will use the term m-tuples to describe these patterns hereafter. For example, when m = 3, Z consists of the following seven 3-tuples: 000, 001, 010, 101, 010, 100 and 011. Using these patterns, a frequency distribution table can be constructed to examine the frequency of these 3-tuples. The frequency distribution table for Z is shown in Table 2.1. These frequency distribution tables will be used to examine the m-tuple distributions in keystream sequences produced by nonlinear filter generators in Chapter 3 and by linearly filtered nonlinear feedback shift registers in Section 4.1. Ideally, the m-tuple distributions will be almost uniform, for an adequate length of keystream.

One other method of studying the properties of sequences is through the Hamming Distance [62]. The Hamming Distance between two binary sequences of equal length

is the number of bit positions in which the sequences differ. This can be calculated by XORing two sequences together and counting the number of ones in its result. For example, assume we have two five-bit sequences $Z_0 = 01010_2$ and $Z_1 = 11110_2$. Then $Z_0 \oplus Z_1 = 01010_2 \oplus 11110_2 = 10100_2$. The Hamming Distance between Z_0 and Z_1 has weight two.

If the state size is greater than or equal to $l + j$, where l and j are the key and IV sizes in bits respectively, and the state-update functions used during the diffusion and keystream generation are well-designed, the number of distinct initial states and keystream which can be generated will be 2^{l+j}. In Chapters 5 and 6, we investigate the case where this does not happen, and discuss the security implications which arise from this. If the state-update and output functions take as input register stages whose contents are bit-based, the functions are called Boolean functions.

2.2 Components in keystream generators

In this section, we review components which are commonly used in modern keystream generators. These include Boolean functions, state-update functions and output functions.

2.2.1 Boolean Functions

A Boolean function $g(x) : \mathbb{Z}_2^n \to \mathbb{Z}_2$ such that $x = (x_0, x_1, \ldots, x_{n-1})$ is a mapping from n binary inputs to a binary output, for $n > 0$. A Boolean function is a common component used for both state-update functions and output functions in keystream generators. There are two common ways of expressing a Boolean function: the truth table and Algebraic Normal Form (ANF). The truth table of a Boolean function is a list of the output for all possible 2^n inputs. An example of a truth table for a Boolean function where $n = 3$ is shown in Table 2.2 A second expression of a nonlinear Boolean function is the Algebraic Normal Form (ANF). For every Boolean function there is a unique ANF representation. The ANF for a Boolean function is expressed in terms of an XOR sum of AND products of the input variables [76]. That is,

$$g(x) = \bigoplus_{I \in P(N)} a_I \bigotimes_{i \in I} x_i$$

where $P(N)$ denotes the power set of $N = \{1, \ldots, n\}$ [28] and $A_I \in 0$ for all I. If the ANF contains no AND product terms, the Boolean function is a linear Boolean function. If

Table 2.2: Truth table for Boolean function $g(x_0, x_1, x_2)$

x_0	x_1	x_2	$g(x)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

the ANF contains at least one AND product term, the Boolean function is a nonlinear Boolean function. For example, the ANF for the nonlinear Boolean function given by the truth table shown in Table 2.2 is

$$g(x) = x_1 \oplus x_0x_2 \oplus x_0x_1 \tag{2.1}$$

Using the Algebraic Normal Form, we can determine a property of a Boolean function, its algebraic order. The *algebraic order* a_o of a Boolean function is defined as the number of variables of the largest product term the Boolean function's ANF contains [90]. The algebraic order for the Boolean function example given in Equation 2.1 is two. Other important properties of nonlinear Boolean functions are balance, nonlinearity, correlation immunity and resilience.

Balance

If a Boolean function is balanced, the number of zeros of the Boolean function's output is equal to the number of ones in the Boolean function's output. That is, if the number of ones (or zeros) in the list of possible outputs of an n-input Boolean function is 2^{n-1}, the Boolean function is said to be balanced. Checking for the balance of a Boolean function is done by examining the Boolean function's truth table output. For example, the Boolean function is shown in Table 2.2 is balanced as the number of zeros and ones is $2^{3-1} = 4$. If the Boolean function is going to be used in keystream generation in stream ciphers, it is important that the Boolean function used is balanced (or close to balanced), otherwise the keystream generated will be biased, which may leave the stream cipher

vulnerable to statistical attacks.

Nonlinearity

The nonlinearity of a Boolean function is an important criteria to consider when evaluating the security of a stream cipher. The nonlinearity of a Boolean function is defined as the minimum Hamming distance to any affine function [28]. That is, the minimum number of bits in the truth table output which differ as compared to any affine function. The definition of Hamming Distance from Section 2.1.2 can also be applied to measure the Hamming Distance between a Boolean function and an affine function. This is done by XORing the truth table output for the Boolean function and for any affine function and counting the number of one's in the result.

For example, assume we have an affine function a_f defined as $x_0 \oplus x_1 \oplus x_2$. The output of a_f is 01101001_2. To compute the Hamming Distance between $g(x)$ and a_f we XOR both functions' truth table output. That is $00110101_2 \oplus 01101001_2 = 01011100_2$. Counting the number of one's in the result, we know that the Hamming Distance between $g(x)$ and a_f is four. After calculating the Hamming Distance between $g(x)$ and all possible affine functions, we discover that the minimum distance between $g(x)$ and an affine function is two. Thus, the nonlinearity of the Boolean function in Table 2.2, is two.

If the Boolean function used in a stream cipher is highly nonlinear, it may be difficult for an attacker to approximate it with a linear function. This may make it harder to form the linear relations necessary to perform linear cryptanalysis [56]. Similarly, a highly nonlinear function will make solving a system of equations using algebraic attacks [31,34] more complex.

Correlation immunity

Correlation immunity is a measure of whether the output of the Boolean function is correlated to a subset of the input variables. An n-input Boolean function $f(x_0, x_1, \ldots x_{n-1})$ is said to be c-order correlation immune if each subset of c random variables $x_{i_0}, x_{i_1}, \ldots x_{i_{c-1}}$ with $1 \leq i_0 \leq i_1 \leq \ldots i_{c-1} \leq n$, the random variable $P = f(x_0, x_1, \ldots x_{n-1})$ is statistically independent of the random vector $x_{i_0}, x_{i_1}, \ldots x_{i_{c-1}}$ [88, Definition 6.52]. For example, in the Boolean function shown in Table 2.2, there is a correlation between the output of the Boolean function and x_2, as $g(x)$ and x_2 are the same with a probability of 0.75. That is, the Boolean function has a correlation immunity of zero. Correlation immunity is an important characteristic for nonlinear combiners and nonlinear filter generators. Having a correlation immunity greater than zero may make the keystream generator res-

istant to divide-and-conquer attacks. There is a tradeoff between the algebraic order of a Boolean function and its correlation immunity. This bound is known as Siegenthaler's bound [100]. Assume there is a balanced n-input Boolean function of algebraic order a_o that satisfies correlation immunity of order c. Then Siegenthaler's bound states that $a_o \leq n - c - 1$. For a given value of n, this bound allows for a trade-off between the algebraic order of a Boolean function and the correlation immunity it provides.

For example, designers who want an $n = 10$-input, balanced nonlinear Boolean function can construct a function such that $a_o = 5$. Using Siegenthaler's bound, it is clear that this function's correlation immunity order can be no greater than four. However, if the designers of a nonlinear Boolean function decide that having a nonlinear Boolean function where $a_0 = 2$ is acceptable, they can construct a nonlinear Boolean function whose correlation immunity order is no greater than seven. In summary, the implications of Siegenthaler's bound means that designers of nonlinear Boolean functions may need to forgo having a high alegraic order if they want a function of high order correlation immunity, or vice-versa.

Resilience

A Boolean function is resilient if it is *both* correlation immune and balanced. For example, a balanced nonlinear Boolean function which is $c = 2$-order correlation immune can be described as a 2-resilient nonlinear Boolean function.

2.2.2 State-update functions

State-update functions are used to update the internal state of registers in keystream generators. At each iteration of the state-update function, the contents of each stage are shifted. Each time the contents of each stage are shifted, there will be a particular stage which does not have any value. The computation of this new stage's contents, known as the feedback bit, is calculated as a linear or nonlinear function of the contents of other stages prior to the register being shifted. Two common types of state-update functions are linear and nonlinear state-update functions.

Linear update functions

Linear feedback shift registers (LFSRs) are common components in many bit-based stream cipher designs due to their simplicity and efficiency in hardware. A binary LFSR R has an internal state consisting of r stages, where each stage stores $\omega = 1$ bit. Therefore,

this LFSR has a state size of $s = 1 \times r$ bits. R's feedback function or connection polynomial, is used to update the internal state of R at each iteration. The feedback function of an LFSR can also be converted to its characteristic polynomial

$$R(x) = c_0 x^r + c_1 x^{r-1} + \ldots + c_{r-1} x + c_r,$$

where the coefficients c_s are the feedback constants, also known as tap positions. If this feedback function of the LFSR is primitive and the initial state is non-zero, all possible $2^s - 1$ non-zero internal states will be generated before a certain internal state repeats. The period p of the sequence produced by this LFSR will be $2^s - 1$ [88]. This sequence satisfies three properties, also known as Golomb's postulates [59]:

1. The number of ones and zeros in a single period of the sequence is such that the number of ones is 2^{s-1} and the number of zeros is $2^{s-1} - 1$.

2. A run in a period of a keystream sequence is a string of consecutive 0's or 1's which is neither preceded nor succeeded by the same symbol [88]. In every period, half the runs must be of length one, a quarter of the runs must be of length two, an eighth of the runs must be of length three, etc. In general, if the run length is denoted as β, the proportion of runs of this length must be $\frac{1}{2^\beta}$, for $\beta \leq s - 1$.

3. Assume we have two keystream sequences, Z^0 and Z^d, of period $p = 2^s - 1$ for an LFSR of length s such that Z^d is a d bit shift of Z^0. The number of agreements, N_a and the number of disagreements N_d between Z^0 and Z^d is counted. The auto-correlation coefficient for both sequences is calculated using the formula $(\frac{N_a - N_d}{p-1})$. The auto-correlation coefficient for all d sequences must be constant.

A sequence which satisfies all three of the properties listed above is called a maximum length sequence or an m-sequence. Golomb's first postulate can be extended for m-tuple distributions where $m > 1$. Consider the case when $m = 2$. The LFSR output sequence of length $2^s - 1 + 2 - 1$ can be viewed as a series of overlapping two-tuples. For the output sequence of an LFSR with a primitive polynomial, each of the 2-tuples 01, 10 and 11 occurs 2^{s-2} times, except the pattern 00, which occurs $2^{s-2} - 1$ times. Similarly, for the LFSR output sequence of length $2^s - 1 + m - 1$, each m-tuple occurs 2^{s-m} times, except for the all-zero m-tuple which occurs $2^{s-m} - 1$ times. The distribution of m-tuples, for $m < s$, in random sequences is expected to be essentially uniform.

The benefit of using an LFSR to generate keystream is that the properties of the keystream it generates are desirable. However, using only an LFSR to generate keystream

is very insecure due to its low linear complexity. The linear complexity (LC) of a binary sequence is defined as the length of the shortest LFSR which can generate that sequence. Given $2 \cdot LC$ bits of a binary sequence, the Berlekamp-Massey algorithm [82] can determine the LFSR which generated that sequence with a maximum time complexity of $O(LC^2)$. For LFSRs with a primitive feedback polynomial, $LC = s$, and the time complexity of reconstructing the LFSR is $O(s^2)$.

An LFSR can either be autonomous or non-autonomous. An autonomous LFSR's feedback bit is calculated using only the register's internal state values and the characteristic polynomial. In contrast, a non-autonomous LFSR's feedback bit is calculated as a function of the internal state values, characteristic polynomial *and* some other external input.

To provide resistance to Berlekamp-Massey attacks [82], the sequences produced by keystream generators should have a large linear complexity. Due to the Berlekamp-Massey attack on LFSR sequences, the sole use of an LFSR to generate keystream for cryptographic purposes is not recommended. Some form of nonlinearity needs to be introduced. This is done either implicitly via clock control mechanisms, or explicitly, through the use of nonlinear state-update functions, or nonlinear Boolean functions as output functions. In the following sections, we review common nonlinear state-update functions used in keystream generators. These include nonlinear feedback shift registers (NLFSRs) and clock control mechanisms.

Nonlinear update functions

Nonlinearity can be introduced into binary sequences through state-update functions which are either explicit or implicit. Examples of explicit and implicit nonlinear state-update functions are nonlinear feedback shift registers and clock-control mechanisms, respectively. These are described below.

Nonlinear feedback shift registers Nonlinear feedback shift registers (NLFSRs) use a nonlinear function to update the internal state of the shift register. A simple five stage NLFSR is shown in Figure 2.3, where $\oplus$ and $\otimes$ are addition and multiplication in GF(2) respectively. The feedback function of this five stage NLFSR is $f(x_0, x_1, x_2, x_3, x_4) = x_0 \oplus x_1 \oplus x_2 \oplus (x_2 \otimes x_4)$. Given a non-zero initial state, this NLFSR will generate a sequence of period $2^5 - 1 = 31$ bits [40].

For certain NLFSR sequences with a period of $2^s - 1$, Gammel and Göttfert [49] show that the distribution of ones and zeros, is almost equally distributed. Gammel

Table 2.3: Properties of certain NLFSRs [48]

Type of NLFSRs	A	B	C	D
Length of shift register	s	s	s	s
Period	$2^s - 1$	$2^s - 1$	$2^s - 2$	$2^s - 2$
Forbidden initialisations	$(0, 0, \ldots, 0)$	$(1, 1, \ldots, 1)$	$(0, 0, \ldots, 0)$, $(1, 1, \ldots, 1)$	$(0, 1, 0, \ldots)$, $(1, 0, 1, \ldots)$
Linear complexity	$2^s - 1$	$2^s - 1$	$2^s - 2$	$2^s - 2$
No. of distinct cycles	2	2	3	2
Distribution of 0's and 1's in full period	almost uniform	almost uniform	uniform	uniform

and Göttfert [48] reviewed the properties of four types of NLFSR sequences which have guaranteed periods. The four NLFSR types were denoted A, B, C, and D. Portions of their review are reproduced by Table 2.3. In a 2006 paper by the same authors [49], the m-bit pattern distribution of sequences produced by bit-based type A and B NLFSRs were shown to be the same as a LFSR with a primitive feedback function. That is, the all-zero m-tuple occurs $2^{s-m} - 1$ times, while all non-zero m-tuples occur 2^{s-m} times. However, little is known about the other properties of these sequences. For example, when different sets of stages are used as inputs to the linear output function, what would the distribution of m-tuples be? The answer to the question just posed, for $m \geq 2$ was not known prior to the analysis performed in this thesis. In Chapter 4 and 5, we analyse stream ciphers which use nonlinear feedback shift registers in their design. In particular, the effect of different selections of stages used as inputs to the linear output function have on the distribution of m-bit tuples in keystream sequences produced by linearly filtered nonlinear filter generators are examined in Section 4.1.

An NLFSR can either be autonomous or non-autonomous. An autonomous NLFSR's feedback bit is calculated as a nonlinear function of the register's internal state value as input. In contrast, a non-autonomous LFSR's feedback bit calculated as a nonlinear function of the register's internal state value and some other external input.

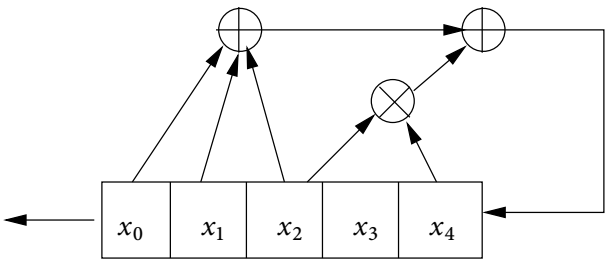


Figure 2.3: Diagram of a NLFSR

Clock control mechanisms For a regularly clocked shift register, the internal state of the shift register is always updated once at each clock. Clock-controlled keystream generators generally use the contents of one or more registers to control the clocking of itself and/or other registers. For irregularly clocked shift registers, the register state can be updated more than once or not at all at each clock, depending on the type of mechanism used. Two examples of clock-control mechanisms are majority-decision and integer functions.

Majority-decision Majority-decision rule mechanisms use the contents of an odd number of register stages to determine which of two possible actions will occur. For example, which binary shift register is clocked each time the state-update function is invoked and which shift register is not clocked. This majority decision rule requires that the keystream generator which uses this mechanism has an odd number of registers, so that at least $\lceil \frac{h}{2} \rceil$, where h is the number of registers in the keystream generator, have the same value.

An example of the use of this majority-decision rule is in the state-update function of A5/1. A5/1 is a stream cipher based on three LFSRs. For each shift register, one register stage is used as the clocking tap. This clocking tap is used as input to the majority-decision function to determine if that particular register will be clocked. If the contents of two or more clocking taps are the same, the registers with clocking taps which agree are clocked. In Section 6.2.1, we analyse the A5/1 stream cipher and show that this majority-decision rule causes state convergence.

Integer function Integer functions take as input the contents of selected stages from a binary shift register into a function, which outputs an integer value $c(x)$. A separate register is then clocked $c(x)$ times. Integer functions are used in the state-update functions of LILI-II [29] and Mixer [79]. In Chapter 5 we analyse the Mixer stream cipher and in Section 6.4 we analyse other keystream generators which use integer functions in their state-update functions and show that this can cause state convergence.

2.2.3 Output functions

Output functions take input from selected stages of a keystream generator's registers and produce output which form the keystream during a keystream generator's keystream generation phase. These output functions can be linear or nonlinear. During a keystream

generator's keystream generation phase, selected stages from a register are used as input to the output function. We call these stages used as input to the output function *tap settings*. These tap settings can be consecutive, or non-consecutive.

Let $\Upsilon = \{r_i : 0 \leq i \leq n-1\}$ be a set of n non-negative integers, where r_i is a certain stage index in a register whose stages are used as input to the output function. There are four methods of selecting tap settings. These are: consecutive taps, non-consecutive but evenly spaced, non-consecutive (or uneven taps), and Full Positive Difference Set taps.

Consecutive taps If an output function takes as input, consecutive taps, the distance between r_i and r_{i+1} or (r_{i-1}) will be one for values of i.

Non-consecutive but evenly-spaced taps If an output function takes as input, non-consecutive but evenly-spaced taps, the distance between r_i and r_{i+1} or (r_{i-1}) will be equidistant for all values of i.

Non-consecutive taps If an output function takes as input, non-consecutive taps, the distance between r_i and r_{i+1} (or r_{i-1}) will be greater than or equal to one for all values of i.

Full-Positive Difference taps The term Full-Positive Difference Set was used by Golić [54] to describe the selection of stages used as input to the output function. If an output function takes as input, taps which form a Full Positive Difference Set (FPDS), the positive differences between r_i and r'_i for $0 \leq i \leq n-1$ and $0 \leq i' \leq n-1$, are distinct for all values of i and i' for all cases where $i \neq i'$.

2.3 Combining update and output functions

The previous sections reviewed common components used in the construction of keystream generators. In this section, we describe how we can combine the various components to construct a keystream generator. We form categories based on the type of state-update and output functions. The stream ciphers analysed in this thesis can be grouped into one of these categories. These categories include

- Linear state-update and linear output
- Linear state-update and nonlinear output
- Nonlinear state-update and linear output
- Nonlinear state-update and nonlinear output

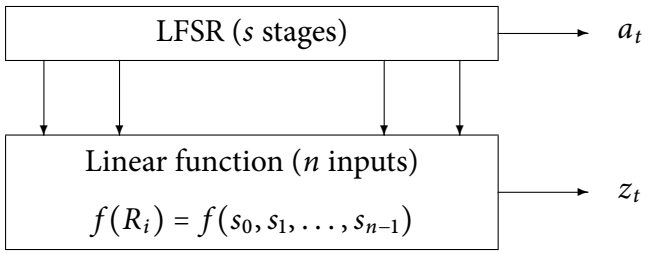


Figure 2.4: Linear output from LFSR

2.3.1 Linear state-update and linear output

Using a linear state-update function and linear output function is a simple method for generating keystream. Such a keystream generator is shown in Figure 2.4. Properties of these keystream generators which take a single output for use as keystream are well-known, and were reviewed in Section 2.2.2. Rueppel [96] showed that the sequence produced by keystream generators which take linear combinations from selected stages in a LFSR is a periodic sequence which can be formed by another LFSR of some length with a different feedback function. Unfortunately, this type of keystream generator can also be easily broken using the Berlekamp-Massey algorithm. Consequently, use of this type of keystream generator is not recommended in secure cryptographic applications. We give no further consideration to keystream generators of this sort in this thesis, but include the description here for the sake of completeness.

2.3.2 Linear state-update and nonlinear output

Nonlinear combiners and nonlinear filter generators are common examples of keystream generators which use a combination of linear state-update followed by nonlinear output. Since the update functions for the LFSRs are linear, the security of the NLFGs and the nonlinear combiner rely on the characteristics of the nonlinear functions. Important characteristics for both NLFGs and nonlinear combiners include nonlinearity and algebraic order.

The nonlinear combiner consists of two or more LFSRs and a nonlinear Boolean function. To generate keystream, the output of each LFSR is used as input to the nonlinear function. A diagram of a nonlinear combiner is shown in Figure 2.5. Properties of sequences produced by nonlinear combiners were studied by Rueppel [96]. He found that if the combiner consists of n LFSRs with primitive feedback polynomials whose

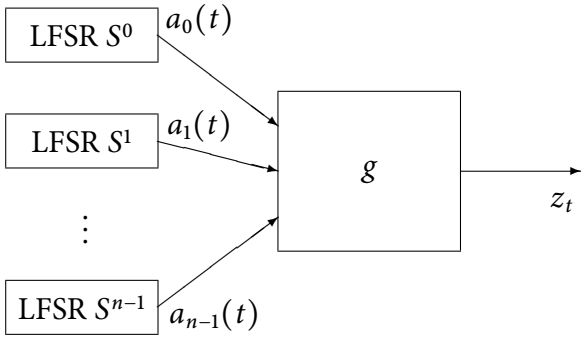


Figure 2.5: Nonlinear Combiner

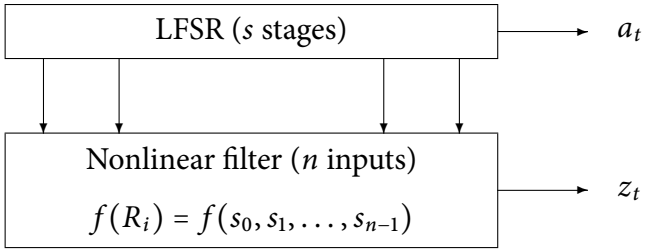


Figure 2.6: Nonlinear Filter Generator

lengths $L_0, L_1, \ldots L_{n-1}$ are pairwise relatively prime, the period and the linear complexity of the sequence produced are $\prod_{i=0}^{n-1}(2^{L_i} - 1)$ and $g(L_0, L_1, \ldots L_{n-1})$ respectively, where g is the nonlinear combiner's Boolean filter function, evaluated over integers instead of GF(2).

NLFGs typically consist of an LFSR and a nonlinear Boolean function. To generate keystream, the internal state of the LFSR is first updated linearly. Selected stages from the LFSR are used as input to the nonlinear function. The output of the nonlinear function is used as keystream. A diagram of a NLFG is shown in Figure 2.6. An example of a stream cipher which uses such a construction is Sfinks [21].

Some properties of nonlinear filter generators are known. Golić [54] showed that the distribution of m-tuples, for $m = 1$ of an output sequence produced by a nonlinear filter generator is almost uniform if (and only if) the n-input nonlinear Boolean function $f(x_0, \ldots, x_{n-1})$ is balanced for each value of $(x_1, \ldots, x_{n-1})$, that is, if

$$f(x_0, \ldots, x_{n-1}) = x_0 \oplus g(x_2, \ldots, x_{n-1})$$

or if the Boolean function $f(x_0, \ldots, x_{n-1})$ is balanced for each value of $(x_0, \ldots, x_{n-2})$,

that is, if

$$f(x_0, \ldots, x_{n-1}) = x_{n-1} \oplus g(x_0, \ldots, x_{n-2}).$$

Simpson [105] later showed that if the feedback function of the LFSR is primitive and the nonlinear Boolean function is balanced, the period of the keystream produced will be $2^s - 1$ and the distribution of m-tuples, when $m = 1$, in a single period will be almost uniform. Less is known about the other properties of keystream sequences generated by such a technique. For example, little is known about the distribution of overlapping m-tuples, when $m > 1$, across a single period. Little is also known about the effect different tap settings to the output functions have on the distribution of m-tuples in keystream sequences produced by nonlinear filter generators. In Chapter 3, we perform computer simulations to analyse these m-tuple distributions produced by these keystream generators in more detail.

2.3.3 Nonlinear state-update and linear output function

Using a nonlinear state-update with a linear output function is another method of generating a nonlinear keystream sequence. Instead of using a linear function to update the internal state of the keystream generator, a nonlinear function is used. A linear combination of state bits is then used as keystream output. A diagram of such a keystream generator is shown in Figure 2.7.

Some NLFSRs can produce a sequence which has a period of $2^s - 1$ [40], while other NLFSRs produce a de Bruijn sequence [38] which has a period of 2^s, and some have much shorter periods. Gammel and Göttfert [49] proved that sequences generated by some types of NLFSRs have good cryptographic properties like large periods, large linear complexities and almost uniform m-tuple distributions. However, very little is known about the properties of other types of NLFSRs. Many modern ciphers use NLFSRs where even basic properties, such as the period are not known. Hu and Gong [72] show that the output sequences of Grain [66] and Trivium [26] are periodic with high probability, although proving what this period is remains an open question. Little is also known about the distribution of overlapping m-tuples across a single period when the linear output function takes as input more than two stages from the NLFSRs. The effect different tap settings to the output functions have on the distribution of m-tuples in keystream sequences produced by linearly filtered nonlinear feedback shift registers is also an open question. In Section 4.1, we perform computer simulations to analyse

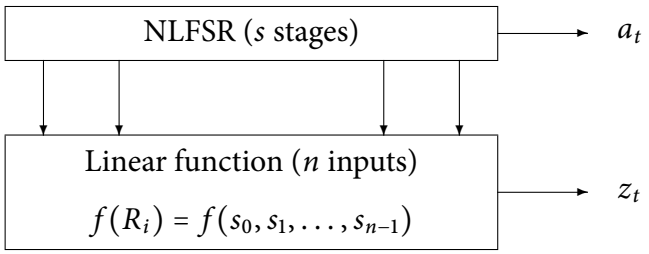


Figure 2.7: Keystream generator with nonlinear state-update and linear output function

these m-tuple distributions produced by these keystream generators in more detail.

Examples of stream ciphers which use such a technique are Trivium [26] and its variants. These ciphers are analysed in Section 4.2 and Section 4.3, using algebraic techniques.

2.3.4 Nonlinear state-update and nonlinear output

A keystream generator can produce a nonlinear keystream sequence through a combination of a nonlinear state-update function and a nonlinear output function. This technique uses a nonlinear function to update the internal state of the keystream generator and uses selected stages of the keystream generator as input into a different nonlinear function, the output of which will be used as keystream. A general diagram of such a keystream generator is shown in Figure 2.8.

Similar to keystream generators which use a nonlinear state-update function and linear output functions, very little is known about the properties of the keystream these types of keystream generators produce. Since this keystream generator generates keystream using two separate nonlinear functions, the sequence generated may be highly nonlinear. This may make it resistant to algebraic attacks. However, other properties like the period of the keystream, and the distribution of m-tuples are open questions. The Mixer cipher [79] analysed in Chapter 5 is an example of such a generator.

2.4 Stream cipher cryptanalysis

The usual goals of stream cipher cryptanalysis are to recover either the initial state or the secret key of the keystream generator. If the attacker recovers the initial state of a keystream generator, they can generate keystream to decrypt only one frame of a

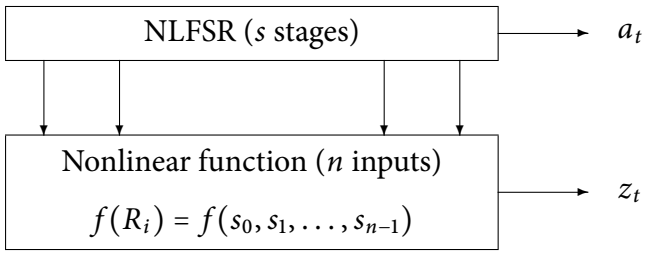


Figure 2.8: Keystream generator with nonlinear state-update and nonlinear output function

communication. However, if the attacker recovers the secret key, they can reproduce the keystream for all frames to decrypt the entire encrypted communication.

The complexity of a cryptanalytic attack can be described using a series of variables. In this thesis, the following notation when discussing the complexity of attacks are applied: D is the amount of data the attacker needs, P is any pre-computation time needed before the actual attack algorithm is run, M is the memory required for the attack, and the time needed for the attack is denoted by T.

In stream cipher cryptanalysis, it is usually assumed that the attacker has access to the ciphertext, and possibly a certain amount of plaintext corresponding to the known ciphertext, or an amount of keystream. A stream cipher designer needs to take into account all of these possibilities when considering if the cipher is vulnerable to certain attacks.

Where the attacker has access to the ciphertext, two attack types should be considered. These are the ciphertext-only (or known-ciphertext) attack and the chosen ciphertext attack. If it is possible to attack a stream cipher using the ciphertext-only attack, without making use of plaintext, this means that the stream cipher's keystream generator is very weak. A chosen ciphertext attack allows the attacker to select ciphertexts to be decrypted and attempts to recover the key from this ciphertext. This model is not relevant to synchronous stream ciphers since the keystream generated is not ciphertext-dependent. Where the attacker has possession of some plaintext and the corresponding ciphertext, two attack types can be considered: known-plaintext attack and chosen plaintext attack. For known plaintext attacks, an attacker gains access to an amount of keystream by XORing the plaintext and its corresponding ciphertext. For chosen-plaintext attacks, the attacker chooses the plaintext they want encrypted and obtains the corresponding ciphertext. For most binary-additive stream ciphers, the outcome of the

known plaintext and chosen plaintext attacks is the same: a segment of the keystream is also revealed.

These attack models provide the framework for generic attacks which can be applied to stream ciphers. The attacks covered in the review are the guess-and-determine attack, distinguishing attacks, divide-and-conquer attacks, linear cryptanalysis, differential cryptanalysis, time-memory-data tradeoff attacks and algebraic attacks.

2.4.1 Exhaustive key search

In an exhaustive key search, an attacker tries all possible 2^l secret keys to determine which secret key can correctly decrypt an encrypted frame. A brute-force attack is an inefficient form of attack as the value of l in modern stream ciphers is quite large, and trying all 2^l possible keys to determine which is the correct key can take a long time. A successful attack will require that an attacker is able to recover the secret key or initial state faster than the time needed to try all 2^l possible keys.

2.4.2 Guess and determine attacks

Guess and determine attacks (G&D) are techniques used with other cryptanalytic techniques like correlation attacks or linear cryptanalysis to recover the initial state of the keystream generator. Thus, G&D attacks can be applied to any stream cipher which uses linear or nonlinear components.

G&D attacks require the attacker to make assumptions about how certain values in the internal state were produced. The attacker then guesses the values of certain register stages and uses these guesses to determine what the contents of the remaining stages are. Using the reconstructed internal state, the attacker produces keystream and compares it to the observed keystream. If it matches, the attacker can be confident that they have recovered the correct internal state. To ensure that the G&D attack on a stream cipher has a complexity which is less than exhaustive keysearch, the number of bits guessed should be less than the keylength l.

The attack on SNOW by Hawkes and Rose [63] is an example of a G&D attack. SNOW [43] is a word-based stream cipher that accepts keysizes of either 128 or 256 bits. Their attack on SNOW has four phases. In the first phase, the attacker makes assumptions on the contents of internal state of SNOW at two points in time. In the second phase, the attacker guesses 192 bits of the internal state. In the third phase, the attacker determines the internal state of the LFSR using the values guessed in the second

phase. In the fourth phase, the attacker tests if their guess for the LFSR's internal state is correct by producing a keystream and comparing it with the observed keystream. If the keystreams match, the attacker can be confident that they have recovered the correct keystream. If they do not match, the attacker can try further guesses in second Phase. If the attacker exhausts their guesses in the second phase, the assumptions made in the first phase are incorrect and the attacker returns to the first phase, makes new assumptions about the internal state at a different point in time, and performs the attack again. Hawkes and Rose estimate that an attack on SNOW has an estimated time complexity of 2^{224} operations and has a data complexity of $O(2^{95})$. The version of SNOW which uses a 128 bit key to generate keystream is resistant to this G&D attack. However, the version of SNOW which uses a 256 bit key to generate keystream is vulnerable to this attack.

2.4.3 Distinguishing attacks

The goal of a distinguishing attack is to find some statistical characteristic in the output sequence of a keystream generator which will allow an attacker to determine if a binary sequence was generated from a particular keystream generator or a truly random source. Statistical packages like the NIST test suite [91], Diehard statistical tests suite [81] and CryptX [61] have a series of tests which can be applied to keystream sequences for randomness properties. Note that passing these tests does not necessarily prove that a keystream generator is resistant to all distinguishing attacks, but failing the test indicates that the keystream generator has some form of bias which may be exploitable by distinguishing attacks.

An example of a distinguishing attack is the attack on RC4 by Mantin and Shamir [80]. In analysing a large group of keystreams generated by RC4's keystream generator, they discovered that in the second byte of keystream, the zero-byte keystream sequence occurred with a probability of $\frac{1}{128}$, twice the expected value of $\frac{1}{256}$. Using 256 output bytes from random keys, Mantin and Shamir were able to construct a distinguisher which is able to distinguish a RC4 keystream sequence from a truly random sequence.

2.4.4 Divide and conquer attacks

Divide and conquer attacks target an individual component of the keystream generator, rather the keystream generator components in their entirety. For example, consider divide and conquer attacks applied to nonlinear combiners. A particular nonlinear combiner consists of $h = 3$ LFSRs, of sizes $s^1 = 128$ bits, $s^2 = 64$ bits and $s^3 = 64$ bits,

giving a total state size of 256 bits. The secret key of this nonlinear combiner is the same size as the total state size, that is, 256 bits. A brute force attack on this nonlinear combiner will have a complexity of $O(\prod_{i=1}^{n}(2^{s^i}-1)) \approx 2^{256.00}$. In contrast, a successful divide and conquer attack may have a reduced attack complexity of $O(\sum_{i=1}^{n}(2^{s^i}-1)) \approx 2^{128.00}$. The divide and conquer strategy is often used in combination with other attacks. Some of these attacks include the basic correlation attack and the fast correlation attack.

Basic correlation attack

The basic correlation attack [100] was proposed as a means to recover the initial state of bit-based stream ciphers, usually in a divide and conquer approach on keystream generators with multiple component parts like the nonlinear combiner. Correlation in the context of binary sequences, is the measure of the extent two binary sequences approximate to each other. For keystream generators, the keystream sequence is treated as a noisy version of the output sequence of an underlying LFSR sequence. In the case of regularly clocked keystream generators, both the output sequence of the underlying LFSR and the keystream sequence are of the same length, so the Hamming Distance, described in Section 2.1.2 between the two sequences can be used to measure the level of correlation. In the case of irregularly clocked keystream generators, the Hamming Distance cannot be used as both sequences will be of different lengths. In this case, the Levenshtein distance can be used to determine the level of correlation between the two sequences [52].

In the case of nonlinear combiners, the basic correlation attack will succeed when the output of a particular LFSR is the same as the keystream bit with a probability $P > 0.5$. For example, consider a nonlinear combiner consisting of three LFSRs, A, B, C, with the nonlinear Boolean function whose ANF is given in Equation 2.1, where x_0, x_1 and x_2 are the outputs of A, B and C respectively. The keystream equation for this particular nonlinear combiner is:

$$z_t = x_1 \oplus x_0 x_2 \oplus x_0 x_1$$

The probability that $x_2 = z_t = 0$ is 0.75. Therefore, when the keystream output from this particular nonlinear combiner is zero, an attacker knows that there is a 75% chance that the output of x_2 is zero.

The basic correlation attack on nonlinear combiners using Siegenthaler's technique [101] requires the attacker to try all possible initial states for a particular component LFSR.

After the correlation values for all possible initial states of a particular component have been calculated, the initial state whose correlation has a high value is most likely the correct initial state which generated the keystream. The approach may be applied sequentially to multiple components. Resistance to the basic correlation attack can be provided by ensuring that the size of individual shift register components used in the keystream generator is larger than the size of the secret key and by using a resilient nonlinear Boolean function

Fast correlation attack

A disadvantage of the basic correlation attack is that it requires exhaustive search of the contents of a shift register component. If the length of the shift register(s) was larger than the secret key size, a successful basic correlation attack will be worse than exhaustive key search. Fast correlation attacks are a type of correlation attack which may have an attack complexity which is faster than exhaustive key search. In the fast correlation attack, proposed by Meier and Staffelbach [87], an attacker uses parity checks instead of the correlation between the output of the LFSRs and the keystream. Parity checks are any linear relationship satisfied by a LFSR sequence [87]. Certain LFSRs have a low-weight polynomial, that is, they have a small number of non-zero coefficients in its characteristic polynomial. If the LFSR has a low-weight polynomial, the number of linear relationships which can be formed using the repeated squaring method [87] may be more than the number of linear relationships which can be formed if the LFSR does not have a low weight polynomial. The more linear relationships which can be formed, the greater the chance of the attack succeeding. The fast correlation attack will use the observed keystream and try to convert it to the underlying LFSR sequence. Using parity checks previously formed using the repeated squaring method, it may be possible to determine which bits in the keystream and the underlying polynomial are the same. An attacker can then use error correction algorithms like those proposed by Golić et al. [57] or Mihaljevic and Golic [89] to recover the rest of the LFSR state from the keystream. Once a candidate internal state has been recovered, the attacker can use this internal state to generate keystream. If this keystream sequence can successfully decrypt the encrypted frame, the attacker can be confident that they have recovered the correct internal state. Various fast correlation attack methods [27, 53, 98] have been proposed since the seminal paper by Meier and Staffelbach.

An example of using a fast correlation attack on a stream cipher can be found in the fast correlation attack on the LILI-128 [107] stream cipher by Jönsson and Johansson [78].

They claim that a fast correlation attack can recover the initial state of LILI-128 with a complexity of about $O(2^{71})$. This is a successful attack against LILI-128 as the attack complexity is less than trying all possible 2^{128} keys to determine which is the correct key.

2.4.5 Linear cryptanalysis

Linear cryptanalysis [83], which is normally applied to block ciphers, analyses the linear relations of internal state bits and linear combinations of keystream bits. These relationships are formed by approximating the nonlinear function used in the keystream generator to linear ones. This is in contrast to correlation attacks, where an attacker is looking for correlations between a single keystream bit and a set of internal state bits. Similar to the fast correlation attack, once linear relations have been established, parity-check equations can be formed and the internal state may be recovered using iterative error-correction algorithms. Linear cryptanalysis was applied to the Bluetooth stream cipher [19] by Golić et al. [56]. In their analyses, they can recover the 128 bit secret key of the Bluetooth stream cipher with a complexity of about $O(2^{70})$, which is faster than exhaustive key search.

2.4.6 Differential cryptanalysis

Differential cryptanalysis [16] is a technique which is commonly applied to block ciphers, but also has applications in stream cipher cryptanalysis. Differential cryptanalysis analyses how differences in the input (either the key, IV, or internal state) affect the output of the cipher (this can be either the internal state or the keystream output.) In the context of stream ciphers, there are three possible differential characteristics [13]:

- a difference in the key and/or IV generates a particular difference in the internal state,
- a difference in the internal state generates a particular difference in the internal state,
- a difference in the internal state generates a particular difference in the keystream output.

If an attacker is able to find a differential characteristic which occurs with a high probability, an attacker may be able to exploit this in initial state or key recovery attacks. An example of this is the key recovery attacks on the Py family of stream ciphers [73, 113].

2.4.7 Time-memory-data tradeoff attacks

Time-memory-data tradeoff (TMDT) attacks are performed in two phases: the pre-computation phase and the online phase. In the pre-computation phase, a lookup table is constructed. This table has two columns. For state recovery, the first column consists of selected initial states of the stream cipher. For secret key recovery, the first column consists of selected secret keys (and might also include the IV). In both scenarios, the second column consists of a segment of keystream generated using either the corresponding key-IV pair, initial state or internal state. In the online-phase of the attack, the attacker compares the captured keystream to the second column of the lookup table. If a match is detected, the attacker assumes that the obtained initial state, internal state or key-IV pair is correct.

The complexity of a TMDT attack can be described using a series of variables. D is the amount of data the attacker needs in the online-phase of the attack to recover the secret key. P is the pre-computation time needed to construct the lookup table. M is the memory needed to construct and store the table. During the online phase, the attacker attempts to recover the initial state or secret key by searching through the lookup table. The time taken for the search is denoted by T. The success of the attack depends on T or M or the sum of $T + M$ being less than 2^l or 2^s, depending on the particular TMDT attack being used. Since P is a one-off operation, it is assumed that the attacker has already pre-computed the lookup table beforehand and the time taken for this operation is not considered when measuring the complexity of the TMDT attack. In this section, the major TMDT attacks on stream ciphers are reviewed.

Babbage and Golić

Babbage [7] and Golić [55] independently applied the TMDT attacks to stream ciphers. Their initial state recovery attack is referred to as the BG attack hereafter.

In the pre-computation phase, an attacker selects either M different initial states or internal states. For each of these, the attacker produces some keystream of length s. The attacker then stores the initial state-keystream pair in a lookup table, sorted according to the keystream.

In the real-time phase of the attack, the attacker takes a segment of keystream of length $D + \log s - 1$ they have captured and uses a sliding-window to produce all D possible keystream sub-strings of length s. The attacker then searches the lookup table to see if any of these substrings match. If there is a match, the initial state corresponding to the keystream sub-string is considered to be the initial state which generated the

captured keystream. If the TMDT satisfies the following equations

$$T \cdot M = 2^s \text{ with } P = M \tag{2.2}$$

the attack complexity is less than that of exhaustive keysearch. To provide resistance to this attack, both Babbage and Golić recommend that the size of the internal state of the stream cipher should be at least twice the key size.

Biryukov and Shamir

Biryukov and Shamir [17] combine the concepts of Hellman's TMDT attack on block ciphers [67] and the BG attack to provide a more efficient TMDT attack on stream ciphers. Their initial state recovery attack is referred to as the BS attack hereafter.

The pre-computation phase of the BS attack is similar to Hellman's pre-computation phase. The attacker defines a function f, which generates the keystream in the stream cipher. The attacker also chooses random permutations to take place for a function h. h is a function which maps the s-bit state to another s-bit state. The attacker defines $g = h \circ f$ and creates lookup tables using Hellman's lookup table construction method. In the online phase, the attacker uses any instance c in the D keystreams obtained and iteratively applies g to $h(c)$ until the s-bit value $h(c)$ matches an entry in the second column of the lookup table. Once a match is found, the initial state which generated the keystream is recovered using the method used in Hellman's online phase attack. The tradeoff curve of the BS attack is given by:

$$T \cdot M^2 \cdot D^2 = 2^{2s} \text{ with } P = \frac{2^{2s}}{D} \text{ and } 1 \leq D^2 \leq T \tag{2.3}$$

The BS attack reiterates the importance that the size of the internal state of a keystream generator needed to be at least twice the secret key size so that TMD tradeoffs are worse than exhaustive keysearch.

Hong and Sarkar

Hong and Sarkar's TMDT attack [70, 71] aims to recover the secret key, as opposed to recovering the internal state or initial state in the earlier attacks. Their secret key recovery attack will be referred to as the HS attack hereafter.

In the pre-computation phase of the HS attack, the attacker first chooses random session keys, storing the secret key and the IV in the first column of the lookup table.

For each key-IV pair, the attacker generates a keystream of length $l + j$ bits, where l and j are the lengths (in bits) of the key and IV respectively. The tradeoff curve from the HS attack is the same as BS curve, but instead of it being an internal-state to keystream mapping, the HS attack uses a key-IV to keystream mapping:

$$T = M = 2^{2(l+j)} \text{ with } D = 2^{\frac{1}{4}(l+j)} \tag{2.4}$$

Thus, if the attacker has access to $D = 2^{\frac{1}{4}(l+j)}$ bits of keystream, the attacker can recover the secret key with a time and memory complexity of $T = M = 2^{\frac{1}{2}(l+j)}$. If $j < l$, the complexity of the attack is less than exhaustive key search. In order to resist the HS attack, Hong and Sarkar recommend that the IV size is at least as long as that of the secret key.

Dunkelman and Keller

The TMDT attack by Dunkelman and Keller [41], referred to hereafter as the DK attack, is a secret key recovery attack. In the DK attack, an attacker constructs lookup tables for chosen IVs. This approach is different from the HS attack, where each lookup table would consist of arbitrary IVs. Constructing lookup tables for each IV allows the attacker to take advantage of the fact the IV is a publicly known value. By constructing tables for specific IVs, the following tradeoff curve is obtained.

$$T \cdot M^2 \cdot D^2 = 2^{2(l+j)} \tag{2.5}$$

Note that this is the same tradeoff curve as the HS and BS attack. However, because of the IV table-based approach, this approach does not use multiple keystreams and hence, imposes no restrictions on the parameters. If $T = M = D$, the complexity of the attack is less than exhaustive key search as long as $2^{2j} < 2^{3l}$. That is, a stream cipher would be resistant to the DK attack if the $j \geq 1.5l$.

2.4.8 Algebraic attacks

Algebraic attacks involve solving a series of multivariate equations with a large number of unknowns, $x_0, x_1, \ldots x_{p-1}$, and which are related to a set of keystream bits $z_1, z_2, \ldots z_q$

via some functions f such that:

$$\begin{aligned} f_0(x_0, x_1, \ldots, x_{p-1}, z_1, z_2, \ldots z_q) &= 0 \\ f_1(x_0, x_1, \ldots, x_{p-1}, z_1, z_2, \ldots z_q) &= 0 \\ &\vdots \\ f_{n-1}(x_0, x_1, \ldots, x_{p-1}, z_1, z_2, \ldots z_q) &= 0 \end{aligned} \tag{2.6}$$

In the context of modern stream ciphers, $f_0, f_1, \ldots f_{n-1}$ are nonlinear functions, while $x_0, x_1, \ldots, x_{p-1}$ can either be the initial state or secret key of the cipher and z_1, z_2, z_q are the keystream bits of the stream cipher which are assumed known. Solving this series of equations to recover $x_0, x_1, \ldots, x_{p-1}$ is known as the algebraic attack.

Algebraic attacks were first used to attack stream ciphers by Courtois [32] and Courtois and Meier [34]. For a successful algebraic attack, there are three steps that need to be met. These are:

1. *Pre-computation*: In this phase, equations that relate key or internal state bits to the keystream bits are generated.

2. *Substitution and reduction*: The keystream bits are substituted into the equations formed in Step 1. These equations will now relate the unknown key or internal state bits to the keystream bits. An attacker will then try to reduce the degree of the formed equations as much as possible. Whilst one might be able to solve the series of equations without trying to reduce the degree of the equations, the time required to solve such equations might be more than an exhaustive search of the entire key space. Reducing the complexity of the equations will usually significantly reduce the time required to solve the equations.

3. *Solving*: In this phase, the system of equations is solved. This can be done using algorithms such as linearisation and Gröbner basis computations [23].

If an attacker uses linearisation to solve the system of equations, they require $\binom{i}{d}$ keystream bits, where i is the number of variables in the equation and d is the maximum degree of the equations. The maximum number of monomials E which can appear in the system of equations is $E = \sum_{a=1}^{d} \binom{i}{a}$. The complexity for solving the system of equations is approximately $E^{2.807}$ [108].

There are two methods an attacker can use to reduce the degree of the equations in generated in Step 1. One method is to guess some state bits and another is to find

low-degree multiples g of the nonlinear function f. This low-degree and non-zero g is also known as an annihilator of a Boolean function f if $f \cdot g = 0$ [86] or in the case where $f \cdot g = h$ where degree of h is less than the degree of f. Meier et al. introduced the concept of algebraic immunity to measure what the degree of this annihilating function can be. The algebraic immunity is the minimum value of d such that f or $f + 1$ admits an annihilating function of degree d [86]. It is claimed that a nonlinear function with a large algebraic immunity is resistant to algebraic attacks. However, the fast algebraic attack on Sfinks [21] by Courtois [33] indicates that a Boolean function with a high algebraic immunity value does not preclude a stream cipher immune from a successful algebraic attack.

Gröbner bases and the F4 algorithm

The Gröbner basis method introduced by Buchberger [23] is a tool which can be used to solve multivariate polynomial equations. This makes it ideal for solving systems of nonlinear equations generated by stream ciphers. This section gives a general overview of the Gröbner basis method and the F4 algorithm.

The main idea of the Gröbner basis approach is to transform a set of polynomials F to another set of polynomials G with 'certain nice properties' called a Gröbner basis such that F and G generate the same ideal [24]; that is, have the same set of solutions. These 'nice properties' may allow problems which are difficult for the polynomial F to be easily solved for the polynomial G. The Gröbner basis approach first describes the problem as a set of multivariate polynomials. The second step involves the transformation of the said multivariate polynomials before the system is solved.
Assume we have the following system of equations in GF(2):

$$f_0 : x_0 x_1 \oplus x_2 \oplus 1 = 0$$
$$f_1 : x_0 \oplus x_2 \oplus 1 = 0$$
$$f_2 : x_1 \oplus x_2 \oplus 0 = 0$$
$$F = \{f_0, f_1, f_2\}$$

As we are solving a system of equations in GF(2), the field equations below are added to

the system.

$$x_0^2 \oplus x_0 = 0$$
$$x_1^2 \oplus x_1 = 0$$
$$x_2^2 \oplus x_2 = 0$$

The set F becomes

$$F = \{x_0x_1 \oplus x_2 \oplus 1, x_0 \oplus x_2 \oplus 1, x_1 \oplus x_2, x_0^2 \oplus x_0, x_1^2 \oplus x_1, x_2^2 \oplus x_2\}$$

The Gröbner basis of F, which can be obtained by mathematical software like Magma [20], is:

$$G = \{x_0, x_1 \oplus 1, x_2 \oplus 1\} \tag{2.7}$$

Since F and G generate the same ideal, F and G will have the same solutions. The elimination property in Gröbner bases ensures that, in case G has a finite set of solutions, G contains a univariate polynomial in x [24]. In this example, the univariate polynomials in x contained in G is x_0, $x_1 \oplus 1$, $x_2 \oplus 1$. Solving G for x_0, x_1, x_2 gives the following solutions:

$$x_0 = 0, x_1 = 1, x_2 = 1$$

In this way, we can obtain all the solutions of G, and hence, of F. The Gröbner basis method has an advantage over linearisation that the Gröbner basis method requires less keystream to solve the system of equations than with the linearisation method and in some cases, a solution to the system of equations can be found in less time than with the linearisation method. These two characteristics were demonstrated by Al-Hinai [1] in his algebraic analyses of clock-controlled stream ciphers.

The F4 algorithm [46] in Algorithm 1, reproduced here from the description in Al-Hinai's thesis [1], is an algorithm published by Faugère to compute a Gröbner basis. Using the Gröbner basis method, the reduction of polynomials in the system of equations is performed using a pair selection strategy. Although this pair selection strategy may be efficient for small systems of equations, it may not necessarily be the case if the system of equations is large. In the F4 algorithm, linear algebraic techniques are used to reduce these polynomials. The polynomials are linearised and a matrix of their coefficients is then constructed, and the row echelon form of this matrix computed. Using this matrix,

the F4 algorithm can process the polynomials in the system of equations more efficiently, and speed up the calculation of a Gröbner basis. This thesis uses the F4 algorithm implemented in Magma [20] for the computation of slid pairs in Section 4.2 and the the recovery of the initial states of some Trivium-like stream ciphers in Section 4.3.

Algorithm 1 The F4 algorithm

Inputs

1: Input: A set of polynomials F in n unknowns.
2: Output: Set of solutions to the system $F = 0$ for n unknowns.

Steps

1: Let $G = F$
2: Select a set of polynomials S from G.
3: Linearise polynomials in S and form their coefficient matrix, A.
4: Add additional reduction polynomials into A.
5: Compute the row echelon form $\tilde{A}$ of A.
6: Form reduced polynomials $\tilde{S}$ of S from A.
7: Add $\tilde{S}$ to G and delete unnecessary polynomials in G.
8: Repeat steps 2 to 7 until G is a Gröbner basis.
9: Perform back substitution from the univariate polynomial to obtain solution to the system of equations.

2.5 Conclusion

This chapter presented an overview of the structure of keystream generators for binary additive stream ciphers and the techniques commonly used to analyse them. In Section 2.1, the phases involved in using a keystream generator were discussed. These include initialisation and keystream generation. In Section 2.2.1, nonlinear Boolean functions, a common component used to provide explicit nonlinearity in stream ciphers was defined. Various properties of nonlinear Boolean functions were also discussed, while in Section 2.2.2, the two types of state-update functions used to update the internal state of shift registers were introduced. In Section 2.2.3, the various methods of selecting tap settings to the output function was discussed. The various ways a keystream generator can use the components introduced in Sections 2.2.1–2.2.3 was discussed in Section 2.3. These combinations provide a framework in which analyses of various keystream generators presented in this thesis is based. Common cryptanalysis techniques used for analysing stream ciphers are reviewed in Section 2.4. These include guess and determine attacks, distinguishing attacks, divide-and-conquer attacks, correlation

attacks, linear, and differential cryptanalysis, time-memory-data tradeoff attacks, and algebraic attacks. In particular, the F4 algorithm will be used in the algebraic analysis in Section 4.2 and Section 4.3. The combination of a divide-and-conquer attack and the algebraic attack will be used in the analysis of Trivium-like stream ciphers in Section 4.3. The effectiveness of state convergence on time-memory-data tradeoff attacks, correlation attacks, algebraic attacks and differential attacks are investigated in Section 6.5.

Chapter 3

m-tuple distributions in nonlinear filter generators

This chapter examines the distribution of m-tuples in keystream sequences formed by keystream generators using a linear state-update and a nonlinear output function. If a keystream sequence produced was truly random, the distribution of m-tuples is expected to be uniform. The apparent randomness of the keystream generated by these keystream generators is one of the many criteria used in evaluating the security of a keystream generator. In particular, the uniformity of m-tuple distributions is an important criterion, as non-uniformity of these distributions is a key indication that the keystream sequence is not random.

This chapter is organised as follows: In Section 3.1, we describe previous work which have been done with regards to the anaylsis of m-tuple distributions of keystream sequences produced by NLFGs. In Section 3.2, we describe the experimental goals and design in our analysis of m-tuple distributions of keystream sequences produced by NLFGs. In Section 3.3, we describe the results of our experiments, while in Section 3.4, we discuss the potential impact of biased m-tuple distributions in keystream sequences produced by NLFGs. Section 3.5 concludes this chapter.

3.1 Existing analysis on m-tuple distributions of NLFGs

Groth [60] investigated the application of non-linear functions to LFSRs to generate sequences with high linear complexities. Groth noted that certain multiplier arrangements

(his term for nonlinear filter generators) did not produce sequences with noise-like characteristics. Groth's analysis of noise-like characteristics was based on the analysis of the distributions of run lengths. He also described a method for constructing multiplier arrangements which had ideal noise-like characteristics. Extending the work of Groth, Siegenthaler, Kleiner and Forré [102] show that if an LFSR of length $n \times r$, where n and r are arbitrary integers, has a primitive feedback function and the Boolean function used as the output function is balanced, the keystream sequence produced by the NLFG has an ideal distribution for non-overlapping m-tuples.

Golić [54] showed that the distribution of m-tuples, for $m = 1$ of keystream sequences produced by a nonlinear filter generator is almost uniform if (and only if) the n-input nonlinear Boolean function $f(x_0, \ldots, x_{n-1})$ is balanced for each value of $(x_1, \ldots, x_{n-1})$, that is, if

$$f(x_0, \ldots, x_{n-1}) = x_0 \oplus g(x_2, \ldots, x_{n-1})$$

or if the Boolean function $f(x_0, \ldots, x_{n-1})$ is balanced for each value of $(x_0, \ldots, x_{n-2})$, that is, if

$$f(x_0, \ldots, x_{n-1}) = x_{n-1} \oplus g(x_0, \ldots, x_{n-2})$$

Simpson [105] later showed that, for any given NLFG, a balanced nonlinear Boolean function applied to the stages of a LFSR with primitive feedback function and non-zero initial state results in an output keystream that is close to uniform. That is, the difference between the number of zeroes and the number of ones occurring in one period of the keystream sequences is exactly one. Much less is known about the frequency distribution of m-bit patterns in the NLFG output sequence for $m > 1$.

Anderson [4] discusses the distribution of m-tuples in the NLFG output sequence in the context of a correlation attack on the NLFG. Anderson considers that the common NLFG correlation attack strategy, which regards the keystream as a series of individual bits, discards information about the nonlinear structure of the filter function. Instead, for a given m-input Boolean filter function, he defines an augmented function which maps a $2m - 1$-bit input to an m-bit output. To illustrate this, he constructed a NLFG. The nonlinear 5-input Boolean function used in his NLFG had the following nonlinear Boolean function:

$$x_0 \oplus x_1 \oplus (x_0 \oplus x_2)(x_1 \oplus x_3 \oplus x_4) \oplus (x_0 \oplus x_3)(x_1 \oplus x_2)(x_4)$$

The feedback function of the nine bit LFSR used was not specified. Anderson applied the 5-input filter function 5 times in succession, assuming that the inputs to the filter function are from consecutive positions of the underlying sequence, although this is not stated explicitly in the paper. For this augmented function Anderson examined the number of inputs which produced each 5-tuple output and reported that the distributions were unbalanced.

For another NLFG which uses the following nonlinear Boolean function:

$$x_0 \oplus x_1 \oplus x_2 \oplus x_0x_3 \oplus x_1x_4 \oplus x_2x_5$$

Anderson noted that certain m-tuples did not occur at all, while the all-zero tuple occurred 100 times. Leakage of state information also occurred. For example, 12 different input states gave the same 5-tuple output. However, in all 12 cases, the contents of some internal state stages used as input were all the same. The particular function used was a bent function, which, due to its unbalanced nature, is not suitable as the filter function for a nonlinear filter generator. However, it was not clear from Anderson's paper whether distributions with non-occurring m-tuples are possible when balanced functions are used. Furthermore, the relationship between the characteristics of the Boolean function and the degree of bias in the output is not revealed.

3.2 Experimental goals and design

There are two main components of a NLFG: the LFSR and the nonlinear Boolean function. The goal of our experiment was to determine if some properties of sequences produced by nonlinear filter generators are affected by combination of choices in LFSR and nonlinear Boolean functions. In particular, the following questions which can arise when analysing output sequences produced by nonlinear filter generators are answered:

- What are the m-tuple distribution of the sequences produced by NLFGs?
- Do different choices in the LFSR feedback function and tap settings to the nonlinear Boolean function affect the distribution of the m-tuples of sequences produced by NLFGs?

Our work extends the earlier work of Anderson [4], where the value of m was used as both the number of inputs to the filter function and the length of the bit patterns examined in the NLFG output sequence, due to the selection of inputs from consecutive

stages of the LFSR. In our investigation, we make a clear distinction between these two parameters. We denote the number of inputs to the filter function by n, and consider the distribution of m-tuples in the NLFG output sequence for $m = \{1, 2, \ldots, s\}$, where s is the length of the LFSR.

To provide resistance to guess-and-determine style attacks, NLFG-style designs now commonly take the inputs to the filter from positions in the LFSR which are not consecutive. Ideally these tap settings form a full positive difference set. The effect of this change in the positions of the input stages of the m-tuple pattern distribution of the NLFG output also sequence warrants further investigation.

In order to accurately determine the m-tuple distribution, it is necessary to produce an entire period of the keystream sequence. This constrained the length of the LFSRs used in our experiments. We chose 10 different LFSRs with lengths ranging from 13 to 20 bits for use in our experiments. The primitive polynomials used as the LFSR feedback functions are:

$$R1 : x^{13} + x^4 + x^3 + x^1 + 1$$
$$R2 : x^{13} + x^{12} + x^{10} + x^9 + x^6 + x^3 + 1$$
$$R3 : x^{15} + x^{10} + x^5 + x^1 + 1$$
$$R4 : x^{15} + x^1 + 1$$
$$R5 : x^{16} + x^5 + x^3 + x^2 + 1$$
$$R6 : x^{16} + x^{15} + x^{14} + x^8 + x^4 + x^3 + 1$$
$$R7 : x^{18} + x^5 + x^2 + x^1 + 1$$
$$R8 : x^{18} + x^{16} + x^{15} + x^{12} + x^{11} + x^9 + x^7 + x^6 + x^5 + x^4 + x^2 + x^1 + 1$$
$$R9 : x^{20} + x^{19} + x^4 + x^3 + 1$$
$$R10 : x^{20} + x^{19} + x^{18} + x^{15} + x^{14} + x^{12} + x^{11} + x^{10} + x^4 + x^2 + 1$$

The three balanced nonlinear Boolean functions which were chosen for use as nonlinear filters appear in the cryptographic literature. We denote these $F1$, $F2$, and $F3$. $F1$ is a 5-bit Boolean function used in the Grain stream cipher [66]. $F2$ is a 6-bit Boolean function obtained from a report by Faugère and Ars [47]. $F3$ is a 7-bit Boolean function

Table 3.1: Cryptographic characteristics of the Boolean functions

Function	Algebraic order	Nonlinearity	Correlation immunity
$F1$	3	12	1
$F2$	3	24	0
$F3$	4	56	2

Table 3.2: Tap settings used in our experiments

LFSR	$F1$		$F2$		$F3$	
	$T1$	$T2$	$T1$	$T2$	$T1$	$T2$
$R1$ & $R2$	0,1,4,8,12	0,1,3,7,12	0,1,2,5,9,12	0,2,5,7,10,12	0,1,3,5,19,11,12	0,2,3,5,8,10,12
$R3$ & $R4$	0,1,4,9,14	0,4,6,13,14	0,1,4,8,10,14	0,2,3,10,13,14	0,1,4,5,10,13,14	0,2,3,7,9,11,14
$R5$ & $R6$	0,1,3,11,15	0,1,5,11,15	0,1,4,8,13,15	0,3,7,8,11,15	0,1,3,6,10,12,15	0,2,3,7,10,13,15
$R7$ & $R8$	0,3,8,13,17	0,2,5,9,17	0,2,3,8,15,17	0,1,4,10,12,17	0,1,3,9,11,14,17	0,2,8,9,12,15,17
$R9$ & $R10$	0,1,3,7,19	0,2,7,9,19	0,1,3,7,12,19	0,1,4,11,13,19	0,1,3,8,11,17,19	0,3,5,9,10,14,19

used in the Pomraranch stream cipher [75]. The ANFs of $F1$ and $F2$ are:

$$F1 : x_1 \oplus x_4 \oplus x_0x_3 \oplus x_2x_3 \oplus x_3x_4 \oplus x_0x_1x_2 \oplus x_0x_2x_3 \oplus x_0x_2x_4 \oplus x_1x_2x_4 \oplus x_2x_3x_4$$

$$F2 : x_3 \oplus x_4 \oplus x_0x_1x_2 \oplus x_1x_2x_5 \oplus x_0x_1 \oplus x_2x_3 \oplus x_4x_5$$

The ANF of $F3$ is omitted from this chapter due to its size. However, the truth table output is shown in Appendix A. Specific characteristics of these three Boolean functions (the algebraic order, nonlinearity and correlation immunity) are shown in Table 3.1.

For each of the feedback polynomials, three different sets of tap settings for the Boolean functions were chosen. One set of tap settings used consecutive taps from the LFSR and two sets used uneven (or FPDS where possible) taps from the LFSR. In the uneven tap settings scenario, two sets of tap settings were used. These are denoted $T1$ and $T2$ in Table 3.2.

For each LFSR, nonlinear filter function and tap setting combination, the NLFG was run to generate a sequence $2^s - 1 + m - 1$ in length. The frequency distribution of m-tuples was calculated for $m = 2$–13. From this, the m-tuple which occurs least and most frequently for m-tuples of sizes 2–13 were noted. The standard deviation of the frequencies of occurrence is a useful summary measure for the m-tuple distribution of

Table 3.3: 3-tuple distribution of a NLFG sequence

m-tuple	Expected occurrences	Observed occurrences	Standard Deviation	Proportion of all 3-tuples	Standard deviation of all 3-tuples
000	4095	2815	768.208	0.085 910	0.023 445
001	4096	4352		0.132 817	
010	4096	4864		0.148 442	
011	4096	4352		0.132 817	
100	4096	4352		0.132 817	
101	4096	4864		0.148 442	
110	4096	4352		0.132 817	
111	4096	2816		0.085 940	

a sequence. The smaller the standard deviation, the closer the m-tuple distribution of the sequence is to a uniform distribution. To enable comparisons where different sized LFSR was used, the proportions of each m-tuple across 2^m possible tuples is calculated and the standard deviation of this proportion is also calculated.

Recall the m-tuple distribution for the maximum length sequence produced by a LFSR is almost uniform. For example, the 3-tuple distribution for a 15-bit LFSR $R3$ with the nonlinear filter function $F1$, which takes as input consecutive taps from the nonlinear Boolean function is given in Table 3.3. The m-tuple distribution of the NLFG sequence generated is far from uniform. This is shown by the large standard deviation.

3.3 Experimental results

Our experiments involved 90 NLFGs, comprising combinations of 10 different LFSRs, three nonlinear Boolean functions and three sets of tap settings for each LFSR-nonlinear Boolean function combination. For each NLFG, a sequence of length $2^s + m - 1$ bits was generated and the output sequence was examined for m-tuples for values of m ranging from 2–13 bits. We make a number of observations based on the results of our experiment. The factors which could impact on the m-tuple distribution include the positions of the inputs to the filter functions, the number of inputs to the filter function, and the length and feedback function of the LFSR. Detailed results from the experiments can be found in Appendix B.

Observation 3.1 The m-tuple distribution of NLFG output sequences is generally non-uniform.

Note that this observation supports the earlier findings by Anderson. We also note that the degree of non-uniformity varies depending on the combination of LFSR feedback function, the nonlinear filter functions and positions of input taps to the filter function. There are a few cases when the m-tuple output of the NLFG was almost uniform for smaller m-tuple values. For example, the 3-tuple distribution for a NLFG using the $R5$, $F1$ and T1 combination had the m-tuple distribution expected of a maximum length sequence. However, as the value of m increased, the distribution became less uniform. Close to uniform distributions were more frequent when $m < 4$ and when uneven tap settings were used. With the exception of one case when $m = 5$, all m-tuple distributions when $m > 4$ were not uniform for NLFGs which used uneven tap settings. Almost uniform m-tuple distributions never occurred for consecutive tap settings for all m-tuples tested.

Observation 3.2 The m-tuple distribution is less uniform when tap settings are consecutive.

When comparing the m-tuple distribution for the output sequences obtained from NLFGs with the same LFSR and filter function but with different positions in the LFSR selected for inputs to the filter function, the distributions when the tap settings are consecutive are more varied than when the tap settings are uneven. For example, in the case for a 5-tuple distribution of a NLFG using the feedback function $R3$ with consecutive tap settings and the $F1$ as the nonlinear filter, the least frequent 5-tuple occurred 320 times and the most frequent 5-tuple occurred 1665 times. The standard deviation obtained was 320.056. When the same feedback function and filter function was used in a NLFG with uneven tap setting $T1$, the least frequent 5-tuple occurred 731 times and the most frequent 5-tuple occurred 1315 times. The standard deviation obtained 133.982. For the same LFSR and nonlinear filter with the uneven tap setting $T2$, the minimum obtained was 1005 and the maximum obtained was 1043 and the standard deviation obtained 10.264. This trend was apparent for every NLFG sequence examined.

Table 3.4: Value of m when non-occurring m-tuples start appearing

Length of LFSR	Minimum value of m
13	11
15	12
16	13

Observation 3.3: For some NLFGs with balanced Boolean functions, some m-tuples do not occur.

It is possible that particular m-tuples may not appear in a NLFG output sequence. In our experiments, we noted that this can occur when the nonlinear Boolean functions are balanced. The number of different m-tuples which do not appear in the output sequence is higher for NLFGs using consecutive tap settings than when uneven tap settings are used. For example, a NLFG using the feedback function $R1$, $F2$ Boolean function and the consecutive tap settings has 20 non-occurring 10-bit tuples. In contrast, a NLFG using the same feedback function and Boolean function with the $T1$ tap setting has only three non-occurring 10-bit tuples.

Equivalently, the minimum value of m for which some m-tuples do not occur is lower for consecutive tap settings than when uneven tap settings are used. For example, when the NLFG uses the $R1$ feedback function with the $F1$ nonlinear Boolean function, the value of m when non-occurring m-tuples occur starts at $m = 8$ for consecutive tap settings. In contrast, the value of m when non-occurring m-tuples occurs for the uneven tap settings $T1$ and $T2$ is $m = 11$.

For uneven tap settings, this minimum value of m increases with the length of the LFSR. For example, when the NLFG uses the $F1$ nonlinear Boolean function taking as input selected stages from the LFSR using the tap setting $T1$, the value of m when non-occurring m-tuples appear for a given LFSR length is shown in Table 3.4

As the m-tuple size increases, the number of non-occurring m-tuples also increases for both consecutive and uneven tap settings. We also noted from our experiments that, for a given choice of filter function and tap setting, as the size of the LFSR increased, the number of m-tuples which do not appear in the output sequence remained constant once a certain size was reached for the LFSRs we tested. For example, when using consecutive tap settings, there were 17 non-occurring 10-bit m-tuples for the 6-input Boolean function $F2$ when $s \geq 15, \ldots, 20$.

Observation 3.4: Distribution of m-tuples for NLFGs using consecutive tap settings are similar regardless of the size of the LFSR.

The standard deviations of the proportions of m-tuples for NLFGs using consecutive tap settings were similar regardless of the size of the LFSR but varied with the choice of Boolean functions. For example, the standard deviation in terms of proportions for various NLFGs when used with the $F1$ nonlinear Boolean function for $m = 4$ ranged from 0.015 625 to 0.015 642.

However, this was not the case for uneven tap settings. For NLFGs using uneven tap settings, the standard deviation of the m-tuples in terms of proportions are different for different LFSR lengths, tap settings and Boolean functions. This difference can be slightly or significantly different. For example, consider the example when a NLFG was constructed using the $F1$ Boolean function and the $R1$ feedback function. When these particular components were used with the $T1$ tap setting, the standard deviation for $m = 10$ in terms of proportions was 0.000 352. When the $T2$ tap setting was used, the standard deviation was 0.000 346. The difference between these two standard deviations, 0.000 006, is quite small.

However, consider the situation when a NLFG was constructed using the $F2$ Boolean function and the $R9$ feedback function. When these particular components were used with the $T1$ tap setting, the standard deviation for $m = 4$ in terms of proportions was 0.001 007. When the $T2$ tap setting was used, the standard deviation was 0.002 889. The difference between these two standard deviations, 0.001 882, is large by comparison.

Observation 3.5: Distribution of m-tuples for NLFGs for a given sized LFSR using consecutive tap settings are largely unaffected by the LFSR feedback polynomial.

The distribution of m-tuples for NLFGs for a fixed LFSR length using consecutive tap settings and a fixed Boolean function are largely unaffected by the LFSR feedback polynomial. For example, consider the m-tuple distribution, for $m = 5$, of NLFGs constructed using the F2 Boolean function and the feedback functions $R3$ and $R4$. In both examples, the standard deviation and the standard deviation in terms of proportions is 219.335 and 0.006 697 respectively.

3.4 Discussion

In this section, we consider the potential impact of biased m-tuple distributions in the output sequences from NLFGs. These sequences are used as keystream for stream ciphers, in initialisation functions and as building blocks for message authentication codes (MACs). The potential impact in each case is discussed below.

Some stream ciphers use the output of NLFGs as keystream to encrypt messages. There is a potential major flaw in this design choice if the NLFG has a highly biased m-tuple distribution. Firstly, there is a possibility of mounting a distinguishing attack on the keystream. An attacker who is able to perform a statistical analysis on the m-tuple outputs might be able to mount a distinguishing attack based on the frequency of the various m-tuples in the keystream. Another possible attack is a ciphertext-only attack on the stream cipher. Biased m-tuple distribution combined with the redundancy of the plaintext may provide leakage of information to allow an attack to partially decrypt ciphertext messages without initial knowledge of the secret key. An example of a ciphertext-only attack which exploits biased eight-tuple distributions in RC4 [5] is the ciphertext-only attack by Mantin and Shamir [80].

Recall our description of the initialisation phase of stream ciphers in Section 2.1.1. Modern stream ciphers use a secret key and a publicly known IV as input to an initialisation function to generate the initial state of the keystream generator. This initialisation function should be nonlinear. A potential problem with using the output of a nonlinear filter for initialisation is that if some m-tuples occur more often than others, then it is possible that some initial states will occur more often than others, resulting in biased keystream distribution. In the case where some m-tuples do not occur at all, this means some initial states might not occur at all for any key-iv pair, reducing the effective key space of the stream cipher.

In recent years, stream cipher designers have proposed ciphers which aim to provide simultaneous confidentiality and integrity protection. These are commonly called authenticated encryption (AE) stream ciphers. Some AE stream ciphers use nonlinear filter generators in components used to compute the Message Authentication Code (MAC) tag. One example of such a cipher is Sfinks [21]. For MAC algorithms which make use of nonlinear filters, the distribution of MAC tags for messages may not be uniform. An attacker may be able to exploit this in a MAC collision attack. A successful MAC collision attack may allow an attacker to generate a forged message which has the same MAC value as an authentic message, which could deceive a third party into believing that the contents of a forged message is authentic.

3.5 Conclusion

In this chapter, we have extended the work of Anderson, who reported that the 5-tuple distribution a NLFG was not uniform. Anderson also observed that the m-tuple distribution of the keystream produced by this particular NLFG was not uniform. However, Anderson did not investigate the effect different tap settings can have on the m-tuple distributions of keystream produced by NLFGs nor did he investigate if non-occurring m-tuples were present for NLFGS which use balanced nonlinear Boolean functions as output functions.

We examined the keystream produced by 90 different NLFGs. We analysed the distribution of m-tuples in the output sequence for $m = \{2, 3, \ldots 13\}$. Our investigation came up with three main results. Firstly, we show that the m-tuple distributions of NLFGs are non-uniform except in some cases for small values of m (where $m \leq 4$), regardless of tap settings used, although the bias is generally greater when the tap settings to the filter function are consecutive. Secondly, in some cases, there are some m-tuples which do not occur at all in the outputs. This happens for small m-values ($m \geq 8$) if the NLFGs use consecutive tap settings rather than uneven tap settings ($m \geq 11$). Thirdly, our experiments also show that the frequency distributions of m-tuples for NLFGs using consecutive tap settings are similar regardless of the size of the LFSR.

The findings in this experiment may have cryptanalytic implications. The significant m-tuple bias in the output sequence may be exploited in attacks ranging from distinguishing attacks [80] to ciphertext-alone attacks [37]. If a NLFG is to be used in a cryptographic application, the use of consecutive tap settings in NLFGs is not recommended.

In Section 4.1, we analyse the distribution of m-tuples in keystream sequences produced by the complementary concept of the NLFG, the linearly filtered nonlinear feedback shift register.

Chapter 4

Analysis of linearly filtered nonlinear feedback shift registers

The complementary concept to the nonlinearly filtered LFSR sequence analysed in Chapter 3 is a linearly filtered nonlinear feedback shift register sequence, which is investigated in this chapter. In this case, the internal state of a component shift register is updated using a nonlinear feedback function. Linear combinations of the values in certain stages of this internal state form the keystream output.

When comparing the keystreams produced by an LFSR and a NLFSR, clearly the NLFSR sequence will not be vulnerable to Berlekamp-Massey attack. Also, NLFSRs may provide resistance to the algebraic [31, 34] and correlation attacks [87, 100] which keystreams formed by nonlinear combinations of LFSR-only sequences are susceptible to.

The disadvantage is that the properties of the sequences produced by the NLFSRs used in most keystream generators are not well-known. In particular, important properties like the period of the keystream produced for some NLFSRs are not known and designers can only estimate what this period may be. For example, in the specifications of Trivium [26, page 261], the designers state

> "Because of the fact that the internal state of Trivium evolves in a non-linear way, its period is hard to determine."

As this period is difficult to determine, keystream generators which use NLFSRs to produce keystreams run the risk of producing keystream sequences with short periods.

The minimum period of keystream sequences produced by NLFSRs is also an open problem.

The objective of this chapter is to analyse stream ciphers which use the model of a linearly filtered NLFSR to generate keystream, with a specific focus on the Trivium stream cipher. An analysis of the distributions of overlapping m-tuples for linearly filtered NLFSR sequences is performed to determine the uniformity of its distribution. Two separate analyses of Trivium are conducted. The first analysis investigates the characteristics of Trivium's state-update function, while the second analysis investigates the security of its keystream generation process against algebraic attacks.

The contributions of this chapter are as follows. In Section 4.1, we perform an experimental analysis of overlapping m-tuple distributions for linearly filtered NLFSR sequences. In Section 4.2.3, we extend the work of Priemuth-Schmid and Biryukov [92] and search for additional types of slid pairs in the Trivium cipher, and in doing so, attempt to establish a new lower-bound on the minimum period of Trivium. In Section 4.3, we perform a new algebraic analysis on Trivium and its scaled-down variants, Bivium-A and Bivium-B, using the technique introduced by Berbain et al. [10] that was originally used to attack a modified version of the Grain cipher. This technique is combined with Raddum's relabelling technique, that was originally applied to his analysis on Trivium [93], to determine whether this combined technique can be applied to Trivium and its variants.

4.1 m-tuple distributions in linearly filtered nonlinear feedback shift registers

In Chapter 3, we analysed the m-tuple distribution of sequences produced by NLFGs. The same questions posed at the beginning of the Chapter 3 form the basis into the investigation of m-tuples of sequences produced by linearly filtered NLFSRs. That is:

- What are the m-tuple distributions of the sequences?
- Do choices of tap positions to the linear function affect the distribution of the m-tuples?

In this section, we perform computer experiments simulating a series of linearly filtered NLFSRs to answer these questions. Some properties of some NFLSR sequences are known. These properties were reviewed in Section 2.2.2. However, Gammel and

Göttfert's paper appears to only consider the direct output functions and does not analyse the m-tuple distribution of keystreams generated from output functions consisting of linear combinations of the NLFSR outputs. In this section, we analyse the m-tuple distributions of 20 different linearly filtered NLFSRs, and analyse some properties of these sequences.

4.1.1 Experimental goals and design

There are two main components of a linearly filtered NLFSR, the NLFSR and the linear combining function. The goal of the experiments was to determine how parameters associated with these components affect the output sequence of the linearly filtered NLFSR. In particular, we analyse how the number of inputs and the tap positions for inputs into the linear combining functions affect the m-tuple distribution of the keystream for keystream generators based on a single NLFSR.

In order to accurately determine the m-tuple distribution, it is necessary to produce an entire period of the keystream sequence. This constrained the length of the NLFSRs used in our experiments. Four NLFSRs, $R1$, $R2$, $R3$, and $R4$, of length s, for s equal to 22, 25, 28, and 29 bits were chosen. These NLFSRs are the same NLFSRs used in a stream cipher designed by Gammel et al. [50] and produce sequences which have a period of $2^s - 1$. This allows us to compare their m-tuple distributions to the m-tuple distribution of maximal-length LFSR sequences. The state-update functions of the four selected NLFSRs are:

$$A_i(t+1) = \begin{cases} A_0(t) \oplus A_5(t) \oplus A_6(t) \oplus A_7(t) \oplus A_{10}(t) \oplus A_{11}(t) \oplus \\ A_{12}(t) \oplus A_{13}(t) \oplus A_{17}(t) \oplus A_{20}(t) \oplus A_2(t)A_7(t) \oplus \\ A_4(t)A_{14}(t) \oplus A_8(t)A_9(t) \oplus A_{10}(t)A_{11}(t) \oplus \\ A_1(t)A_4(t)A_{11}(t) \oplus A_1(t)A_4(t)A_{13}(t)A_{14}(t) & i = 21, \\ A_{i+1}(t) & 0 \le i \le 20. \end{cases}$$

$$B_i(t+1) = \begin{cases} B_0(t) \oplus B_1(t) \oplus B_3(t) \oplus B_5(t) \oplus B_6(t) \oplus B_7(t) \oplus \\ B_9(t) \oplus B_{12}(t) \oplus B_{14}(t) \oplus B_{15}(t) \oplus B_{17}(t) \oplus \\ B_{18}(t) \oplus B_{22}(t) \oplus B_1(t)B_6(t) \oplus B_4(t)B_{13}(t) \oplus \\ B_8(t)B_{16}(t) \oplus B_{12}(t)B_{15}(t) \oplus B_5(t)B_{11}(t)B_{14}(t) \oplus \\ B_1(t)B_4(t)B_{11}(t)B_{15}(t) \oplus B_2(t)B_5(t)B_8(t)B_{10}(t) & i = 24, \\ B_{i+1}(t) & 0 \le i \le 23. \end{cases}$$

$$C_i(t+1) = \begin{cases} C_0(t) \oplus C_1(t) \oplus C_2(t) \oplus C_7(t) \oplus C_{15}(t) \oplus C_{17}(t) \oplus \\ C_{19}(t) \oplus C_{20}(t) \oplus C_{22}(t) \oplus C_{27}(t) \oplus C_9(t)C_{17}(t) \oplus \\ C_{10}(t)C_{18}(t) \oplus C_{11}(t)C_{14}(t) \oplus C_{12}(t)C_{13}(t) \oplus \\ C_5(t)C_{14}(t)C_{19}(t) \oplus C_6(t)C_{10}(t)C_{12}(t) \oplus \\ C_6(t)C_9(t)C_{17}(t)C_{18}(t) \oplus C_{10}(t)C_{12}(t)C_{19}(t)C_{20}(t) & i = 27, \\ C_{i+1}(t) & 0 \le i \le 26. \end{cases}$$

$$D_i(t+1) = \begin{cases} D_0(t) \oplus D_4(t) \oplus D_5(t) \oplus D_6(t) \oplus D_9(t) \oplus D_{10}(t) \oplus \\ D_{11}(t) \oplus D_{16}(t) \oplus D_{18}(t) \oplus D_{22}(t) \oplus D_{25}(t) \oplus \\ D_1(t)D_2(t) \oplus D_3(t)D_{10}(t) \oplus D_5(t)D_8(t)D_{10}(t) & i = 28, \\ D_{i+1}(t) & 0 \le i \le 27. \end{cases}$$

Five types of linear output functions were chosen. The function parameters include the number of inputs and also the position of inputs. Recall from Section 2.2.3 the definition of a FPDS. Inputs which form a Full Positive Difference Set (FPDS) ensure that the differences between all pair-wise indexes of stages used as inputs to the output function are distinct. For example, inputs to a four-input output function from a ten-stage NLFSR which form a FPDS can be $\{0, 1, 4, 9\}$.

1. A single-input output function $T1$, which takes the output (the 0'th stage) of each NLFSR as the keystream bit.

2. A three input linear function $T2$, where the input positions form a Full Positive Difference Set (FPDS), with the restriction that two inputs must be from the 0'th

Table 4.1: Tap settings used in experiments

Taps	$R1$	$R2$	$R3$	$R4$
$T1$	0	0	0	0
$T2$	0,1,21	0,1,24	0,1,27	0,1,28
$T3$	0,1,3,8,18,21	0,1,3,7,19,24	0,1,3,8,17,23,27	0,1,3,8,12,22,28
$T4$	0,1,2	0,1,2	0,1,2	0,1,2
$T5$	0,1,2,3,4,5,6	0,1,2,3,4,5,6	0,1,2,3,4,5,6	0,1,2,3,4,5,6

and $(s-1)$'th stage of the respective NLFSRs.

3. A six or seven input linear function $T3$, where the input positions form a Full Positive Difference Set (FPDS), with the restriction that two input bits must be from the 0'th and $(s-1)$'th stage of the respective NLFSRs.

4. A three input linear function $T4$ which takes inputs from three consecutive stages (CONS) 0, 1, 2 of the respective NLFSRs as the output for the keystream.

5. A linear function $T5$ which takes inputs from seven consecutive stages in the respective NLFSR.

The tap settings for each function $T1$, $T2$, ..., $T5$ when applied to each of the selected NLFSRs $R1$, $R2$, $R3$, and $R4$ is summarised in Table 4.1.

4.1.2 Experimental results

Our experiments consisted of forming keystream generators using one of four NLFSRs and five different linear combining functions, giving us a total of 20 linearly filtered nonlinear feedback shift register-based keystream generators. A sequence of $2^s-1+m-1$ bits for each keystream generator was generated and the output sequence was examined for m-tuples with m ranging from 2–s. We make a number of observations based on the results of our experiment. These results are summarised in Appendix C, which records four observations in our experiment for each of the output functions $T2$ to $T5$. These are:

- Min: In a period of a particular keystream generator, the count for the minimum occurrence of an m-tuple.

- Max: In a period of a particular keystream generator, the count for the maximum occurrence of an m-tuple.
- N.O: In a period of a particular keystream generator, the number of non-occuring m-tuples.
- S.D In a period of a particular keystream generator, the standard deviation of the frequencies of occurrence of all m-tuples.

Observation 4.1: The m-tuple distribution of keystreams generated using single-input linear functions are almost uniform.

Our experiments confirm Gammel and Göttfert's results that the keystream sequence generated using NLFSRs which have a period of $2^s - 1$ have an almost uniform m-tuple distribution for sizes ranging from 1 to s. That is, given an output sequence of length $2^s - 1 + m - 1$ each m-tuple occurs 2^{s-m} times, except for the all-zero m-tuple which occurs $2^{s-m} - 1$ times. Note that this observation only holds for single-input linear functions.

Observation 4.2: The m-tuple distribution of keystreams generated using NLFSRs with CONS or FPDS linear functions can be non-uniform

This phenomenon is observable in all keystream generators tested. We take the m-tuple distributions of keystream generators using $R2$ with different linear functions as an example, a portion of which is reproduced in Table 4.2. When $R2$ was used with the T2 linear function, almost uniform distribution of m-tuples were observed until $m = 5$, but the m-tuple distributions were non-uniform from then onwards. When $R2$ was used with the T4 linear function, almost uniform distribution of m-tuples were observed until $m = 23$ and were non-uniform from then onwards.

Observation 4.3: The m-tuple distribution of linear functions with FPDS inputs is less uniformly distributed than linear functions with consecutive inputs.

The m-tuple distribution of the keystream generated by linear functions which takes as inputs, taps which form a FPDS, are generally less uniformly distributed as compared to the m-tuple distribution of the keystream generated by linear functions which takes as inputs taps which are consecutive. Furthermore, this non-uniformity started occurring

Table 4.2: Excerpt for Observation 4.2

Uneven Taps ($T2$)			CONS taps ($T4$)		
m-tuple	Min	Max	m-tuple	Min	Max
4	2 097 151	2 097 152	21	15	16
5	1 048 575	1 048 576	22	7	8
6	523 967	524 608	23	3	4

Table 4.3: Excerpt for Observation 4.3

Uneven Taps ($T3$)				CONS taps ($T5$)			
m-tuple	Min	Max	S.D	m-tuple	Min	Max	S.D
7	2 097 151	2 097 152	0.088 39	21	127	128	0.000 69
8	1 048 575	1 048 576	0.062 50	22	64	32	0.000 49
9	524 128	524 448	115.377 11	23	24	40	4.109 61

earlier for keystream generators using a FPDS-input function compared to keystream generators using a CONS-input function.

We take the m-tuple distributions for sequences produced by keystream generators using $R3$ with different linear functions as an example, a portion of which is reproduced in Table 4.3. For example, when $R3$ used the $T5$ function, the m-tuple distribution was the same as that expected of an maximal length sequence up to $m = 22$. In contrast, the m-tuple distribution of $R3$ when it used the FPDS linear $T3$ function was the same as that for a primitive LFSR only up to $m = 8$.

The greater unevenness of the distribution can also be seen in the higher standard deviation values for $R3$ when used with the $T3$ function as compared the standard deviation values of $R3$ when used with the $T5$. For example, when $m = 23$, the standard deviation obtained with the $T5$ function is 4.109 61, as compared to the standard deviation value of 5.639 63 obtained when the $T3$ function was used.

Note that this finding is the direct opposite of the experimental results of the m-tuple distribution of NLFGs performed in Chapter 3. In that chapter, Observation 3.2 noted that the m-tuple distribution was less uniform when the tap settings were consecutive.

Table 4.4: Excerpt for Observation 4.4

Uneven Taps ($T2$)		CONS taps ($T4$)	
m-tuple	N.O	m-tuple	N.O
28	36 324 779	28	16 777 216

Observation 4.4: Non-occurring m-tuples are identified for CONS, and FPDS input linear functions

Non-occurring m-tuples are identified for linearly filtered NLFSRs whose output function takes as input, taps which are consecutive or FPDS. This happens regadless of the choice of NFLSRs and linear functions. We take the m-tuple distributions of keystream generators using $R4$ with different linear functions as an example, a portion of which is reproduced in Table 4.4. For example, $R4$ using the $T4$ function had 16 777 216 non-occurring 28-bit tuples, and $R4$ with the $T2$ function had 36 324 779 non-occurring 28-bit tuples.

4.1.3 Discussion

A common strategy to prevent guess and determine attacks for nonlinear filter generators (NLFGs) is to use inputs to the nonlinear function which span a full positive difference (FPDS) set. Furthermore, computer experiments performed for NLFGs in Chapter 3 show that using a NLFG with FPDS inputs to the nonlinear Boolean function can give a more uniform m-tuple distribution compared to the m-tuple distribution of the keystream generated NLFG with consecutive inputs to the Boolean function. However, in the case of the keystream generators presented in this section, the opposite is true. That is, the m-tuple distribution for sequences produced by linearly filtered NLFSRs using FPDS inputs to the linear output function is *less* uniformly distributed.

Our findings can have implications for attacks commonly applied to keystream generators. For example, based on our experiments, a linearly filtered NLFSR using a linear function with consecutive taps will have a more uniform distribution compared to one which uses uneven or FPDS taps. While the former may provide better resistance against statistical attacks, the use of consecutive taps may leave it vulnerable to a guess-and-determine attack. Inputs to a output function which form a full positive difference set provide resistance to this [53].

4.2 Slid pairs in Trivium

This section presents the first of two new contributions to the analysis of the Trivium stream cipher. Our first contribution towards the analysis of Trivium comes in the form of searching for slid pairs in Trivium. We extend the work of Priemuth-Schmid and Biryukov [92], and Zeng and Qi [115] and search for additional types of slid pairs. We show that by forming a new system of equations, the size of the search space for these types of slid pairs can be significantly reduced, and the time and memory requirements needed will be significantly reduced as compared to searching for the same type of special slid pairs using Priemuth-Schmid and Biryukov's system of equations. Through the computation of slid pairs, we also establish a new lower bound on keystream sequences produced by the Trivium stream cipher.

4.2.1 Trivium Specifications

Trivium [26] is a bit-based stream cipher designed by De Canniére and Preneel in 2005 and was submitted to the eSTREAM project [45]. Trivium has been selected as one of the stream ciphers in the final portfolio. It uses an 80 bit key and a 80 bit IV to generate keystream.

Components of Trivium

Although the original specifications of Trivium describes it as been having a single 288 bit register, there have been other representations in the public literature. The representation of Trivium by Bernstein [11] describes Trivium as having three non-autonomous binary NLFSRs: A, B, and C, of sizes 93, 84 and 111 bits respectively, giving a total state size of 288 bits. Figure 4.1 shows a diagram of Trivium's three NLFSRs, along with its state-update (in solid lines) and keystream generation functions (in dotted lines).

State-update function of Trivium

In this thesis, to make it easier to apply Raddum's relabelling technique described in Section 4.3.3, the state-update function we use for Trivium is different from the Berstein's representation. In Berstein's representation, the feedback bit for the respective registers after the first iteration is represented by $A_0(t+1)$, $B_0(t+1)$, and $C_0(t+1)$. In contrast, the feedback bits in the state-update function presented in this thesis are reversed, and

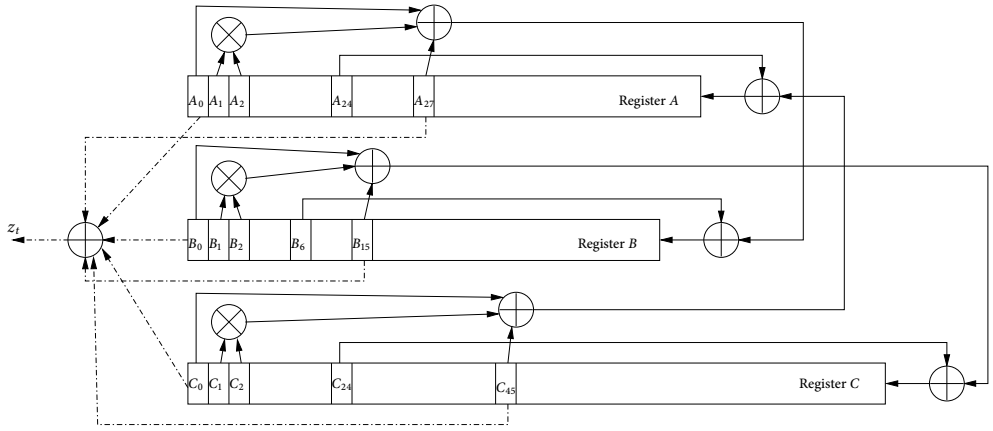


Figure 4.1: Diagram of the Trivium stream cipher

the feedback bits after the first iteration are represented by $A_{93}(t+1)$ and $B_{84}(t+1)$ and $C_{111}(t+1)$. The tap settings to the nonlinear functions are adjusted accordingly. The state-update functions for Trivium are:

$$A_i(t+1) = \begin{cases} A_{24}(t) \oplus C_{45}(t) \oplus C_0(t) \oplus C_2(t)C_1(t) & i = 92, \\ A_{i+1}(t) & 0 \le i \le 91. \end{cases} \tag{4.1}$$

$$B_i(t+1) = \begin{cases} B_6(t) \oplus A_{27}(t) \oplus A_0(t) \oplus A_1(t)A_2(t) & i = 83, \\ B_{i+1}(t) & 0 \le i \le 82. \end{cases} \tag{4.2}$$

$$C_i(t+1) = \begin{cases} C_{24}(t) \oplus B_{15}(t) + B_0(t) \oplus B_2(t)B_1(t) & i = 110, \\ C_{i+1}(t) & 0 \le i \le 109. \end{cases} \tag{4.3}$$

Initialisation of Trivium

During the key and IV loading phases, the key and IV are transferred to A and B such that $A_i(0) = k_{i-13}$ for $13 \le i \le 92$ and $B_i(0) = v_{i-4}$ for $4 \le i \le 83$. All the remaining registers are set to zero, with the exception of $C_0(0), C_1(0), C_2(0)$, which are set to one. During the diffusion phase, 1152 iterations of Trivium's state-update function is run and the outputs produced during these phases are discarded.

Keystream generation

The state-update function for Trivium during keystream generation is the same as the one used during initialisation. To produce a keystream bit, the following register stages are linearly combined.

$$z(t) = A_{27}(t) \oplus A_0(t) \oplus B_{15}(t) \oplus B_0(t) \oplus C_{45}(t) \oplus C_0(t) \text{ , } t \geq 0 \qquad (4.4)$$

The designers note that since the internal state of Trivium is updated in a nonlinear way, the period of keystream sequences produced by Trivium is difficult to determine. They note however, due to the way register C is loaded, the period of keystream sequences produced by Trivium is at least 111 bits. They also claim that the probability of a key-IV pair producing a keystream sequence of period less than 2^{80} bits in length is 2^{-208}. Hu and Gong [72] claim that keystream sequences produced by Trivium are periodic with high probability. This is clear since the state-update function is deterministic. However, they did not establish a lower bound on what period keystream sequences produced by Trivium may be.

Current cryptanalysis

There are no known attacks on Trivium which are faster than exhaustive keysearch. The designers claim that a naive guess-and-determine attack on Trivium has a complexity of 2^{195} operations [26]. Maximov and Biryukov [84] claim that their state recovery attack can recover the initial state of Trivium with a time complexity of about $2^{83.5}$. Raddum [93] used an algebraic relabelling technique and techniques from graph theory and estimated that the initial state of Trivium can be recovered in about 2^{164} operations. We will discuss this relabelling method in more detail in Section 4.3.3. McDonald et al. [85] attempted to recover the initial state by treating Trivium as a Boolean Satisfiability (SAT) problem and used a SAT solver to recover the initial state. They estimated that the complexity of recovering the initial state of Trivium using a SAT solver has a time complexity of about $2^{159.9}$. If the number of initialisation rounds during the diffusion phase was reduced from 1152 to 767 rounds, Dinur and Shamir [39] claim that an algebraic cube attack can recover the entire 80 bit secret key with a complexity faster than exhaustive keysearch.

4.2.2 Overview of Slid Pairs

Slid pairs in stream ciphers can occur during initialisation in a keystream generator. A slid pair is a loaded state S^0, which after α iterations of a state-update function, gives another loaded state S^1. If, and only if, the same state-update function is used during keystream generation, the keystream Z^1 produced by S^1 will be an α bit shifted version of the keystream Z^0 produced by S^0.

When the internal state of a keystream generator is viewed as a finite-state machine, the existence of two session keys which produce keystreams which are shifted copies of each other is not uncommon. However, as De Cannière et al. [25] note, some stream ciphers have a property which causes these session keys to "cluster" together, instead of distributing them evenly over the entire state space. This clustering property may make the stream cipher more susceptible to slid pairs, which in turn may make them more vulnerable to attacks or may allow us to determine the minimum period of keystream sequences produced by a particular keystream generator.

Applications of slid pairs in key recovery attacks

The existence of slid pairs in stream ciphers has been used in the analysis of stream ciphers. De Cannière et al. [25] outlined two attacks on the Grain family of stream ciphers [65]. In the first attack, they use a related-key attack to attack Grain by exploiting the fact that there is a mathematical relationship between two keys $K1$ and $K2$ and the corresponding keystream, Z^1 and Z^2 respectively. For example, by observing Z^1 and Z^2, an attacker may know that the very first key bit for both $K1$ and $K2$ is 1. Knowledge of any particular key bit may make recovery of the entire key easier compared to exhaustive key search. In the case for the Grain family of stream cipher, De Cannière et al. [25] claim that for any session key there exists another distinct session key which will produce a α shifted copy of the keystream produced by the first session key with a probability of $2^{-2\cdot\alpha}$. They note that knowledge of the relationship between two session keys can allow the attacker to determine particular session key bits, which may speed up the recovery of the other session key bits due to the simpler equations which can be formed. In the second attack, they claim that due to the sliding property in the Grain family of stream ciphers, the cost of exhaustive keysearch can be reduced by half.

Priemuth-Schmid and Biryoukov [92] claimed that the existence of slid pairs in Salsa20 can result in a birthday attack on Salsa20 with a 256 bit key with a complexity of $O(2^{192})$. Priemuth-Schmid and Biryoukov also describe the existence of slid pairs in Trivium and estimate that there may be approximately 2^{39} slid pairs which are within 221

bits shifts of each other. Wu and Preneel [114] apply the slid attack on LEX. They claim that if each of approximately $2^{60.8}$ IVs were used to produce 160, 000 bits of keystream, they are able to recover the master key of LEX. Kircanski and Youssef use slide attacks to analyse SNOW 2.0 and SNOW 3G. Their paper describes how slide attacks can be used to recover the 256 bit secret key of SNOW 2.0 with a complexity less than exhaustive key search.

The presence of slid pairs in stream ciphers may also allow an attacker to use an algebraic cube attacks [39, 97] to recover the loaded state L^1 of a keystream generator which eventually produces the keystream sequence Z^1. Assume that the attacker was previously able to successfully mount an algebraic cube attack on another keystream sequence, whose keystream sequence Z^2 is an $\alpha = 100$ bit shift of Z^1. In this scenario, the attacker does not need to go through the (computationally-intensive) task of recovering the loaded state L^1. Since Z^1 and Z^2 are 100-bit shifts apart, an attacker can simply use the loaded state L^2, and run the state-update function an additional (or fewer) 100 iterations, and obtain the loaded state L^1. Note that this application of the cube attack is only possible if the state-update function used during the diffusion, and the keystream generation process of the stream cipher is the same. If the stream cipher uses a different state-update function for the diffusion process and keystream generation, the sliding property of the keystream sequences may be due to the state-update function used during keystream generation, and running the diffusion process additional (or less) iterations may not allow an attacker to successfully recover a correct loaded state.

Establishing the period of keystream generators using slid pairs

As was stated at the beginning of Chapter 4, the minimum period of most modern keystream generators are difficult to establish, due to the nonlinear way the internal state of the keystream generator is updated. To compound the issue, the size of the internal state of modern stream cipher proposals is usually large (larger than 80 bits). This makes it computational impractical to try all possible states of an NLFSR and measure all the period of the keystream sequences produced by these NFLSR states.

Slid pairs may allow us to establish a lower bound on the period of keystream sequences produced by modern keystream generators. For example, assume we have a set of loaded states Γ_α, which after α iterations will give another set Θ_α of loaded states. If none of the states in Γ_α is in Θ_α, that is, if $\Gamma_\alpha \cap \Theta_\alpha$ is an empty set, we know that there are no short cycles in the keystream generator of length α. The sets Γ_α and Θ_α can be constructed using algebraic methods. In this thesis, we use the F4 algorithm [46]

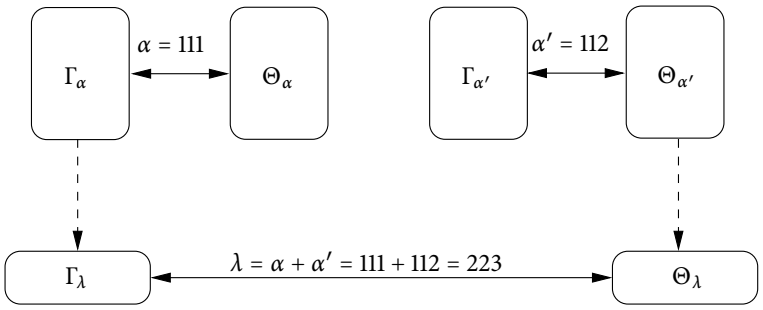


Figure 4.2: Searching for slid pairs when $\lambda = 223$

implemented in Magma [20] to algebraically solve systems of equations to obtain these sets (see Section 2.4.8 for a description of the F4 algorithm) for $111 \leq \alpha \leq 121$.

A problem which can arise when trying solve this system of equations algebraically is that the degree of the equations increases as the number of iterations of the state-update function increases. Solving the system of equations will be infeasible if the system of equations are complex. Consider a particular value of α, which we denote λ, for which the system of equations is too complex to solve directly. One method to find partial information regarding $\Gamma_\lambda \cap \Theta_\lambda$ is to consider λ as the sum of two smaller integers, so that $\lambda = \alpha + \alpha'$, where we are able to solve the equations for α and α'. Figure 4.2 illustrates this concept for $\alpha = 111$, $\alpha' = 112$, and $\lambda = \alpha + \alpha' = 111 + 112 = 223$. As the state-update function of Trivium is deterministic and the loading process of Trivium, described in Section 4.2.1, implies that getting from one loaded state to another loaded state requires at least 111 iterations of Trivium's state-update function, the set $\Gamma_{111} \cap \Gamma_{112}$ is necessarily an empty set. Similarly, the set $\Theta_{111} \cap \Theta_{112}$ is empty. However, it is not immediately clear if the set $\Gamma_{111} \cap \Theta_{112}$ is empty. Similarly, it is unknown if the set $\Theta_{111} \cap \Gamma_{112}$ is empty. If the set $\Gamma_{111} \cap \Theta_{112}$ is not empty, it means that the minimum period of Trivium is at least $111 + 112 = 223$. Note that if $\Gamma_{111} \cap \Theta_{112}$ is not empty, it does not imply that the minimum period of Trivium is greater than 223. Similarly, if the set $\Gamma_{111} \cap \Theta_{112}$ is not empty, it means that we are able to find state cycles which have two or more consecutive loaded states.

4.2.3 Existing Work on Trivium Slid Pairs

In Trivium, a loaded state is such that $A_i(0) = k_{i-13}$ for $13 \leq i \leq 92$, and $B_i(0) = v_{i-4}$ for $4 \leq i \leq 83$. The remaining 13 stages in A and four stages in B are set to zero. The majority of the contents of the 111-bit register in Trivium C are set to zero, with the exception of the last three stages, C_0, C_1, C_2 which are each set to one. A slid pair in Trivium is a pair of loaded states (L_1, L_2) such that when L_1 is loaded, after α iterations of Trivium's

state-update function, L_2 is obtained. Due to the format of register C in a loaded state, α has to be greater than or equal to 111 for another loaded state to occur. In this chapter, we use the term *starting state* to denote the initial loaded state and the term *target state* to denote a particular loaded state which is obtained after α iterations. The set Γ_α is the set of loaded states which after α iterations will give another set Θ_α of loaded states. Since the state-update function of Trivium is one-to-one, both sets Γ_α and Θ_α are the same size.

The existence of slid pairs in Trivium was first pointed out by Hong [68]. He notes that the naïve system of equations which needs to be solved to obtain Γ_{111} has 128 equations in 160 variables. Solving this system of equations will allow us to determine how many slid pairs there are for $\alpha = 111$ iterations. These equations are the 111 equations for register C and 17 equations which satisfy the loading requirements for registers A and B. Priemuth-Schmid and Biryukov observe that for $\alpha = 111$, the last 24 bits of key do not occur in this system of equations. These 24 bits are known as free variables and can take on any value. An additional 16 bits are given a priori due to the 13 zeros in A and three ones in register C. The 13 zeros are set for $A_{42}, A_{43}, \ldots, A_{54}$ while the 3 ones are set for B_{15}, B_{16}, B_{17}. This gives us a system of 112 equations in 120 variables.

Priemuth-Schmid and Biryukov [92] used the F4 algorithm in mathematical software Magma [20] to calculate the number of slid pairs for $111 \leq \alpha \leq 115$. They show that the size of the set Γ_{111} and Θ_{111} is approximately 2^{32}. They also checked for the existence of two particular types of slid pairs. First, they checked for slid pairs where the keys in two states are the same after α iterations. Then they checked for slid pairs where the IV in two states are the same after α iterations. Priemuth-Schmid and Biryukov showed that the two particular slid pairs described did not exist for $111 \leq \alpha \leq 143$.

Zeng and Qi [115] extended the search for slid pairs to $\alpha = 195$ iterations. In contrast to the approach of Priemuth-Schmid and Biryukov, they used a SAT solver called MiniSat [42] to solve the system of equations generated by Trivium's state-update function. By using MiniSat to solve the system of equations, Zeng and Qi are able to search for slid pairs without guessing any key-IV bits and solve a particular system of equations faster than the F4 algorithm will take to solve the same system of equations. For example, Priemuth-Schmid and Biryukov estimate that completing the search for slid pairs for $\alpha = 115$ using the F4 algorithm will take 5 529 600 seconds (64 days) if 10 key/IV bits were guessed. In contrast, Zeng and Qi were able to complete the search for slid pairs for $\alpha = 115$ in 0.03 seconds without any bit guessing. Zeng and Qi make further observations on the number of free variables which appear in the system of equations for $\alpha = 111$–134.

For α = 111–134, there would be $24 - (\alpha - 111)$ free variables which do not appear in the system of equations. Thus, the set Θ_α (and Γ_α) will consist of at least $2^{24-(\alpha-111)}$ loaded states for α = 111–134. Unlike Priemuth-Schmid and Biryukov however, Zeng and Qi did not check for slid pairs where the keys in two states are the same after α iterations, nor did they check for slid pairs where the IV in two states are the same for α = 144–195 iterations, and the presence of such slid pairs remains an open question.

4.2.4 Experiment goals

Although the works of Priemuth-Schmid and Biryukov, and Zeng and Qi establish what the size of Θ_α is, for $111 \leq \alpha \leq 195$, both groups of researchers did not check if $\Gamma_\alpha \cap \Theta_\alpha$ is an empty set. The main goals of our experiments is to extend their work and determine if a new lower bound on the period of keystream sequences produced by Trivium can be established. We also investigate if it is possible to find state cycles which consist of two or more consecutive loaded states. We accomplish this by performing two sets of experiments. In these two experiments, we:

1. check if $\Gamma_\alpha \cap \Theta_\alpha$ is a empty set. That is, if any of the loaded states in Γ_α can also be found in Θ_α.

2. check if $\Gamma_\alpha \cap \Theta_{\alpha'}$, where $\alpha \neq \alpha'$, is an empty set. That is, if any of the loaded states in Γ_α can be found in $\Theta_{\alpha'}$.

4.2.5 Experimental Design

In this section, we describe the steps we used in our experiments to search for slid pairs. Experiment 1 describes the experimental design for our first goal, while Experiment 2 describes the experimental design for our second goal.

Experiment 1

There are two phases involved in our experiments. First, a system of equations needs to be solved. Solving this system of equations allows us to construct the set Γ_α, which after α iterations, will gives us a loaded state in the set Θ_α. In our experiments, we use the F4 algorithm implemented in Magma to solve this system of equations. Solving this system of equations using Priemuth-Schmid and Biryukov's technique requires a large amount of memory and time for $\alpha > 112$, as evidenced by the time and memory entries for their technique in Table 4.5. Furthermore, the total number of loaded states in Γ_{111}

is $5\,704\,253\,440 \approx 2^{32}$. While the size of this set can be searched through in the second step of our experiments using todays' computers, it is not an optimal way of doing so as some of the loaded states in Θ_α obtained using Priemuth-Schmid and Biryukov's approach are not in the 'proper' format. If we were to impose additional conditions on what the format a loaded state needs to be, the size of Γ_α (and Θ_α) can be reduced. This will make checking if $\Gamma_\alpha \cap \Theta_\alpha$ is an empty set faster. In this thesis, these reduced sets are denoted Γ'_α and Θ'_α. The conditions for ensuring the loaded states are of the 'proper' format in our system of equations, for $\alpha = 111$, are as follows.

- For any starting state, $A_{42}, A_{43}, \ldots, A_{54}$ need to be zero as these will comprise the first 13 zeros a target state's register A needs to have after 111 iterations. Therefore, the contents of $A_{42}, A_{43}, \ldots, A_{54}$ in any target state need to be zero as well. The equations describing these stages after 111 iterations can be added into our system of equations. If for an example, the contents of A_{42} in a particular target state is one, this target state will never be in the set Γ_α.

  The equations describing the stages A_i, for $42 \leq i \leq 50$, for the target state have nine linear equations expressed in terms of key bits k_i, for $71 \leq i \leq 79$ respectively and can be fixed for a starting state. The stages which contain these key bits for any loaded state are A_i, for $84 \leq i \leq 92$ respectively. Hence, the equations which describe stages A_i, for $84 \leq i \leq 92$ for a target state needs to be added into the system of equations.

- We fix the values of three registers stages, B_{16}, B_{17}, and B_{18}, as these will form the first three ones of C after 111 iterations. The equations which describe these three registers stages after 111 iterations need to be added into the system of equations.

For $\alpha = 111$, 16 new equations need to be added to our system of equations. When these are added to the original equations in Priemuth-Schmid and Biryukov's approach, the new system consists of 128 equations in 123 variables. To compare the time and memory requirements needed to solve our system of equations with the requirements of Priemuth-Schmid and Biryukov's system of equations, we also solved their system of equations using the same computer used to solve our system of equations. We used QUT HPC's supercomputer for the experiments. This computer is a SGI Altix XE Cluster and the computer node which Magma runs on has an X5650@2.66 GHz 64bit Intel Xeon processor with access to a maximum of 96 GB of RAM. The time and memory requirements required to solve the two systems of equations are listed in Table 4.5. In both systems of equations, we do not guess any bits. This allows us to obtain the actual

number of loaded states in Γ_α and Γ'_α. This is in contrast to the bit-guessing approach undertaken by Priemuth-Schmid and Biryukov, which was necessary to ensure that they could find solutions in a feasible time.

Once Magma solves our system of equations, we can construct the set Γ'_α and we can proceed with the second step of our experiment. The second step of our experiment involves using a C program to check if the set $\Gamma'_\alpha \cap \Theta'_\alpha$ is a empty set. To do this, we pick a loaded state L_1 from the set Γ'_α and run it for α iterations to obtain a target state L_2 in Θ'_α. The program then checks if this target state can be found in Γ'_α. This process is repeated until all the loaded states in Θ'_α have been used to generate a target state and all target states have been compared to all states in Γ'_α.

Experiment 2

Experiment 2 runs in two steps. First, a system of equations needs to be solved; this system of equations allows us to construct the set Γ_α, which after α iterations, will gives us a loaded state in the set $\Theta_{\alpha'}$, where $\alpha \neq \alpha'$. This set is equivalent to forming a system of equations which, when solved, forms the set $\Gamma_{\alpha+\alpha'}$ where a loaded state in the set $\Gamma_{\alpha+\alpha'}$ forms another loaded state which can be found in the set $\Theta_{\alpha+\alpha'}$ after $\alpha + \alpha'$ iterations.

The second step of Experiment 2 involves checking if a loaded state $L^1 \in \Gamma_\alpha$ also exists as a loaded state $L^2 \in \Theta_{\alpha'}$, a similar C program is used. This program picks a loaded state from $\Gamma_{\alpha'}$ and runs it for α' iterations to obtain L^2. L^2 is then checked to see if $L^2 \in \Gamma_\alpha$. This process is repeated until all states in $\Theta_{\alpha'}$ have been checked against the states in Γ_α.

4.2.6 Experimental Results

Experiment 1

Table 4.6 lists the results for Experiment 1, while Table 4.5 lists the time in seconds (s) and memory requirements in megabytes (MB) of both Priemuth-Schmid and Biryukov's approach and ours. A DNF entry in the table means that we were not able to obtain a solution using Magma either because it took too long to compute or failed to finish due to a lack of computer memory. The Solns column gives the number of solutions obtained when we used Magma to solve the system of equations. The β column gives the number of free variables, that is, the number of key bits which do not occur in the system of equations and can take on any value. Γ_α is the total number of loaded state's in each of the Priemuth-Schmid and Biryukov's, and our work's respective columns which

Table 4.5: Memory and time measurements for solving the system of equations for slid pair Experiment 1

	P-S & Biryukov		This work	
α	Memory (MB)	Time (s)	Memory (MB)	Time (s)
111	4915.2	5520	70.2	23
112	7782.4	22 860	151	20
113	67 584	208 080	13.8	2
114	DNF	DNF	72	20
115	DNF	DNF	78	17
116	DNF	DNF	37	14
117	DNF	DNF	35	20
118	DNF	DNF	71	24
119	DNF	DNF	131	97
120	DNF	DNF	214	120
121	DNF	DNF	4505.6	5400

after α iterations, will give another loaded state. The value of Γ_α can be calculated by calculating $\Gamma_\alpha = \text{Solns} \times 2^\beta$. In the case where the experiments did not finish, we give the approximate size of the set Γ_α based on the calculations of Zeng and Qi. The last column, $|\Gamma_\alpha \cap \Theta_\alpha|$ lists the size of the set $\Gamma_\alpha \cap \Theta_\alpha$.

The results show that for our work, when attempting to solve the system of equations for $\alpha = 111$–121, Magma was only able to obtain solutions for $\alpha = 114$, 120, and 121. The number of solutions, represented by Solns in Table 4.6 are 32, 24 and 20 respectively. For all three cases, the set $\Gamma_\alpha \cap \Theta_\alpha$ is an empty set. Furthermore, Table 4.5 shows that solving our system of equations require significantly less time and memory than compared to Priemuth-Schmid and Biryukov's approach. For example, for $\alpha = 113$, solving Priemuth-Schmid and Biryukov's system of equations took 67 584 seconds and required 208 080 MB of memory, whereas our technique took only two seconds and needed only 13.8 MB of memory.

Experiment 2

Table 4.7 lists the results of Experiment 2. Γ_α is total number of loaded states, which after α iterations, will give a state with a loaded format which can be found in the $\Theta_{\alpha'}$, where $\alpha \neq \alpha'$.

The Solns column gives the number of solutions obtained when we used Magma to solve the system of equations. The β column gives the number of free variables, that is,

Table 4.6: Results of Trivium slid pairs for Experiment 1

P-S & Biryukov						This work					$\lvert\Gamma_\alpha \cap \Theta_\alpha\rvert$
α	Var	Eqns	Solns	β	$\lvert\Gamma_\alpha\rvert$	Var	Eqns	Solns	β	$\lvert\Gamma'_\alpha\rvert$	
111	120	112	340	24	5 704 253 440	122	129	0	12	0	0
112	122	113	440	23	3 690 987 520	125	128	0	14	0	0
113	124	114	848	22	3 556 769 792	127	128	0	16	0	0
114	126	115	DNF	21	$\leq$ 2 097 152	129	128	32	18	8 388 608	0
115	127	115	DNF	20	$\leq$ 1 048 576	128	128	0	19	0	0
116	128	115	DNF	19	$\leq$ 524 288	128	128	0	19	0	0
117	129	115	DNF	18	$\leq$ 262 144	129	128	0	18	0	0
118	130	115	DNF	17	$\leq$ 131 072	130	128	0	17	0	0
119	131	115	DNF	16	$\leq$ 65 536	131	128	0	16	0	0
120	132	115	DNF	15	$\leq$ 32 768	132	128	24	15	786 432	0
121	133	115	DNF	14	$\leq$ 16 384	132	128	20	14	327 680	0

Table 4.7: Results of Trivium slid pairs for Experiment 2

α	α'	Solns	β	$\lvert\Gamma_\alpha\rvert$	$\lvert\Gamma_\alpha \cap \Theta_{\alpha'}\rvert$
111	112	16	12	65 536	0
111	113	0	22	0	0
112	111	28	14	458 752	0
112	113	0	22	0	0
113	111	16	24	268 435 456	0
113	112	0	23	0	0

the number of key bits which do not occur in the system of equations and can take on any value. Γ_α is the total number of loaded state's which after α iterations, will produce a state in $\Theta_{\alpha'}$. The value of Γ_α can be calculated by calculating $\Gamma_\alpha = \text{Solns} \times 2^\beta$. $\Gamma_\alpha \cap \Theta_{\alpha'}$ lists the number of states which exists in both Γ_α and $\Theta_{\alpha'}$. Table 4.7 shows that Magma was able to find solutions for (α, α') for (111,112), (112,111), and (113,111). However, upon running the two steps of Experiment 2, we find that $\Gamma_\alpha \cap \Theta_{\alpha'}$ are empty sets.

4.2.7 Discussion

In our first experiment, we have shown that the set $\Gamma_\alpha \cap \Theta_\alpha$ is empty, for $111 \leq \alpha \leq 121$, and in our second experiment, we have shown that the sets $\Gamma_\alpha \cap \Theta'_\alpha$, for $111 \leq \alpha \leq 113$ and $111 \leq \alpha' \leq 113$, are also empty. If any sets in either experiments were non-empty, we could begin to form chains of sequential loaded states. If an endpoint of such a chain is also the beginning of a chain, then we have established the period for that particular cycle.

Given the small values of α used in our experiments, we were unable to find more than two consecutive loaded states. In particular, our second experiment allows us to find particular slid pairs for larger α value than was explored by either Priemuth-Schmid and Biryukov [92], or Zeng and Qi [115].

Knowledge of a slid pair L^1 and L^2 can also aid in an algebraic key recovery attack as well. Cube attacks [39, 97] have been able to recover the secret key if the number of initialisation rounds of Trivium is reduced from 1152 to 767. This allows us to use cube attacks in conjunction with a related-key attack described in Section 4.2.2. If the keystream Z^1 produced by the loaded state L_1 is the same keystream Z^2 that is produced by the loaded states L_2 after $(t + 767)$ clocks, we may be able to mount a key recovery attack by first using a cube attack to recover the secret key of L^1. Since the keystream generation function of Trivium is the same as the initialisation function, we can load the key for the loaded state L_1 and perform 767 initialisation rounds, instead of the 1152 rounds suggested in the Trivium proposal. After 767 initialisation rounds, Trivium should now be in the loaded state L^2 which would produce the keystream z^2. Since the loading process of Trivium is linear, all 80 secret key bits (in register A) can now be easily recovered.

4.3 New algebraic analysis on Trivium and its variants

This section presents the second of two new analyses on Trivium and Trivium-like ciphers. Trivium-like ciphers analysed in this section include Bivium-A and Bivium-B, which are described in Section 4.3.1 and Trivium-A and Trivium-AB which are described in Section 4.3.7. Trivium-A and Trivium-AB are new variants of Trivium we introduce in this section for our new analyses. In our analysis, we combine the algebraic technique presented by Berbain et al. [10] to analyse the Grain family of stream ciphers and its variants combined with the technique Raddum [93] used to analyse the Trivium stream cipher. We show that by using an algebraic divide-and-conquer approach and solving two systems of equations resulting from this divide-and-conquer approach sequentially, it may be possible to break some Trivium-like ciphers faster than exhaustive keysearch. In Section 4.3.8, we discuss exceptions to this attack and a possible countermeasure to successful attacks. The effect distances between stages used as inputs to the output function and the state size of Trivium-like ciphers have on the effectiveness of the new attack are also investigated.

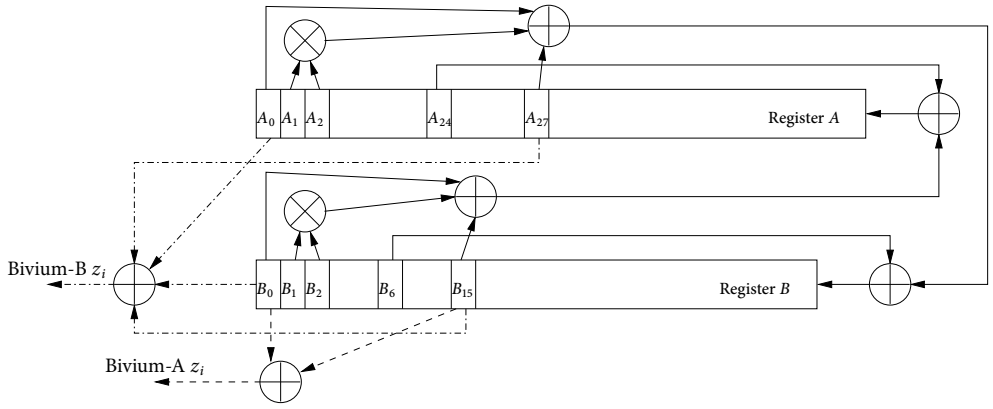


Figure 4.3: Diagram of the BiviumA/B stream cipher

4.3.1 Bivium-A and Bivium-B

To aid the analyses of Trivium-like ciphers, two scaled-down versions of Trivium, Bivium-A and Bivium-B were introduced by Raddum [93]. Bivium-A and Bivium-B both consist of two NLFSRs, A and B of sizes 93 and 84 bits respectively, giving a total state size of 177 bits. Figure 4.3 shows the components of Bivium-A/B, along with its state-update (in solid lines) and keystream generation functions (in dotted lines). The state-update functions for the two registers are as follows:

$$A_i(t+1) = \begin{cases} A_{24}(t) \oplus B_{15}(t) \oplus B_0(t) \oplus (B_1(t) \cdot B_2(t)), & i = 92, \\ A_{i+1}(t), & 0 \le i \le 91. \end{cases} \tag{4.5}$$

$$B_i(t+1) = \begin{cases} B_6(t) \oplus A_{27}(t) \oplus A_0(t) \oplus (A_1(t) \cdot A_2(t)), & i = 83, \\ B_{i+1}(t), & 0 \le i \le 82. \end{cases} \tag{4.6}$$

Initialisation of Bivium-A and B

During the key and IV loading phases, the key and IV are transferred to A and B such that $A_i(0) = k_{i-13}$ for $13 \le i \le 92$ and $B_i(0) = v_{i-4}$ for $4 \le i \le 83$. All the remaining registers are set to zero. During the diffusion phase, 708 iterations of the state-update function are performed and the output produced during this phase is discarded.

Keystream generation

The state-update function for Bivium-A and Bivium-B during keystream generation is the same as the one used during initialisation. To produce a keystream bit, the following register stages are linearly combined. The linear function used to generate keystream for Bivium-A is

$$z(t) = B_{15}(t) \oplus B_0(t)\ ,\ t \geq 0$$

while the linear function used to generate keystream for Bivium-B is

$$z(t) = A_{27}(t) \oplus A_0(t) \oplus B_{15}(t) \oplus B_0(t)\ ,\ t \geq 0$$

4.3.2 Overview of Berbain's et al.'s technique

Berbain et al. [10] presented a technique of expressing new feedback bits of a NLFSR as linear combinations of keystream bits and the internal state. For example, consider a 14 stage binary NLFSR A with the state-update function

$$A_i(t+1) = \begin{cases} A_0(t) \oplus A_1(t) \oplus A_2(t) \oplus A_7(t)A_{12}(t), & i = 13, \\ A_{i+1}(t), & 0 \leq i \leq 12. \end{cases} \tag{4.7}$$

The linear output function for this NLFSR is:

$$z(t) = A_0(t) \oplus A_4(t), \text{ for } t \geq 0 \tag{4.8}$$

The initial state of this NLFSR is $A_i = x_i$, for $0 \leq i \leq 13$. Written in terms of initial state bits x_i, for $0 \leq i \leq 13$, the first five feedback bits, $A_{13}(t)$, for $1 \leq t \leq 5$ are:

$$A_{13}(1) = x_7x_{12} \oplus x_0 \oplus x_1 \oplus x_2 \tag{4.9}$$

$$A_{13}(2) = x_8x_{13} \oplus x_1 \oplus x_2 \oplus x_3 \tag{4.10}$$

$$A_{13}(3) = x_7x_9x_{12} \oplus x_0x_9 \oplus x_1x_9 \oplus x_2x_9 \oplus x_2 \oplus x_3 \oplus x_4 \tag{4.11}$$

$$A_{13}(4) = x_8x_{10}x_{13} \oplus x_1x_{10} \oplus x_2x_{10} \oplus x_3x_{10} \oplus x_3 \oplus x_4 \oplus x_5 \tag{4.12}$$

$$\begin{aligned} A_{13}(5) = {} & x_7x_9x_{11}x_{12} \oplus x_0x_9x_{11} \oplus x_1x_9x_{11} \oplus x_2x_9x_{11} \oplus \\ & x_2x_{11} \oplus x_3x_{11} \oplus x_4x_{11} \oplus x_4 \oplus x_5 \oplus x_6 \end{aligned} \tag{4.13}$$

The degree of the equations representing the feedback bits can increase at each iteration since each feedback bit is calculated as a function of degree two over 14 internal state variables. In our example, the first two feedback bit equations are of degree two, the next two are of degree three, while the last equation is of degree four. Similarly, the degree of the keystream equations produced by this NLFSR, can also increase the more keystream equations are obtained. This may make algebraic attack methods infeasible as the equations become too complex. By presenting a technique which writes each internal state bit as linear combinations of keystream and initial state bits, Berbain et al. were able to prevent the degree of equations representing the feedback bits increasing at each clock. These new equations representing the new feedback bits always remain as linear equations. Consequently, the keystream equations generated will also be linear, which may make algebraic attacks possible.

They were able to accomplish this by observing that by reordering the keystream equations of linearly filtered NLFSRs, it is possible to represent the feedback bits in terms of keystream bits and internal state bits. Recall the linear output function of our toy NLFSR in Equation 4.8. At $t = 10$, the equation representing the 11th bit of keystream is

$$z(10) = A_{10}(0) \oplus A_{13}(1) \tag{4.14}$$

By reordering Equation 4.14, we can obtain the first feedback bit of A:

$$A_{13}(1) = A_{10}(0) \oplus z(10) \tag{4.15}$$

Note that the feedback bit $A_{13}(1)$ represented by Equation 4.15 is linear, compared to equation representing the same feedback bit in Equation 4.9, which is of degree two. Therefore, if we were to represent the NLFSR's first five feedback bits $A_{13}(t)$, for $1 \leq t \leq 5$ as linear combinations of keystream and internal state bits, we have the following equations:

$$\begin{aligned}
A_{13}(1) &= A_{10}(0) \oplus z_{10} \\
A_{13}(2) &= A_{11}(0) \oplus z_{11} \\
A_{13}(3) &= A_{12}(0) \oplus z_{12} \\
A_{13}(4) &= A_{13}(0) \oplus z_{13} \\
A_{13}(5) &= A_{13}(1) \oplus z_{14} = A_{10}(t) \oplus z_{10} \oplus z_{14}
\end{aligned}$$

Table 4.8: Values of q, q', and j for Trivium-like stream ciphers

	Bivium-A	Bivium-B	Trivium
q	2	2	3
q'	1	1	1
j	1	2	3

The feedback bits of our example NFLSR are now all represented as linear combinations of keystream and internal state bits, compared to equations of degree two or greater previously as shown in Equations 4.9–4.13.

Berbain et al. applied their attack to a modified version of Grain-128. Like the original version of Grain-128 [64], their version of Grain-128 also uses a 128-bit secret key. In Berbain et al.'s version of Grain-128, $q = 1$ bits of internal state are non-linearly updated at each step while $q = 1$ linear combinations of internal state are output as keystream. They claim that they are able to recover the initial state of this modified version of Grain-128 with a complexity of 2^{105} computations and 2^{39} bits of keystream.

Berbain et al. claim that their technique can be extended to ciphers in which $q > 1$ bits of internal state are non-linearly updated at each step and q or more linear combinations of the state are output as keystream. However, whether these techniques can be extended to ciphers in which $q > 1$ bits of internal state are non-linearly updated, while only $q' < q$ linear combinations of the state-bits are output remains an open question. Bivium-A, Bivium-B, and Trivium are ideal candidates to analyse in an attempt to answer this open question, as $q' < q$ in all three cases. To assist us in our algebraic analysis, we introduce a new variable j, which describes how many registers the keystream generation function takes as input. The values of q', q and j for Bivium-A, Bivium-B, and Trivium are shown in Table 4.8.

4.3.3 Review of Raddum's analysis of Trivium

One problem of algebraically analysing Trivium is that due to the use of NLFSRs, the degree of the equations generated can increase at each iteration. For example, Raddum [93] noted that the first $\alpha = 66$ iterations representing the keystream bit of Trivium give linear equations, the next 82 iterations, for $67 \leq \alpha \leq 148$ give quadratic equations and the next 66 iterations, for $149 \leq \alpha \leq 214$ give cubic equations [103], and so on. The degree of equations continues to increase in this manner as α increases.

In 2006, Raddum presented an algebraic analysis on Trivium [93], where he constructed a sparse system of equations and used techniques from graph theory to solve the system of equations. To construct a sparse system of equations, Raddum used additional variables to represent the feedback bits generated at each iteration. Due to this relabelling technique, equations representing the keystream bit will be at most quadratic, regardless of how many keystream bits were generated.

State-update Function of Trivium

The state-update function of Trivium, described in Section 4.2.1 using Raddum's relabelling technique of the first three feedback bits for regisers A, B and C are as follows:

$$\begin{aligned} A_{93} &= A_{24} \oplus C_{45} \oplus C_0 \oplus C_1C_2 \\ A_{94} &= A_{25} \oplus C_{46} \oplus C_1 \oplus C_2C_3 \\ A_{95} &= A_{26} \oplus C_{47} \oplus C_2 \oplus C_3C_4 \end{aligned} \tag{4.16}$$

$$\begin{aligned} B_{84} &= B_6 \oplus A_{27} \oplus A_0 \oplus A_1A_2 \\ B_{85} &= B_7 \oplus A_{28} \oplus A_1 \oplus A_2A_3 \\ B_{86} &= B_8 \oplus A_{29} \oplus A_2 \oplus A_3A_4 \end{aligned} \tag{4.17}$$

$$\begin{aligned} C_{111} &= C_{24} \oplus B_{15} \oplus B_0 \oplus B_1B_2 \\ C_{112} &= C_{25} \oplus B_{16} \oplus B_1 \oplus B_2B_3 \\ C_{113} &= C_{26} \oplus B_{17} \oplus B_2 \oplus B_3B_4 \end{aligned} \tag{4.18}$$

The equations representing the first three keystream bits are

$$\begin{aligned} z_0 &= A_{27} \oplus A_0 \oplus B_{15} \oplus B_0 \oplus C_{45} \oplus C_0 \\ z_1 &= A_{28} \oplus A_1 \oplus B_{16} \oplus B_1 \oplus C_{46} \oplus C_1 \\ z_2 &= A_{29} \oplus A_2 \oplus B_{17} \oplus B_2 \oplus C_{47} \oplus C_2 \end{aligned}$$

At each iteration, three new variables and four equations are introduced. After clocking Trivium 288 times, we have a system of equations consisting of 1152 equations in 1152 unknowns. Raddum also noted that during the last 66 clocks, the new variables

introduced are not used in keystream generation. By dropping the variables and their equations introduced in the last 66 clocks and only including the equation representing the keystream bits for these 66 clocks, the system of equations can be reduced to 954 equations in 954 variables.

Using these systems of equations, Raddum [93] attempted to recover the initial state of Trivium. He estimated that the complexity of the attack is about $O(2^{164})$ operations. Simonetti et al. [103] attempted to use the F4 algorithm implemented in Magma to solve Raddum's set of equations for Trivium, but the computation did not finish.

State-update Function of Bivium

The state-update function of Bivium-A and Bivium-B for the first three feedback bits of registers A and B, described in Section 4.2.1, when applied with Raddum's relabelling technique are as follows:

$$\begin{aligned} A_{93} &= A_{24} \oplus B_{15} \oplus B_0 \oplus B_1 B_2 \\ A_{94} &= A_{25} \oplus B_{16} \oplus B_1 \oplus B_2 B_3 \\ A_{95} &= A_{26} \oplus B_{17} \oplus B_2 \oplus B_3 B_4 \end{aligned} \tag{4.19}$$

$$\begin{aligned} B_{84} &= B_6 \oplus A_{27} \oplus A_0 \oplus A_1 A_2 \\ B_{85} &= B_7 \oplus A_{28} \oplus A_1 \oplus A_2 A_3 \\ B_{86} &= B_7 \oplus A_{29} \oplus A_2 \oplus A_3 A_4 \end{aligned} \tag{4.20}$$

The equations representing the first three keystream bits for Bivium-A are

$$\begin{aligned} z_0 &= B_{15} \oplus B_0 \\ z_1 &= B_{16} \oplus B_1 \\ z_2 &= B_{17} \oplus B_2 \end{aligned}$$

and the equations representing the first three keystream bits for Bivium-B are

$$z_0 = A_{27} \oplus A_0 \oplus B_{15} \oplus B_0$$
$$z_1 = A_{28} \oplus A_1 \oplus B_{16} \oplus B_1$$
$$z_2 = A_{29} \oplus A_2 \oplus B_{17} \oplus B_2$$

At each iteration, two new variables and one equations are introduced. After clocking Trivium 177 times, we have a system of equations consisting of 531 equations in 531 unknowns. Similar to Trivium's system of equations, Raddum also noted that during the last 66 clocks, the new variables introduced are not used in keystream generation. By dropping the variables introduced in the last 66 clocks, the system of equations can be reduced to 399 equations in 399 unknowns.

Using these systems of equations, Raddum [93] attempted to recover the initial state of Bivium-A and Bivium-B. He was able to recover the initial state of Bivium-A in about a day, while it will take about 2^{56} seconds to recover the initial state of Bivium-B. Simonetti et al. [103] used the F4 algorithm implemented in Magma to solve Bivium-A and Bivium-B's system of equations. Simonetti et al. conducted two different experiments.

The first experiment involved solving a system of equations consisting of 177 variables in 320 equations. That is, forming a system of equations required 320 bits of keystream. They found that for Bivium-A, if they did not guess any bits, they were able to recover the initial state in 68 732.180 seconds, while guessing five bits, the computation finished in 40.819 seconds. For Bivium-B, they were not able to recover the initial state if they did not guess any bits, while guessing 56 bits allowed them to recover the initial state in 483.640 seconds.

The second experiment used Raddum's approach, where they formed a system of equations consisting of $2n + 177$ variables in $3n$ equations. That is, forming a system of equations in their second experiment required n bits of keystream. For Bivium-A, if they did not guess any bits, they were able to solve the system of equations in 400.810 seconds using $n = 2000$ bits of keystream. If they guessed two bits, they were able to solve the system of equations in 17.930 seconds using $n = 800$ bits of keystream. For Bivium-B, the computation did not finish if they did not guess any bits, whereas if they guessed 56 variables, they were able to solve the system in 1006.259 seconds using $n = 2000$ bits of keystream.

4.3.4 New algebraic analysis on Bivium-A

In this section, we apply Berbain et al.'s and Raddum's technique to Bivium-A and discuss the effectiveness of their technique on an algebraic attack on Bivium-A.

Rewriting the equations for Bivium-A

The equation representing the first keystream bit at $t = 0$ is:

$$z_0 = B_0 \oplus B_{15} \tag{4.21}$$

Bivium-A's keystream equation at time t depends on two stages of B, B_{15+t} and B_t. At $t = 69$, the keystream equation is

$$z_{69} = B_{84} \oplus B_{69} \tag{4.22}$$

Applying Berbain at al.'s technique to Equation 4.22, we can determine the first feedback bit B_{84} by calculating

$$B_{84} = z_{69} \oplus B_{69} \tag{4.23}$$

The feedback bits for the first three feedback bits of B, when Berbain et al.'s and Raddum's technique are applied can be written as follows:

$$\begin{aligned} B_{84} &= z_{69} \oplus B_{69} \\ B_{85} &= z_{70} \oplus B_{70} \\ B_{86} &= z_{71} \oplus B_{71} \end{aligned} \tag{4.24}$$

In this section, we present three possible approaches for recovering the initial state of Bivium-A. The first approach allows us to recover the entire initial state of Bivium-A using one system of equations, while the second and third approach uses a divide-and-conquer approach to form two systems of equations. The two systems of equations are solved sequentially to recover the internal states of Bivium-B's registers B and A respectively.

First approach. In our first approach, we use a single system of equations to recover the initial state of Bivium-A. In Bivium-A, using Berbain et al.'s technique, the feedback bit of B can be expressed in terms of keystream and internal state bits of register B, as

shown in Equation 4.23. In our first approach, we combine our technique with Raddum's relabelling technique to represent the feedback bits of Bivium-B's register B. Since the feedback bits of A can not be written in as linear combinations of keystream and internal state bits, Raddum's relabelling technique is used to calculate the new feedback bits of A. We start to form this system of equations with 177 variables. At each iteration, three equations and two variables are added into the system of equations — one equation for the keystream output bit, one variable and equation each for the feedback bits of A and B. An algebraic representation of the first two sets of equations and variables added are:

$$A_{93} \oplus B_{15} \oplus B_0 \oplus B_1 B_2 \oplus A_{24} = 0 \tag{4.25}$$

$$B_{84} = z_{69} \oplus B_{69} \tag{4.26}$$

$$z_0 \oplus B_{15} \oplus B_0 = 0 \tag{4.27}$$

$$A_{94} \oplus B_{16} \oplus B_1 \oplus B_2 B_3 \oplus A_{25} = 0 \tag{4.28}$$

$$B_{85} = z_{70} \oplus B_{70} \tag{4.29}$$

$$z_1 \oplus B_{16} \oplus B_1 = 0 \tag{4.30}$$

After 69 clocks, the keystream equations start to cancel out and are therefore redundant in the system of equations. To illustrate this, consider the equation describing the first feedback B_{84}. This equation is

$$B_{84} = z_{69} \oplus B_{69}$$

When we try to add the equation describing z_{69} at the 69'th clock into the system of equations, we have

$$z_{69} \oplus B_{69} \oplus B_{84} = 0$$
$$z_{69} \oplus B_{69} \oplus z_{69} \oplus B_{69} = 0$$

After the 69th clock, we drop the keystream equation. However, if we continued to add remaining new variables and equations into the system of equations, we only add two variables and equations at each iteration, which adds unnecessary complexity to solving the system of equations as the number of variables will never match (or exceed) the number of equations, and will always be under-defined. Thus, after 69 clocks, we have completed the construction of the system of equations. This system of equations

consists of 207 equations in 315 equations. Solving this equation using the F4 algorithm will yield 2^{108} possible solutions, which is worse than exhaustive keysearch.

Second approach. In the second approach, we use an algebraic divide-and-conquer approach to recover the initial state of Bivium-A. This consists of two phases.

1. Form the first system of equations to recover the initial state of B.
2. For each of the possible solutions obtained in the first system of equations:
   (a) Substitute the solutions in obtained in the first system of equations into a second system of equations.
   (b) Solve second system of equation to recover initial state of A.

Sequentially solving the two systems of equations may allow us to reduce the complexity recovering the initial state of Bivium-A, since we are solving two systems of equations to recover the internal state of B and A sequentially, compared to solving a single system of equations to recover the initial state of A and B at the same time, as was the case in previous approaches.

We start to form the first system of equations with 84 variables. The first 69 equations can be expressed in terms of initial state bits and keystream bits. The first two equations are as follows:

$$z_0 \oplus B_{15} \oplus B_0 = 0$$
$$z_2 \oplus B_{16} \oplus B_1 = 0$$

To calculate the feedback bits of B, we can use Berbain et al.'s technique to reorder the keystream equations, the first three of which were given in Equation 4.24. After 69 clocks, we can no longer add the keystream equations into the system as these equations would have already been used in the feedback bits of register B, and adding them again into the system of equations is redundant. Therefore, after 69 clocks, we can stop adding any more equations into our system, and have a final system of equations consisting of 84 variables in 69 equations. Solving this system of equations will give us $2^{15} = 32\,768$ possible solutions.

Upon recovering the initial state of B, we can form the second system of equations to recover the initial state of A, substituting the initial state bits of B recovered in the first system of equations into the second system of equations. We start to form the

second system of equations with 93 variables. For every iteration of this second system of equations, we add one keystream equation and use the original state-update function for A and B to update the respective registers A and B. The first two sets of equations in this system are

$$z_0 \oplus B_{15} \oplus B_0 = 0$$
$$z_1 \oplus B_{16} \oplus B_1 = 0$$

After 177 iterations, we have a final system of equations consisting of 93 variables and 177 equations. Note that it is not necessary to run the state-update function of B 177 times to generate 177 bits of keystream. After 111 state-update function iterations of register B, we will have generated enough internal state bits for B to generate 177 bits of keystream. Although this observation does not reduce the number of equations added into the system of equations, it will allow us to reduce the complexity of the equations generated as we are not using Raddum's technique to keep the degree of the equations to two. After 111 iterations, we have a system of equations consisting of 93 variables in 111 equations. Once the second system of equations is solved, the entire initial state of Bivium-A is recovered and an attacker can use this initial state to generate keystream to check if it matches the captured keystream.

A minor drawback of this approach is that solving the sets of equations without knowing any correct initial state bits may not yield a unique solution. However, the number of possible solutions is expected to be small, and wrong initial states can be discarded by checking if the keystream generated by a candidate initial state matches the captured keystream. The bottleneck of our new attack is the first system of equations. If we do not know any correct initial state bits which we can use in our construction of the system of equations, we need to try, in the worst case, all 2^{15} possible states for register B before the correct solution is found for the second system of equations.

If we were to know of any fifteen correct bits of register B, the first system of equations consisting of 69 variables in 69 equations can be constructed. This should yield a unique solution for the first system of equations, which would also make solving for the second system of equations using the F4 algorithm is a lot faster as it only needs to attempt to solve one solution, as opposed to 2^{15} possible solutions.

Third approach. In our third approach, we combine the techniques of Raddum and Berbain et al. to recover the initial state of Bivium-A. In our second approach, we recovered the initial state of B (the first 84 bits of register B) and did not use a new

variable to represent the feedback bits of Bivium-B's registers. In Raddum's technique, the feedback bits of the registers are not written in terms of the initial state bits, but a new variable is introduced. This, as described earlier, has the benefit of keeping the system of equations generated in the Trivium cipher and its variants to a maximum degree of two. For the algebraic analyses presented hereafter, we will use the combined techniques of Raddum and Berbain et al.'s to analyse Trivium-like ciphers and do not consider the technique described in our second approach.

Similar to our second approach, we use an algebraic divide-and-conquer approach to recover the initial state of Bivium-A. This involves forming two system of equations and solving them sequentially. Our third approach consists of three phases:

1. Form the first system of equations using the combined technique of Berbain and Raddum to recover the internal state of B.

2. For each of the possible solutions obtained in the first system of equations:

   (a) Substitute the solutions in obtained in the first system of equations into a second system of equations, formed using the combined technique of Berbain et al. and Raddum.

   (b) Solve second system of equation to recover internal state of A.

Sequentially solving the two systems of equations may allow us to reduce the complexity recovering the initial state of Bivium-A, since we are solving two systems of equations to recover the internal state of B and A sequentially, compared to solving a single system of equations to recover the initial state of A and B at the same time, as was the case in previous approaches.

We start to form the first system of equations with 84 variables. For the first 69 clocks, we add two equations and one variable into the system of equations: one equation representing the keystream, and one equation and variable representing the feedback bit. The first two equations are:

$$z_0 \oplus B_{15} \oplus B_0 = 0 \tag{4.31}$$

$$B_{84} \oplus z_{69} \oplus B_{69} = 0 \tag{4.32}$$

$$z_1 \oplus B_{15} \oplus B_0 = 0 \tag{4.33}$$

$$B_{85} \oplus z_{70} \oplus B_{70} = 0 \tag{4.34}$$

where Equations 4.31 and 4.33 are the first two keystream equations for Bivium-B and Equations 4.32 and 4.34 are new equations derived from our new analysis. After the first 69 clocks, we do not need to add the subsequent keystream equations, as these would have already been added into the system of equations and are redundant. For the last 39 clocks, we only add the equations representing the feedback bit for B and would have generated enough internal state bits for register B to have generated 177 bits of keystream. This gives us a final system of equations after 108 clocks consisting of 192 variables in 177 equations. Solving this system of equations using the F4 algorithm gives us 2^{15} possible solutions.

Upon recovering the internal state of B, we can form the second system of equations to recover the initial state of A, substituting the initial state bits of B recovered in the first system of equations into the second system of equations. We start to form the second system of equations with 93 variables. At each iteration, we add two equations and one variable: one equation and variable relating the feedback bit of A with the internal state bits of A and B, and one equation relating the feedback bit of B with the internal state bits of A and B. It is not necessary to add the keystream equations into this system of equations, as these would have already been solved in the first system of equations. The first two sets of equations in this system of equations are:

$$A_{93} \oplus B_{15} \oplus B_0 \oplus B_1 B_2 \oplus A_{24} = 0$$
$$B_{84} \oplus A_{27} \oplus A_0 \oplus A_1 A_2 \oplus B_6 = 0$$

$$A_{94} \oplus B_{16} \oplus B_1 \oplus B_2 B_3 \oplus A_{25} = 0$$
$$B_{85} \oplus A_{28} \oplus A_1 \oplus A_2 A_3 \oplus B_7 = 0$$

After 108 iterations, we have a system of equations consisting of 201 variables in 216 equations. Solving this system of equations using the F4 algorithm can recover the initial state of A. An attacker can then use this initial state to generate some keystream, and check if the keystream generated matches that which was captured. If it is, the attacker can be confident they have recovered the correct initial state.

In the first system of equations, it is necessary to generate a system of equations where there are more equations than variables. Let us assume if we had only generated 69 new variables in B. In that case, we would have 2^{40} free variables of B in the second system of equations. When this system of equations is solved, we would have had 2^{40} possible solutions. Combined with the number of solutions in the first system of

equations, this gives us a total number of $2^{15} \times 2^{40} = 2^{55}$ possible solutions. While this is still better than exhaustive keysearch, it is clearly more than the 2^{15} possible solutions we have to originally try in our proposed third approach.

Fourth approach. The second system of equations formed in our third approach consists of 201 variables in 216 equations. As we have more equations than variables, some of these equations can be removed from our second system of equations without increasing the number of expected solutions. In this section, we investigate what effect removing these equations and variables will have on the system of equations obtained.

At each iteration, we add two variables and one equation into the second system of equation, which starts with 93 variables. After 93 iterations, we have the following equations:

$$\begin{aligned} A_{93} &= B_{15} \oplus B_0 \oplus B_1 B_2 \oplus A_{24} \\ B_{84} &= A_{27} \oplus A_0 \oplus A_1 A_2 \oplus B_6 \\ &\vdots \\ A_{185} &= B_{107} \oplus B_{92} \oplus B_{93} B_{94} \oplus A_{116} \\ B_{176} &= A_{119} \oplus A_{92} \oplus A_{93} A_{94} \oplus B_{98} \end{aligned}$$

We are able to obtain a second system of equations consisting of 186 variables in 186 equations after 93 iterations of Bivium-A's state-update functions. To build this second system of equations, we need 84 + 93 = 177 variables in register B to form the second system of equations. This means adding 177 − 84 = 93 sets of equations into our first system of equations. We then use the same construction method used in forming the first system of equations for our third approach to form the system of equations in our fourth approach. This first system of equations consist of 177 variables in 162 equations. When this first system of equations is solved, we get 2^{15} possible solutions.

Taking into account the new step of checking how many variables are needed for the second system of equation, our fourth approach consists of three steps. These are:

1. Determine how many equations and variables are needed in the second system of equations.

2. Form the first system of equations using the combined technique of Berbain and Raddum to recover the internal state of B.

3. For each of the possible solutions obtained in the first system of equations:

(a) Substitute the solutions in obtained in the first system of equations into a second system of equations, formed using the combined technique of Berbain et al. and Raddum.

(b) Solve the second system of equations to the recover internal state of A.

Comparison of attacks

Details of the systems of equations formed for Raddum's approach and our third, fourth approach are shown in Table 4.9. The linear equation column lists the number of linear equations in the respective system of equations, while the quadratic column entry lists lists the number of quadratic equations for the respective system of equations. The total equation column lists the total number of equations for the respective system equations, while the K.S. column lists the length of keystream needed to solve the relevant system of equations. The variable column lists the number of variables in the respective system of equations. Finally, the expected solution (Expected solns) column lists the expected number of solutions obtained by solving the respective system of equations.

For Raddum's technique, we need to solve a single system of equations consisting of 399 variables in 399 equations and requires 177 bits of keystream to solve it. Of these 399 equations, 177 are linear and 222 are quadratic. Our third approach requires solving two system of equations sequentially. The first system of equations has 192 variables in 177 linear equations and requires 177 bits of keystream to solve, while the second system has 108 linear and 108 quadratic equations and does not require any keystream bits to solve. Our fourth approach's first system of equations has 177 variables in 162 equations and requires 162 bits of keystream to solve. The second system has 93 linear, 93 quadratic equations and does not require any keystream bits to solve. Since we are solving both systems of equations separately, the complexity of our attacks may be less complex as compared to Raddum's technique. We evaluate these findings in our experimental results, as is shown in Table 4.10.

We attempted, via computer experiments, to recover the initial state of Bivium-A using five different approaches: Simonetti et al.'s [103] approach where they solved Bivium-A's system of equations consisting of 177 variables using Gröbner basis, Raddum's and all of our approaches. The time and memory complexities of the five approaches were compared. We solve the system of equations in these techniques using the F4 algorithm implemented in Magma. In all four cases, the keystream were generated using the same initial state. We then attempted to solve the system of equations in two ways.

1. Without guessing any bits, solve the system of equations for Bivium-A.

Table 4.9: Details of equations for Bivium-A for various approaches

Technique	Linear eqn.	Quadratic eqn.	Total eqn.	K.S. (bits)	Variables	Expected Solns
Raddum	177	222	399	177	399	1
3rd approach (Step 1)	177	0	177	177	192	2^{15}
3rd approach (Step 2)	108	108	216	0	201	1
4th approach (Step 1)	162	0	162	162	177	2^{15}
4th approach (Step 2)	93	93	186	0	186	1

2. Load 15 correct initial state bits of Register B, and solve the system of equations for Bivium-A.

The F4 algorithm implemented in Magma was used to solve these equations. Magma runs on a SGI Altix XE Cluster. The node which Magma runs on has an X5650@2.66GHz 64-bit Intel Xeon processor. The maximum amount of memory set in our experiments is 87 040 MB (85 GB). Table 4.10 lists the time and memory requirements needed for Magma to solve the system of equations. A DNF (Did Not Finish) entry signifies that Magma was not able to solve the system of equations with allocated amount of memory or the computation was not able to finish within 360 000 seconds (100 hours). Magma was not able to solve Raddum's and Simonetti et al's system of equations despite the allocation of 87 040 MB of RAM for the experiments when we did not guess any bits in the system of equations, whereas the second and third new approaches were able to obtain a solution in 15 048 seconds (4.18 hours) using 120 MB of RAM, and 50 040 seconds (13.9 hours) using 135.56 MB of RAM respectively.

If the 15 correct bits of B were loaded into register B, Simonetti's, Raddum's and our second and third approach were able to recover the initial state within nine seconds using at most 13.34 MB of RAM. Our fourth approach required 31.79 seconds and 113.12 MB of RAM. Magma was not able to solve the system of equations in the fourth approach within the allocated time of 360 000 seconds (100 hours), and the log file stated that Magma, at the point of terminating solving the system of equations, had used up 180.13 MB of memory.

There have also been several other attacks which did not use the F4 algorithm on Bivium-A and Bivium-B, and are faster than exhaustive keysearch. Raddum [93] used his relabelling technique along with techniques from graph theory and was able to recover the initial state of Bivium-A in about a day and estimated that the complexity of

Table 4.10: Time, memory, and data complexities for recovering initial state of Bivium-A

	No Guessing			Load correct 15 bits into register B		
	Time (s)	Memory (MB)	K.S (bits)	Time (s)	Memory (MB)	K.S (bits)
Simonetti et al. [103]	DNF	87 040	177	2.16	11.16	177
Raddum [93]	DNF	87 040	177	8.95	13.34	177
2nd approach	15 048	120	177	1.43	10.56	177
3rd approach	50 040	135.56	177	3.29	13.31	177
4th approach	DNF	180.13	162	31.79	113.12	162

recovering the initial state of Bivium-B will take about 2^{56} seconds (2.28×10^9 years). McDonald et al. [85] attempted to recover the initial state of Bivium-A and Bivium-B by treating Bivium-A and Bivium-B as a Boolean Satisfiability (SAT) problem and used a SAT solver to recover the initial state. They were able to recover the initial state of Bivium-A in about sixteen seconds, and estimated that the average time needed to recover the initial state of Bivium-B was about $2^{42.7}$ seconds (about 262 144 years). It should be noted that it is diffcult to compare the effectiveness of our approaches against the attacks by Raddum [93] and McDonald et al. [85] as these attacks were implemented on different hardware platforms, which may have an effect on the time and memory required to recover the initial state of Bivium-A.

4.3.5 New Algebraic Analysis on Bivium-B

The system of equations using Raddum's relabelling technique which can be formed for Bivium-B is the same as Bivium-A's. That is, after 177 clocks of Bivium-B's state-update function, we have a single system of equations which consists of 399 equations in 399 variables. The keystream generation function of Bivium-B takes as input one linear combination of internal state bits: two stages from A and two stages from B. The equation representing the first keystream bit for Bivium-B is:

$$z_0 = A_{27} \oplus A_0 \oplus B_{15} \oplus B_0 \tag{4.35}$$

The 67th keystream bit is:

$$z_{66} = A_{93} \oplus A_{66} \oplus B_{81} \oplus B_{66} \tag{4.36}$$

and the 70th keystream bit is:

$$z_{69} = A_{96} \oplus A_{69} \oplus B_{84} \oplus B_{69} \tag{4.37}$$

Using Berbain et al.'s approach to reorder Equations 4.36 and 4.37, we can determine the first three of register A's feedback bit by calculating:

$$\begin{aligned} A_{93} &= z_{66} \oplus A_{66} \oplus B_{81} \oplus B_{66} \\ A_{94} &= z_{67} \oplus A_{67} \oplus B_{82} \oplus B_{67} \\ A_{95} &= z_{68} \oplus A_{68} \oplus B_{83} \oplus B_{68} \end{aligned} \tag{4.38}$$

Similarly, the first three feedback bits for register B can be calculated as:

$$\begin{aligned} B_{84} &= z_{69} \oplus A_{96} \oplus A_{69} \oplus B_{69} \\ B_{85} &= z_{70} \oplus A_{97} \oplus A_{70} \oplus B_{70} \\ B_{86} &= z_{71} \oplus A_{98} \oplus A_{71} \oplus B_{71} \end{aligned} \tag{4.39}$$

However, we can not use these both sets equations to simultaneously represent the feedback bits for A and B. For example, calculating B_{84} requires knowledge of the feedback bit A_{96}. The equation representing A_{96} is:

$$A_{96} = z_{69} \oplus A_{69} \oplus B_{84} \oplus B_{69} \tag{4.40}$$

If we substitute Equation 4.40 into the equation representing the feedback bit B_{84} from Equation 4.39, we get

$$\begin{aligned} B_{84} &= z_{69} \oplus z_{69} \oplus A_{69} \oplus B_{84} \oplus B_{69} \oplus A_{69} \oplus B_{69} \\ &= B_{84} \end{aligned}$$

which does not allow us to express B_{84} in terms of the current internal state bits and keystream. Therefore, our technique only allows us to express the feedback bits for a single register in terms of internal state and keystream bits.

If we were to express the feedback bits of register A using our technique and the feedback bits of B using Raddum's technique, we add three equations and two variables at each iteration into the system of equations for the first 66 clocks. The first two sets of

these equations and variables are:

$$B_{84} \oplus A_{27} \oplus A_0 \oplus A_1A_2 \oplus B_6 = 0 \tag{4.41}$$
$$A_{93} \oplus z_{66} \oplus A_{66} \oplus B_{81} \oplus B_{66} = 0 \tag{4.42}$$
$$z_0 \oplus A_{27} \oplus A_0 \oplus B_{15} \oplus B_0 = 0 \tag{4.43}$$

$$B_{85} \oplus A_{28} \oplus A_1 \oplus A_2A_3 \oplus B_7 = 0 \tag{4.44}$$
$$A_{94} \oplus z_{67} \oplus A_{67} \oplus B_{82} \oplus B_{67} = 0 \tag{4.45}$$
$$z_1 \oplus A_{28} \oplus A_1 \oplus B_{16} \oplus B_1 = 0 \tag{4.46}$$

where Equations 4.41 and 4.44 are the relabelled bits introduced using Raddum's technique; Equations 4.42 and 4.45 are the the feedback bit equations introduced using our approach; Equations 4.43 and 4.46 are the equations representing the first two keystream bits respectively. Similar to the first approach used in solving Bivium-A in Section 4.3.4, we can only add equations describing keystream equations up to z_{65} as the equations describing the keystream equations from z_{66} are redundant. Therefore, after 66 iterations, we have formed our system of equations for Bivium-B. This system consist of 198 equations in 309 variables. Solving this system of equations using the F4 algorithm will give 2^{111} possible solutions, which is worse than exhaustive keysearch.

The comparison of both systems of equations is shown in Table 4.11. The linear equation column lists the number of linear equations in the respective system of equations, while the quadratic column entry lists lists the number of quadratic equations for the respective system of equations. The total equation column lists the total number of equations for the respective system equations, while the K.S. column lists the length of keystream needed to solve the relevant system of equations. The variable column lists the number of variables in the respective system of equations. Finally, the expected solution (Expected solns) column lists the expected number of solutions obtained by solving the respective system of equations. Our system of equations has less quadratic equations compared to Raddum's technique: Raddum's technique gives 222 quadratic equations, compared 111 quadratic equations in ours. The drawback of our system of equations however, is that we have a greater excess of variables over equations. In Raddum's technique, solving a system consisting of 399 variables in 399 equations will yield a unique solution. In contrast, solving the system of equations in our approach, which consists of 399 variables in 288 equations, using the F4 algorithm of will yield 2^{111} possible solutions (initial states).

Table 4.11: Details of equations for Bivium-B

Technique	Linear eqn.	Quadratic eqn.	Total eqn.	K.S. (bits)	Variables	Expected solns
Raddum	177	222	399	177	399	1
Our approach	132	66	198	132	309	2^{111}

4.3.6 New Algebraic Analysis on Trivium

Berbain et al.'s technique of expressing any internal state variable of a particular NLFSR as a linear combination of initial state variables and keystream bits can be applied to Trivium. For example, the equation representing the 67'th keystream bit is

$$\begin{aligned} z_{66} =& A_{93} \oplus A_{66} \oplus B_{81} \oplus B_{66} \oplus \\ & C_{111} \oplus C_{66} \end{aligned} \tag{4.47}$$

where A_{93} and C_{111} are the first feedback bits for A and C. If we look at the equation representing the 70'th keystream bit, we have

$$z_{69} = A_{96} \oplus A_{69} \oplus B_{84} \oplus B_{69} \oplus C_{114} \oplus C_{69} \tag{4.48}$$

where B_{84} is the first new feedback bit for B. Instead of using the relations given in Equation 4.16, 4.17 or 4.18 to update the internal state of A, B, and C, we can re-order Equations 4.47 and 4.48 to obtain the relevant feedback bits. The three possible equations representing the next three feedback bits for all three registers are:

$$\begin{aligned} A_{93} &= z_{66} \oplus A_{66} \oplus B_{81} \oplus B_{66} \oplus C_{111} \oplus C_{66} \\ A_{94} &= z_{67} \oplus A_{67} \oplus B_{82} \oplus B_{67} \oplus C_{112} \oplus C_{67} \\ A_{95} &= z_{68} \oplus A_{68} \oplus B_{83} \oplus B_{68} \oplus C_{113} \oplus C_{68} \end{aligned} \tag{4.49}$$

and

$$\begin{aligned} B_{84} &= z_{69} \oplus A_{96} \oplus A_{69} \oplus B_{69} \oplus C_{114} \oplus C_{69} \\ B_{85} &= z_{70} \oplus A_{97} \oplus A_{70} \oplus B_{70} \oplus C_{115} \oplus C_{70} \\ B_{86} &= z_{71} \oplus A_{98} \oplus A_{71} \oplus B_{71} \oplus C_{116} \oplus C_{71} \end{aligned} \tag{4.50}$$

and

$$\begin{aligned} C_{111} &= z_{66} \oplus A_{66} \oplus A_{93} \oplus B_{81} \oplus B_{66} \oplus C_{66} \\ C_{112} &= z_{67} \oplus A_{67} \oplus A_{94} \oplus B_{82} \oplus B_{67} \oplus C_{67} \\ C_{113} &= z_{68} \oplus A_{68} \oplus A_{95} \oplus B_{83} \oplus B_{68} \oplus C_{68} \end{aligned} \tag{4.51}$$

Similar to Bivium-B's equations, we can only express the feedback bits of one register in terms of initial state and keystream bits, as using more than one of these new equations will cause the equations to cancel out, and leave us with the inability to express the feedback bits in terms of initial state, and keystream bits. In the following, we express the feedback bits of A using our approach, and the feedback bits of B and C using Raddum's technique. This new system of equations starts off with 288 variables and zero equations. For each of the first 66 clocks, we add three new variables and four new equations to the system of equations. The first two sets of equations for the first 66 clocks are:

$$A_{93} \oplus z_{66} \oplus A_{66} \oplus B_{81} \oplus B_{66} \oplus C_{111} \oplus C_{66} = 0 \tag{4.52}$$
$$B_{84} \oplus B_6 \oplus A_{27} \oplus A_0 \oplus A_1A_2 = 0 \tag{4.53}$$
$$C_{111} \oplus C_{24} \oplus B_{15} \oplus B_0 \oplus B_1B_2 = 0 \tag{4.54}$$
$$z_0 \oplus A_{27} \oplus A_0 \oplus B_{15} \oplus B_0 \oplus C_{45} \oplus C_0 = 0 \tag{4.55}$$

$$A_{94} \oplus z_{67} \oplus A_{67} \oplus B_{82} \oplus B_{67} \oplus C_{112} \oplus C_{67} = 0 \tag{4.56}$$
$$B_{85} \oplus B_7 \oplus A_{28} \oplus A_1 \oplus A_2A_3 = 0 \tag{4.57}$$
$$C_{112} \oplus C_{25} \oplus B_{16} \oplus B_1 \oplus B_2B_3 = 0 \tag{4.58}$$
$$z_1 \oplus A_{28} \oplus A_1 \oplus B_{16} \oplus B_1 \oplus C_{46} \oplus C_1 = 0 \tag{4.59}$$

where Equations 4.53, 4.54, 4.57 and 4.58 are the equations describing the relabelled bits introduced using Raddum's technique; Equations 4.52 and 4.56 are the feedback bit equations introduced using our approach and Equations 4.55 and 4.59 are the equations related to the keystream bits for z_i, for $0 \leq i \leq 65$. After 66 iterations, similar to Bivium-A and Bivium-B, adding the keystream equations into the system of equations are redundant, and we can stop adding variables and equations into the system. This gives us a final system of equations consisting of 486 variables in 264 equations. The comparison of both systems of equations is shown in Table 4.12. The linear equation column lists the number of linear equations in the respective system of equations, while

Table 4.12: Details of equations for Trivium

Technique	Linear eqn.	Quadratic eqn.	Total eqn.	K.S. (bits)	Variables	Expected solns
Raddum	288	666	954	288	954	1
Our approach	132	132	264	132	486	2^{222}

the quadratic column entry lists lists the number of quadratic equations for the respective system of equations. The total equation column lists the total number of equations for the respective system equations, while the K.S. column lists the length of keystream needed to solve the relevant system of equations. The variable column lists the number of variables in the respective system of equations. Finally, the expected solution (Expected solns) column lists the expected number of solutions obtained by solving the respective system of equations.

Our system of equations has less quadratic equations compared to Raddum's technique: Raddum's technique has 666 quadratic equations, compared to 132 quadratic equations in ours. The drawback of our system of equations however, is that our system of equations have a greater excess of variables over equations to Raddum's technique. In Raddum's technique, solving a system consisting of 954 variables in 954 equations will yield a unique solution. In contrast, solving our system of equations consisting of 486 variables in 264 equations using the F4 algorithm will yield 2^{222} possible solutions, which is worse than exhaustive keysearch.

4.3.7 Algebraic Analysis on Trivium Variants

To determine if Berbain's technique can be applied for Trivium-like ciphers for which $j > 1$ and $q > 1$, two alternative versions of Trivium are proposed: Trivium-*A*, Trivium-*AB*. The two versions can be differentiated by their letters designation, with the letter representing the register(s) whose stages are used by the linear function for keystream generation. The state-update functions for the two ciphers are the same as the original Trivium proposal. The equations representing the first three keystream bits of Trivium-*A*,

Table 4.13: Values of q, q', and j for Trivium-A, Trivium-AB

	Trivium-A	Trivium-AB
q	3	3
q'	1	1
j	1	2

Trivium-AB are:

$$z_0 = A_{27} \oplus A_0$$
$$z_1 = A_{28} \oplus A_1$$
$$z_2 = A_{29} \oplus A_2$$

and

$$z_0 = A_{27} \oplus A_0 \oplus B_{15} \oplus B_0$$
$$z_1 = A_{28} \oplus A_1 \oplus B_{16} \oplus B_1$$
$$z_2 = A_{29} \oplus A_2 \oplus B_{17} \oplus B_2$$

respectively. The values of q', q and j for Trivium-A and Trivium-B are shown in Table 4.13.

Analysis of Trivium-A

The keystream generation equation Trivium-A can be reordered so that the feedback bits for Trivium-A can rewritten as a linear combination of keystream and internal state bits of register A. The first three feedback bits of A written in terms of keystream and internal state bits are:

$$A_{93} = z_{66} \oplus A_{66}$$
$$A_{94} = z_{67} \oplus A_{67}$$
$$A_{95} = z_{68} \oplus A_{68}$$

Analysis of Trivium-A using our third approach. Our first system of equations is used to recover the initial state of register A. We start to form the first system of equations

with 93 variables. For the first 66 clocks, we add two equations and one variable into the system of equations. For the next 156 iterations, we only add one equation and one variable for the equation which describes the feedback bit for register A. We no longer add the equations describing keystream as z_{66} has already been used to describe A_{93} in terms of internal state bits and keystream. After a total of 222 iterations, we have calculated enough internal state bits of for register A to generate 288 bits of keystream, and we have a system of equations consisting of 315 variables in 288 equations. Solving this system of equations using the F4 algorithm will give us 2^{27} possible solutions. The first two equations in our system are:

$$A_{93} \oplus z_{66} \oplus A_{66} = 0 \tag{4.60}$$

$$z_0 \oplus A_{27} \oplus A_0 = 0 \tag{4.61}$$

$$A_{94} \oplus z_{67} \oplus A_{67} = 0 \tag{4.62}$$

$$z_1 \oplus A_{28} \oplus A_0 = 0 \tag{4.63}$$

where Equations 4.60 and 4.62 are the feedback bits introduced using our approach, and Equations 4.61 and 4.63 are the equations related to the keystream bits for z_0, for $1 \leq i \leq 65$.

The second system of equations is used to recover the initial state of registers B and C. As the contents of register A has already been recovered, we can substitute the values of A into the appropriate stages. We start to form the second system of equations with 195 variables. As we are using Raddum's relabelling technique, at each iteration, we add two variables and three equations relating the feedback bits of A, B and C to the internal state. The first two equations are:

$$A_{93} \oplus A_{24} \oplus C_{45} \oplus C_0 \oplus C_1C_2 = 0$$
$$B_{84} \oplus B_6 \oplus A_{27} \oplus A_0 \oplus A_1A_2 = 0$$
$$C_{111} \oplus C_{24} \oplus B_{15} \oplus B_0 \oplus B_1B_2 = 0$$

$$A_{94} \oplus A_{25} \oplus C_{46} \oplus C_1 \oplus C_2C_3 = 0$$
$$B_{85} \oplus B_7 \oplus A_{28} \oplus A_1 \oplus A_2A_3 = 0$$
$$C_{112} \oplus C_{25} \oplus B_{16} \oplus B_1 \oplus B_2B_3 = 0$$

We no longer need to add the keystream equations as both of these would have been already recovered in the first system of equations. After 222 clocks, we have a system of equations consisting of 639 variables in 666 equations. Solving this system of equations using the F4 algorithm will allow us to recover the initial state of B and C. Once we recover internal state of B and C, we use the recovered contents of A, B and C and check if the keystream it generates is the same keystream we captured. If it is, we have successfully recovered the initial state of Trivum-A.

Analysis of Trivium-A using our fourth approach. The second system of equations formed in our third approach consists of 639 variables in 666 equations. As we have more equations than variables, some of these equations can be removed from our second system of equations without increasing the number of expected solutions. In this section, we investigate what effect removing these equations and variables will have on the system of equations obtained.

At each iteration, we add two variables and three equations into the second system of equations, which starts with 195 variables. These equations are:

$$
\begin{aligned}
&A_{93} \oplus A_{24} \oplus C_{45} \oplus C_0 \oplus C_1C_2 = 0\\
&B_{84} \oplus B_6 \oplus A_{27} \oplus A_0 \oplus A_1A_2 = 0\\
&C_{111} \oplus C_{24} \oplus B_{15} \oplus B_0 \oplus B_1B_2 = 0\\
&\vdots\\
&A_{287} \oplus A_{218} \oplus C_{239} \oplus C_{194} \oplus C_{195}C_{196} = 0\\
&B_{278} \oplus B_{200} \oplus A_{221} \oplus A_{194} \oplus A_{195}A_{196} = 0\\
&C_{305} \oplus C_{218} \oplus B_{209} \oplus B_{194} \oplus B_{195}B_{196} = 0
\end{aligned}
$$

We are able to obtain a system of equations consisting of 585 variables in 585 equations after 195 iterations of Trivium-A's state-update functions. To build this second system of equations, we need $93 + 195 = 288$ variables in register A. This means adding 195 sets of equations for register A into our first system of equations, using the same construction method used in form the first system of equations in our third approach. This first system of equations consists of 288 variables in 261 equations. Solving this system of equations using the F4 algorithm will give us 2^{27} possible solutions.

Table 4.14: Details of equations for Trivium-A

Technique	Linear eqn.	Quadratic eqn.	Total eqn.	K.S. (bits)	Variables	Expected solns
Raddum	288	666	954	288	954	1
3rd approach (Step 1)	288	0	288	288	315	2^{27}
3rd approach (Step 2)	222	444	666	0	639	1
4th approach (Step 1)	261	0	261	261	288	2^{27}
4th approach (Step 2)	195	390	585	0	585	1

Comparison of system of equations for Trivium-A. Details of the systems of equations formed for Raddum's and our approach to recovering the initial state of Trivium-A are shown in Table 4.14. The linear equation column lists the number of linear equations in the respective system of equations, while the quadratic column entry lists lists the number of quadratic equations for the respective system of equations. The total equation column lists the total number of equations for the respective system equations, while the K.S. column lists the length of keystream needed to solve the relevant system of equations. The variable column lists the number of variables in the respective system of equations. Finally, the expected solution (Expected solns) column lists the expected number of solutions obtained by solving the respective system of equations.

For Raddum's technique, we need to solve a system of equations consisting of 954 variables in 954 equations and requires 288 bits of keystream. Of these 954 equations, 288 are linear and 666 are quadratic. Our third approach requires solving two separate systems of equations: the first system consists of 288 linear equations in 315 variables and requires 288 bits of keystream, while the second system of consists of 222 linear equations, 444 quadratic equations in 639 variables and does not require any keystream. Our fourth approach requires solving two separate systems of equations: the first system consists of 261 linear equations in 288 variables and requires 261 bits of keystream, while the second system of consists of 195 linear equations, 390 quadratic equations in 585 variables and does not require any keystream. Since we are solving both systems of equations separately, it is expected that both our approaches will be less complex as compared to Raddum's technique. This is due to the fact that the first system of equations is linear, and the second system of equations has less quadratic equations than in Raddum's technique.

Analysis of Trivium-AB

The keystream generation equation Trivium-AB can be reordered so that the feedback bits for Trivium-AB can rewritten as a linear combination of keystream and internal state bits of register A or a linear combination of keystream and internal state bits of register B. The first three feedback bits of register A written in terms of keystream and internal state bits are:

$$A_{93} = z_{66} \oplus A_{66} \oplus B_{81} \oplus B_{66}$$
$$A_{94} = z_{67} \oplus A_{67} \oplus B_{82} \oplus B_{67}$$
$$A_{95} = z_{68} \oplus A_{68} \oplus B_{83} \oplus B_{68}$$

The first three feedback bits of register B written in terms of keystream and internal state bits are:

$$B_{84} = z_{69} \oplus A_{96} \oplus A_{69} \oplus B_{69}$$
$$B_{85} = z_{70} \oplus A_{97} \oplus A_{70} \oplus B_{70}$$
$$B_{86} = z_{71} \oplus A_{98} \oplus A_{71} \oplus B_{71}$$

Analysis of Trivium-AB using our third approach The first system of equations recovers the internal state of A and B. Similar to Bivium-A and Trivium, we can not use both sets of equations to simultaneously represent the feedback bits for A and B. Therefore, we can only use Berbain et al.'s technique to express the feedback bits in terms of keystream and internal state bits for a single register. In the first system of equations, we express the feedback bits of register A using the combined techniques of Raddum and Berbain et al.'s, and express the feedback bits of register B using Raddum's technique. For the first 66 clocks, we add two variables and three equations into the system. The

first two sets of these are:

$$A_{93} \oplus z_{66} \oplus A_{66} \oplus B_{81} \oplus B_{66} = 0 \tag{4.64}$$
$$B_{84} \oplus B_6 \oplus A_{27} \oplus A_0 \oplus A_1 A_2 = 0 \tag{4.65}$$
$$z_0 \oplus A_{27} \oplus A_0 \oplus B_{15} \oplus B_0 = 0 \tag{4.66}$$

$$A_{94} \oplus z_{67} \oplus A_{67} \oplus B_{82} \oplus B_{67} = 0 \tag{4.67}$$
$$B_{85} \oplus B_7 \oplus A_{28} \oplus A_1 \oplus A_2 A_3 = 0 \tag{4.68}$$
$$z_1 \oplus A_{28} \oplus A_1 \oplus B_{16} \oplus B_1 = 0 \tag{4.69}$$

where Equations 4.65 and 4.68 are the equations describing the relabelled bits introduced using Raddum's technique, Equations 4.64 and 4.67 are the feedback bit equations introduced using our approach and Equations 4.66 and 4.69 are the equations related to the keystream bits for z_i, for $0 \leq i \leq 65$. For the next 156 clocks, we drop the keystream equations and only add the equations for the feedback bits for registers A and B. This is due to the fact that from z_{66} onwards, the keystream equations have already been used for the feedback bit equations for register A, and adding them into the system of equations is redundant. After 222 clocks, we have a final system of equations consisting of 621 variables in 510 equations. Solving this system of equations will give us 2^{111} possible solutions.

The second system of equations recovers the internal state of register C. This system of equations starts with 111 variables. For each of the 222 iterations, we add one variable and two equations into the system of equations. The first two sets of equations are:

$$A_{93} \oplus C_{24} \oplus C_{45} \oplus C_0 \oplus C_1 C_2 = 0$$
$$C_{111} \oplus C_{24} \oplus B_{45} \oplus B_0 \oplus B_1 B_2 = 0$$

$$A_{94} \oplus C_{25} \oplus C_{46} \oplus C_1 \oplus C_2 C_3 = 0$$
$$C_{112} \oplus C_{25} \oplus B_{16} \oplus B_1 \oplus B_2 B_3 = 0$$

We no longer need to add the keystream equations as these would have been already recovered in the first system of equations. We add the equations representing the feedback bits of A into the system of equations as register A's state-update function takes as input stages from register C. We do not include the equations representing the

feedback bits of B as the state-update function only takes as inputs stages from A and B, which have already been solved in the first system of equations. After 222 clocks, we have a system of equations consisting of 333 variables in 444 equations.

Analysis of Trivium-AB using our fourth approach. The second system of equations formed in our third approach consists of 333 variables in 444 equations. As we have more equations than variables, some of these equations can be removed from our second system of equations without increasing the number of expected solutions. In this section, we investigate what effect removing these equations and variables will have on the system of equations obtained.

At each iteration, we add two variables and three equations into the second system of equations, which starts with 111 variables. These equations are:

$$\begin{aligned}
&A_{93} \oplus C_{24} \oplus C_{45} \oplus C_0 \oplus C_1C_2 = 0\\
&C_{111} \oplus C_{24} \oplus B_{45} \oplus B_0 \oplus B_1B_2 = 0\\
&\vdots\\
&A_{203} \oplus C_{134} \oplus C_{155} \oplus C_{110} \oplus C_{111}C_{112} = 0\\
&C_{221} \oplus C_{134} \oplus B_{155} \oplus B_{110} \oplus B_{111}B_{112} = 0
\end{aligned}$$

We are able to obtain a system of equations consisting of 222 variables in 222 equations after 111 iterations of Trivium-A's state-update functions. To build this second system of equations, we need $93 + 111 = 204$ variables in register A and $45 + 111 = 156$ variables in register B to form the second system of equations. During the construction of the first system of equations, we add $204 - 93 = 111$ equations describing the feedback bits of register A and add $156 - 84 = 72$ equations describing the feedback bits of register B. This first system of equations consists of 453 variables in 342 equations. Solving this system of equations using the F4 algorithm will give us 2^{111} possible solutions.

Comparison of system of equations for Trivium-AB. Details of the systems of equations formed for Raddum's and our approach to recovering the initial state of Trivium-AB are shown in Table 4.15. The linear equation column lists the number of linear equations in the respective system of equations, while the quadratic column entry lists lists the number of quadratic equations for the respective system of equations. The total equation column lists the total number of equations for the respective system equations, while the K.S. column lists the length of keystream needed to solve the relevant system of equations.

Table 4.15: Details of equations for Trivium-AB

Technique	Linear eqn.	Quadratic eqn.	Total eqn.	K.S. (bits)	Variables	Expected Solns
Raddum	288	666	954	288	954	1
3rd approach (Step 1)	288	222	510	288	621	2^{111}
3rd approach (Step 2)	222	222	444	0	222	1
4th approach (Step 1)	270	72	342	270	453	2^{111}
4th approach (Step 2)	111	111	222	0	222	1

The variable column lists the number of variables in the respective system of equations. Finally, the expected solution (Expected solns) column lists the expected number of solutions obtained by solving the respective system of equations. For Raddum's technique, we need to solve a system of equations consisting of 954 variables in 288 linear and 666 quadratic equations. Solving Raddum's system of equations requries 288 bits of keystream. Our approaches also requires solving two separate system of equations: The first system in our third approach consists 621 variables in 288 linear equations and 222 quadratic equations. The first system of equations in our first approach also requires 288 bits of keystream to solve, while the second system of consists 222 variables in 222 linear equations and 222 quadratic equations. The first system in our fourth approach consists 453 variables in 270 linear equations and 72 quadratic equations. The first system of equations in our first approach requires 270 bits of keystream to solve, while the second system of consists 111 variables in 111 linear equations and 111 quadratic equations.

Our approaches may be less complex to solve as compared to Raddum's system of equations as they have less quadratic equations. The drawback of our system of equations however, is that the first system of equations in both our approaches has a greater excess of variables over equations compared to Raddum's technique. In Raddum's technique, solving a single system consisting of 954 variables in 954 equations will yield a unique solution. In contrast, solving the first system of equations in both our approaches will yield 2^{111} possible solutions, which is worse than exhaustive keysearch.

4.3.8 Discussion

In this section, we have presented analyses on the feasibility of recovering the initial state of Trivium-like ciphers using the combined approaches of Berbain et al. and Raddum. The analysis was done through the approaches named the third, and fourth approach.

Our third approach did not take into account the number of variables and equations needed for the construction of the second system of equations, leading to a system of equations which has more equations than variables. Our fourth approach determined how many variables and equations were needed for the second system of equations, and adjusted the number of variables calculated in the first system of equations accordingly. The fourth approach's system of equations has the benefit of having less equations, variables, and needed less keystream compared to our third approach and Raddum's approach.

In their paper, Berbain et al. [10] left the application of their technique to keystream generators which updated $q > 1$ bits of internal state at each iteration and only output $q' < q$ linear combinations of the state-bits as an open question. In this section, we have answered this open question. We have also shown that the possibility of recovering the initial states of some Trivium-like ciphers using our analysis with complexity less than exhaustive keysearch depends on the relationship between j (the number of registers the keystream generation function takes as input to generate keystream) and q (the number of registers whose internal state is updated at each iteration).

In the case of Bivium-A, Trivium-A and Trivium-AB, where $j < q$, it was shown that it may be possible to recover the initial state of the respective ciphers using the F4 algorithm with a complexity that is less than recovering the same initial state using Raddum's relabelling technique. It is shown through computer simulations that the recovery of Bivium-A's initial state using our approaches can be significantly faster than Raddum's technique if the same amount of keystream is needed. We also demonstrated a successful initial state recovery attack if an attacker knew 15 correct initial state bits of Bivium-A's register B and had access to 162 bits of keystream. This attack however, has a higher time and memory complexity compared to Raddum's technique.

In the case of Trivium-A, the actual complexity of recovering its initial state using our technique is unknown but is expected to be less than solving it using Raddum's technique. The only cipher where a divide-and-conquer approach is not possible is for Trivium-AB. When the first system of equations for Trivium-AB is solved, it gives 2^{111} possible solutions and trying all possible solutions to determine which is the correct initial state is worse than exhaustive keysearch. For all three ciphers, it is possible to use a divide-and-conquer approach as $j < q$. The success of solving these equations with a complexity less than exhaustive keysearch however, would depend on the system of equations generated. The complexity of our new approaches can be calculated as follows: Let T_f denote the total time taken needed for the F4 algorithm to solve both systems

of equations; with T_1 and T_2 denoting the time needed for the F4 algorithm to solve the first and second system of equations respectively, and N_A denoting the number of solutions obtained in solving the first system of equations. The complexity of recovering the initial state of Trivium-like ciphers where $j < q$ is

$$T_f = T_1 + (N_A \times T_2) \tag{4.70}$$

In the case of Bivium-B and Trivium, $j = q$, and it is not possible to use a divide-and-conquer approach to recover the initial state of the ciphers. To recover the initial state of these ciphers using our approach requires us to solve a single system of equations. However in the situation of $j = q$, using our approach to build this system of equations has a drawback. Since a keystream equation is essentially being used to represent a feedback bit, it does not allow us to add an additional (keystream) equation after some iterations, and the number of variables and equations added into the system of equations at each iteration is the same, as was demonstrated in our analysis of Bivium-B and Trivium. If the number of variables is greater than the number of equations prior to us not adding the keystream equation into the system, this system will always have more variables than equations. In both Bivium-B and Trivium, trying all possible solutions obtained to determine which is the correct initial state when the system of equations is solved is worse than exhaustive keysearch. The complexity of recovering the initial state of Trivium-like ciphers where $j = q$ is

$$T_f = T_s + N_A \tag{4.71}$$

where T_s is the time taken to solve the system of equations and N_A is the number of solutions obtained when the system of equations is solved.

In our analyses of Trivium-like ciphers, the number of solutions obtained when a particular system of equations is solved is found by counting the number of equations and variables added at each iteration. This set of solutions can be generalised. Let S_R be the total size of the registers whose state-update is not re-written using our approach *and* whose stages are used during the construction for the particular system of equations. For Trivium-like ciphers where $j < q$, this system of equations is the first system of equations. For Trivium-like ciphers where $j = q$, this system of equation is the sole system of equations obtained when we perform an algebraic analysis on them. For the register whose feedback bit is written in terms of keystream and internal state bits, let D_{l-0} be the "distance" between the largest and stage "o", both of which are used as input

to the output function. The size of the set of solutions, N_A obtained when the system of equations is solved is:

$$2^{S_R} \times 2^{D_{l-0}} = N_A \tag{4.72}$$

To illustrate this, we use the system of equations constructed for Trivium using our approach which was analysed in Section 4.3.6. For the case of Trivium, S_R will be the combined size of registers B and C, $S_R = 84 + 111 = 195$. D_{l-0} is the "distance" between the largest stage (A_{27}) and A_0 of register A which are used as inputs to Trivium's output function. That is, $D_{l-0} = 27 - 0 = 27$. The total set of solutions obtained when the respective values for S_R and D_{l-0} are substituted into Equation 4.72 is

$$2^{195} \times 2^{27} = 2^{222}$$

which is the same size of solutions obtained in our previous analysis in Section 4.3.6.

A summary of the results on our analyses of Trivium-like ciphers with regards to the number of equations, variables, and solutions obtained when their system of equations are formed and solved using our approaches, and their relationship with S_R and D_{l-0}, is shown in Table 4.16. It shows the number of equations and variables obtained from solving the various system of equations. For Bivium-A, Trivium-A and Trivium-AB, this is divided into Step 1 and Step 2 for our third and fourth approaches respectively. Since only one system of equations needs to be solved for Bivium-B and Trivium before their initial state can be recovered, the appropriate values for the number of equations, variables and solutions are recorded in the Step 1 of the 3rd approach column of Table 4.16.

Relationship between inputs to output function and success of attacks

Our new attacks on Bivium-A and Trivium-A can be prevented by making changes to the keystream output function. Recall the keystream generation equation for Bivium-A shown in Equation 4.21 and the equation representing the first feedback bit of register B, B_{84}, shown in Equation 4.23. As was discussed in our previous analyses of Bivium-A in Section 4.3.4, the number of solutions obtained when the first system of equations is solved has an important effect on the effectiveness of our attack. If the number of solutions obtained is less than the total number of possible keys, it may be possible to recover the initial state of the cipher faster than exhaustive keysearch, otherwise it will be worse than exhaustive keysearch. In our third and fourth approach, a keystream

Table 4.16: Details on systems of equations in our third and fourth approaches for Trivium-like ciphers

	Parameters		3rd approach (Step 1)			3rd approach (Step 2)			4th approach (Step 1)			4th approach (Step 2)		
	S_R	D_{l-0}	Eqn	Var	Soln (N_A)	Eqn	Var	Soln	Eqn	Var	Soln (N_A)	Eqn	Var	Soln
Bivium-A	0	15	177	192	2^{15}	216	201	1	162	177	2^{15}	186	186	1
Bivium-B	84	27	198	309	2^{111}	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.
Trivium-A	0	27	288	315	2^{27}	666	639	1	261	288	2^{27}	585	585	1
Trivium-AB	84	27	510	621	2^{111}	444	222	1	342	453	2^{111}	222	222	1
Trivium	195	27	264	486	2^{222}	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.

equation is used to represent a feedback bit, preventing us from adding keystream equation at some future point due to it being redundant. In Equation 4.72, we showed how it is possible to calculate the size of the set of solutions when a system of equations is solved for Trivium-like ciphers. To make an attack on such a cipher worse than exhaustive keysearch we need modify the output function of Trivium-like ciphers such that this happens early on in the first system of equations. For example, suppose the output function of Bivium-A was, instead,

$$z_0 = B_{0+i} \oplus B_{83+i}, \text{ for } i \geq 0 \tag{4.73}$$

The first feedback bit B_{84}, rewritten in terms of linear combinations of keystream and internal state bits is now

$$B_{84} = z_1 \oplus B_1$$

We start to form the first system of equations with 84 variables. For the first iteration, two equations and one variable are added. From the second iteration onwards, the keystream equation no longer needs to be added as it is already used to represent the feedback bit of register B. From the second iteration onwards, we will only add one variable and one equation into the system of equations. At this point, the difference between the number of variables will always be 83 more than the number of equations. If this system of equations is solved, it will give 2^{83} possible solutions. Equation 4.72 can also be used to determine the size of N_A. In this example, the distance D_{l-0} between B_{83} and B_0 is 83. $S_R = 0$, as the contents of register A are not used in during the construction of the first system of equations. Therefore, substituting the values of D_{l-0} and S_R into Equation 4.72 will also yield

$$2^U = 2^{S_R} \times 2^{D_{l-0}} = 2^0 \times 2^{83} = 2^{83}$$

Since the set of solutions for the first system of equations is already larger than the number of all possible keys, it is not worthwhile to generate and solve the second system of equations. In general, for Trivium-like ciphers where $j < q$, if $D_{l-0} > 80$, the number of solutions obtained when the first system of equations is solved will be more than the total possible number of secret keys and the complexity of the entire attack will be worse than exhaustive keysearch. It should also be noted that Equation 4.72 only gives an indication of how large the set of solutions will be when the system of equations is

solved. It does not take into account the complexity of solving the system of equations. As was shown in Equation 4.70, the time T_1 and T_2 needs to be taken in account when calculating the total time complexity T_f of solving the system of equation. If T_f has a complexity which is worse than exhaustive keysearch, the cipher is still considered secure against the our algebraic analyses.

Conversely, assume D_{l-0} was a small value. We use Trivium to illustrate how this change reduces the number of solutions obtained when the system of equations was solved. Assume that the output function for Trivium was instead,

$$z_{0+i} = A_{i+1} \oplus A_i \oplus B_{15+i} \oplus B_i \oplus C_{45+i} \oplus C_{0+i} \text{ for } i \geq 0 \tag{4.74}$$

We use our approach to represent the feedback bit equations for register A, and Raddum's method to represent the feedback bits equations for registers B and C. In this case, $D_{l-0} = A_1 - A_0 = 1$ and $S_R = 111 + 84 = 195$. Substituting these values into Equation 4.72 will give the following value

$$2^U = 2^{S_R} \times 2^{D_{l-0}} = 2^{195} \times 2^{1-0} = 2^{196}$$

Recall from Section 4.3.6 that the size of the set of solutions when the system of equations formed using our approach is solved was 2^{222}. By changing the inputs to the output function, the size of the set of solutions obtained is 2^{196}, a 2^{26} factor decrease in the number of possible solutions. However, going through all 2^{196} possible solutions to determine which is the correct initial state is worse than exhaustive keysearch.

For Trivium-like ciphers where $j = q$, Equation 4.72 allows us to determine how many solutions are obtained when the system of equations are solved. In the case of Trivium, the size of each register is more than 80. Therefore, changing the position of the stages used as input into its output function such that D_{l-0} is as small as possible may not be sufficient for our algebraic analysis to succeed as 2^{S_R} will always be larger than the number of possible secret keys. It should also be noted that Equation 4.72 only gives an indication of how large the set of solutions when the system of equations is solved will be. It does not take into account the complexity of solving the system of equations. As was shown in Equation 4.71, the time T_s needs to be taken in account when calculating the total time complexity T_f of solving the system of equation. If T_f has a complexity which is worse than exhaustive keysearch, the cipher is still considered secure against our algebraic analysis.

Relationship between size of registers and success of attacks

The size of the registers which are used in the formation of the first system of equations can have an effect on the number of solutions obtained when the first system of equations. For example, assume that Trivium's three registers, A, B, and C had lengths 198, 45 and 45 bits. During the construction of the system of equations, we rewrite the feedback bits of register A in terms of initial state and keystream bits. Thus, $D_{l-0} = 27 - 0 = 27$. For this Trivium cipher, $S_R = 45 + 45$, and $N_A = 2^{90} \times 2^{27} = 2^{117}$. Recall from Section 4.3.6 that the size of the set of solutions obtained when the original system of equations using the our approach is solved was 2^{222}. By changing the size of the registers, the size of the set of solutions obtained is 2^{117}, a 2^{105} factor decrease in the number of possible solutions. However, going through all 2^{117} possible solutions to determine which is the correct initial state is worse than exhaustive keysearch.

4.4 Conclusion

This chapter has analysed keystream generators which use a linearly filtered nonlinear feedback shift register to generate keystream. In Section 4.1, the m-tuple distribution of keystream sequences formed by linearly filtered nonlinear feedback shift register was analysed. Our findings indicate that the keystream formed by these generators can have a non-uniform distribution if the linear output function takes as input more than three stages in the nonlinear feedback shift register. Non-occurring m-tuples were also observed for all keystream generators used in our experiments. Note that these findings were also present when the m-tuple distributions of keystream sequences formed by nonlinear filter generators (NLFGs) were analysed in Chapter 3. The m-tuple distribution of the keystream generated by linear functions which take as input taps which form a FPDS are generally less uniformly distributed as compared to the m-tuple distribution of the keystream generated by linear function which take as inputs taps which are consecutive. This is in contrast to the m-tuple distribution of NLFGs. In the m-tuple distribution of keystreams generated by NLFSRs, the m-tuple distribution of keystream formed by a nonlinear function which takes as inputs taps which are consecutive is generally less uniform than the m-tuple distribution of keystream formed by a nonlinear function which takes as inputs taps which which form a full-positive difference set.

In Section 4.2, the work on slid pairs on Trivium done by Priemuth-Schmid and Biryukov [92] and Zeng and Qi [115] was extended. In our work, we have shown two

things. Firstly, for $111 \leq \alpha \leq 121$, $\Gamma_\alpha \cap \Theta_\alpha$ is a empty set. That means none of the loaded states in the set Γ_α can be found in the loaded states in the set Θ_α, where Θ_α contains the set of loaded states formed by a particular loaded state in Γ_α. Secondly, for $111 \leq \alpha \leq 113$ and $111 \leq \alpha' \leq 113$, the set $\Gamma_\alpha \cap \Theta_{\alpha'}$ is also empty. If any of those sets were non-empty, an attacker may have been able to use a related-key attack to recover the initial state of a particular keystream based on the the keystream generated by another initial state. An example of such an attack was highlighted in Section 4.2.7. These two new investigations have also allowed us to establish that it is not possible to have a state cycle within 225 bits of each other, which have two loaded states as its contents.

In Section 4.3, the combined algebraic techniques of Berbain [10] and Raddum [93] were used to analyse the Trivium stream cipher and its variants through two of our new approaches. Our analysis answers Berbain's et al.'s open question posted at the end of their paper which posed the question of whether their technique could be extended to keystream generators which updated $q > 1$ bits of internal state at each iteration and only output $q' < q$ linear combinations of the state-bits. In particular, we demonstrated two new algebraic attacks on Bivium-A. The first attack has less time and memory requirements than other techniques which use the F4 algorithm [46] to recover Bivium-A's initial state. The second attack has a higher time and memory complexity, but requires less keystream to recover the initial state of Bivium-A. When our new approachces is applied to Bivium-B and Trivium however, the time taken to search through the number of solutions the system of equations produces is worse than exhaustive key search. We also demonstrated that if $j \neq q$, it may be possible to mount a divide-and-conquer algebraic attack which can recover the initial state of the keystream generator which has less complexity compared to an exhaustive keysearch over the entire keyspace.

For Trivium-like ciphers, we show how the size of the registers used in the construction of the first system of equations, and the selection of stages used as input to the output function can affect the number of solutions obtained when the system of equations is solved. By changing this distance between register stages, the complexity of our algebraic attack can be worse than exhaustive keysearch. In other cases, even if the distance between the register stages is as small as possible, the complexity of our algebraic attack is still worse than exhaustive keysearch as the total state size of the registers whose stages are used in the construction of the system of equations is larger than the key size.

Chapter 5

State convergence in Mixer

The previous chapters of this thesis have analysed keystream generators which either used a linear state-update function and nonlinear output function to produce keystream or used nonlinear state-update function and linear output function to produce keystream. This chapter presents our analysis for a stream cipher which uses nonlinear functions for both state-update and output functions. Nonlinearity is introduced into the keystream sequence through a combination of implicit and explicit methods.

This chapter analyses a stream cipher called Mixer [79] with a particular focus on the initialisation function. A well-designed initialisation function should ensure that each distinct session key generates a distinct keystream. If this does not occur, state convergence has occured. State convergence is undesirable as it implies the effective keysize of the stream cipher is reduced, decreasing the security which can be provided by a stream cipher. If two distinct initial states of a stream cipher generates the same keystream, this further reduces the security which the stream cipher can provide. The keystream generation function of Mixer is also examined to determine if this occurs.

This chapter is organised as follows: In Section 5.1, we describe the specifications of Mixer. In Section 5.2, we define what state convergence is. In Section 5.3.1, we show why state convergence occurs in the Mixer stream cipher during its initialisation process, while in Section 5.3, we show how two distinct Mixer initial states can produce the same keystream. Section 5.4 concludes this chapter.

5.1 Mixer specifications

Mixer is a binary additive stream cipher designed by Kanso in 2008 which uses both explicit and implicit methods to provide nonlinearity. The keystream generator has two component registers: one with linear feedback and one with nonlinear feedback. The explicit nonlinearity comes from the use of a nonlinear feedback shift register, while the implicit nonlinearity is derived from the use irregular clocking (a function which controls how many times a particular shift register is clocked) of one of the component register.

The Mixer keystream generator design has much in common with two well-known stream ciphers, the LILI family of stream ciphers [29, 107] and Grain-80 [65]. Like both LILI and Grain, the Mixer design is based on two shift registers. The clock-control operation from LILI and the identical nonlinear Boolean function from Grain-80 are used in Mixer to provide nonlinearity.

Components of Mixer

The keystream generator of Mixer is based on two binary shift registers, denoted A and B, of lengths 128-bits and 89-bits, respectively. Register A is a regularly clocked LFSR with the feedback function $A(x)$. Register B is an irregularly clocked NFSR with the feedback function $B(x)$, with clocking controlled by register A. The feedback functions for the two registers are as follows.

The feedback function of A is the weight 67 primitive polynomial $A(x)$ defined as:

$$\begin{aligned} A(x) = 1 &+ x + x^6 + x^7 + x^8 + x^9 + x^{11} + x^{14} + x^{15} + x^{16} + x^{17} \\ &+ x^{18} + x^{23} + x^{25} + x^{29} + x^{30} + x^{35} + x^{36} + x^{39} + x^{40} \\ &+ x^{43} + x^{44} + x^{47} + x^{49} + x^{50} + x^{51} + x^{52} + x^{53} + x^{55} \\ &+ x^{56} + x^{57} + x^{58} + x^{60} + x^{61} + x^{65} + x^{66} + x^{67} + x^{70} \\ &+ x^{71} + x^{72} + x^{73} + x^{74} + x^{77} + x^{79} + x^{80} + x^{81} + x^{82} \\ &+ x^{87} + x^{90} + x^{94} + x^{96} + x^{102} + x^{103} + x^{104} + x^{105} \\ &+ x^{106} + x^{108} + x^{111} + x^{115} + x^{117} + x^{119} + x^{122} \\ &+ x^{123} + x^{124} + x^{125} + x^{126} + x^{128} \end{aligned}$$

where + denotes addition in GF(2).

The feedback function for NLFSR $B(x)$ is the composition of a linear feedback

polynomial $B_L(x)$ and the nonlinear function $B_{NL}(x)$. That is, $B(x)$ is defined as:

$$B(x) = B_L(x) + B_{NL}(x) \tag{5.1}$$

where $B_L(x)$ is defined as:

$$1 + x + x^{39} + x^{42} + x^{53} + x^{55} + x^{80} + x^{83} + x^{89} \tag{5.2}$$

and $B_{NL}(x)$ is:

$$B_{NL}(x) = \prod_{j=1}^{88}(1 \oplus B_j(t)) \tag{5.3}$$

Note that $B_{NL}(x)$ is equal to 1 with probability 2^{-88}. In the analysis presented in this paper, the feedback function $B(x)$ is approximated by $B_L(x)$, an approximation that holds with very high probability.

The clocking of Register B is under the control of register A. An integer function, F_{INT}, takes the contents of w selected stages of register A as input and outputs an integer c_b, which is the number of times register B is clocked. F_{INT} is defined as:

$$c_b(t) = F_{INT} = 1 + 2^0 A_{i_0}(t) + 2^1 A_{i_1}(t) + \ldots + 2^{w-1} A_{i_{w-1}}(t)$$

where $w \in \{0, 1, \ldots, 127\}$ and $i_0, i_1, \ldots, i_{w-1} \in \{0, 1, 2, \ldots, 127\}$. Note that the Mixer specification does not fix the value for w, nor does it specify the tap positions for the inputs to F_{INT}, but it does recommend that $w \in \{2, 3, \ldots 7\}$ be used for efficiency reasons.

A nonlinear Boolean function, $g(x)$, is used to determine whether the output of Register B will be used or discarded. The function $g(x)$ is defined as:

$$\begin{aligned} g(x) =& x_1 \oplus x_4 \oplus x_0x_3 \oplus x_2x_3 \oplus x_3x_4 \oplus x_0x_1x_2 \oplus x_0x_2x_3 \oplus x_0x_2x_4 \oplus \\ & x_1x_2x_4 \oplus x_2x_3x_4 \end{aligned}$$

The inputs x_0, x_1, x_2, x_3, x_4 at time t are the contents of the five stages of register A: $A_7(t)$, $A_{37}(t)$, $A_{73}(t)$, $A_{91}(t)$ and $A_{123}(t)$ respectively.

Figure 5.1 illustrates the components of Mixer and the interaction of its components during both the initialisation (includes both solid and dotted lines) and keystream generation (solid lines only) processes.

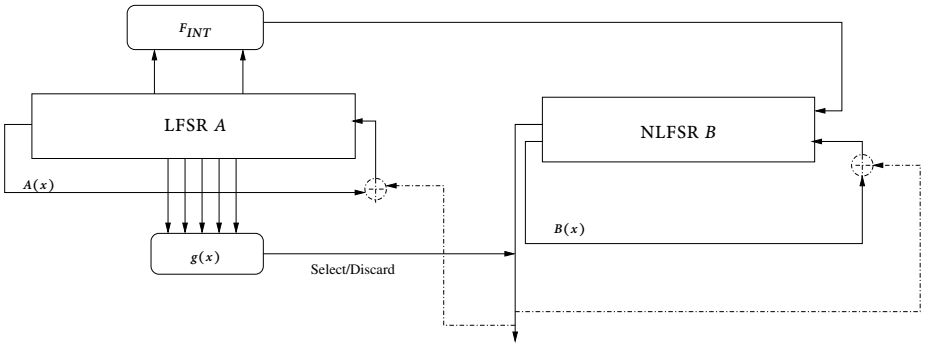


Figure 5.1: Mixer state update functions

Initialisation of Mixer

The Mixer initialisation process can be divided into three phases: the key loading phase, the IV loading phase and the diffusion phase. During the key and IV loading phases, the master secret key and IV are loaded into the shift registers.

Let the individual bits of the 128-bit key K be denoted $k_0, k_1, \ldots k_{127}$, and the individual bits of the 64-bit IV, V be denoted by $v_0, v_1, \ldots v_{63}$. The key and IV loading phases of Mixer are performed as follows. The master secret key, K, is loaded into register A such that $A_i = k_i$, for $0 \leq i \leq 127$. The IV, V, is loaded into register B such that $B_i = v_i$, for $0 \leq i \leq 63$. The remaining stages of register B are filled with ones. The Mixer internal state at the completion of the loading phase is in a *loaded state*.

To complete the initialisation process, the diffusion phase must be performed. This involves performing 200 iterations of the initialisation state update function. Consider S_t, the internal state at time t, (the contents of the two registers $A(t)$ and $B(t)$).

For each iteration the following process, as shown in Figure 5.1 (with both solid and dotted lines), is followed:

1. Clock register A once.
2. For the updated state $A(t+1)$, calculate:
   (a) the integer value $c_b(t+1)$ using F_{INT}.
   (b) the output of the nonlinear Boolean function $g(x)$.
3. Clock register B $c_b(t+1)$ times.
4. If at time $t+1$, $g(x) = 0$ then this iteration is complete.
5. If at time $t+1$, $g(x) = 1$ then XOR the output bit B_0 of register B (after $c_b(t+1)$ clocks) with the contents of both register stages A_{127} and B_{88}.

In this chapter, the XOR operation described in Step 5 is referred to as the *mixing* operation. This is the only operation in the initialisation state update function where contents of the two registers are directly combined. The output bit B_0 from register B which is XORed is referred to as *the mixing bit*, and denoted φ.

During the initialisation phase, no keystream is produced. After 200 iterations of the initialisation state update function, an *initial state* has been produced and keystream generation can commence.

Keystream generation

During keystream generation the state update function is similar to the initialisation state update function. The only difference is that the mixing operation is not used, and the output of register B is used directly as the keystream. That is, for each iteration the following process, as shown in Figure 5.1 (with solid lines only), is followed:

1. Clock register A once.
2. For the updated state $A(t+1)$, calculate:
   (a) The integer value $c_b(t+1)$ using F_{INT}.
   (b) The output of the nonlinear Boolean function $g(x)$.
3. Clock register B $c_b(t+1)$ times.
4. If at time $t+1$, $g(x) = 0$, no keystream is produced and this iteration is complete.
5. If at time $t+1$, $g(x) = 1$ then the output bit of B becomes the keystream bit z_j.

This process continues until sufficient keystream bits have been generated to encrypt or decrypt the message.

5.2 State convergence in stream ciphers

To prevent time-memory-data tradeoff attacks, modern stream ciphers have internal states which are at least the size of the key-IV pair. That is, $2^s \geq 2^{l+j}$. Since the state space is at least the size of the space spanned by all key-IV pairs, it is reasonable to expect that the initialisation process will be one-to-one, that is, each distinct key-IV pair should map to a distinct state at the end of initialisation.

Some initialisation processes are not one-to-one. Considering the state-update function in the forwards direction, for a given $S(t)$, there is a single $S(t+1)$. However, when considering the reverse direction, for a given value of $S(t+1)$, there may be multiple values for $S(t)$. That is, multiple states converge during one iteration of the initialisation state update function. If this state convergence occurs at any point during the initialisation process, the same initial state will be attained at the end of initialisation from different loaded key-IV pairs. Multiple distinct key-IV pairs will then generate the same keystream. If the state-update function used during keystream generation is not one-to-one, it is possible to have distinct initial states generate the same keystream, further reducing the security of the keystream generator.

The term entropy is sometimes used when discussing the internal state of a keystream generator. Entropy is defined by Shannon as the measure of unpredictability in information contained in a random variable [99]. More formally, assume that there is a set $X = \{x_i\}_{i \in I}$, with each element x_i appearing with probability p_i, the entropy $H(X)$ of set X is defined by Hong and Kim [69] as:

$$H(X) = -\sum_{i \in I} p_i \log_2(p_i)$$

If the set X has $l + j$ elements, we have:

$$H(X) = -\sum \frac{1}{2^{l+j}} \log_2(\frac{1}{2^{l+j}}) = \log_2(2^{l+j}).$$

In the context of stream ciphers, the entropy $H(X)$ is the size of the key and IV in bits. That is, $l + j$, where l and j are the key and IV sizes in bits respectively. If after an iteration of the state-update function, this entropy is still $l + j$, the state-update function is one-to-one, and state covnergence does not occur. If this entropy falls below $l + j$, state convergence is said to have occurred. The discussion of state entropy in the context of state convergence in keystream generators is used in the analysis of Mickey-v2 and the summation generator in Sections 6.2.2 and 6.3.3 respectively.

Where state convergence occurs in keystream generators for stream ciphers, there are three possible scenarios. They are:

Scenario 1 The same secret key used with different IVs generates the same initial state. That is, (K^1, V^1) and (K^1, V^2) produce the same keystream.

Scenario 2 The same IV used with different secret keys generates the same initial state. That is, (K^1, V^1) and (K^2, V^1) produce the same keystream.

Scenario 3 Distinct key-IV pairs generate the same initial state. That is, (K^1, V^1) and (K^2, V^2) produce the same keystream.

These scenarios are irrelevant if the attacker's goal is initial state recovery as it will not matter which key-IV pair generated any particular initial state. However, the attacker will not be able to use this initial state to decrypt the rest of the communication, as the other frames would have been encrypted using the same secret key but with a different IV. The state recovery attack must be repeated for each frame of the communication. If the attacker's goal is to decrypt the entire communication, a better alternative would be a secret key recovery attack, which can recover the actual secret key.

For Scenario 1 and Scenario 3, an attacker can, upon recovering a secret key, check if the IV observed along with the keystream is the same as that recovered in the attack. If it is, an attacker can be confident that they have recovered the correct key. If Scenario 2 state convergence occurs, an attacker needs to check if the recovered key can properly decrypt other messages before the attacker can be confident they have recovered the correct key. If they have not recovered the correct key they would need to perform the attack again for the next frame. Alternatively, if the initialisation process is easily invertible, an attacker could run the initialisation process backwards to recover all the possible keys which could have generated that particular state.

5.3 Analysis of Mixer

Mixer takes a 128-bit master key, K, and a 64-bit IV, V, as inputs to the initialisation process. The total Mixer internal state size is the sum of the lengths of registers A and B: $128+89 = 217$, so there are a total of 2^{217} possible internal states. Since the loading process is linear, there are a total of 2^{128+64} distinct loaded states. Ideally, the initialisation process would result in 2^{192} distinct session keys for keystream generation, and Mixer could produce 2^{192} distinct possible keystream sequences. That is, each key-IV pair should produce a distinct keystream. However this is *not* the case for Mixer. This analysis explores two main causes for the reduction in the state space of the Mixer keystream generator: the internal state convergence that occurs during the initialisation process, and the existence of "equivalent states" — distinct session keys which produce the same keystream. These two problems each result in a reduction of the effective keyspace of Mixer.

5.3.1 Analysis of Mixer's initialisation process

The analysis of the initialisation process begins with the observation that the state update function is not one-to-one. In this section we examine the state convergence during one iteration of the initialisation state update function, and across multiple iterations of the initialisation process.

State transition possibilities during initialisation

The Mixer initialisation state update function requires calculation of $g(x)$, as the update of A_{127} and B_{88} with the mixing value φ is conditional on the value of $g(x)$. The possibilities for the state transitions from $S(t)$ to $S(t+1)$ are:

1. $g(x) = 0$. No mixing operation occurs, regardless of the value of φ.
2. $g(x) = 1$ and $\varphi = 0$. The mixing operation occurs but the contents of A_{127} and B_{88} remain unchanged after the mixing operation. That is, the outcome is the same as when $g(x) = 0$.
3. $g(x) = 1$ and $\varphi = 1$: The mixing operation occurs, and the contents of A_{127} and B_{88} are complemented.

A is an LFSR with a primitive feedback function and $g(x)$ is a balanced nonlinear Boolean function. If A was autonomous, that is, no additional inputs other than the feedback bit calculated using the characteristic polynomial of A itself is used to update the internal state of A, then the probability that $g(x) = 1$ would be very close to 0.5. After the first iteration the feedback from B complicates this. However, assuming this probability is still very close to 0.5 and considering the four possible combinations of $g(x)$ and φ values, effective mixing occurs with a probability of 0.25.

Consider inverting the initialisation state update function. That is, given S_{t+1} we want to obtain S_t. Recall that A is a regularly clocked LFSR, which controls the clocking of B. The value of $g(x)$ at time $t+1$ is readily calculated. The possibilities for the state transitions from S_{t+1} to S_t are conditional on $g(x)$ at time $t+1$ and φ:

1. At time $t+1$, $g(x) = 0$. No mixing occurred. In this case, we use A_{t+1} to calculate $c_b(t+1)$, and clock A back once and B back $c_b(t+1)$ times.
2. At time $t+1$, $g(x) = 1$. Mixing has occurred, but the effect depends on the value of φ:

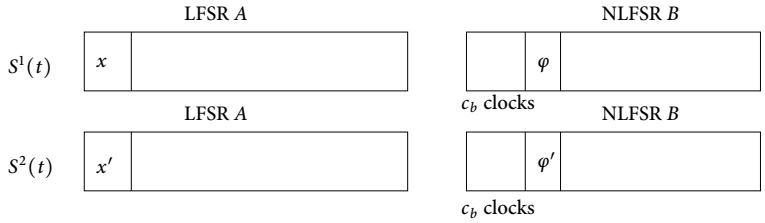


Figure 5.2: States which converge to the same next state.

(a) If $\varphi = 0$ then again use A_{t+1} to calculate $c_b(t+1)$, and clock A back once and B back $c_b(t+1)$ times.

(b) If $\varphi = 1$ then complement both $A_{127}(t+1)$ and $B_{88}(t+1)$, and then use $A(t+1)$ to calculate $c_b(t+1)$, and clock A back once and B back $c_b(t+1)$ times.

The difficulty in inverting the state update function lies with computing the value of φ. We cannot obtain this directly from $B(t+1)$ as it is discarded from B after the mixing operation. Therefore, given $g(x) = 1$ at time $t+1$, we consider two possibilities (φ equals 0 or 1). Thus there are two possible previous states. Figure 5.2 shows the format of two states at time t which converge to the same state at time $t+1$. Note that x' and φ' represent the complements of x and φ respectively. The contents of the other register stages must be the same.

Bounds on number of equivalent loaded states

For the first iteration of the diffusion phase 50% of all loaded states have $g(x) = 0$. Each of these produces a distinct next state. For the other 50%, $g(x) = 1$ and these states can be grouped into pairs that converge to the same next state. Thus, after the first iteration of the state update function, the number of distinct states is only 75% of the number of loaded states.

At the next iteration, we consider firstly those states for which $g_1(x) = 0$. Applying the argument above, after the second iteration the number of distinct states is 75% of the size of this group. For the states where $g_1(x) = 1$, the pairing argument may not hold (some of the relevant states may have been eliminated in the previous iteration) so the number of remaining states may be more than 75%.

Combining these results gives upper and lower bounds on the number of distinct states after two iterations of 62.5% and 56.25% of the number of loaded states, respectively. Continuing these arguments for α iterations gives lower bounds n_l, and upper

Table 5.1: Bounds on the predicated number of distinct initial states, n_l and n_u, after an α iteration initialisation process

[b]

α	0	25	50	100	150	200
n_l	2^{192}	$2^{181.62}$	$2^{171.25}$	$2^{150.50}$	$2^{129.74}$	$2^{109.00}$
n_u	2^{192}	$2^{191.0}$	$2^{191.0}$	2^{191}	$2^{191.0}$	$2^{191.0}$

bounds n_u, on the proportion of states remaining as

$$n_l = N \times 0.75^{\alpha} \tag{5.4}$$

and

$$n_u = \frac{N}{2}(1 + 2^{-\alpha}) \tag{5.5}$$

where N is the number of loaded states.
Equations 5.5 and 5.4 above give an upper and lower bounds on the predicted number of distinct initial states that will be obtained after an α iteration process. Table 5.1 provides the computed value for n_u and n_l for various values of α between 0 and 200.

Computer simulations and results

As an alternative approach to estimating the degree of state convergence, we ran some computer simulations for a reduced-round diffusion phase. We set $w = 2$ and took inputs to F_{INT} from A_{70} and A_{71}. In our experiments, 100 loaded states were randomly generated. For each loaded state, α iterations of the Mixer initialisation process were performed, for $\alpha = 1, 2, \ldots, 30$. We refer to the initial state resulting from this process as the *target* initial state. For each value of α and for each target obtained, the state was clocked back α times and all loaded states which generate the same target were recovered.

Data corresponding to $\alpha = 5, 10, 15, 20, 25$ and 30 have been collated to form Table 5.2. For each value of α, the table includes:

- The total number (Total) of loaded states found for all 100 target initial states.
- The minimum number (Min) and maximum number (Max) of loaded states

Table 5.2: Number of Mixer loaded states for 100 target initial states.

α	**Total**	**Min**	**Max**	**Mean**	**S.D**
5	766	1	32	7.66	6.47
10	3327	2	256	33.27	35.07
15	8120	2	1024	81.20	96.52
20	14 239	4	1152	142.39	149.07
25	20 328	4	1344	203.28	211.74
30	23 180	4	1848	231.80	242.39

found for any target.

- The mean and standard deviation (S.D.) of the number of loaded states for each target initial state.

Using the experimental results, a graph of the mean number of loaded states per target, n, against α was plotted. Two versions of this experiment were run: one in which candidate loaded states must conform to the specifications (with $B_{64}, \ldots, B_{88} = 1, \ldots, 1$) and another without this restriction. These are labelled Format check and No Format check respectively in Figure 5.3. For reference, the figure also includes the graphs of two other curves: $n = 1.25^{\alpha}$ and $n = 1.5^{\alpha}$. From Table 5.1, for 200 iterations of Mixer's initialisation process, the lower bound on the total number of distinct states is 2^{109} and the upper bound would result in 2^{191} distinct initial states, which is less than the total key-space of Mixer. Table 5.2 shows that as α increases, the number of loaded states corresponding to a target session key also increases. That is, the number of loaded states which converge to a particular session key increases with α. Similar conclusions can also be drawn from Table 5.1, where the total number of distinct session keys decrease as α increases. Note that the standard deviation values recorded in Table 5.2 increases as the value of α increases. This tells us that the rate of state convergence is not uniform across all key-IV pairs which form the loaded states.

As a way of measuring the accuracy of our theoretical estimates on the number of distinct initial states after an α initialisation process in Table 5.1 against our experimental results in Table 5.2, we combined the results from both tables. The estimated number of distinct initial states e_{α} after an α iteration initialisation process based on the mean number in Table 5.2 was calculated. The calculated results are shown in Table 5.3. Our calculations show that the experimental estimate e_{α} on the total number of distinct initial

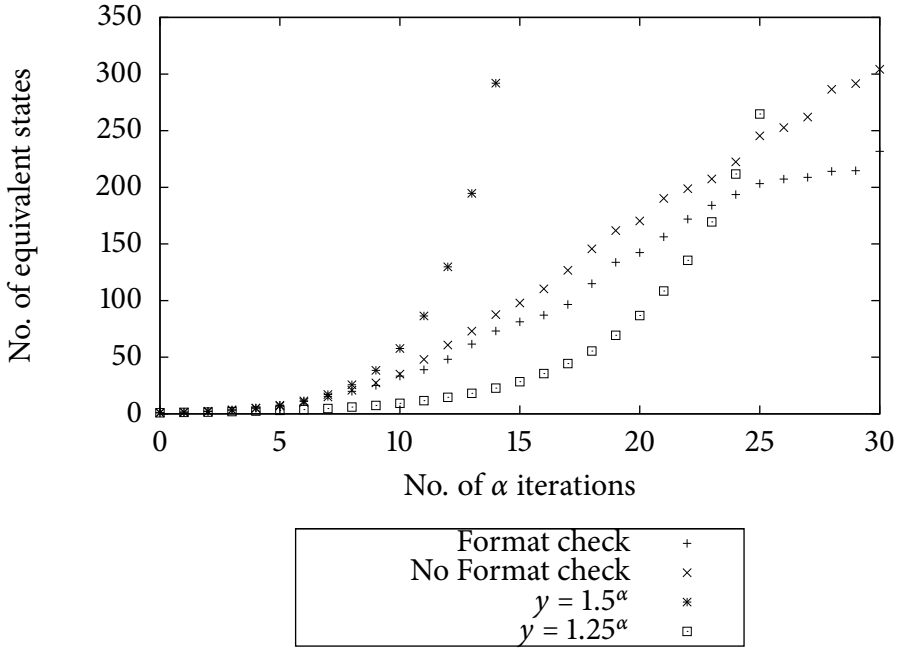


Figure 5.3: Mean number of loaded states per target for various α.

Table 5.3: Comparison of e_α against n_u and n_l

α	Theoretical estimate (n_l)	Experimental estimate (e_α)	Theoretical estimate (n_u)
5	$2^{189.92}$	$2^{189.06}$	2^{191}
10	$2^{187.84}$	$2^{186.94}$	2^{191}
15	$2^{185.77}$	$2^{185.66}$	2^{191}
20	$2^{183.70}$	$2^{184.85}$	2^{191}
25	$2^{181.62}$	$2^{184.33}$	2^{191}
30	$2^{179.55}$	$2^{184.14}$	2^{191}

states after an α iteration initialisation process is always lower than the our estimated upper bound n_u on the total number of distinct initial states. However e_α is, with the exception for α = 5–15, larger than n_l for all calculated values of α. We expect the number of distinct states which can be obtained after α = 200 iterations of Mixer's initialisation process will be between 2^{109}–2^{191}.

Limitations of experiments. Our experimental sample size of 100 trials represents a very small fraction of the 2^{192} possible loaded states. This, coupled with the non-uniform rate of convergence, may have affected the accuracy of our estimate of the number of loaded states converging to each initial state after α iterations. This may explain why our estimates for the number of distinct initial states for Mixer after an α = 5–15 initialisation process is lower than our lower bound n_l, while the other estimates of α were higher than n_l.

5.3.2 Analysis of Mixer's keystream generation process

During keystream generation, LFSR A is autonomous. That is, there is no mixing operation introducing values from register B into A. Therefore, state convergence due to the 'mixing' operation will not occur during the keystream generation process.

Assume that β is the total number of distinct initial states we can obtain after all 2^{192} possible key-IV pairs have been used in initialisation. At this point, we would expect that Mixer will produce β distinct keystreams. However, this is not the case. This is because the Mixer keystream generator employs a shrinking-generator style mechanism to determine whether the output of register B will be used as a keystream bit or discarded. Thus it will also suffer from a known weakness of shrinking generators. For the shrinking generator there exist distinct initial states, known as equivalent states, which produce the same keystream [104].

Recall from Section 5.1, if the value of $g(x)$ is 0, the output of register B is discarded and no keystream is produced. It is not until the value of $g(x)$ is 1 that the first bit of keystream is produced. Suppose we have an initial state, $S^0(0)$, with component register states $A^0(0)$ and $B^0(0)$. For this initial state $g(x) = 1$, so the first keystream bit is produced immediately. Let the keystream produced when keystream generation is commenced in this state be denoted Z. Now consider an alternative initial state, $S^1(0)$, with component register states $A^1(0)$ and $B^1(0)$. Suppose for this initial state $g(x) = 0$, so no keystream bit is produced. Further suppose that after the state update function is applied, $S^0(0) = S^1(1)$. That is, we are now in the state from which the production of Z

began. Therefore the two distinct states $S^0(0)$ and $S^1(0)$ can be considered equivalent, as both produce the same keystream sequence, Z.

Note that the binary sequence formed by successive outputs of $g(x)$ is the output of the nonlinear filter generator formed by applying $g(x)$ to the five stages of LFSR A. Statistics regarding run lengths for the binary sequences produced by LFSRs are well known, but less is known regarding the distribution of a NLFSR. The analysis of m-tuple distributions of NLFGs in Chapter 3 show that they are not uniform for small LFSR sizes. This implies that the number of initial states generated for each keystream may not uniformly distributed. However, as $g(x)$ is balanced, we expect that $P(g(x) = 0) = 0.5$ so the number of distinct keystreams is expected to be about half the number of distinct initial states, that is, $\frac{\beta}{2}$.

5.4 Summary

In this chapter we analysed Mixer, a stream cipher which uses nonlinear functions for both updating the internal state and computing the output. It is shown that, due to Mixer's initialisation function not being one-to-one, state convergence occurs. This leads to a reduction in the number of distinct initial states obtained after its initialisation phase is complete. As the number of iterations of the initialisation process increases, the number of distinct initial states which can be obtained decreases. We estimate that the total number of initial states, after 200 initialisation rounds is between 2^{109} and 2^{191}. Furthermore, we also show that due to the shrinking-generator method used to generate keystream, two distinct initial states can actually generate the same keystream. This further reduces the effective keysize of Mixer.

To increase the stream cipher's resistance to correlation and algebraic attacks, the diffusion phase needs to be run for a number of iterations. This is accompanied by a corresponding decrease in efficiency. For some applications, an appropriate tradeoff can be identified. However, where the nonlinear function is not one-to-one, as is the case for Mixer, increasing the number of iterations decreases the efficiency with no corresponding increase in security.

In Chapter 6, the state convergence problem is examined in more detail. We discuss methods which can be used to detect state convergence, and identify mechanisms which can cause state convergence. We also analyse the impact state convergence has on stream cipher cryptanalysis.

Chapter 6

State convergence and its effects on cryptanalysis

Well-designed initialisation and keystream generation processes for stream ciphers should ensure that each key-IV pair generates a distinct keystream. However, for some stream ciphers distinct key-IV pairs produce the same keystream. This phenomenon, known as state convergence, was described in Section 5.2. In Chapter 5, we showed that state convergence occurs in the stream cipher Mixer. In this chapter, we analyse the state convergence problem in more detail.

We describe the techniques used to detect state convergence in Section 6.1. We divide the analyses of state convergence in certain stream ciphers into two sections: Section 6.2 analyses irregularly clocked ciphers which experience state convergence. In particular, we show why a modified version of A5/1 which uses Step-1/2 clocking for its diffusion phase experiences state convergence, and provide counter-arguments to the claims made by Mickey-v2's designers that increasing the state size of the keystream generator's registers reduces the degree of state convergence experienced by Mickey-v2. In Section 6.3 we analyse regularly clocked ciphers which experience state convergence. We give a general example of why state convergence can occur in the F-FCSR stream cipher and the summation generator, both of which use an addition-with-carry state-update function to update the internal state of its registers.

In Section 6.4, we classify the stream ciphers which experience state convergence by mechanisms we identified which can cause state convergence and show why state convergence can occur in keystream generators using such mechanisms. In Section 6.5,

we discuss the effect state convergence has on some common cryptanalytic techniques used against bit-based stream ciphers. These include time-memory-data tradeoff attacks, correlation attacks, algebraic attacks and differential attacks.

6.1 State convergence detection

To detect state convergence in stream ciphers, we first need to thoroughly understand what happens when the state-update process is applied. We can take either of two approaches: Try to find two distinct states $S(t)$ and $S'(t)$ which both give $S(t+1)$; or given a state at time $t+1$, try to determine if there are multiple pre-image states $S^i(t)$, for $i \geq 2$, which generated $S(t+1)$. We use two main techniques to determine if state convergence occurs in stream ciphers. These are:

- State transition tables.
- Analysis of the possible combinations for clocking registers backwards.

6.1.1 State transition tables

One way to determine if state convergence exists in keystream generators is to take each of the 2^s initial states, run the state-update function once and obtain the 2^s next states and count how many distinct next state values occur. If each of the states occurs only once, we can be confident that state convergence does not occur. However, constructing a state transition table is infeasible if s is large. In this case there are two possible alternatives. These are:

1. Construct a full state-transition table for a scaled-down version of a cipher.
2. Construct a state transition table for a small subset of registers in the full version of the cipher.

State-transition table for a scaled-down version of a cipher

The first alternative is to implement a scaled-down version of the stream cipher. This is only possible for some ciphers which use mechanisms which can be easily scaled for both small and large values of s. For example, Alhamdan [3] used a scaled-down version of A5/1 to investigate the number of distinct initial states remaining after A5/1's initialisation process was completed. This scaled-down version for A5/1 had a 15-bit

internal state (LFSR lengths of 4, 5, and 6 bits) and used a similar majority-clocking scheme used in the original A5/1. Note that the majority-clocking scheme mechanism is not dependent on the size of s or function-dependent (as long as the LFSRs used in the scaled-down A5/1 use primitive feedback functions, which are also used in the original A5/1). In this case, using a scaled-down version of A5/1 provides a useful indication of the degree of state convergence experienced in the original A5/1 cipher.

State-transition table for a small subset of a stream cipher's registers

For certain ciphers like the F-FCSR stream cipher [6] described in Section 6.3.2, the contents of three stages of its registers at time $t+1$ are determined by three other registers stages at time t. Thus, the state-update function for these particular three stages can be treated as a 3×3 S-Box (relating each 3-bit input to a 3-bit output). If this S-Box is balanced, the state-update function is one-to-one, and state convergence is not present in the cipher. However, if it is not balanced, as was the case in the F-FCSR cipher, this indicates that state-convergence occurs and the state-update function should be investigated further. Later in this chapter, we use this technique to explain why state convergence occurs in both the F-FCSR stream cipher and the summation generator, in Section 6.3.2 and Section 6.3.3, respectively.

6.1.2 Analysing various combinations for clocking registers backwards

Another technique of detecting state convergence is to determine if there are multiple pre-images for a state $S(t+1)$. First, we randomly populate the state $S(t+1)$. We then attempt to reverse the state-update function to obtain the pre-image(s) $S^i(t)$, for $i \geq 1$. If there are more than one valid pre-images, then state convergence clearly occurs since all of $S^i(t)$ clock to $S(t+1)$. Conversely, if there are no valid pre-images for $S(t+1)$, it follows that at least one other state $S'(t+1)$ must have multiple valid pre-images as for any state-update function modelled as a finite state machine, a total of 2^s states must have had 2^s pre-images. In this thesis, we use the backwards clocking technique to explain why state convergence occurs in A5/1, Mixer and Mickey.

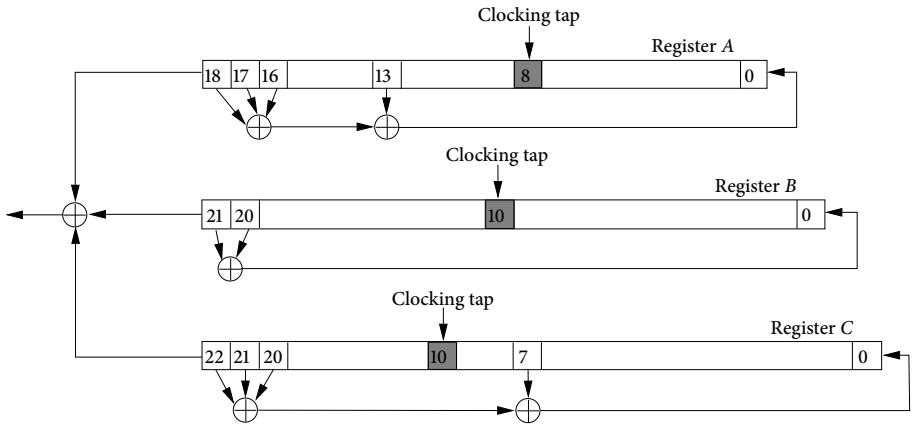


Figure 6.1: Diagram of A5/1

6.2 Irregular clocking and state convergence

A number of stream ciphers which use irregular clocking mechanisms are known to experience state convergence. These ciphers include A5/1 [22], Mickey [8], and Mixer [79]. The causes of state convergence for the stream cipher Mixer have been discussed in Chapter 5 in detail, and hence are omitted from this chapter.

6.2.1 A5/1

A5/1 is a well-known bit based stream cipher based on three binary LFSRs; denoted A, B and C; of lengths 19, 22, and 23 bits respectively. Hence A5/1 has a total state size of 64 bits. A single 64-bit secret key is used for each communication, and a 22-bit frame number is used as the IV for each frame in the communication. A diagram showing the structure of A5/1 can be found in Figure 6.1. The three registers are regularly clocked during loading of the key and IV (frame number). Following this, a majority clocking mechanism is used for the diffusion phase and also for keystream generation. This majority clocking is the only nonlinear operation performed.

To implement the majority clocking scheme, each register has a clocking tap: stages $A_8(t)$, $B_{10}(t)$ and $C_{10}(t)$. The contents of these stages determine which registers will be clocked at the next iteration: those registers for which the clock control bits agree with the majority value are clocked. For example, if $A_8(t) = 0$, $B_{10}(t) = 1$ and $C_{10}(t) = 0$, then the majority value is 0 and registers A and C are clocked. Thus, either two or three registers are clocked at each step.

Initialisation and keystream generation processes

As the total size of key and IV for A5/1 (64 + 22 = 86 bits) exceeds the 64 bit state size, compression of the key-IV space into the state space occurs during the loading phases of initialisation. In fact, as the state-update function is linear during the loading phases, it can be shown that there are 2^{22} key-IV pairs corresponding to each possible loaded state. The diffusion phase involves performing 100 iterations of the initialisation state update function using the majority clocking scheme. At the end of this phase an initial state is obtained.

Keystream is generated as a linear combination of the contents of one stage from each of the three registers: at time t, $z_t = A_{18}(t) \oplus B_{21}(t) \oplus C_{22}(t)$, where $\oplus$ denotes binary addition or xor. The majority clocking process continues until sufficient keystream, 228 bits, is generated to encrypt the frame. Then the keystream generator is re-initialised using the same secret key and the next frame number, to produce keystream for the next frame.

State convergence in A5/1

Using majority clocking as the state-update function introduces nonlinearity during the diffusion phase and during keystream generation. In fact, it is the only nonlinear operation in A5/1. However considering the possible prior states for given states of a certain format or pattern reveals an interesting phenomenon. For some patterns there are no prior states while others have one or more prior states. Clearly the state update function is not one-to-one.

Golić [51] considered the inverse mapping for the majority clocking function and identified some states with no pre-image and which therefore cannot be reached from any loaded state in a single iteration. He demonstrated that these states comprise $\frac{3}{8}$ of the loaded states of the system. Thus, the usable state space shrinks by a factor of $\frac{5}{8}$ (from 2^{64} to $5 \times 2^{61} \approx 2^{63.32}$) at the first iteration of the diffusion phase. Golić also identified some states with unique pre-images and others with up to four pre-image states. Figure 6.2 presents a graphical summary of the six cases identified by Golić. In this figure, (R_i, R_j, R_k) is any permutation of the set $\{A, B, C\}$ of a particular set of registers stages in A5/1, where the shaded stage in each register is its clocking tap. The symbol **x** represents either 0 or 1, while # represents the complement of **x**; a blank square represents a bit which can take either value. The proportion of loaded states for each case in Figure 6.2 is presented in Table 6.1, along with the corresponding number of pre-images. Note that the case identified as (i) cannot be clocked back to any valid state.

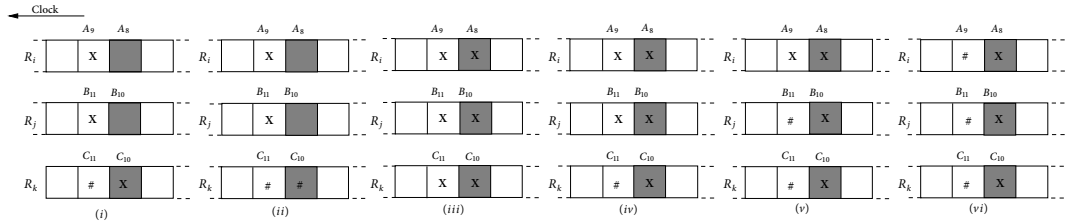


Figure 6.2: A5/1 preimage cases identified by Golić's cases [51]

Table 6.1: Proportions of states in A5/1 for Golić's cases [51]

Case	(i)	(ii)	(iii)	(iv)	(v)	(vi)
Proportion of states	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{1}{32}$	$\frac{3}{32}$	$\frac{3}{32}$	$\frac{1}{32}$
Number of pre-images	0	1	1	2	3	4

That is, states of this form cannot be reached after the first iteration of A5/1 initialisation state update function.

We now illustrate why a particular A5/1 state, which falls into case (i) shown in Figure 6.3, cannot be reached after the first iteration of A5/1 state-update function. Assume we have an A5/1 state at time $t+1$ whose register stages have the following values: $A_9(t+1) = B_{11}(t+1) = C_{10}(t+1) = 1$ and $A_8(t+1) = B_{10}(t+1) = C_{11}(t+1) = 0$. Recall that for A5/1's state-update function, register stages whose contents agree are clocked *once*. At time $t+1$, the contents of $A_9(t+1)$ and $B_{11}(t+1)$ agree, while the contents of $C_{11}(t+1)$ disagree. By applying the rules of A5/1's majority-clocking scheme, we clock A and B back once. That is, the contents of $A_9(t+1)$ and $B_{11}(t+1)$ are shifted to $A_8(t)$ and $B_{10}(t)$ respectively, while the contents of $A_8(t+1)$ and $B_{10}(t+1)$ are shifted to $A_7(t)$ and $B_9(t)$ respectively. We do not clock C back as the contents of $C_{11}(t+1)$

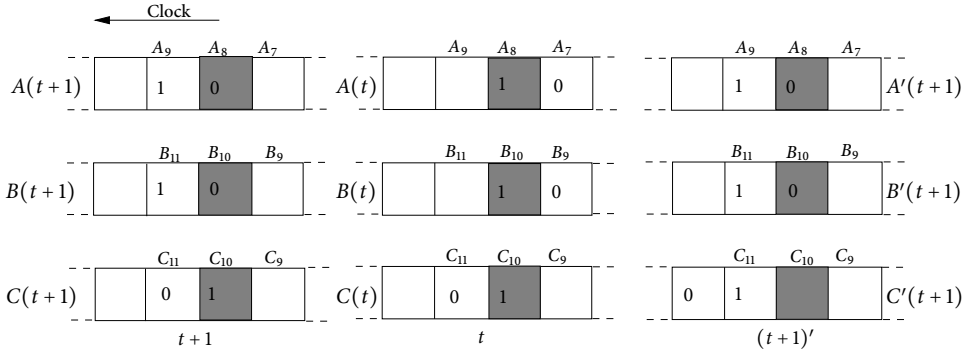


Figure 6.3: A5/1 preimage case(i) example

does not agree with $A_9(t+1)$ and $B11(t+1)$. This single application of the majority-clocking scheme gives us the A5/1 state with the register contents A, B, and C at time t. The contents of the A5/1's clocking taps at time t are $A_8(t) = B_{10}(t) = C_{10}(t) = 1$. By applying the rules of A5/1's majority-clocking scheme to this A5/1 state, we clock all three registers once, as the contents of the clocking taps all agree. giving us the state at time $(t+1)'$. However, the contents of A5/1's state at time $(t+1)'$, consisting of $A'(t+1)$, $B'(t+1)$, and $C'(t+1)$ is not the same as $A(t+1)$, $B(t+1)$, and $C(t+1)$. Therefore, the state at time $t+1$ is not a state which can be attained after one iteration of A5/1 initialisation state update function. The register stages needed for analysing the number of pre-images which can be obtained when an A5/1 state at time $t+1$ is clocked back are A_8, A_9, B_i and C_i for $i \in \{10, 11\}$. If we were to treat these stages as a 6-tuple and examine the possible ways we can clock A5/1's registers backwards for each of the $2^6 = 64$ possible values the register stages A_8, A_9, B_i and C_i for $i \in \{10, 11\}$ can attain at $t+1$, we are able to obtain Golić's six preimage cases shown in Figure 6.2.

Biryukov et al. [18] also provided convergence estimates when exploring the efficiency of their proposed attacks on A5/1. They reported that, of 10^8 randomly chosen states, only about 15% can be clocked back 100 iterations. That is, 85% of chosen states could not be obtained after 100 iterations of the majority clocking process.

We extended the existing work on state convergence in A5/1 and explored the patterns of states which can not be accessed after $2 \leq \alpha \leq 6$ iterations. We note that the rate of state convergence is not uniform at each iteration. The proportion of inaccessible states increases as α increases. Table 6.2 summarises the proportion of inaccessible states for $1 \leq \alpha \leq 6$. From these results, we estimate the number of accessible states at the end of initialisation ($\alpha = 100$) to be approximately 5% of the number of loaded states. However, an experiment by Alhamdan [3] using a scaled-down version of A5/1 yielded an estimated number of accessible states after initialisation of approximately 19.2% of the number of loaded states. The latter result is similar to previous experimental results for A5/1 reported by Biryukov et al. [18].

The above analyses of the A5/1 state-update function demonstrate that increasing the number of iterations of the state-update function during initialisation decreases the number of distinct initial states. Also if two loaded state converge during initialisation, the same keystream will be produced

Table 6.2: Proportion of available states in A5/1 after α iterations

α (number of iterations)	1	2	3	4	5	6
new proportion inaccessible	$\frac{3}{8}$	$\frac{3}{64}$	$\frac{9}{512}$	$\frac{57}{4096}$	$\frac{423}{32768}$	$\frac{6453}{524288}$
cumulative proportion inaccessible	0.375	0.422	0.439	0.453	0.466	0.479
proportion accessible	0.625	0.578	0.561	0.547	0.534	0.521
number of accessible states	$2^{63.322}$	$2^{63.209}$	$2^{63.165}$	$2^{63.129}$	$2^{63.094}$	$2^{63.061}$

New analysis on occurrence of state convergence in modified A5/1

To determine if state convergence also exists when another clock-control mechanism is employed, we analysed a modified version of A5/1 (called A5/1M). In A5/1M, majority clocking is still used to determine how the registers will be clocked. However rather than stop/go clocking, Step-1/2 clocking is now used. That is, all registers are clocked *twice* if all the clocking taps agree, otherwise, clock the registers which agree twice, and clock the remaining register once. For example, if $A_8(t) = 0$, $B_{10}(t) = 1$, and $C_{10}(t) = 0$, registers A and C are clocked *twice*, and register B is clocked *once*.

Our analysis of A5/1M reveals that it also suffers from state convergence and there exist some states which have 0–4 pre-images. In this section, Case i are the states which have no pre-image, Case ii are the states which have one pre-image, Case iii are the states which have two pre-images, and Case iv are the states which have three pre-images and Case v are the states which have four pre-images. Figures 6.4 list the states in Cases i and ii and Figure 6.5 lists the states which have two, three, and four pre-images. In these figures, (R_i, R_j, R_k) is any permutation of the set $\{A, B, C\}$ of registers and the shaded stage in each register is its clocking tap. The symbol **x** represents either a 0 or 1, while # represents the complement of **x**; a blank square represents a bit which can take either value.

We now illustrate why a particular A5/1M state, which falls into case (i) shown in Figure 6.6, cannot be reached after the first iteration of A5/1M state-update function. Assume we have an A5/1M state at time $t+1$ whose register stages have the following values: $A_9(t+1) = A_{10}(t+1) = B_{12}(t+1) = C_{11}(t+1) = 1$ and $B_{11}(t+1) = C_{12}(t+1) = 0$. Recall that for A5/1's state-update function, register stages whose contents agree are

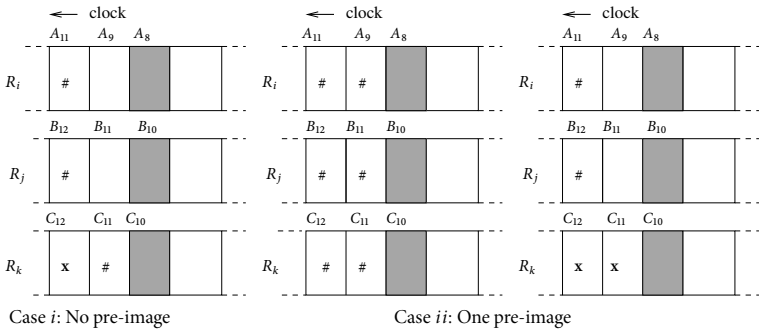


Figure 6.4: Case i and ii: A5/1M states which have no pre-image, and one pre-image respectively

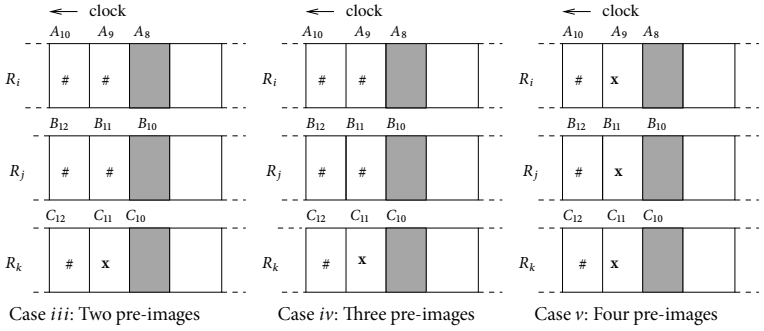


Figure 6.5: Case iii, iv, and v: A5/1M states which have two, three, and four pre-images respectively

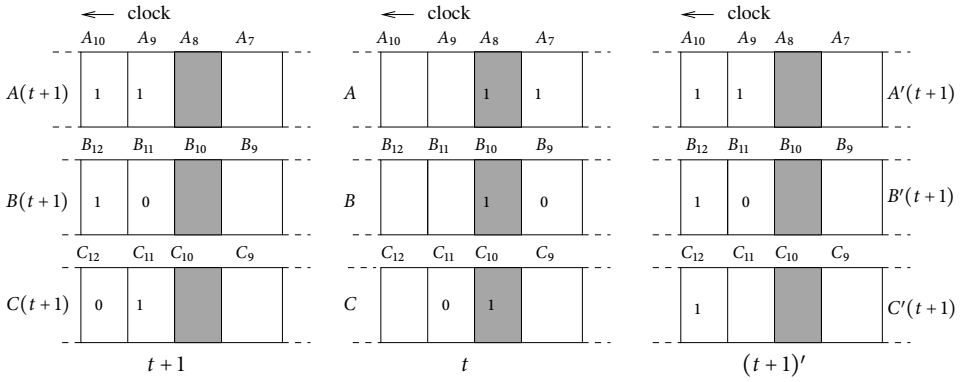


Figure 6.6: A5/1M preimage case(i) example

Table 6.3: Proportions of states in A5/1M

Case	(i)	(ii)	(iii)	(iv)	(v)
Proportion of states	$\frac{3}{8}$	$\frac{13}{32}$	$\frac{3}{32}$	$\frac{3}{32}$	$\frac{1}{32}$
Number of pre-images	0	1	2	3	4

clocked *twice* and the remaining stage is clocked *once*. This means that the contents of each register is shifted at least once at each clock. At time $t+1$, the contents of $A_{10}(t+1)$ and $B_{12}(t+1)$ agree, while the contents of $C_{12}(t+1)$ disagree. By applying the rules of A5/1M's majority-clocking scheme, we clock A and B back twice, and clock C back once. That is, the contents of $A_{10}(t+1)$ and $A_9(t+1)$ are shifted to $A_8(t)$ and $A_7(t)$ respectively, while the contents of $B_{12}(t+1)$ and $B_{11}(t+1)$ are shifted to $B_{10}(t)$ and $B_9(t)$ respectively. The contents of $C_{12}(t+1)$ and $C_{11}(t+1)$ are shifted to $C_{11}(t)$ and $C_{10}(t)$ respectively. This single application of the majority-clocking scheme gives us the A5/1M state with the register contents A, B, and C at time t. The contents of the A5/1M's clocking taps at time t are $A_8(t) = B_{10}(t) = C_{10}(t) = 1$. By applying the rules of A5/1M's majority-clocking scheme to this A5/1M state, we clock all three registers twice, as the contents of the clocking taps all agree. giving us the state at time $(t+1)'$. However, the contents of A5/1M's state at time $(t+1)'$, consisting of $A'(t+1)$, $B'(t+1)$, and $C'(t+1)$ is not the same as $A(t+1)$, $B(t+1)$, and $C(t+1)$. Therefore, the state at time $t+1$ is not a state which can be attained after one iteration of A5/1*M* initialisation state update function. The register stages needed for analysing the number of pre-images which can be obtained when an A5/1M state at time $t+1$ is clocked back are A_9, A_{10}, B_i and C_i for $i \in \{11, 12\}$. If we were to treat these stages as a 6-tuple and examine the possible ways we can clock A5/1's registers backwards for each of the $2^6 = 64$ possible values for which the register stages A_9, A_{10}, B_i and C_i for $i \in \{11, 12\}$ can attain at $t+1$, we are able to obtain the five preimage cases shown in Figures 6.4 and 6.5.

The proportion of states in each case, along with the relevant number of pre-images can be found in Table 6.3. Note that the number of pre-images for each case in A5/1M is exactly the same as Golić's analysis. That is, for case i, the proportion of states is $\frac{3}{8}$. Similarly, $\frac{3}{8} + \frac{1}{32}$ of states in Golić's analysis have one pre-image; which is the same as $\frac{13}{32}$ in A5/1M's case (ii). Case iii, iv, and v have the same proportion of states in both A5/1 and A5/1M; $\frac{3}{32}$, $\frac{3}{32}$, and $\frac{1}{32}$ states respectively.

From our new analysis on A5/1M, we can see that even with a different clocking

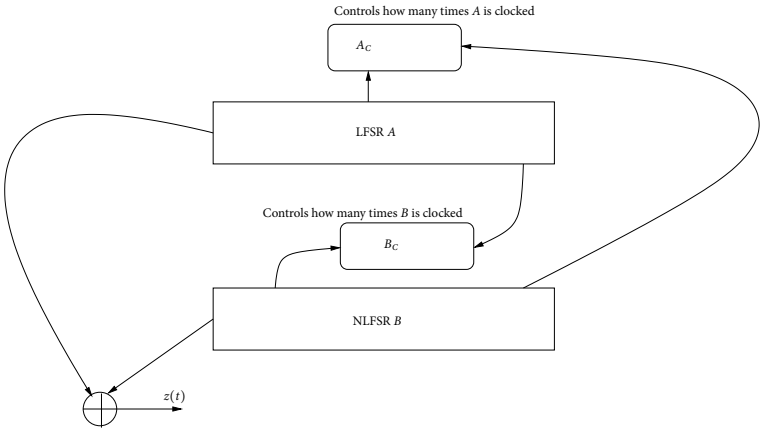


Figure 6.7: General diagram for the Mickey stream cipher

mechanism, A5/1M still suffers from state convergence. Since both A5/1 and A5/1M use majority clocking to determine the number of times a particular register gets clocked, state convergence in both stream ciphers could be attributed to the use of a majority clocking technique.

6.2.2 Mickey

Mickey-v1 [8] is a bit-based stream cipher designed by Babbage and Dodd in 2005 and was submitted to the eSTREAM project [45]. After state convergence was reported by Hong and Kim [69] in Mickey-v1, an updated version, referred to as Mickey-v2 [9], was submitted in 2006 and is one of the stream ciphers in the final eSTREAM portfolio.

Keystream generation process for Mickey-v1

Mickey-v1 has two 80-bit shift registers: LFSR A and NLFSR B, giving a total state size of 160 bits. A diagram showing the components and their interactions is shown in Figure 6.7. An 80-bit key and an 80-bit IV are used to initialise the internal state.

A keystream bit is generated as a linear combination of the contents of one stage from each register: $z(t) = A_0(t) \oplus B_0(t)$. This is done *before* Mickey's internal state is updated. The keystream generation state-update function of Mickey-v1 is similar to A5/1. Each register has clocking taps which are used in the calculation of a control bit which determines how a particular register is clocked. The control bit for LFSR A, denoted A_c, is calculated as $B_{27} \oplus A_{53}$. If $A_c = 0$, A is clocked once. If $A_c = 1$, the state is updated using an operation proposed by Jansen [74], which is equivalent to clocking A $2^{40} - 23$ times. Since A is an LFSR, once the control bit for A is known, it is possible to

calculate $A(t)$ given $A(t+1)$.

Similarly, the control bit for NLFSR B, denoted B_c, is calculated as $B_{53}(t) \oplus A_{27}(t)$. The choice of nonlinear function used to update register B depends on the value of B_c. The designers of Mickey note that the feedback function of B is invertible in both cases. That is, once the control bit of B is known, it is possible to calculate the internal state $B(t)$ given $B(t+1)$. They also impose a restriction on the amount of keystream which can be generated using a single key-IV pair: a maximum of 2^{40} bits should be generated before rekeying.

State convergence in Mickey-v1

State convergence in Mickey-v1 during keystream generation was reported by Hong and Kim [69]. They claim that this state convergence is due to the fact that the state-update function of Mickey-v1 uses two control bits, which consequently affect the very bits used to obtain the control bits, and claim that this self-dependent operation usually produces state convergence in the keystream generator. In their paper, they describe the algorithm they applied to determine how many pre-images a random state has. This algorithm is as follows:

1. Choose random states for both A and B.

2. Calculate the reverse clocking of register A, assuming the control bit is set to 0, and call this state A'. Calculate what the state of A would have been assuming that the control bit was 1, and call this state A''. Do the same for register B, and call the clocked-back states B' and B'', where B' is the state B clocks back to when the control bit is 0, and B'' is the state B clocks back to when the control bit is 1.

3. For each of the four possible (A, B) pairs after A and B are clocked back, calculate the two control bits, check to see if these match with the control bits actually used and count the number of matches.

For each of 2^{20} randomly chosen initial states, Hong and Kim ran the algorithm described above. They found that some states had none, one, two, or four pre-images. None of the 2^{20} randomly chosen inital states had three pre-images. The total number of pre-images Hong and Kim found for the 2^{20} randomly chosen states can be found in Table 6.4.

Table 6.4: 2^{20} randomly chosen states, and the number of pre-images which produce them

No. of pre-images	0	1	2	3	4
No. of states	307 988	452 017	279 418	0	9153

Causes of state convergence in Mickey v1

Assume that the state of Mickey is $(A(t+1), B(t+1))$. It is noted by Hong and Kim [69] that if the values of the controls of $A_c(t)$ and $B_c(t)$ were known, it would be possible to calculate a unique pre-image for $(A(t+1), B(t+1))$. However, to calculate $A_c(t)$ and $B_c(t)$, we need to know the original state $(A(t), B(t))$ which generated $(A(t+1), B(t+1))$. However, given the state $(A(t+1), B(t+1))$, it is not possible to calculate $A_c(t)$ and $B_c(t)$ as the state has already been updated after the values of $A_c(t)$ and $B_c(t)$ were calculated. Consequently, we have to assume each possible combination of $\{A_c(t), B_c(t)\}$ integer function outputs was possible. Using each possible combination $\{A_C(t), B_c(t)\}$ outputs, we clock back registers A and B and check if the integer value output of $A_c(t)$ and $B_c(t)$ at time t match the number of times we clocked back registers A and B. If there is more than one match, there is more than one pre-image for that particular $(A(t+1), B(t+1))$ state.

Although their discussion focuses on the keystream generation phase, the same approach can be used to demonstrate convergence during the 80-step diffusion phase between key-IV loading and keystream generation. Hong and Kim present their results in terms of entropy loss during the keystream generation process. This approach takes into account the (estimated) distribution of reachable states as well as their number. They estimate that about 39 bits of entropy are lost during the 2^{40} iterations of keystream generation. They also display some graphical results for an estimate based only on the number of reachable states; these results suggest that the number of distinct internal states will reduce from 2^{160} to about 2^{122} during these 2^{40} iterations.

Mickey-v2

In response to the attacks by Hong and Kim, the designers of Mickey-v1 released a new version of Mickey, referred to as Mickey-v2 [9] in 2006. The register lengths of A and B were increased to 100 bits each, and the tap positions of the clocking bits were adjusted accordingly. The actual mutual clocking mechanism remains unchanged. As the general

structure and components are similar to Mickey-v1, the reader is referred to Figure 6.7 for a diagram of Mickey-v2.

The designers argue that while state convergence is still possible, the expected entropy will drop from 200 bits to about 160 bits after 2^{40} keystream bits have been generated, and since this is "twice the key size, ... we no longer have a problem" [9].

This argument is clearly not correct. The initial entropy can not be larger than 160 bits of loaded key and IV data. Although the registers are now longer, as the mechanism which causes convergence is unchanged the number of distinct states will still decrease with additional iterations of the register update functions. In fact, the effectiveness of their strategy requires the assumption that the initialisation phase distributes the 2^{160} possible loaded states randomly within the state space of size 2^{200}. This assumption is equivalent (under the arguments of Hong and Kim) to having started with a full state entropy of 200 bits and then having undertaken approximately 2^{41} state update iterations. This then implies that the further 2^{40} iterations (for keystream generation) will result in an additional entropy loss of only approximately 0.6 bits ($\log_2(\frac{2^{41}+2^{40}}{2^{41}})$). While this is not a large enough reduction to give rise to a serious attack, it does show that state convergence needs to be considered carefully and allowed for appropriately when designing ciphers. It is also clear that further entropy loss (state convergence) would result if a user of this cipher contravened the design restrictions and generated keystream in excess of the 2^{40} bits permitted.

6.3 Regular clocking and state convergence

In the previous section, state convergence for the ciphers examined was associated with irregular clocking. However, the use of regular clocking is not sufficient to avoid state convergence. In this section we analyse some regularly clocked, bit-based stream ciphers for which state convergence occurs. These include the Sfinks [21] stream cipher, F-FCSR [6], and the summation generator [95].

6.3.1 Sfinks stream cipher

The Sfinks stream cipher [21] was designed by Braeken et al. in 2005 and was submitted to the eSTREAM project. It is a bit based and hardware oriented stream cipher that uses an 80-bit secret key and 80-bit IV as inputs and has a 256-bit internal state.

This cipher is based on a regularly clocked 256-bit LFSR and a nonlinear one-to-one inversion function *INV*. The *INV* function may considered as a 16×16 bit S-box, with

input bits $(x_{15}, \ldots, x_0)$ and output bits $(y_{15}, \ldots, y_0)$. The interactions between these components differ between the initialisation and keystream generation processes.

Initialisation process

During initialisation, as shown in Figure 6.8, the 16 output bits from the *INV* function are delayed and combined with the shift register contents so that the shift register is updated according to the following update functions:

$$R_i(t) = \begin{cases} R_{i+1}(t-1) & \text{for } i \in \{0, 1, \ldots, 254\} \text{ except } \{11, 17, 41, 52, \\ & 66, 80, 111, 118, 142, 154, 173, 179, 204, \\ & 213, 232, 247\} \\ R_{i+1}(t-1) \oplus y_{(i \bmod 16)}(t-7) & \text{for } i \in \{11, 17, 41, 52, 66, 80, 111, 118, 142, \\ & 154, 173, 179, 204, 213, 232, 247\} \\ \bigoplus_j R_j(t-1) & \text{for } i \in \{255\} \text{ and } j \in \{212, 194, 192, 187, 163, \\ & 151, 125, 115, 107, 85, 66, 64, 52, 48, 14, 0\} \end{cases}$$

State convergence in Sfinks

Alhamdan et al. [2] noted that although the individual components of the Sfinks state-update function during initialisation are one-to-one, state convergence still occurs because the *combination of these components* is not one-to-one. Combining the bits from the LFSR and the S-box output, together with the choice of input taps to the S-box, gives rise to state convergence during initialisation. In particular, the spacing of seven time steps between register stages R_{161} (which is an S-box input bit) and R_{154} (which is affected directly by an S-box output bit) makes it possible for two states which differ in R_{161} at time $t-7$ to converge to the same state at time t. The requirement for this to occur is that a change in the content of the S-box input bit R_{161} $(= x_{11})$ with all other input bits held constant must result in a change in the S-box output bit y_{10}. In this case, $R_{154}(t) = R_{155}(t-1) \oplus y_{10}(t-7) = R_{161}(t-7) \oplus y_{10}(t-7) = \bar{R}_{161}(t-7) \oplus \bar{y}_{10}(t-7)$ where $\bar{R}$ and $\bar{y}$ represent the complements of R and y respectively.

Alhamdan et al. [2] considered all possible pairs of S-box input bits that differ only in bit x_{11} and determined the number of corresponding pairs of S-box output that differ in bit y_{10}. They found 273 such pairs out of a possible 2^{15}. Hence the probability of convergence at each iteration is estimated as $\frac{273}{2^{15}} = 2^{-6.9}$. Based on this result, the estimated number of distinct states drops from 2^{160} after key-IV loading to approximately $2^{158.55}$ distinct states by the end of the initialisation process.

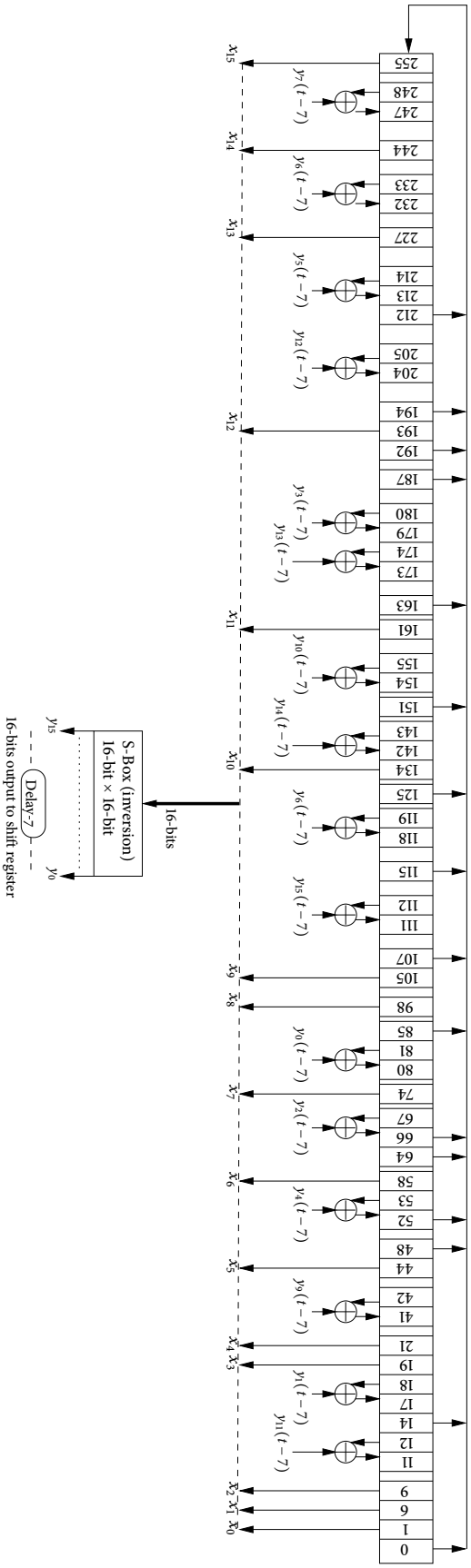


Figure 6.8: Initialisation processes of Sfinks stream cipher, reproduced from the diagram in Alhamdan et al. [2]

6.3.2 F-FCSR

The F-FSCR [6] bit-based stream cipher was designed by Arnault and Berger in 2005. It uses a 128 bit key and a 64 bit IV, or a 96 bit key and a 64 bit IV to generate keystream, depending on usage needs.

Description of a FCSR

The F-FCSR stream cipher is based on the operation of a Feedback with Carry Shift Register (FCSR). FCSRs consist of two registers: a main register A and a carry register B. The operation of a FCSR is based on the theory of 2-adic fractions of the form $\frac{p}{q}$, where q is a negative odd integer and $0 \leq p \leq |q|$. The period of the binary sequence generated from the 2-adic fraction is the order of 2 modulo q. If the prime number q is such that the order is $|q|-1$, then the period of any initial state (represented by the value p) would also be $|q|-1$.

The main register A consists of a binary stages, where a is the bit length of $d = \frac{|q|-1}{2}$. The carry register B has b binary stages, where $b+1$ is the Hamming weight of d. Given the binary expansion of $d = \sum_{i=0}^{a-1} d_i \cdot 2^i$, we set $I_d = \{i | 0 \leq i \leq a-2 \text{ and } d_i = 1\}$ and $b = \#I_d$.

If the FCSR internal state at time t is $(A(t), B(t))$, the state-update function for any FCSR will update the state at $t+1$ to $(A(t+1), B(t+1))$ using the following state-update function:
For $0 \leq i \leq a-2$ and $i \notin I_d$,

- $A_i(t+1) = A_{i+1}(t)$

For $0 \leq i \leq a-2$ and $i \in I_d$,

- $A_i(t+1) = A_{i+1}(t) \oplus B_i(t) \oplus A_0(t)$
- $B_i(t+1) = A_{i+1}(t)B_i(t) \oplus B_i(t)A_0(t) \oplus A_0(t)A_{i+1}(t))$

For $i = a-1$,

- $A_{a-1}(t+1) = A_0(t)$

For example, if an FCSR has $q = -347$, then $d = 174$, $a = 8$ and $b = 4$, Figure 6.9 illustrates the FCSR which would have been constructed.

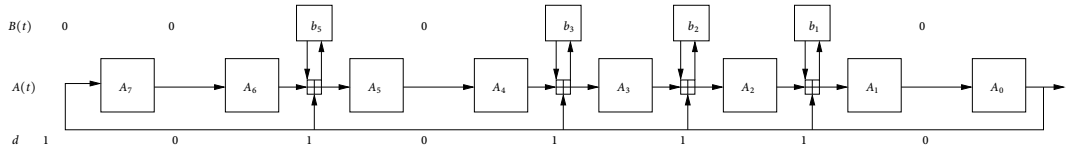


Figure 6.9: Example of an FCSR when $q = -347$, $d = 174$, $a = 8$, and $b = 4$

Components of F-FCSR

For the F-FCSR stream cipher proposed by Arnault and Berger,

$$-q = 49387740064344360888838204820078394382 7$$

The binary expansion of $d = (|q| + 1)/2$ is:

```
10111001 11000110 10101001 11101010
10110111 11100010 01011111 11010110
10011110 10000110 00110110 10011010
00011000 01010110 11101100 01001010.
```

Register A has $a = 128$ stages. The Hamming weight of d is 69 and there are $b = 69 - 1$ stages in register B.

Initialisation of F-FCSR

During the loading phase, the key is loaded directly into A, while the IV is loaded directly into B. Note that the length of $b = 68$, while F-FCSR uses a 64 bit IV. The F-FCSR paper does not specify how a user of F-FCSR populates the remaining four stages of B. We assume that the remaining four stages of B, that is B_i, for $64 \leq i \leq 67$, are set to zero. During the diffusion phase, the state-update function for the FCSR is applied six times. Upon completion of the diffusion phase, F-FCSR is ready to begin keystream generation.

Keystream generation

The propagation of carries provides implicit nonlinearity for the FCSR. Similar to LFSRs however, using the FCSR output directly as keystream is insecure. Thus, the designers of F-FCSR propose using a linear function which takes as inputs the contents of Register A, to use as keystream output. There are four proposals for F-FCSR, namely: F-FCSR-SF1, F-FCSR-SF8, F-FCSR-DF1, F-FCSR-DF8. The linear filters used for F-FCSR-SF1

and F-FCSR-SF8 are fixed and known. F-FCSR-SF1 outputs one keystream bit at each iteration, while F-FCSR-SF8, through the use of eight independent linear filters, outputs eight bits of keystream at each iteration. The linear filters used for F-FCSR-DF1 and F-FCSR-DF8 are key-dependent and are derived from the secret key and an invertible nonlinear function g. The designers of F-FCSR recommend using the AES S-box for g. Since the AES S-box operates on bytes, we first divide K into 16 byte blocks K_i, for $i = 1, 2, \ldots, 16$ and compute the output of $g(K_i)$. For the design of F-FCSR-DF8, the eight sub-filters constructed using $g(K)$ are used to output eight bits of keystream each time the F-FCSR state is updated.

State convergence in F-FCSR

The state-convergence problem in F-FCSR was first pointed out by Jaulmes and Muller [77]. In their paper, Jaulmes and Muller use the small FCSR described in Figure 6.9 to explain why a FCSR state-update function is not invertible. The example they gave in their paper only looked at some particular stages and showed how state convergence can occur. In the example below, we extend their work and give a general example of why a FCSR state-update function is not invertible.

Let $A_i(t)$, $B_i(t)$ and $A_i(t+1)$, $B_i(t+1)$ denote the contents of the i'th stage of the main register A and carry register B at time t and $t+1$ respectively. If $A_{a-1}(t+1) = 0$, $A_i(t+1) = B_i(t+1) = 1$, and $i \in I_d$, two incompatible constraints for $A_0(t)$ at time t must be satisfied to obtain it. Since $A_{a-1}(t+1) = 0$, the state-update function requires that $A_0(t) = 0$. However, if $A_i(t+1) = B_i(t+1) = 1$, we must also have $A_0(t) = 1$. Since $A_0(t))$ cannot have the value 1 and 0 at time t, the state when $A_{a-1}(t+1) = 0$, $A_i(t+1) = B_i(t+1) = 1$, and $i \in I_d$ is not invertible. Note that when $i \notin I_d$, the carry register B is not updated and the state-update function for the corresponding stages A_i of the FCSR is linear. Thus, given the state $A_i(t+1)$, it is easy to calculate the pre-image state $A_i(t)$.

The state-transition table for all possible combination values of $A_0(t)$, $A_{i+1}(t)$, and $B_i(t)$ at time t, and the values of $A_{a-1}(t+1)$, $A_i(t+1)$, and $B_i(t+1)$ at time $t+1$ if $i \in I_d$, are shown in Table 6.5, while Table 6.6 lists the number of possible three-tuple states for the states $A_0(t)$, $A_{i+1}(t)$, and $B_i(t)$ which lead to each possible $A_{a-1}(t+1)$, $A_i(t+1)$, and $B_i(t+1)$ combination.

Table 6.5 and 6.6 show that the internal state values of $A_{a-1}(t+1) = 1$, $A_i(t+1) = B_i(t+1) = 0$, and $A_{a-1}(t+1) = 0$, $A_i(t+1) = B_i(t+1) = 1$, are states which cannot be reached from any combination of the states $A_0(t), A_{i+1}(t), B_i(t)$. In contrast, the state

Table 6.5: State-transition table for certain stages in a FCSR when $i \in I_d$

$A_0(t)$	$A_{i+1}(t)$	$B_i(t)$	$A_{a-1}(t+1)$	$A_i(t+1)$	$B_i(t+1)$
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	0	1
1	0	0	1	1	0
1	0	1	1	0	1
1	1	0	1	0	1
1	1	1	1	1	1

Table 6.6: Three-tuple distribution for $A_i(t+1)$, $A_{a-1}(t+1)$, and $B_i(t+1)$ when $i \in I_d$

$A_{a-1}(t+1)$	$A_i(t+1)$	$B_i(t+1)$	Count
0	0	0	1
0	0	1	1
1	0	0	0
1	0	1	2
0	1	0	2
0	1	1	0
1	1	0	1
1	1	1	1

values for $A_i(t+1) = 0$, $A_{a-1}(t+1) = B_i(t+1) = 1$ can be generated by two different sets of internal states: $A_0(t) = B_i(t) = 1$, $A_{i+1}(t) = 0$ and $A_0(t) = A_{i+1}(t) = 1$, $B_i(t) = 0$, while the state values for $A_i(t+1) = 1$, $A_{a-1}(t+1) = B_i(t+1) = 0$ can be generated by two different sets of internal states: $A_0(t) = B_i(t) = 0$, $A_{i+1}(t) = 1$ and $A_0(t) = A_{i+1}(t) = 0$, $B_i(t) = 1$. For some states, there is a one-to-one mapping at time t to $t+1$, and vice-versa when $i \in I_d$. These states are:

- $A_i(t+1) = A_{a-1}(t+1) = B_i(t+1) = 0$
- $A_i(t+1) = A_{a-1}(t+1) = 0$, $B_i(t+1) = 1$
- $A_i(t+1) = A_{a-1}(t+1) = 1$, $B_i(t+1) = 0$
- $A_i(t+1) = A_{a-1}(t+1) = B_i(t+1) = 1$

Note that during any one-to-one mapping, $A_i(t+1) = A_{a-1}(t+1)$ and holds with probability one. If a FCSR state at $t+1$, has these values, only one previous state would have generated that particular state.

Jaulmes and Muller [77] estimate that any two related states in F-FCSR will converge to the same internal state with very high probability, after 128 iterations of the F-FCSR state-update function. As a result, they also claim the same probability that after 128 iterations of the F-FCSR state-update function from any given initial state, the internal state will be in a unique cycle and state convergence will no longer occur. Thus, if the first 128 bits of output from the F-FCSR are discarded, there is a very large probability that all internal states encountered from that point on will be part of a unique cycle. Consequently, the real entropy of the internal state of F-FCSR is of $\log_2(|q|-1) \approx 128$ bits. If both registers of an FCSR are randomly initialised, then the initial entropy is $a+b$ bits. In this case, Röck [94] shows that after one iteration, the entropy of the FCSR is

$$a + \frac{b}{2} \tag{6.1}$$

She also claims if all 2^{a+b} initial states are equally likely, a FCSR will have at least $a + 2^{b-a}\frac{7}{12\ln(2)}$ bits of entropy. Thus, for any given FCSR, its entropy will never go below a. In the case of F-FCSR, its entropy will always be at least $128 + 2^{63-128}\frac{7}{12\ln(2)} \approx 128.000$ bits.

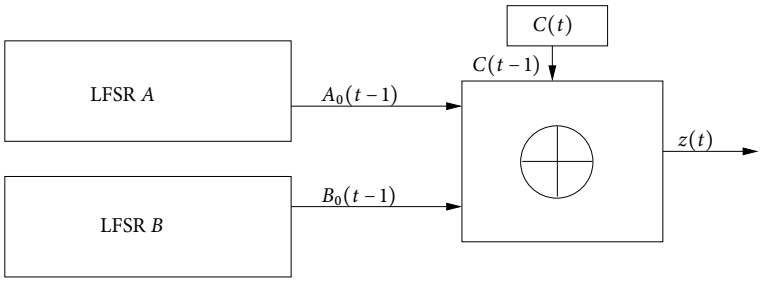


Figure 6.10: Summation generator diagram

6.3.3 Summation generator

The summation generator [95] is a bit-based stream cipher proposed by Ruppel in 1985. The original summation generator proposed by Ruppel consisted of two binary symmetric source output A and B of sizes a bits and b bits respectively, and a single memory bit C. This gives the summation generator a state space S of $s = a + b + 1$ bits. The keystream output from a summation generator is formed by combining the output of A and B and the previous contents of C. Therefore, a summation generator can be viewed as a nonlinear combiner with a single memory bit. In this section, the two binary symmetric source outputs are assumed to be two LFSRs A and B, whose state-update functions are implemented using primitive feedback polynomials. The output of two LFSRs A and B of lengths a and b, at time t is combined with a memory bit $C(t-1)$ to form a keystream bit $z(t)$, and the memory bit $C(t)$ is calculated using the following functions respectively:

$$z(t) = A_0(t-1) \oplus B_0(t-1) \oplus C(t-1) \tag{6.2}$$

$$C(t) = A_0(t-1)B_0(t-1) \oplus (A_0(t-1) \oplus B_0(t-1))C(t-1) \tag{6.3}$$

A diagram of the summation generator can be found in Figure 6.10. Note the similarity between these equations and those for the FCSR obtained in the previous section. In the next section, we show why state convergence occurs in the summation generator and estimate the loss of state entropy in the summation generator arising from this state convergence.

State convergence in the summation generator

If the state-update functions of A and B are implemented using primitive feedback polynomials, there is a one-to-one mapping from a particular state of either shift register at time t to another state at time $t + 1$. Furthermore, the memory bit C is not used to

Table 6.7: State transition table for $C(t)$ and keystream generation output

$A_0(t-1)$	$B_0(t-1)$	$C(t-1)$	$C(t)$	$z(t)$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

update A or B. Thus, no state convergence occurs in either register A and B.

However, when we consider the entire state of the keystream generator, we see that it is possible for combinations of $A_0(t-1)$, $B_0(t-1)$ and $C_0(t-1)$ to generate the same $C(t)$ value. Table 6.7 shows how the keystream bit $z(t)$, and the memory bit $C(t)$ are calculated based on the values of $A_0(t-1)$, $B_0(t-1)$, and $C(t-1)$. From Table 6.7, we can observe that when $A_0(t-1) = B_0(t-1)$, the calculation of the value $C(t)$ is not affected by $C(t-1)$. Consider two distinct summation generators with states at time $t-1$ comprising $A(t-1) = A'(t)$, $B(t) = B'(t)$ and $C(t) \neq C'(t)$. For example, let $A_0(t-1) = A'_0(t-1) = B'_0(t-1) = B'_0(t-1) = 0$, and $C(t-1) = 1$, and $C'(t-1) = 0$. At time t, $A(t) = A'(t)$ and $B(t) = B'(t)$ have the same values, as the state-update functions for A and B are autonomous and are not affected by the value of $C(t-1)$. However, note the value of $C(t)$ is now 0. Hence, two distinct states $A(t-1), B(t-1), C(t-1)$, and $A'(t-1), B'(t-1), C'(t-1)$ have now converged to the same state, and will generate the same keystream from this point on. Likewise, the states

- $A'_0(t-1) = B'_0(t-1) = 1$, $C(t-1) = 0$ and
- $A'_0(t-1) = B'_0(t-1) = 1$, $C(t-1) = 1$

will also converge to the same state at the next clock step, and will produce identical keystream from that point on. If the initial state of a summation generator was initialised randomly, $\frac{1}{2}$ of the states can be paired up, and each pair will generate the same state at time $t+1$. At time $t+2$, $\frac{1}{4}$ of the initial states can be paired up, and each pair will generate the same state at time $t+3$. Therefore, after two iterations of the summation generator's state-update function, $\frac{1}{2} + (\frac{1}{2} \times \frac{1}{2}) = \frac{3}{4}$ of the states would have experienced

Table 6.8: Causes of state convergence summary table

Type of mechanism	Stream ciphers
Mutual clock-control	• A5/1 (and A5/1M) • Mickey-v1/v2
Self-update	• Mixer • Sfinks
Addition-with-carry state-update	• F-FCSR • Summation generator

state convergence. After α iterations, the summation generator would have lost $(\sum_{i=0}^{\alpha} \frac{1}{2^{\alpha}})$ bits of entropy, approximately 1 bit for suitably large α. As mentioned above, if A and B are autonomous LFSRs, state convergence will not occur in either register. In this section, it is shown that state convergence can occur in the carry register C. Thus, the summation generator will at most lose one bit of entropy, resulting from the state convergence caused by C. As a result, the effective state space of the summation generator is $a + b$ bits.

6.4 Mechanisms which can cause state convergence

So far in this chapter we have studied stream ciphers which have state convergence problems. Based on this study, we have identified three mechanisms which cause state convergence. These are: .

- Mutual clock-control,
- Self-update mechanisms,
- Addition-with-carry mechanisms.

The classification of the studied stream ciphers into the three categories is shown in Table 6.8.

All the mechanisms shown in Table 6.8 are nonlinear operations. However, the types of registers used in the stream ciphers can either be linear and/or nonlinear. The stream ciphers which update the internal state in a linear way do so via the XOR operation. For A5/1 and Mixer, this is accomplished through the feedback polynomial of the LFSR. Note that although Mixer uses a nonlinear function to update the internal state of one of its registers, it can be easily approximated with a linear function. Details of this approximation can be found in Section 5.1. For Sfinks, the update of the internal state is also accomplished through the LFSR's feedback polynomial, along with the direct XOR of S-Box output bits with specified bits of the LFSR. The remaining ciphers, the summation generator, the F-FCSR and Mickey, consist of ciphers which use a combination of linear and nonlinear functions to update the internal state of the keystream generator. In Section 6.4.1–6.4.3, we discuss why these mechanisms can cause state convergence in keystream generators.

6.4.1 Mutual clock-control

Mutual clock-control mechanisms are the combination of two different mechanisms: mutual update, and clock-control mechanisms. In this section, we discuss both mechanisms and discuss the situations where state convergence may or may not occur in keystream generators which use clock-control mechanisms in their keystream generators.

Mutual-update mechanisms

Mutual-update mechanisms are mechanisms where each register is updated using input from the stages of another register. That is, none of the registers are autonomous. A mutual update mechanism is used for the state-update function of Trivium [26], whose state-update equations were reviewed in Section 4.3.3. It should be noted that the state-update function of Trivium is reversible, that is, each internal state has only one prior state and one next state. Therefore, state convergence will not occur either during initialisation, or keystream generation.

Clock-control mechanisms

Clock control mechanisms are used in keystream generators which have two or more registers. They typically use an integer function which takes inputs from selected stages of a particular register. Clock-control mechanisms can be found in ciphers like the Step-1/2 generator proposed by Gollmann and Chambers [58], and the LILI family of

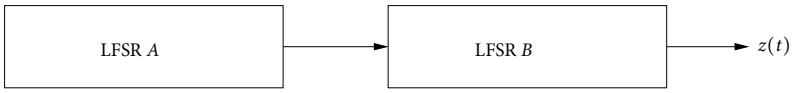


Figure 6.11: Step-1/2 generator

stream ciphers [29, 107]. In the Step-1/2 generator, clocking register A controls how many times the clock-controlled generator B is clocked for. If the output of A is 0, B is clocked once, and if the output of A is 1, B is clocked twice. A diagram showing the components in the Step-1/2 generator can be seen in Figure 6.11. State convergence will not occur in the Step-1/2 generator, as the state-update function is bijective. Given a state $S(t+1)$ at time $t+1$, we perform the following operations to obtain the unique pre-image of $S(t+1)$.

1. Calculate the previous output bit of A by clocking A backwards once.
2. Once the previous output bit of A is obtained, we know how many times B was clocked, and can clock it the appropriate number of times to obtain the state at time t.

The LILI family of stream ciphers make use of an integer function during the diffusion and keystream generation process. At time t, this integer function I_B takes as input stages from the clock-control register A and outputs the integer value $I_B(t)$. The clock-controlled register B is then clocked $I_B(t)$ times. Similar to the Step-1/2 register, this state-update function is bijective. Given a state at time t+1, we perform similar operations to obtain $t+1$'s unique pre-image.

1. Calculate the state of A at time t by clocking A backwards once.
2. Once the previous state of A is obtained, calculate the integer value $I_B(t)$ obtained at time t using the integer function, and clock B backwards that number of times.

Before we explain why state convergence occurs in mutually clock-controlled ciphers, we discuss why state convergence *does not* occur in both the Step-1/2 generator, and the LILI family of stream ciphers. In both of these keystream generators, the bijectiveness of the state-update function is due to the fact the register A is autonomous and regularly clocked, while register B relies on some output derived from selected stages of register A to determine how many times to clock. Analysing the state update function for A and B, we know that since B relies on the output of A to determine how many times B is clocked, it would be possible for two distinct B states , B, and B', at time t to converge

to the same state at time $t+1$ for two different clocking values obtained from register A. However for that to happen, register A at time $t+1$ must have had two distinct preimages, A and A', at time t, since a different number of clocks would be required for the two B distinct states at time t to be equal at $t+1$. But register A can only have one pre-image at time t, since register A is clocked regularly, so state convergence is not possible for the state-update function in the Step-1/2 generator and the LILI family of stream ciphers. In contrast, using outputs from a clock-controlled register to control how the clock-control register is clocked has the potential to cause state convergence.

To illustrate why this can occur, we use modified versions of LILI, LILI-M1 and LILI-M2, as case studies.

LILI-M1

The LILI-family of stream ciphers consists of two stream ciphers. LILI-II [107] and LILI-128 [29]. Although both ciphers differ in the various functions used, the structure of both ciphers can be generalised. This structure consists of two LFSRs: LFSR A, LFSR B, a nonlinear output function g and an integer function I_B. I_B takes as input selected stages from A and outputs an integer $I_B(t)$. $I_B(t)$ determines how many times register B is clocked.

In LILI-M1, we retain these components and add an additional integer function I_A, which takes as input selected stages from register B and outputs an integer $I_A(t)$. $I_A(t)$ determines how many times register A is clocked. The functions for I_A and I_B output the integer value $I_A(t)$ and $I_B(t)$ using the following equations.

$$I_A(t) = 2 \times B_{12}(t) + B_{20}(t) + 1$$
$$I_B(t) = 2 \times A_{12}(t) + A_{20}(t) + 1$$

The outputs of I_A and I_B can be seen in Table 6.9. During each iteration of LILI-M1's state-update function, the integer values $I_B(t)$ and $I_A(t)$ are calculated from the internal state $A(t)$ and $B(t)$ respectively. Registers $A(t)$ and $B(t)$ are then clocked $I_A(t)$ and $I_B(t)$ times respectively. Diagrams showing the difference in structures of the original LILI family of stream ciphers and LILI-M1 is shown in Figure 6.12 and Figure 6.13 respectively.

Why state convergence occurs in LILI-M1. Assume that the current internal state of LILI-M1 is $(A(t+1), B(t+1))$. To calculate how many times $A(t+1)$ needs to be

Table 6.9: Output of I_A and I_B based on their inputs.

$B_{12}(t)$	$B_{20}(t)$	$I_A(t)$	$A_{12}(t)$	$A_{20}(t)$	$I_B(t)$
0	0	1	0	0	1
0	1	2	0	1	2
1	0	3	1	0	3
1	1	4	1	1	4

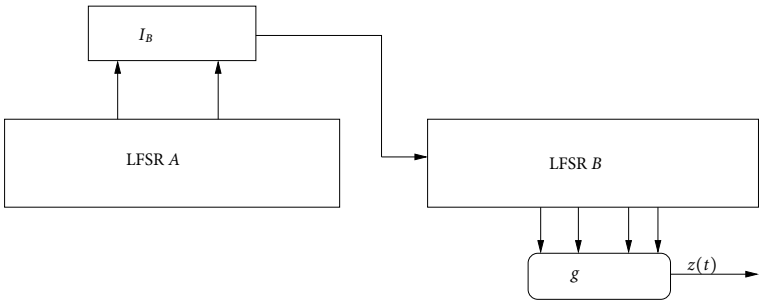


Figure 6.12: General structure and components of the LILI keystream generators

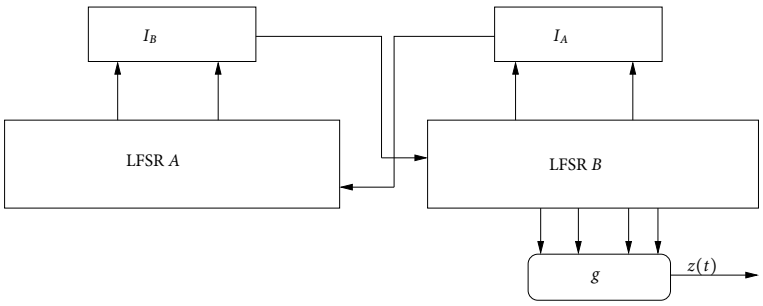


Figure 6.13: Structure and components of LILI-M1

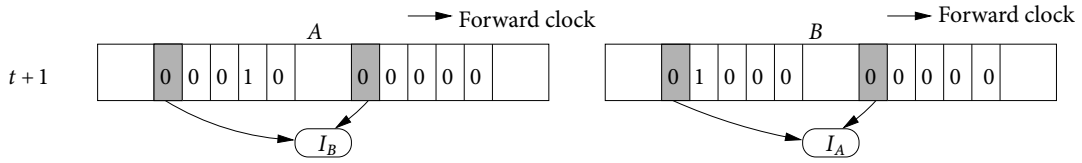
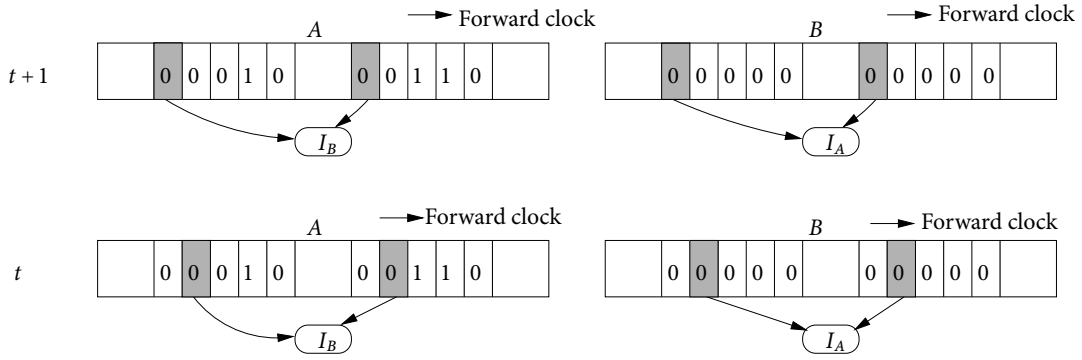


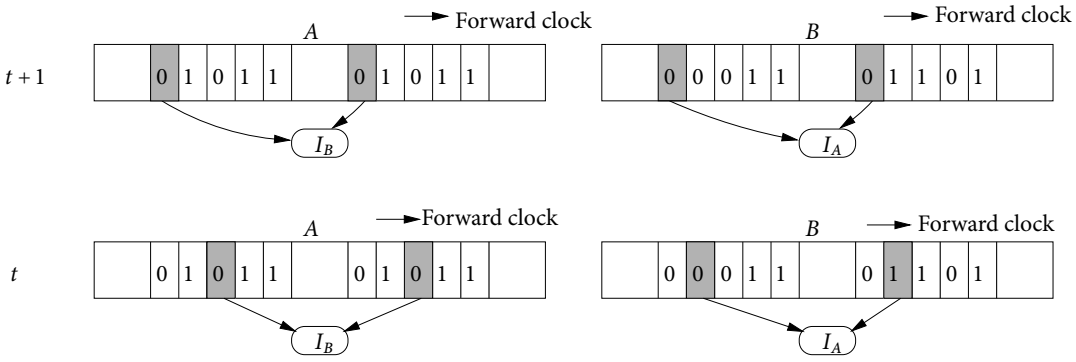
Figure 6.14: LILI-M1 state at time $t+1$ which has 0 pre-images



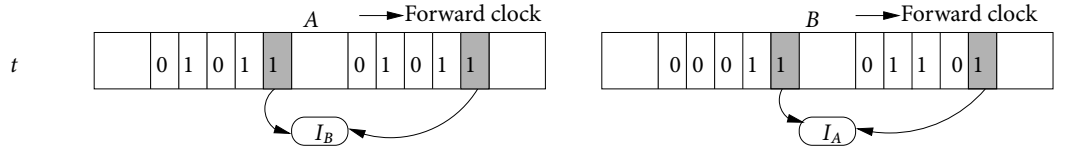
State obtained after clocking both $A(t+1)$ and $B(t+1)$ back once.

Figure 6.15: LILI-M1 state at time $t+1$ which has 1 pre-image

clocked back in LILI-M1, we need to know the value of $I_A(t)$. The determination of $I_A(t)$ requires knowledge of what the state $B(t)$ was, which in turn requires knowledge of $I_B(t)$. Determining $I_B(t)$ requires knowledge of $A(t)$ which is not possible without knowing what $I_A(t)$. If we knew what were the values of $I_A(t)$ and $I_B(t)$ from the states $A(t+1)$ and $B(t+1)$, we would be able to determine $(A(t+1), B(t+1))$'s unique pre-image. However we are not able to determine the original values of $I_A(t)$ and $I_B(t)$ from $A(t+1)$ and $B(t+1)$ as the relevant stages used in the calculation of I_B and I_A have already been clocked forward. Consequently, we have to assume each possible combination of $\{I_A(t), I_B(t)\}$ integer function outputs was equally likely. Using each possible combination $\{I_A, I_B\}$ outputs, we clock back registers A and B and check if the integer value output of $I_A(t)$ and $I_B(t)$ at time t match the number of times we clocked back registers A and B. If there are no matches, the state $(A(t+1), B(t+1))$ has no valid pre-images. If there is one matching pair, the state $(A(t+1), B(t+1))$ has one unique pre-image. If there is more than one match, there is more than one pre-image for that particular $(A(t+1), B(t+1))$ state. Figures 6.14, 6.15, 6.16, 6.17, and 6.18 show examples of LILI-M1 states which have 0–4 pre-images respectively. In the figures, the shaded stages represent inputs to the integer function I_A and I_B. The left shaded stage in each register are the stages A_{12} and B_{12} while the shaded stage on the right in each register represent the stages A_{20} and B_{20}.

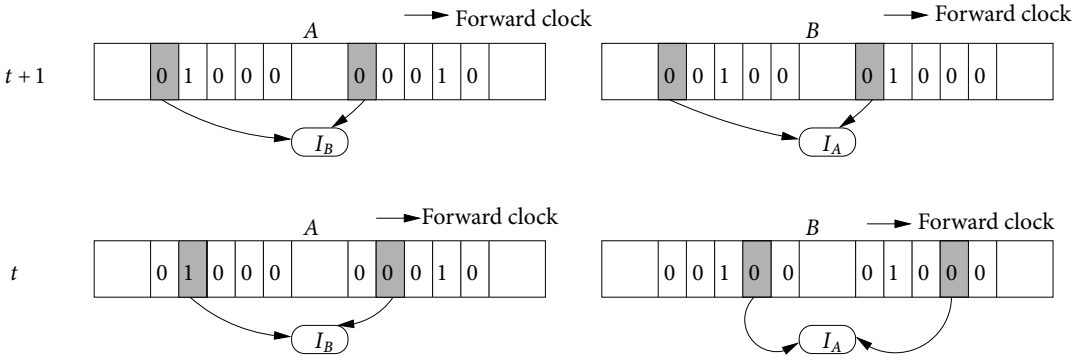


State obtained after clocking $A(t+1)$ back twice and $B(t+1)$ back once.

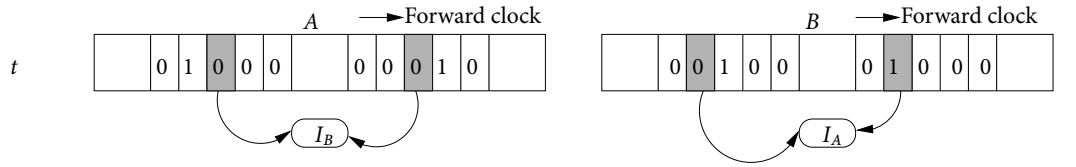


State obtained after clocking both $A(t+1)$ and $B(t+1)$ back four times.

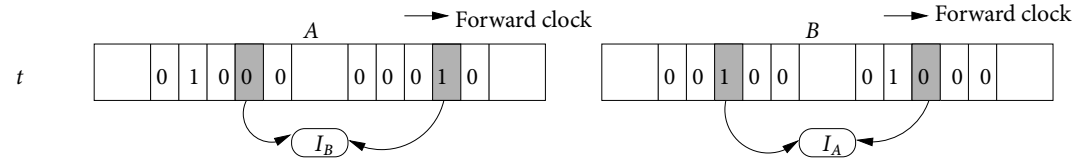
Figure 6.16: LILI-M1 state at time $t+1$ which has 2 pre-images



State obtained after clocking $A(t+1)$ back once and $B(t+1)$ back three times.

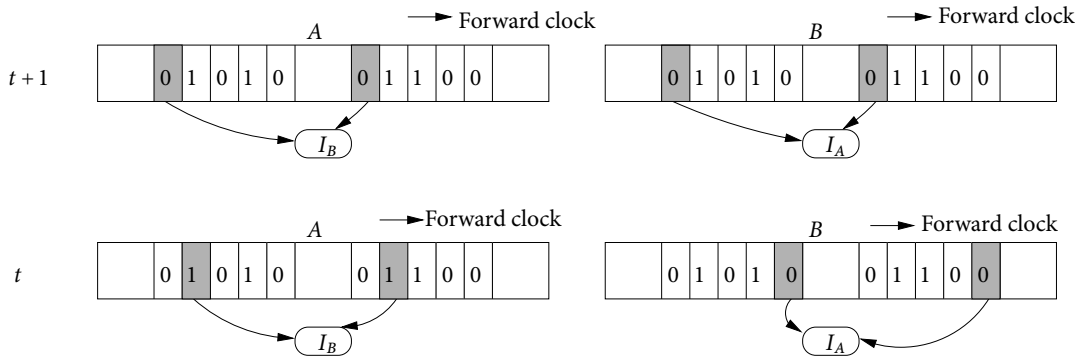


State obtained after clocking $A(t+1)$ back twice times and $B(t+1)$ back once.

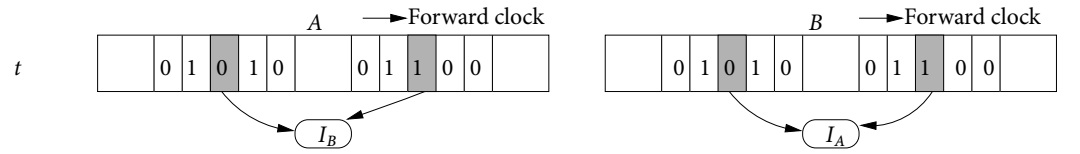


State obtained after clocking $A(t+1)$ back three times and $B(t+1)$ back twice.

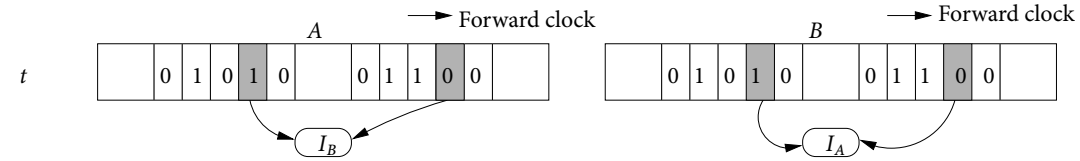
Figure 6.17: LILI-M1 state at time $t+1$ which has 3 pre-images



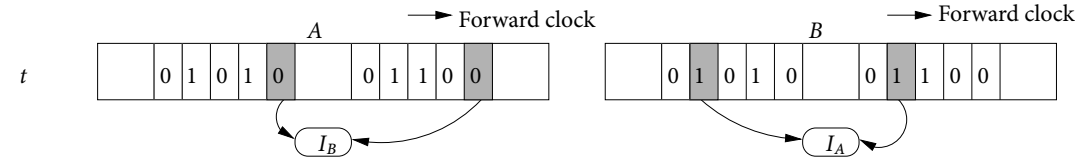
State obtained after clocking $A(t+1)$ back once and $B(t+1)$ back four times.



State obtained after clocking both $A(t+1)$ and $B(t+1)$ back twice.



State obtained after clocking both $A(t+1)$ and $B(t+1)$ back three times.



State obtained after clocking $A(t+1)$ back four times and $B(t+1)$ back once.

Figure 6.18: LILI-M1 state at time $t+1$ which has 4 pre-images

Table 6.10: Output of LILI-M2's I_A function based on its inputs.

$A_{12}(t)$	$B_{20}(t)$	$I_B(t)$
0	0	1
0	1	2
1	0	3
1	1	4

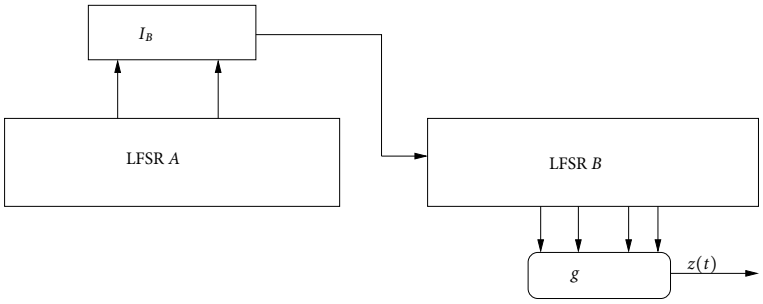


Figure 6.19: Structure and components of the LILI keystream generators

LILI-M2

LILI-M2 retains all the components of the LILI stream ciphers. The only modification comes from the selection of stages which are used in the clocking function I_B. In the original LILI stream ciphers, both input stages to I_B came from register A. In LILI-M2, one of the input stages comes from register A, while the other comes from register B. The function I_B is defined as

$$I_B(t) = 2 \times A_{12}(t) + B_{20}(t) + 1$$

The output of I_B based on its respective inputs can be seen in Table 6.10. Thus, in LILI-M2, register A is regularly clocked, as was the case in the original LILI stream ciphers, while the clocking of register B is determined by I_B. A comparison of the original LILI family of stream ciphers and LILI-M2 can be seen in Figure 6.19 and 6.20 respectively.

Why state convergence occurs in LILI-M2. In LILI-M2, register A is regularly clocked and autonomous, while the clocking of register B is determined by the output of the control function $I_B(t)$. Since register A is autonomous, given the state $A(t+1)$, it is possible to determine the unique pre-image $A(t)$ and determine its bit contribution in the

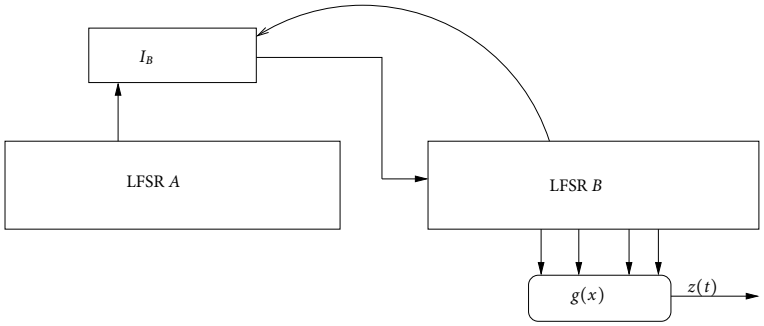


Figure 6.20: Structure and components of the LILI-M2

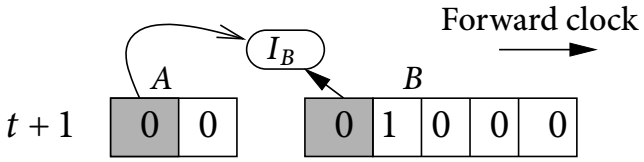
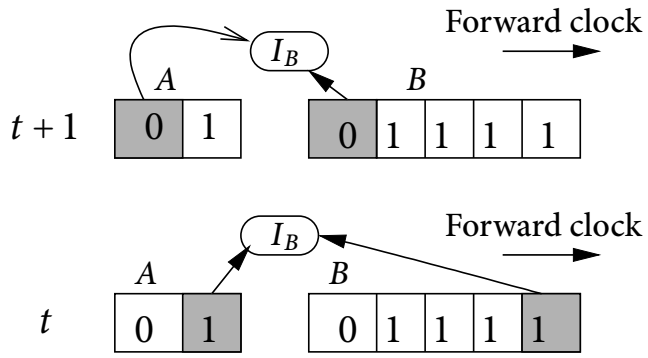


Figure 6.21: LILI-M2 state at time $t+1$ which has o pre-images

calculation of the $I_B(t)$. The state convergence problem arises when we try to determine $B(t+1)$'s unique pre-image. Since we do not know what the actual value of $B_c(t)$ was, we are not able to determine which update function to use to clock $B(t+1)$ back to $B(t)$. Therefore, we have to assume that either control bit value (0 or 1) is possible. Clocking $B(t+1)$ back assuming the control bit was 0 or 1 may yield two possible pre-images, $B(t)$ and $B'(t)$. Consequently, two unique states, $(A(t),B(t))$ and $(A(t),B'(t))$ which are obtained when clocking back register B assuming that $B_c(t)$ was 0 and assuming $B_c(t)$ was 1, may yield two pre-images which when clocked forwards, give $(A(t+1),B(t+1))$. Figures 6.21, 6.22 and 6.23 show examples of LILI-M2 states which have 0–2 pre-images respectively. In the figures, the shaded stages represent inputs to the integer function I_B. The shaded stage in register A is A_{12}, while the shaded stage in register B is B_{20}.

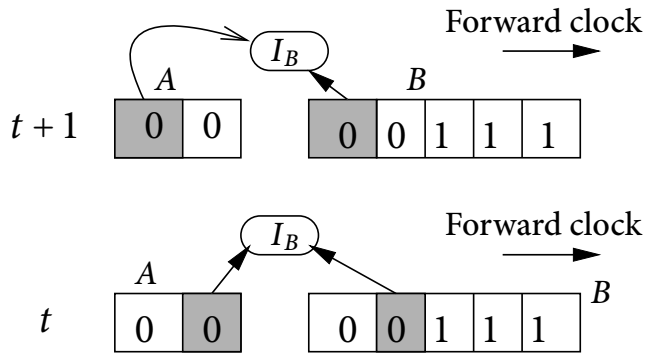
Avoiding state convergence in mutually-clocked keystream generators

Our analyses of LILI-M1 and LILI-M2 have shown that use of stages in a particular register as input to an integer function which controls how both registers are clocked may allow for the occurrence of state convergence. Mickey, which uses a similar method to determine how both of its registers are clocked, also experiences state convergence. Therefore, it is clear that keystream generators which use mutual-clocking mechanisms can experience state convergence. To prevent state convergence from occurring, it is recommended that mutual clock-control mechanisms are not used as the state-update function for clock-controlled keystream generators. However, the traditional

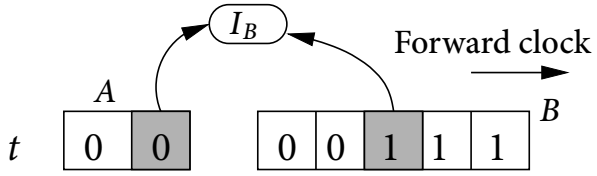


State obtained clocking $A(t+1)$ back once and $B(t+1)$ back four times.

Figure 6.22: LILI-M2 state at time $t+1$ which has 1 pre-image



State obtained clocking $A(t+1)$ and $B(t+1)$ back once.



State obtained clocking $A(t+1)$ once and $B(t+1)$ back twice.

Figure 6.23: LILI-M2 state at time $t+1$ which has 2 pre-images

clock-control mechanism, which uses output from a regularly clocked, autonomous register to control how another register is clocked, appears to be immune from this state convergence problem.

6.4.2 Self-update mechanisms

Self-update operations for stream ciphers are mechanisms which use the output from a particular register to update the same register. Self-update mechanisms can found in the initialisation functions of Mixer [79], which was analysed in Chapter 5, and Sfinks [21].

Cause of state convergence in self-update mechanisms

In Mixer, the cause of state convergence is due to using the output bit of B in the mixing operation. Since the output bit has already been used to update $A_{127}(t+1)$ and $B_{88}(t+1)$, via the XOR operation, it would not be possible to determine what the actual bit was, forcing us to assume both $\varphi = 0$ or $\varphi = 1$ were possible. To prevent this problem from occurring, the diffusion process of Mixer should not employ the mixing operation, but should use another method to diffuse the key-IV bits across the two registers.

In Sfinks, the self-update operation comes from the design decision of using the outputs of the Sfinks's S-box, after a delay of seven clocks, to update selected stages of Sfinks's LFSR. In Section 6.3.1, the cause of state convergence in Sfinks occurs was discussed. The cause was due to the fact that the one of the stages used as input to the S-box to calculate the S-box's 16-bit output was later updated using one of the S-box's output bits via the XOR operation. For Sfinks, the delay of seven iterations means that during initialisation, $A_{154}(t+7)$ is updated by xoring it with the a bit from the S-box output calculated at time t (specifically, $y_{10}(t)$), of which $A_{161}(t)$ is one of the S-Box inputs. Since Sfinks' register A is a regularly clocked LFSR, $A_{161}(t) = A_{154}(t+7)$ after seven clocks. By flipping the bit at $A_{161}(t)$, there is a chance that, with all other stages in A being the same, the flipped bit in $A_{161}(t)$ will result in a different value in $y_{10}(t)$. When $y_{10}(t)$ is used to update $A_{154}(t+7)$, there is a chance two distinct states will converge to the same state. The equations which have to be met for this to happen are detailed in Section 6.3.1. Alhamdan et al. [2] suggested that one possible way of avoiding state convergence is to change the time delay by which the stages in Sfinks' LFSR are updated by the S-Box output word. However, they also note that care must be taken when selecting the amount of delay to ensure that state convergence does not re-occur.

6.4.3 Addition-with-carry state-update operations

The summation generator and the F-FCSR keystream generator are ciphers which use addition with carry operations to provide nonlinearity to the cipher. For FCSR, state convergence can occur in some registers due to the state-update function being non-invertible. For the summation generator, state convergence will occur if the memory bit C of the summation generator was randomly initialised. That is, if two summation generator states had the same contents for registers A and B, and the contents of memory bit C for the two states are complements of each other, state convergence will occur. To prevent state convergence, register C of a summation generator has to be initialised to a fixed value (either 0 or 1). If the memory bit C was fixed during initialisation, the state convergence problem described above will not occur. Precise details of how state convergence can occur in keystream ciphers which use addition-with-carry state-update functions are described in Section 6.3.2 and Section 6.3.3. However, it should be noted that by fixing the value of that memory bit, the state entropy will still be reduced by one bit.

6.4.4 State convergence during the loading phase

In earlier subsections, a classification of the causes of state convergence in stream ciphers was presented. In this section, we review some other causes of state convergence which are not due to any particular mechanism in the keystream generator, but still happen due to the way the key and IV are loaded into the keystream generator's internal state. State convergence of this type occurs in two ciphers: A5/1 [22] and LILI-II [29].

A5/1

During the loading phase in A5/1, the 64 bit key and 22 bit IV are linearly loaded into the register. As the size of A5/1's register is 64 bits, state compression occurs and it is not possible for each key-IV pair to generate a unique loaded state; it can be shown that 2^{22} key-IV pairs correspond to each loaded state. This further compounds the state convergence which is experienced by A5/1 during initialisation and keystream generation. A simple fix to state compression is to ensure that the size of the internal state in a keystream generator is at least as large as the sum of the size of the key and IV.

LILI-II

Biham and Dunkelman [13] observed that the LILI-II stream cipher may also experience state convergence during the loading phase. During LILI-II's loading phase, the 128 bit register A is loaded using the XOR of the 128 bit key and 128 bit IV. To load the 127 bit register B, we first drop the first bit of the key, and drop the last bit of the IV, then XOR the two 127 bit values together. During the diffusion phase, 255 bits of output is generated. This 255 bit value is loaded into the two registers (128 bits in A, and 127 bits in B). The cipher is run again to produce 255 bits of output. As before, this 255 bit value is loaded into the two registers (128 bits in A, and 127 bits in B). At this point, LILI-II is in an initial state and ready to produce keystream.

However, the linear operation of XORing the key and IV together causes state convergence during the loading phase. Biham and Dunkelman [13] noted that if two distinct keys-IV pairs (K^1, V^1), and (K^2, V^2) satisfy the differential $K^1 \oplus K^2 = V^1 \oplus V^2 = 1^{128}$, the 255 bit output produced during the diffusion phase will be the same for the two key-IV pairs. Since the same 255 bit output is directly loaded into the two registers to form the state which is used during the diffusion phase, the two key-IV pairs will produce the same initial state and consequently, the same keystream. To prevent this from occurring, XORing the key-IV pair during the keystream generator's loading process should be avoided, and a direct loading of the key and IV into a keystream generator's internal state should be used instead.

6.5 State convergence and stream cipher cryptanalysis

The usual goal of stream cipher cryptanalysis is to recover either the initial state or the secret key of a keystream generator. In Section 5.2, three possible scenarios for state convergence in keystream generators are described. These are:

Scenario 1: The same secret key used with different IVs generates the same initial state.

Scenario 2: The same IV used with different secret keys generates the same initial state.

Scenario 3: Distinct key-IV pairs generate the same initial state.

In this section, we analyse the effectiveness of state convergence on stream cipher cryptanalysis with regards to some common techniques applied to bit-based stream

ciphers. These include time-memory-data tradeoff attacks, correlation attacks, algebraic attacks and differential attacks.

6.5.1 Effect on time-memory-data tradeoff attacks

Various time-memory-data tradeoff (TMDT) algorithms were discussed in Section 2.4.7. These included the initial state recovery TMDT attacks by Babbage [7] and Golić [55], and the secret key recovery TMDT attacks by Hong and Sarkar (HS) [71], and Dunkelman and Keller (DK) [41]. We discuss the effect state convergence has on these attacks below.

Internal and initial state recovery

If the attacker wanted to perform initial state recovery rather than internal state recovery, the parameter s can be replaced with $k + v$ to obtain appropriate tradeoffs. There are factors to consider when choosing whether to mount an internal or initial state TMDT attack. An attacker selecting random internal states to store in the lookup table does not need to spend any time running the initialisation process before the state is stored in the lookup table, and can produce keystream directly from the chosen state. The drawback of this approach is that the internal state selected may not be a valid initial state; that is, one which can be formed from a key-IV pair, and the attacker wastes memory space storing that particular state. If an attacker selects random key-IV pairs and stores the corresponding initial state, the attacker is assured that they are storing a valid initial state and that this initial state will generate a valid keystream sequence. However, the attacker must spend extra computational time performing the initialisation process for each key-IV pair chosen for the lookup table.

If the captured keystream can be found in the lookup table, the attacker uses the corresponding state to generate sufficient keystream to decrypt the entire encrypted frame. Any state which produces the keystream segment enables the attacker to decrypt the remainder of that frame. However, if multiple key-IV pairs generate the same keystream, the attacker does not know the secret key that was used and they will not be able to decrypt other frames in the communication but will need to perform the online phase of the TMDT attack again in order to decrypt these. If the number of distinct initial states I is such that $I < 2^{k+v}$, the tradeoff equation in Equation 2.4.7 will result in reduced time, memory and data requirements if the attacker is aware of this and constructs the lookup table accordingly.

An example of how the reduction in the effective initial state size due to state convergence has a positive effect on TMDT initial state recovery attacks relates to the cipher

Table 6.11: Tradeoffs for Mixer using Biryukov and Shamir's TMDT attack

	Original tradeoffs	New tradeoffs	
s	192 bits	191 bits	109 bits
T	2^{96}	$2^{95.50}$	$2^{54.50}$
M	2^{96}	$2^{95.50}$	$2^{54.50}$
D	2^{48}	$2^{47.75}$	$2^{27.25}$

Mixer, which was analysed in Chapter 5. The estimated total number of distinct initial states after all key-IV pairs undergo the initialisation phase is bounded by 2^{191} and 2^{109}. Table 6.11 shows the possible time, memory and data requirements for initial state recovery TMDT attacks, taking into account Mixer's state convergence problem. It can be seen that state convergence may result in significantly reduced time, memory and data requirements in BS's TMDT attack.

The non-uniform rate of state convergence, which can be observed in A5/1 and Mixer can also be exploited to make the TMDT attack more efficient. For example, Biryukov et al. [18] exploited the fact that A5/1 can generate some segments of keystream more often than others, which in turn, can only have been generated by a certain subset of initial states. By constructing a lookup table using only these initial states, a TMDT attack can be more efficient than a TMDT attack on a stream cipher which did not have this characteristic.

Secret key recovery

Key recovery TMDT attacks require the attacker to construct lookup tables which consist of the secret key, IV, and the corresponding keystream it generates. If the goal of the attacker is to recover the secret key, the attacker can apply Hong and Sarkar's tradeoff equation, shown in Equation 2.4, or Dunkelman and Keller's tradeoff equation, shown in Equation 2.5 to determine if it is possible to recover the secret key faster than exhaustive key search. It should be noted that Hong and Sarkar impose D and T restrictions, such that $T \geq D^2$, while Dunkelman and Keller do not have this restriction. State convergence can have a positive or negative impact on the effectiveness of key recovery TMDT attacks, depending on which state convergence scenario occurs.

Table 6.12: Original and new tradeoffs for ZUC v1.4

	Original HS	New HS		Original DK	New DK	
l	128 bits	100 bits	66 bits	128 bits	100 bits	66 bits
T	2^{128}	2^{114}	2^{97}	$2^{102.4}$	$2^{91.2}$	$2^{77.6}$
M	2^{128}	2^{114}	2^{97}	$2^{102.4}$	$2^{91.2}$	$2^{77.6}$
D	2^{64}	2^{57}	$2^{48.5}$	$2^{102.4}$	$2^{91.2}$	$2^{77.6}$

Secret key recovery and Scenario 1. When a single secret key and different IVs generate the same keystream, the HS attack can recover the correct secret key if that key-IV pair was one of those selected for the construction of the lookup table. After the online phase of the TMDT attack, an attacker can check that the recovered IV is the same as that captured along with the keystream. If it is the same IV, the attacker can be confident that they have recovered the correct secret key and can use that secret key with other IVs to decrypt other frames in the communication. If the IV does not match the one recorded in the table, the attacker knows that they have recovered the wrong secret key and would need to perform the online phase of the attack again.

A similar process happens in the DK attack. In the online phase, the attacker uses the appropriate IV-based lookup table and checks if there is a match on the captured keystream. If there is a match, the corresponding secret key is the correct secret key which generated the captured keystream. The attacker can then use that same secret key with other IVs to decrypt other encrypted frames in the communication.

Reduced secret key sizes may have a positive effect on key recovery TMDT attacks. An example of this can be seen in the ZUC stream cipher. The ZUC stream cipher [36] is a stream cipher which can experience Scenario 1 state convergence. The effective key size of ZUC was estimated by Wu et al. [111] to be between 66 to 100 bits. In Table 6.12 we show the possible time, memory and data requirements for both HS and DK's TMDT attack when the effective key size is 66 or 100 bits and assuming the size of the IV is unchanged at 128 bits. It can be seen that state convergence may result in reduced time, memory and data requirements in both the HS and DK attack.

Secret key recovery and Scenario 2. Where the same keystream is generated by different secret keys for any given IVs, the attacker does not have the confidence that secret key they recovered is the correct one.

Let us assume that three secret keys, K^1, K^2 and K^3, with the same IV, $V1^1$, produce the same keystream for a particular stream cipher. Two secret keys, K^2 and K^3 were selected by the attacker for the construction of the lookup table. The original conversation was encrypted with K^1 and a particular frame in this conversation was encrypted with the IV V^1. During the online phase of the attack, the attacker recovers the key K^2. If an attacker, incorrectly assuming that K^2 was the actual secret key, tries to use K^2 to decrypt other frames, it should not be successful, since K^2 with a different IV V^2 will most likely not generate the same keystream as would have been generated by the K^1-V^2 pair. Since K^1 was not selected during the construction of the lookup table, the secret key recovery TMDT attack in this scenario is equivalent to initial state recovery. For the attack to succeed in this situation, the attacker has to hope that the correct secret key was selected during the construction of the lookup table. If the correct secret key was not used, the attacker will not be able to decrypt other encrypted frames in a single conversation.

If γ is the number of secret keys an attacker obtains at the end of the online phase of the attack and ε, with $\varepsilon < V$, being the effective IV size of the stream cipher, the HS and DK tradeoff would be

$$(T + \gamma) \cdot M^2 \cdot D^2 = 2^{2(k+\varepsilon)} \tag{6.4}$$

The memory and data requirements however, remain the same as would have been obtained in Equation 2.4.7. However, since γ possible secret keys can now appear in the lookup table, an attacker needs to try, on average, $\frac{\gamma}{2}$ keys with other IVs before they can be certain if the secret key they are currently trying is correct.

If a keystream generator uses the same state-update function for initialisation and keystream generation, it can be viewed as a keystream generator which performs an extended version of the initialisation process to generate keystream. Hence, if the segment of keystream used during the construction of the lookup table matched a segment of keystream which was captured not from the beginning of keystream generation, the number of possible secret keys which could have generated the keystream with the particular IV can increase. Therefore, a successful TMDT secret key recovery attack with keystream generator which use the same state-update function for both initialisation and keystream generation can be less likely than an attack on a keystream generator which uses a different state-update function for initialisation and keystream generation.

Secret key recovery and Scenario 3. Where the keystream is generated by different key-IV pairs, during the online phase an attacker will know if they have recovered the correct secret key based on the publicly known IV. If τ and ϵ, with $\tau < 2^k$ and $\varepsilon < 2^v$, are the effective secret key and IV size respectively, the tradeoff curve would be

$$T \cdot M^2 \cdot D^2 = 2^{2(\tau+\varepsilon)} \tag{6.5}$$

Similar to Scenario 1, since the attacker knows the IV used to generate the captured keystream and assuming the secret key used to generate the captured keystream was used during the construction of the lookup table, the attacker can be confident that the secret key they recover during the online phase of the TMDT attack is the correct one. Furthermore, since $\tau < 2^k$ and $\varepsilon < 2^v$, the HS and DK attacks will be less than exhaustive key search.

Biryukov et al.'s [18] TMDT attack on A5/1 describes an attack which is able to recover the secret key in a few minutes at most. The most expensive cost of this attack is the pre-computation complexity, which they calculated to be

$$P = M \cdot \sqrt{T} = 2^{48}$$

where $M = 2^{36}$, $T = 2^{24}$, and 2^{48} is the total number of initial states which will produce a certain 16-bit output prefix ($2^{64} \times 2^{-16}$). They estimated the number of distinct initial states at the end of A5/1's diffusion phase to be $19.2/100 \times 2^{64} \approx 2^{61.62}$ due to state convergence. Using this new estimate, the pre-computation complexity of Biryukov et al's attack is reduced to $2^{61.62} \times 2^{-16} = 2^{45.62}$. This in turn, potentially reduces the time and memory requirements to be $M = 2^{35}$ and $T = 2^{21.24}$.

Summary

This section considered how state convergence could affect the success of TMDT attacks. In the case of initial state TMDT attacks, an attacker potentially needs to guess a smaller set of initial state than what was originally intended by the designers of the stream cipher, since not all distinct key-IV pairs generate a distinct initial state. This could result in less time, data and memory requirements needed for the initial state TMDT attacks to succeed than previously estimated by the designers of the stream cipher. The disadvantage of initial state TMDT attacks is if the attacker wanted to decrypt other encrypted frames in the communication, they would need to re-run the TMDT attack for each frame. If the attacker were to repeatedly use initial state recovery TMDT attacks to

decrypt multiple frames, it can be less efficient than secret key recovery TMDT attacks.

For secret key recovery TMDT attacks, the success of the attacks depend on the type of scenario, as outlined in Section 5.2. Unless the convergence is such that different secret keys with the same IV produce the same initial state, an attacker who recovers a secret key can check if the associated IV value matches that which was observed along with the keystream. If the IV value is the same, the attacker can be confident that they have recovered the correct key. However, if state convergence was such that it was possible that multiple distinct secret keys with the same IV generate the same keystream, there is a possibility that the secret key recovered by the attacker during the online phase of the TMDT attack is not the correct secret key. In this case, it is likely that the attacker can only decrypt a single frame and secret key TMDT attacks maybe less effective than claimed. The attacker can only be confident that they have actually recovered the correct secret key if they can use it to decrypt the contents of all the frames in the communication.

6.5.2 Effect on correlation attacks

Correlation attacks require the attacker to reconstruct the correct initial state from the observed keystream by measuring the level of correlation between a particular initial state and the keystream it generates. If the stream cipher suffers from state convergence, there can be many distinct initial state which generate the same keystream. These different initial states may not give the same level of correlation as the actual initial state used to generate the keystream. However, since all these initial states may generate the same keystream, the attacker can use either one to generate keystream to decrypt the frame in the communication, without worrying if the initial state they are using was the initial state used to encrypt the frame in the first place. Simpson et al. [104] used a correlation attack to recover the initial states for a shrinking generator [30] which generated a particular segment of keystream. In some cases, the correlation value between the keystream and a certain initial state A' was higher than the correlation value of the actual initial state A and the said keystream. Note however, that both A and A' produced the same keystream. Since A and A' generate the same keystream, the attacker can use either one to generate keystream to decrypt the frame in the communication, without worrying if the initial state they are using was the initial state used to encrypt the frame in the first place. Hence state convergence may make correlation attacks even more effective for state recovery.

6.5.3 Effect on algebraic attacks

The usual goal of algebraic attacks, reviewed in Section 2.4.8, is to recover the initial state of an encrypted frame in a communication. For a stream cipher which does not suffer from state convergence, solving a system of equations should yield only one unique solution provided enough equations are generated. If this stream cipher suffers from state convergence during keystream generation, solving a system of equations which has the same number of equations and variables in the previous example would yield more than one unique solution corresponding to all possible initial states that generate the same keystream.

The complexity of algebraic attacks depends on the technique used to solve the system of equations. If a brute-force approach is used to solve the system of equations, the time needed to recover an initial state is expected to decrease because the algorithm can terminate the moment an initial state which can generate the keystream is recovered. That is, the attacker does not need to recover the actual initial state which was originally used to encrypt the frame, as long as the initial state recovered produces the same keystream as the actual initial state. If an algorithm which uses a Gröbner basis-like approach is used, like the F4 algorithm [46] described in Section 2.4.8, the time taken to solve the full system of equations is not affected by whether or not the system contains one or multiple solutions. If we were to use partial key-guessing with the F4 algorithm, the time needed to solve the system of equations is expected to be in between a brute-force approach and solving the same system of equations using the F4 algorithm without partial key-guessing.

If the goal of the attacker is secret key recovery, the system of equations will be more complex, as the system of equations will need to take into account the initialisation phase, which can consist of many iterations of the state-update function in the diffusion phase. However, since the the state-update functions for both initialisation and keystream generation is nonlinear for modern stream ciphers, this can result in a system of equations of high degree and solving this system of equations for a particular stream cipher can take a longer time as compared to solving a system of equations whose goal is initial state recovery for the same stream cipher. The equations for secret key recovery can consist of known IV bits. If the system of equations are formed taking the IV bits into account, the confidence of the attacker of whether the secret key recovered is correct depends on which scenario of state-convergence, discussed at the beginning of Section 6.5 occurs. If Scenario 1 or Scenario 3 state convergence occurs, the attacker can be confident that the secret key recovered is correct. However, if Scenario 2 state

convergence occurs, the attacker does not have the same confidence as that same IV with a different secret key could have also generated the keystream. The attacker would need to check if the recovered secret key can decrypt the other frames in the communication. If it can successfully decrypt the remaining frames, the attacker can be confident they have correctly recovered the correct secret key.

6.5.4 Effect on differential attacks

Biham and Dunkelman propose a general framework for analysing stream ciphers using differential cryptanalysis [13]. In their framework, one of the differential characteristics listed which attackers could exploit was comparing the difference in two or more initial states which were generated using two or more distinct key-IV pairs. For any given key, the existence of IV pairs which generate the same keystream allows an attacker to generate equations in the state bits which lead ultimately to recovery of information on some of the key bits. The frequency with which the relevant IV pairs occur determines the data requirements of the attack.

Successful differential attacks on the Py [14] and Pypy [15] ciphers have been mounted by Wu and Preneel [112, 113] and Isobe et al. [73]. Wu and Preneel [112] noted that if two IVs with a two-byte difference satisfied certain differentials, state convergence can occur with probability $2^{-23.2}$ after the diffusion phase is completed. The key recovery attacks on Py and Pypy by Wu and Preneel [113] and Isobe et al. [73] exploited these differentials in their attacks. Wu and Preneel [113] and Isobe et al. [73] noted that if the size of the IV is 80 bits or more, the key recovery attack on Py and Pypy can be faster than exhaustive key search. For example, in Isobe et al.'s attack, for 24 chosen 128 bit IVs, a 128 bit secret key of Py and Pypy can be recovered with an attack complexity of $T = O(2^{48})$, which is significantly less than the complexity of a brute force attack.

6.6 Conclusion

In this chapter, we have identified two methods which cryptanalyst can use to determine if state convergence is present in a stream cipher. These methods are state transition tables and various combinations which a register can be clocked back. We analysed several stream ciphers known to experience state convergence and discussed why this occurs. We showed, using a general example of why a feedback-with-carry state-update function is not invertible and demonstrated why state convergence occurs in the summation

generator. We also show that the effective state size of Mickey-v2 drops below 160 bits, contrary to the claims of the designers

From these analyses, we identified three mechanisms which cause state convergence and the stream ciphers analysed in Section 6.2 and Section 6.3 were classified into these mechanisms. These mechanisms are: mutual clock-control, self-update operations, and addition-with-carry state-update operations. In particular, we demonstrate, through modified versions of the LILI-family of stream ciphers, why mutual clock-control, an amalgamation of mutual-update mechanisms and clock-control mechanisms, causes state convergence when the individual mechanisms may not cause state convergence. It is demonstrated that use of these mechanisms can cause state convergence and their use in state-update functions in keystream generators should be avoided.

The impact state convergence may have on common stream cipher cryptanalytic techniques like correlation attacks, time-memory-data tradeoffs, algebraic attacks, and differential attacks are also discussed. For correlation attacks, state convergence in stream ciphers may result in the recovering the initial state with an attack complexity that is less than exhaustive keysearch. In the case of time-memory-data tradeoff attacks, state convergence can have a positive impact on the efficiency of initial state recovery, and state convergence Scenarios 1 and Scenarios 3. In the Scenario 2, the attacker needs to try all relevant secret keys recovered during the online phase of the attack before they can be confident that they have recovered the correct secret key. For algebraic attacks, multiple solutions when a system of equations consisting of an equal number of equations and variables is solved is possible; something which would not have been possible if the stream cipher did not suffer from state convergence during its keystream generation process. For differential attacks, if Scenario 2 state convergence occurs, the recovery of secret key bits may be possible. Furthermore, If Scenario 2 state convergence occurs, there is a higher chance of generating the same keystream for two different frames in a single communication, which may allow an attacker to decrypt encrypted frames using a ciphertext-only attack [37].

For some stream ciphers like A5/1 and Mixer, the number of distinct initial states can decrease as the number of iterations of the initialisation state update function performed increases. This decreases the effective key-IV space of the keystream generator while simultaneously decreasing the efficiency of the rekeying process with no corresponding increase in security. If the state-update function used during initialisation is also used to generate keystream, state convergence problems will continue and can result in a further reduction of the effective key-IV space.

Avoiding state convergence requires careful analysis of the state-update functions used during both initialisation and keystream generation to ensure that they are one-to-one. These one-to-one state-update functions should also be carefully used. The state convergence experienced by Sfinks is an example that demonstrates that the use of one-to-one components in composing a state update function is not enough to ensure that state convergence does not occur, and that designers should ensure that the overall state update function is also one-to-one.

Current stream cipher proposals commonly include an analysis section detailing their claimed resistance to various attacks. We recommend that future stream cipher designers pay careful attention to the choice of state-update functions used during initialisation and keystream generation, as functions which are not one-to-one may make many attacks more effective than anticipated.

Chapter 7

Conclusion and Future Research

As presented in Section 1.1, the main aim of this research was to examine the relationship between structural features of keystream generators and security. From this perspective, we have analysed the security provided by keystream generators whose keystream is generated using either of three models: linear update and nonlinear output functions, nonlinear update and linear output functions, and nonlinear update and nonlinear output functions. We have performed both algebraic and statistical analysis of keystream generators which use these three models. An in-depth study of state convergence in keystream generator was conducted. State convergence in stream ciphers may also lead to a reduction in security.

In this chapter, the contributions of this thesis are reviewed in Section 7.1. Directions for future research are explored in Section 7.2.

7.1 Review of Contributions

This section summarises the contributions to knowledge regarding stream cipher analyses made in this thesis.

7.1.1 Contributions in Chapter 3

Aspects of sequences produced by keystream generators which use a linear state update function and a nonlinear output function were examined in Chapter 3. In particular, the m-tuple distributions for the sequences produced by nonlinear filter generators

are examined. Firstly, we show, given a fixed nonlinear Boolean function, the m-tuple distributions sequences produced by any corresponding nonlinear filter generator are non-uniform. Additionally, we show that the sequences produced by nonlinear filter generators which use consecutive tap settings to the filter function may exhibit a larger non-uniformity than those produced by nonlinear filter generators which use uneven tap settings. To the best of our knowledge, a comparison of the m-tuple distribution of nonlinear filter generator sequences which use consecutive tap settings and uneven tap settings, respectively, has not appeared in the public literature.

7.1.2 Contributions in Chapter 4

Chapter 4 examined linearly filtered nonlinear feedback shift register keystream generators. We analyse the m-tuple distribution of keystream sequences produced by the complementary concept of the nonlinear filter generator, the linearly filtered nonlinear feedback shift register. Algebraic analysis of the Trivium stream cipher are also performed.

Contributions of Section 4.1

In Section 4.1, the m-tuple distributions of sequences produced by linearly filtered nonlinear feedback shift register keystream generators is investigated. We show that, for a fixed nonlinear feedback function, the m-tuple distributions of keystream sequences produced by applying a linear output function which takes as inputs consecutive tap settings can have a more uniform m-tuple distribution than keystream sequences produced by a linear output function which takes as inputs uneven tap settings. This is in contrast to our findings in Chapter 3, where the sequences produced by nonlinear filter generators which use uneven tap settings can have more uniform m-tuple distributions. To the best of our knowledge, prior to this research, the findings in this section have not appeared in published literature, and our study has provided further insight into the properties of nonlinear sequences generated by linearly filtered nonlinear feedback shift register keystream generators.

Contributions of Section 4.2

In Section 4.2, we present the first of two new analyses on Trivium-like ciphers and search for slid pairs in Trivium [26]. We extend the work of Priemuth-Schmid and Biryukov [92] and search for additional types of slid pairs in the Trivium cipher. A set

of slid pairs in Trivium is a particular sort of loaded state in the set Γ_α which after α iterations, produces another sort of loaded state in the set Θ_α. Prior to our investigation in this thesis, Priemuth-Schmid and Biryukov [92], and Zeng and Qi [115] were, to our knowledge, the only researchers who investigated slid pairs in Trivium. However, they did not investigate if any of the loaded states in Γ_α is present in Θ_α. This did not allow both groups of researchers to establish a lower-bound on the period of keystream sequences produced by Trivium. In our two experiments, we answered this question for particular α values. In our first experiment, we have shown that the set $\Gamma_\alpha \cap \Theta_\alpha$ is empty, for $111 \leq \alpha \leq 121$, and in our second experiment, we have shown that the sets $\Gamma_\alpha \cap \Theta_{\alpha'}$, for $111 \leq \alpha \leq 113$ and $111 \leq \alpha' \leq 113$, are empty. The second experiment have allowed us to establish that for $111 \leq \alpha \leq 113$, we were unable to find state cycles consisting of more than two consecutive loaded states.

Contributions of Section 4.3

In this section, we present some new algebraic analyses on Trivium-like ciphers. We performed algebraic analyses on Trivium-like ciphers using the combined techniques of Berbain et al. [10] and Raddum [93]. By analysing Trivum-like ciphers using this combination of techniques, we answer Berbain et al.'s open question regarding the possibility of extending their attack to ciphers in which q bits are updated at each step, while only $q' < q$ linear combinations of state bits are output. It is shown that the size of the registers, tap settings, and the number of registers whose stages are used as input into the input function all play a part in determining if the attack can succeed. For the case of Bivium-A [93], we present two successful algebraic divide-and-conquer attacks. The first attack requires 177 bits of keystream, and has less time and memory requirements than attacks done by other researchers. The second attack requires more time and memory, but requires 162 bits of keystream — 15 bits less than our first attack.

7.1.3 Contributions of Chapter 5

Chapter 5 examined keystream generators which produced keystream using a nonlinear feedback shift register and a nonlinear output function. In particular, we analysed the Mixer [79] stream cipher. This is an irregularly clocked, bit-based stream cipher with an internal state size of 217 bits. We show that the state-update function used during Mixer's initialisation process is not one-to-one, and as a consequence, state convergence occurs. The degree of state convergence increases as more iterations of the initialisation state-update function are invoked. After 200 iterations, it is estimated that the number

of distinct initial states is between 2^{109} and 2^{191}. We also show that this state convergence continues during keystream generation, due to Mixer's shrinking-generator fashion of generating keystream.

7.1.4 Contributions of Chapter 6

Chapter 6 investigates the state convergence problem in greater detail. A short discussion on the methods of detecting state convergence is done in Section 6.1. A discussion of ciphers which have been known to have state convergence problems is done in Section 6.2 and Section 6.3. In particular, for irregularly clocked ciphers, we show, through an analysis of state-update function of A5/1 and a modified version of A5/1, A5/1M, that majority-clock update functions can cause state convergence. We also provide counter-arguments to the claim made by the designers of the Mickey-v2 stream cipher, that increasing the state size of the keystream generator decreases the degree of state convergence.

For regularly clocked ciphers, we show how the analysis on the state-update function which causes state convergence in the F-FCSR stream cipher [77] can be also be applied to analyse the state-update function used in the summation generator in Section 6.3.3. As a result, we show how the summation generator suffers from state convergence. Based on the reviews done in Section 6.2 and Section 6.3, we are able to identify three mechanisms which cause state convergence in Section 6.4. These mechanisms are: mutual-clock control, self-update mechanisms and addition-with-carry. In particular, we show why mutual clock-control, an amalgamation of mutual-update mechanisms and clock-control mechanisms, causes state convergence when the individual mechanisms do not cause state convergence.

In Section 6.5, we analyse the effectiveness of state convergence on stream cipher cryptanalysis with regards to some common analysis techniques applied to bit-based stream ciphers. In the case of initial state recovery, we show how state convergence may result in the recovery of the initial state with an attack complexity which is less than exhaustive key search for correlation, and time-memory-data tradeoff attacks.

7.2 Future Directions

The research done in this thesis provides a foundation on which the study of stream ciphers of each model can be continued. A summary of several possible future research directions is given below.

7.2.1 Extending our Work in Chapter 3

Our experiments conducted in Chapter 3 represent only a small subset of possible nonlinear filter generators which can be constructed using a fixed nonlinear Boolean function. More experiments should be performed with more feedback functions and tap settings to verify if the trends observed in our experiments still occur. Secondly, establishing a mathematical proof on the relationship between the feedback function, tap settings to the nonlinear Boolean function, and properties of the nonlinear Boolean function have on the m-tuple distribution of the sequences produced by the nonlinear filter generator remains an open question.

7.2.2 Extending our Work in Chapter 4

Similar to the future work proposed for nonlinear filter generators, more experimental work needs to be done to verify our findings. Secondly, there is a need for a mathematical proof that m-tuple distributions of sequences generated by linearly filtered NLFSRs with consecutive taps to the linear function are more uniform than linearly filtered NLFSRs with taps which form a full positive difference set to the linear function, in the general case

The work on slid pairs in Trivium can also be extended. Zeng and Qi [115] have shown that, by using a SAT solver, searching for slid pairs can be more efficient than searching for slid pairs using the Gröbner basis method. Using a SAT solver to check if any loaded state in Γ_α also exists in the set of loaded states in $\Theta_{\alpha'}$ for $\alpha \geq 114$ and $\alpha' \geq 114$ is future work. The use of slid pairs in algebraic attacks, discussed in Section 4.2.7 is also an avenue for future work.

In Section 4.3, we performed an algebraic analysis using the combined methods of Berbain et al. and Raddum and we showed what conditions were required for their algebraic attack to succeed on Trivium-like ciphers. The second part of Berbain et al.'s paper discussed how their technique, when used in conjunction with correlation attacks, can be used to attack linearly filtered NLFSRs. The feasibility of their correlation attack on Trivium-like ciphers should be investigated.

A recent paper by Simpson and Boztas [106], proposed an alternative representation of Trivium, called Trivium-3D. Trivium-3D uses word-based shift registers, where each register stage holds three bits. Trivium-3D updates nine bits of internal state and outputs three bits of keystream at each iteration. In contrast, the original Trivium proposal updates three bits of internal state and outputs one bit of keystream at each iteration.

Three interesting areas of research with regards to this new word-based representation of Trivium are proposed. Firstly, analyse Trivium-3D algebraically using our approach. Secondly, investigate whether Trivium-3D can be broken using the correlation attack proposed by Berbain et al. Thirdly, analyse whether the increase in both the number of bits updated, and number of keystream bits output at each iteration for Trivium-3D increases (or decreases) the security provided.

The findings in Section 4.1 indicated that some m-tuples do not occur in sequences generated by linearly filtered NLFSRs. Whether non-occurring m-tuples are present in sequences produced by Trivium-like ciphers also remains an open question.

7.2.3 Extending our Work in Chapter 5

In Chapter 5, we analysed the Mixer stream cipher and showed that it suffers from state convergence. An area for future research is to investigate if the state convergence analysis can be extended to a key or initial state recovery attack. Another possible area of research is to investigate the m-tuple distribution of keystream sequences produced by nonlinearly filtered nonlinear feedback shift registers in general. Questions which can be investigated include how choices of nonlinear feedback functions, nonlinear Boolean functions, and tap settings to the nonlinear Boolean function affect the m-tuple distribution of the keystream produced by the keystream generator.

7.2.4 Extending our Work in Chapter 6

In Chapter 6, the state convergence problem in stream ciphers was investigated in more detail. In Section 6.4, three causes of state convergence were identified. One possible area of future research would be to investigate if stream ciphers not reviewed in this chapter also suffer from state convergence, and if so, examine if the cause of state convergence is due to any of the mechanisms we have identified. Another area of future research would be the implementation of cryptanalytic attacks on the ciphers known to suffer from state convergence, to verify the expected reductions in time, memory and data requirements which may arise from state convergence.

Appendix A

Truth Table output for the $F3$ Boolean function

Appendix A lists the output truth table for the nonlinear Boolean function *F*3 used for our experiments in Chapter 3. *F*3 is a 7-bit Boolean function used in the Pomraranch stream cipher [75].

0,1,1,1,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,0,0,0,0,1,0,1,1,1,1,1,0,0,
1,1,0,0,0,1,0,1,1,0,0,0,1,0,0,1,0,0,1,1,1,0,1,1,1,0,1,0,0,1,1,0,
1,0,1,0,1,1,0,0,0,0,1,1,0,0,1,0,0,1,1,0,1,1,1,0,0,1,0,0,0,1,1,1,
0,1,1,0,0,0,0,1,1,0,0,1,1,1,1,1,0,1,0,1,1,0,1,0,1,1,0,1,0,0,0,0

Appendix B

Experimental Results for Chapter 3

Appendix B lists the results from the analysis of the m-tuple distribution, for $m = \{2, 3, \ldots 13\}$, for Boolean functions $F1$, $F2$ and $F3$ in Chapter 3.

LFSR feedback function	m-tuple	CONS					NON-CONS									
							T1					*T2*				
		Min	Max	No. non-occurring tuples	S.D.	S.D (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)
R1	2	1791	2304	0	256.250	0.031284	2047	2048	0	0.433	0.000053	2047	2048	0	0.433	0.000053
	3	703	1216	0	192.209	0.023466	1008	1040	0	15.878	0.001938	1008	1040	0	15.878	0.001938
	4	255	640	0	128.125	0.015642	496	528	0	11.228	0.001371	494	530	0	11.393	0.001391
R2	2	1791	2304	0	256.250	0.031284	2047	2048	0	0.433	0.000053	2047	2048	0	0.433	0.000053
	3	703	1216	0	192.209	0.023466	1023	1024	0	0.331	0.000040	1023	1024	0	0.331	0.000040
	4	255	640	0	128.125	0.015642	511	512	0	0.242	0.000030	509	514	0	2.076	0.000253
R3	2	7167	9217	0	1024.250	0.031259	7679	8704	0	512.250	0.015633	8191	8192	0	0.433	0.000013
	3	2815	4865	0	768.208	0.023445	3520	4672	0	367.804	0.011225	4095	4096	0	0.331	0.000010
	4	1023	2561	0	512.125	0.015629	1632	2464	0	228.622	0.006977	2047	2048	0	0.242	0.000007
R4	2	7167	9216	0	1024.250	0.031259	8191	8192	0	0.433	0.000013	8191	8192	0	0.433	0.000013
	3	2815	4864	0	768.208	0.023445	4095	4096	0	0.331	0.000010	4095	4096	0	0.331	0.000010
	4	1023	2560	0	512.125	0.015629	2040	2056	0	7.941	0.000242	2047	2048	0	0.242	0.000007
R5	2	14335	18432	0	2048.250	0.031254	16383	16384	0	0.433	0.000007	16383	16384	0	0.433	0.000007
	3	5631	9728	0	1536.208	0.023441	8191	8192	0	0.331	0.000005	8191	8192	0	0.331	0.000005
	4	2047	5120	0	1024.125	0.015627	4080	4112	0	15.939	0.000243	4080	4112	0	15.939	0.000243
R6	2	14335	18432	0	2048.250	0.031254	16383	16384	0	0.433	0.000007	16383	16384	0	0.433	0.000007
	3	5631	9728	0	1536.208	0.023441	8191	8192	0	0.331	0.000005	8191	8192	0	0.331	0.000005
	4	2047	5120	0	1024.125	0.015627	4095	4096	0	0.242	0.000004	4095	4096	0	0.242	0.000004
R7	2	57343	73728	0	8192.250	0.031251	65535	65536	0	0.433	0.000002	65535	65536	0	0.433	0.000002
	3	22527	38912	0	6144.208	0.023438	32767	32768	0	0.331	0.000001	32767	32768	0	0.331	0.000001
	4	8191	20480	0	4096.125	0.015626	16383	16384	0	0.242	0.000001	16383	16384	0	0.242	0.000001
R8	2	57343	73728	0	8192.250	0.031251	65535	65536	0	0.433	0.000002	65535	65536	0	0.433	0.000002
	3	22527	38912	0	6144.208	0.023438	32767	32768	0	0.331	0.000001	32767	32768	0	0.331	0.000001
	4	8191	20480	0	4096.125	0.015626	16383	16384	0	0.242	0.000001	16383	16384	0	0.242	0.000001
R9	2	229375	294912	0	32768.250	0.031250	262143	262144	0	0.433	0.000000	262143	262144	0	0.433	0.000000
	3	90111	155648	0	24576.208	0.023438	131071	131072	0	0.331	0.000000	129024	133120	0	2047.875	0.001953
	4	32767	81920	0	16384.125	0.015625	65535	65536	0	0.242	0.000000	63488	67584	0	1448.066	0.001381
R10	2	229375	294912	0	32768.250	0.031250	262143	262144	0	0.433	0.000000	262143	262144	0	0.433	0.000000
	3	90111	155648	0	24576.208	0.023438	131071	131072	0	0.331	0.000000	131071	131072	0	0.331	0.000000
	4	32767	81920	0	16384.125	0.015625	65535	65536	0	0.242	0.000000	65535	65536	0	0.242	0.000000

Table 1: Distribution table for *F1* (2,3,4 bit tuples)

LFSR feedback function	m-tuple	CONS					NON-CONS									
							T1					*T2*				
		Min	Max	No. non-occurring tuples	S.D.	S.D (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)
R1	5	80	416	0	80.056	0.009774	230	290	0	17.833	0.002177	242	270	0	7.299	0.000891
	6	16	256	0	48.606	0.005934	103	154	0	12.619	0.001541	114	146	0	7.977	0.000974
	7	4	148	0	28.873	0.003525	44	87	0	8.925	0.001090	47	78	0	6.863	0.000838
R2	5	80	416	0	80.056	0.009774	236	276	0	16.333	0.001994	242	272	0	7.860	0.000960
	6	16	256	0	48.606	0.005934	101	153	0	12.560	0.001533	111	145	0	7.719	0.000942
	7	4	148	0	28.873	0.003525	44	84	0	9.089	0.001110	45	84	0	6.732	0.000822
R3	5	320	1665	0	320.056	0.009768	731	1315	0	133.982	0.004089	1005	1043	0	10.264	0.000313
	6	64	1024	0	194.345	0.005931	347	700	0	78.005	0.002381	496	530	0	8.300	0.000253
	7	16	592	0	115.458	0.003524	166	380	0	44.267	0.001351	235	277	0	9.466	0.000289
R4	5	320	1664	0	320.056	0.009768	1007	1040	0	7.998	0.000244	980	1068	0	26.226	0.000800
	6	64	1024	0	194.345	0.005931	484	544	0	17.361	0.000530	464	562	0	22.664	0.000692
	7	16	592	0	115.458	0.003524	224	293	0	15.493	0.000473	213	297	0	16.530	0.000504
R5	5	640	3328	0	640.056	0.009767	2016	2096	0	22.628	0.000345	2012	2084	0	19.957	0.000305
	6	128	2048	0	388.664	0.005931	962	1086	0	26.131	0.000399	984	1056	0	15.796	0.000241
	7	32	1184	0	230.905	0.003523	462	561	0	19.939	0.000304	461	559	0	18.176	0.000277
R6	5	640	3328	0	640.056	0.009767	2029	2066	0	11.539	0.000176	2038	2058	0	8.271	0.000126
	6	128	2048	0	388.664	0.005931	1005	1049	0	9.765	0.000149	985	1065	0	18.788	0.000287
	7	32	1184	0	230.905	0.003523	483	538	0	10.089	0.000154	460	560	0	18.699	0.000285
R7	5	2560	13312	0	2560.056	0.009766	8184	8200	0	7.971	0.000030	8112	8272	0	65.932	0.000252
	6	512	8192	0	1554.580	0.005930	3985	4215	0	58.325	0.000222	3924	4268	0	66.576	0.000254
	7	128	4736	0	923.586	0.003523	1853	2258	0	88.290	0.000337	1922	2178	0	44.703	0.000171
R8	5	2560	13312	0	2560.056	0.009766	8128	8256	0	63.969	0.000244	8175	8208	0	16.032	0.000061
	6	512	8192	0	1554.580	0.005930	3900	4284	0	87.166	0.000333	4058	4134	0	20.095	0.000077
	7	128	4736	0	923.586	0.003523	1882	2189	0	64.085	0.000244	2002	2107	0	20.461	0.000078
R9	5	10240	53248	0	10240.056	0.009766	32255	33280	0	512.031	0.000488	30944	34592	0	923.516	0.000881
	6	2048	32768	0	6218.244	0.005930	15808	16960	0	367.671	0.000351	15056	17583	0	544.201	0.000519
	7	512	18944	0	3694.313	0.003523	7648	8768	0	273.423	0.000261	7448	8983	0	310.541	0.000296
R10	5	10240	53248	0	10240.056	0.009766	32608	32928	0	131.962	0.000126	32767	32768	0	0.174	0.000000
	6	2048	32768	0	6218.244	0.005930	16208	16560	0	94.671	0.000090	16000	16768	0	383.984	0.000366
	7	512	18944	0	3694.313	0.003523	8058	8338	0	60.767	0.000058	7770	8614	0	272.083	0.000259

Table 2: Distribution table for *F1* (5,6,7 bit tuples)

LFSR feedback function	m-tuple	CONS					NON-CONS										
							T1					*T2*					
		Min	Max	No. non-occurring tuples	S.D.	S.D (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)
R1	8	0	86	1	16.772	0.002048	19	51	0	5.985	0.000731	17	45	0	5.389	0.000658
	9	0	51	6	9.571	0.001168	6	30	0	4.157	0.000507	6	27	0	3.966	0.000484
	10	0	31	32	5.467	0.000667	1	18	0	2.885	0.000352	1	18	0	2.834	0.000346
R2	8	0	86	1	16.772	0.002048	17	50	0	6.148	0.000751	20	47	0	5.203	0.000635
	9	0	51	6	9.571	0.001168	5	29	0	4.222	0.000515	6	30	0	3.722	0.000454
	10	0	31	31	5.464	0.000667	1	19	0	2.952	0.000360	0	20	0	2.695	0.000329
R3	8	0	344	1	67.072	0.002047	72	209	0	25.263	0.000771	109	149	0	8.101	0.000247
	9	0	204	6	38.277	0.001168	22	120	0	15.297	0.000467	46	80	0	6.257	0.000191
	10	0	122	23	21.493	0.000656	6	70	0	9.221	0.000281	15	50	0	5.078	0.000155
R4	8	0	344	1	67.072	0.002047	103	157	0	10.895	0.000333	94	167	0	12.401	0.000378
	9	0	204	6	38.277	0.001168	42	90	0	8.225	0.000251	43	90	0	8.257	0.000252
	10	0	122	23	21.493	0.000656	14	55	0	5.815	0.000177	16	53	0	5.997	0.000183
R5	8	0	688	1	134.139	0.002047	214	301	0	15.502	0.000237	213	297	0	16.060	0.000245
	9	0	408	6	76.553	0.001168	94	163	0	11.628	0.000177	92	161	0	11.864	0.000181
	10	0	244	23	42.985	0.000656	41	89	0	8.370	0.000128	40	90	0	8.376	0.000128
R6	8	0	688	1	134.139	0.002047	226	294	0	11.147	0.000170	210	301	0	13.686	0.000209
	9	0	408	6	76.553	0.001168	105	163	0	9.271	0.000141	99	158	0	9.626	0.000147
	10	0	244	23	42.985	0.000656	44	88	0	7.254	0.000111	42	85	0	6.983	0.000107
R7	8	0	2752	1	536.542	0.002047	883	1192	0	64.860	0.000247	955	1105	0	27.813	0.000106
	9	0	1632	6	306.205	0.001168	385	650	0	43.728	0.000167	461	580	0	20.653	0.000079
	10	0	976	23	171.937	0.000656	167	373	0	29.149	0.000111	214	319	0	16.127	0.000062
R8	8	0	2752	1	536.542	0.002047	891	1130	0	42.693	0.000163	975	1066	0	17.374	0.000066
	9	0	1632	6	306.205	0.001168	423	600	0	28.488	0.000109	469	555	0	15.926	0.000061
	10	0	976	23	171.937	0.000656	203	320	0	19.132	0.000073	203	298	0	13.616	0.000052
R9	8	0	11008	1	2416.153	0.002047	3600	4784	0	210.909	0.000201	3572	4616	0	176.886	0.000169
	9	0	6528	6	1224.812	0.001168	1616	2584	0	136.766	0.000130	1740	2333	0	101.051	0.000096
	10	0	3904	23	687.745	0.000656	780	1352	0	83.062	0.000079	829	1204	0	59.002	0.000056
R10	8	0	11008	1	2146.153	0.002047	3894	4292	0	81.768	0.000078	3707	4512	0	175.030	0.000167
	9	0	6528	6	1224.812	0.001168	1886	2227	0	60.558	0.000058	1737	2381	0	106.320	0.000101
	10	0	3904	23	687.745	0.000656	918	1136	0	39.530	0.000038	802	1263	0	65.158	0.000062

Table 3: Distribution table for *F1* (8,9,10 bit tuples)

LFSR feedback function	m-tuple	CONS					NON-CONS									
							T1					*T2*				
		Min	Max	No. non-occurring tuples	S.D.	S.D (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)
R1	11	0	19	204	3.230	0.000394	0	12	41	2.007	0.000245	0	11	42	1.977	0.000241
	12	0	16	1011	2.004	0.000245	0	9	550	1.421	0.000173	0	9	530	1.392	0.000170
	13	0	10	3719	1.266	0.000155	0	7	3018	1.002	0.000122	0	7	2972	0.991	0.000121
R2	11	0	19	209	3.173	0.000387	0	13	45	2.070	0.000253	0	13	36	1.928	0.000235
	12	0	13	984	1.909	0.000233	0	9	586	1.459	0.000178	0	8	507	1.359	0.000166
	13	0	8	3597	1.203	0.000147	0	7	3087	1.028	0.000125	0	6	2913	0.973	0.000119
R3	11	0	74	79	11.909	0.000363	1	42	0	5.747	0.000175	5	30	0	3.821	0.000117
	12	0	49	299	6.596	0.000201	0	26	11	3.643	0.000111	0	19	1	2.755	0.000084
	13	0	28	1244	3.721	0.000114	0	18	244	2.317	0.000071	0	13	155	1.981	0.000060
R4	11	0	74	79	11.909	0.000363	2	35	0	4.389	0.000134	3	31	0	4.303	0.000131
	12	0	48	292	6.608	0.000202	0	22	6	3.055	0.000093	0	22	2	2.987	0.000091
	13	0	27	1210	3.775	0.000115	0	15	182	2.086	0.000064	0	14	154	2.054	0.000063
R5	11	0	148	79	23.817	0.000363	14	54	0	5.881	0.000090	14	55	0	5.894	0.000090
	12	0	94	240	13.068	0.000199	4	35	0	4.110	0.000063	4	37	0	4.217	0.000064
	13	0	53	751	7.152	0.000109	0	21	3	2.858	0.000044	0	23	6	2.932	0.000045
R6	11	0	148	79	23.817	0.000363	14	54	0	5.323	0.000081	16	51	0	5.192	0.000079
	12	0	94	240	13.068	0.000199	3	35	0	3.908	0.000060	4	30	0	3.824	0.000058
	13	0	59	777	7.189	0.000110	0	21	1	2.798	0.000043	0	20	1	2.772	0.000042
R7	11	0	592	79	95.266	0.000363	76	217	0	18.982	0.000072	89	177	0	11.747	0.000045
	12	0	376	240	52.273	0.000199	29	127	0	12.076	0.000046	37	94	0	8.260	0.000032
	13	0	222	668	28.458	0.000109	8	75	0	7.688	0.000029	12	54	0	5.753	0.000022
R8	11	0	592	79	95.266	0.000363	89	172	0	13.357	0.000051	98	160	0	10.369	0.000040
	12	0	376	240	52.273	0.000199	35	100	0	9.184	0.000035	38	91	0	7.653	0.000029
	13	0	222	668	28.458	0.000109	13	57	0	6.227	0.000024	13	55	0	5.533	0.000021
R9	11	0	2368	79	381.062	0.000363	344	716	0	48.442	0.000046	392	637	0	36.032	0.000034
	12	0	1504	240	209.091	0.000199	164	380	0	27.685	0.000026	182	343	0	22.633	0.000022
	13	0	888	668	113.832	0.000109	69	208	0	17.958	0.000017	81	185	0	14.614	0.000014
R10	11	0	2368	79	381.062	0.000363	435	603	0	25.343	0.000024	382	654	0	40.131	0.000038
	12	0	1504	240	209.091	0.000199	203	321	0	16.594	0.000016	180	353	0	24.896	0.000024
	13	0	888	668	113.832	0.000109	85	170	0	11.699	0.000011	78	205	0	15.543	0.000015

Table 4: Distribution table for *F1* (11, 12, 13 bit tuple)

LFSR feedback function	m-tuple	CONS					NON-CONS									
							T1					*T2*				
		Min	Max	No. non-occurring tuples	S.D.	S.D (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)
R1	2	1920	2176	0	127.750	0.015596	2047	2048	0	0.433	0.000053	2047	2048	0	0.433	0.000053
	3	864	1183	0	95.792	0.011695	1023	1024	0	0.331	0.000040	984	1064	0	32.835	0.004009
	4	368	655	0	76.616	0.009354	480	544	0	19.597	0.002393	476	555	0	23.549	0.002875
R2	2	1920	2176	0	127.750	0.015596	2047	2048	0	0.433	0.000053	2047	2048	0	0.433	0.000053
	3	864	1183	0	95.792	0.011695	1007	1040	0	16.128	0.001969	992	1056	0	31.876	0.003892
	4	368	655	0	76.616	0.009354	492	548	0	19.989	0.002440	484	548	0	22.881	0.002793
R3	2	7680	8704	0	511.750	0.015618	8191	8192	0	0.433	0.000013	8191	8192	0	0.433	0.000013
	3	3456	4735	0	383.792	0.011713	3936	4256	0	131.788	0.004022	4095	4096	0	0.331	0.000010
	4	1472	2623	0	306.816	0.009364	1888	2239	0	98.509	0.003006	2047	2048	0	0.242	0.000007
R4	2	7680	8704	0	511.750	0.015618	8191	8192	0	0.433	0.000013	8191	8192	0	0.433	0.000013
	3	3456	4735	0	383.792	0.011713	3936	4256	0	131.788	0.004022	4095	4096	0	0.331	0.000010
	4	1472	2623	0	306.816	0.009364	1824	2240	0	113.084	0.003451	2015	2080	0	32.063	0.000979
R5	2	15360	17408	0	1023.750	0.015621	16383	16384	0	0.433	0.000007	16383	16384	0	0.433	0.000007
	3	6912	9471	0	767.792	0.011716	7679	8704	0	512.125	0.007815	8191	8192	0	0.331	0.000005
	4	2944	5247	0	613.749	0.009365	3520	4673	0	367.728	0.005611	3936	4256	0	129.923	0.001983
R6	2	15360	17408	0	1023.750	0.015621	16383	16384	0	0.433	0.000007	16383	16384	0	0.433	0.000007
	3	6912	9471	0	767.792	0.011716	8191	8192	0	0.331	0.000005	8160	8224	0	31.876	0.000486
	4	2944	5247	0	613.749	0.009365	4063	4128	0	32.063	0.000489	3920	4304	0	134.752	0.002056
R7	2	61440	69632	0	4095.750	0.015624	65535	65536	0	0.433	0.000002	65535	65536	0	0.433	0.000002
	3	27648	37887	0	3071.792	0.011718	32767	32768	0	0.331	0.000001	30976	34560	0	1279.825	0.004882
	4	11776	20991	0	2455.348	0.009366	16383	16384	0	0.242	0.000001	14400	17983	0	942.673	0.003596
R8	2	61440	69632	0	4095.750	0.015624	65535	65536	0	0.433	0.000002	65535	65536	0	0.433	0.000002
	3	27648	37887	0	3071.792	0.011718	32767	32768	0	0.331	0.000001	31488	34048	0	1055.363	0.004026
	4	11776	20991	0	2455.348	0.009366	16383	16384	0	0.242	0.000001	14944	17632	0	771.235	0.002942
R9	2	245760	278528	0	16383.750	0.015625	262143	262144	0	0.433	0.000000	262143	262144	0	0.433	0.000000
	3	110592	151551	0	12287.792	0.011719	131071	131072	0	0.331	0.000000	125952	136192	0	4221.909	0.004026
	4	47104	83967	0	9821.746	0.009367	64256	66816	0	1055.561	0.001007	59904	70144	0	3028.938	0.002889
R10	2	245760	278528	0	16383.750	0.015625	262143	262144	0	0.433	0.000000	262143	262144	0	0.433	0.000000
	3	110592	151551	0	12287.792	0.011719	131071	131072	0	0.331	0.000000	125952	136192	0	4221.909	0.004026
	4	47104	83967	0	9821.746	0.009367	65535	65536	0	0.242	0.000000	59904	70144	0	3028.938	0.002889

Table 5: Distribution table for *F2* (2,3,4 bit tuples)

LFSR feedback function	m-tuple	CONS					NON-CONS									
							T1					*T2*				
		Min	Max	No. non-occurring tuples	S.D.	S.D (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)
R1	5	136	336	0	54.3800	0.006690	212	308	0	25.437	0.003105	226	309	0	18.348	0.002240
	6	44	204	0	36.315	0.004434	89	175	0	18.874	0.002304	91	155	0	13.881	0.001695
	7	16	122	0	22.242	0.002715	35	98	0	12.718	0.001553	43	86	0	9.168	0.001119
R2	5	136	336	0	54.800	0.006690	206	314	0	25.655	0.003132	220	297	0	18.314	0.002236
	6	44	204	0	36.315	0.004434	88	175	0	18.347	0.002240	104	151	0	13.106	0.001600
	7	16	122	0	22.242	0.002715	35	97	0	12.104	0.001478	43	87	0	8.851	0.001081
R3	5	544	1344	0	219.335	0.006697	920	1239	0	71.236	0.002174	996	1052	0	18.324	0.000559
	6	176	816	0	145.310	0.004435	451	690	0	44.625	0.001362	456	558	0	21.471	0.000655
	7	67	488	0	88.987	0.002716	189	389	0	29.324	0.000895	201	203	0	18.231	0.000556
R4	5	544	1344	0	219.335	0.006697	848	1184	0	82.317	0.002512	784	1232	0	102.465	0.003127
	6	176	816	0	145.310	0.004435	410	634	0	51.776	0.001580	352	800	0	76.536	0.002336
	7	64	488	0	88.987	0.002716	181	345	0	33.495	0.001022	159	474	0	49.283	0.001504
R5	5	1088	2688	0	438.716	0.006694	1480	2632	0	260.116	0.003969	1828	2228	0	104.035	0.001587
	6	352	1632	0	290.637	0.004435	592	1520	0	166.378	0.002539	856	1220	0	83.546	0.001275
	7	128	976	0	177.982	0.002716	237	803	0	100.244	0.001530	396	638	0	54.050	0.000825
R6	5	1088	2688	0	438.716	0.006694	1944	2128	0	49.644	0.000758	1798	2230	0	106.416	0.001624
	6	352	1632	0	290.637	0.004435	926	1150	0	49.829	0.000760	841	1177	0	83.802	0.001279
	7	128	976	0	177.982	0.002716	423	581	0	33.979	0.000518	398	645	0	54.112	0.000826
R7	5	4352	10752	0	1755.002	0.006695	8032	8352	0	95.990	0.000366	6992	9360	0	593.674	0.002265
	6	1408	6528	0	1162.601	0.004435	3832	4280	0	81.971	0.000313	3392	4752	0	355.949	0.001358
	7	512	3904	0	711.948	0.002716	1840	2304	0	94.994	0.000362	1514	2501	0	210.097	0.000801
R8	5	4352	10752	0	1755.002	0.006695	8175	8208	0	16.032	0.000061	7160	9048	0	478.948	0.001827
	6	1408	6528	0	1162.601	0.004435	4064	4144	0	19.609	0.000075	3480	4688	0	287.135	0.001095
	7	512	3904	0	711.948	0.002716	1930	2166	0	69.253	0.000264	1706	2467	0	166.347	0.000635
R9	5	17408	43008	0	7020.145	0.006695	30912	34368	0	907.368	0.000865	28672	36096	0	1872.423	0.001786
	6	5632	26112	0	4650.454	0.004435	14336	17728	0	806.993	0.000770	13984	18847	0	1105.934	0.001055
	7	2048	15616	0	2847.813	0.002716	6928	9744	0	628.072	0.000599	6688	9919	0	633.344	0.000604
R10	5	17408	43008	0	7020.145	0.006695	32480	33056	0	257.965	0.000246	28752	36176	0	1868.661	0.001782
	6	5632	26112	0	4650.454	0.004435	15432	17287	0	552.438	0.000527	14072	18631	0	1091.873	0.001041
	7	2048	15616	0	2847.813	0.002716	7338	9365	0	470.134	0.000448	6654	9781	0	619.964	0.000591

Table 6: Distribution table for *F2* (5,6,7 bit tuples)

LFSR feedback function	m-tuple	CONS					NON-CONS									
							T1					*T2*				
		Min	Max	No. non-occurring tuples	S.D.	S.D (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)
R1	8	4	65	0	13.144	0.001605	13	55	0	8.235	0.001005	16	47	0	6.130	0.000748
	9	0	41	3	7.612	0.000929	4	34	0	5.257	0.000642	4	28	0	4.315	0.000527
	10	0	23	20	4.453	0.000544	0	21	3	3.433	0.000419	0	17	3	2.955	0.000361
R2	8	4	65	0	13.144	0.001605	13	50	0	7.712	0.000942	18	48	0	5.916	0.000722
	9	0	43	3	7.600	0.000928	4	34	0	5.176	0.000632	5	32	0	4.295	0.000524
	10	0	23	26	4.407	0.000538	0	19	1	3.389	0.000414	1	17	0	3.017	0.000368
R3	8	16	260	0	52.585	0.001605	84	228	0	19.052	0.000581	78	165	0	12.980	0.000396
	9	0	168	3	30.194	0.000921	32	138	0	12.019	0.000367	28	92	0	9.239	0.000282
	10	0	91	17	17.023	0.000520	13	85	0	7.845	0.000239	10	52	0	6.377	0.000195
R4	8	16	260	0	52.585	0.001605	77	199	0	21.110	0.000644	61	274	0	30.385	0.000927
	9	0	168	3	30.194	0.000921	29	109	0	12.965	0.000396	22	172	0	18.407	0.000562
	10	0	91	17	17.023	0.000520	10	64	0	8.375	0.000256	10	93	0	10.923	0.000333
R5	8	32	520	0	105.173	0.001605	82	438	0	58.361	0.000891	181	359	0	33.627	0.000513
	9	0	336	3	60.389	0.000921	33	234	0	33.437	0.000510	73	199	0	20.893	0.000319
	10	0	182	17	34.047	0.000520	12	142	0	19.703	0.000301	26	104	0	13.012	0.000199
R6	8	32	520	0	105.173	0.001605	197	317	0	22.883	0.000349	169	348	0	33.154	0.000506
	9	0	336	3	60.389	0.000921	92	182	0	15.111	0.000231	73	183	0	20.214	0.000308
	10	0	182	17	34.047	0.000520	37	102	0	10.328	0.000158	25	100	0	12.342	0.000188
R7	8	128	2080	0	420.701	0.001605	785	1257	0	83.011	0.000317	705	1349	0	123.493	0.000471
	9	0	1344	0	241.560	0.000921	351	691	0	56.403	0.000215	332	769	0	74.321	0.000284
	10	0	728	17	136.190	0.000520	154	382	0	35.718	0.000136	113	435	0	45.149	0.000172
R8	8	128	2080	0	420.701	0.001605	883	1195	0	62.774	0.000239	800	1266	0	95.444	0.000364
	9	0	1344	3	241.560	0.000921	420	627	0	42.630	0.000163	348	662	0	56.295	0.000215
	10	0	728	17	136.190	0.000520	184	340	0	26.877	0.000103	162	361	0	33.831	0.000129
R9	8	512	8320	0	1682.814	0.001605	3080	5415	0	412.356	0.000393	3218	5161	0	357.407	0.000341
	9	0	5376	3	966.245	0.000921	1370	3013	0	252.397	0.000241	1443	2666	0	203.722	0.000194
	10	0	2912	17	544.764	0.000520	577	1638	0	155.132	0.000148	636	1390	0	118.380	0.000113
R10	8	512	8320	0	1682.814	0.001605	3367	5138	0	314.488	0.000300	3221	5096	0	344.841	0.000329
	9	0	5376	3	966.245	0.000921	1502	2811	0	191.075	0.000182	1520	2640	0	190.614	0.000182
	10	0	2912	17	544.764	0.000520	667	1487	0	115.196	0.000110	711	1395	0	107.391	0.000102

Table 7: Distribution table for *F2* (8,9,10 bit tuples)

LFSR feedback function	m-tuple	CONS					NON-CONS									
							T1					*T2*				
		Min	Max	No. non-occurring tuples	S.D.	S.D (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)
R1	11	0	16	161	2.657	0.000324	0	14	74	2.299	0.000281	0	12	56	2.057	0.000251
	12	0	10	843	1.644	0.000201	0	10	668	1.556	0.000190	0	9	601	1.436	0.000175
	13	0	7	3346	1.086	0.000133	0	7	3161	1.057	0.000129	0	6	3040	1.002	0.000122
R2	11	0	14	162	2.647	0.000323	0	12	61	2.242	0.000274	0	14	44	2.088	0.000255
	12	0	10	806	1.643	0.000201	0	10	650	1.526	0.000186	0	10	580	1.456	0.000178
	13	0	6	3324	1.084	0.000132	0	8	3158	1.050	0.000128	0	7	3075	1.022	0.000125
R3	11	0	51	68	9.453	0.000288	4	51	0	5.123	0.000156	2	34	0	4.425	0.000135
	12	0	31	242	5.257	0.000160	0	31	8	3.339	0.000102	0	21	4	3.076	0.000094
	13	0	18	980	3.028	0.000092	0	18	244	2.230	0.000068	0	15	219	2.130	0.000065
R4	11	0	52	68	9.461	0.000289	2	41	0	5.502	0.000168	0	53	1	6.564	0.000200
	12	0	32	234	5.239	0.000160	0	26	7	3.619	0.000110	0	34	24	4.085	0.000125
	13	0	20	936	3.078	0.000094	0	17	260	2.387	0.00073	0	23	354	2.570	0.000078
R5	11	0	103	66	18.882	0.000288	1	86	0	11.499	0.000175	9	64	0	8.110	0.000124
	12	0	63	207	10.410	0.000159	0	49	1	6.835	0.000104	3	38	0	5.169	0.000079
	13	0	35	682	5.775	0.000088	0	33	53	4.211	0.000064	0	24	11	3.360	0.000051
R6	11	0	103	66	18.882	0.000288	13	59	0	6.914	0.000106	9	57	0	7.598	0.000116
	12	0	59	209	10.471	0.000160	4	36	0	4.606	0.000070	2	38	0	4.847	0.000074
	13	0	37	724	5.838	0.000089	0	24	9	3.178	0.000048	0	23	6	3.221	0.000049
R7	11	0	412	66	75.528	0.000288	66	209	0	21.949	0.000084	41	234	0	27.148	0.000104
	12	0	240	205	41.456	0.000158	21	122	0	13.541	0.000052	18	134	0	17.138	0.000065
	13	0	146	579	22.540	0.000086	9	67	0	8.347	0.000032	3	87	0	10.604	0.000040
R8	11	0	412	66	75.528	0.000288	82	180	0	16.690	0.000064	62	206	0	20.278	0.000077
	12	0	240	205	41.456	0.000158	31	103	0	10.477	0.000040	23	123	0	13.141	0.000050
	13	0	146	579	22.540	0.000086	12	61	0	6.746	0.000026	8	78	0	8.319	0.000032
R9	11	0	1648	0	302.114	0.000288	256	882	0	91.641	0.000087	308	788	0	70.734	0.000067
	12	0	960	205	165.823	0.000158	106	483	0	56.768	0.000054	129	475	0	41.907	0.000040
	13	0	584	579	90.161	0.000086	41	284	0	34.374	0.000033	50	283	0	26.610	0.000025
R10	11	0	1648	66	302.114	0.000288	309	779	0	67.017	0.000064	314	744	0	62.724	0.000060
	12	0	960	205	165.823	0.000158	126	433	0	42.825	0.000041	142	410	0	36.351	0.000035
	13	0	584	579	90.161	0.000086	49	240	0	26.370	0.000025	64	255	0	23.290	0.000022

Table 8: Distribution table for *F2* (11, 12, 13 bit tuple)

LFSR feedback function	m-tuple	CONS					NON-CONS									
							T1					*T2*				
		Min	Max	No. non-occurring tuples	S.D.	S.D (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)
R1	2	1920	2176	0	127.750	0.015596	2047	2048	0	0.433	0.000053	2047	2048	0	0.433	0.000053
	3	848	1135	0	97.181	0.011864	990	1058	0	31.947	0.003900	990	1058	0	31.931	0.003898
	4	376	615	0	67.075	0.008189	460	557	0	26.350	0.003217	471	548	0	23.546	0.002875
R2	2	1920	2176	0	127.750	0.015596	2015	2080	0	32.252	0.003937	2016	2080	0	31.752	0.003876
	3	848	1135	0	97.181	0.011864	986	1094	0	35.364	0.004317	990	1090	0	39.046	0.004767
	4	376	615	0	67.075	0.008189	464	556	0	26.843	0.003277	462	568	0	28.334	0.003459
R3	2	7680	8704	0	511.750	0.015618	8191	8192	0	0.433	0.000013	8191	8192	0	0.433	0.000013
	3	3392	4543	0	389.153	0.011876	4095	4096	0	0.331	0.000010	4087	4104	0	8.131	0.000248
	4	1504	2463	0	268.589	0.008197	2010	2086	0	26.025	0.000794	1990	2098	0	34.969	0.001067
R4	2	7680	8704	0	511.750	0.015618	8064	8320	0	127.750	0.003899	8191	8192	0	0.433	0.000013
	3	3392	4543	0	389.153	0.011876	3968	4224	0	90.333	0.002757	4088	4104	0	7.881	0.000241
	4	1504	2463	0	286.589	0.008197	1914	2145	0	60.198	0.001837	2011	2077	0	19.607	0.000598
R5	2	15360	17408	0	1023.750	0.015621	16383	16384	0	0.433	0.000007	16383	16384	0	0.433	0.000007
	3	6784	9087	0	778.450	0.011878	8191	8192	0	0.331	0.000005	8175	8208	0	16.128	0.000246
	4	3008	4927	0	537.725	0.008198	3989	4203	0	75.554	0.001153	3959	4217	0	76.380	0.001165
R6	2	15360	17408	0	1023.750	0.015621	16383	16384	0	0.433	0.000007	16128	16640	0	255.750	0.003902
	3	6784	9087	0	778.450	0.011878	8143	8240	0	48.126	0.000734	7904	8479	0	183.360	0.002802
	4	3008	4927	0	537.275	0.008198	3934	4210	0	83.065	0.001267	3899	4346	0	114.655	0.001750
R7	2	61440	69632	0	4095.750	0.015624	65535	65536	0	0.433	0.000002	65535	65536	0	0.433	0.000002
	3	27136	36351	0	3114.231	0.011880	32767	32768	0	0.331	0.000001	32767	32768	0	0.331	0.000001
	4	12032	19711	0	2149.389	0.08199	16372	16396	0	11.940	0.000046	16267	16500	0	74.395	0.000284
R8	2	61440	69632	0	4095.750	0.015624	65535	65536	0	0.433	0.000002	65535	65536	0	0.433	0.000002
	3	27136	36351	0	3114.231	0.011880	32767	32768	0	0.331	0.000001	32576	32960	0	191.875	0.000732
	4	12032	19711	0	2149.389	0.008199	16383	16384	0	0.242	0.000001	16088	16680	0	156.879	0.000598
R9	2	245760	278528	0	16383.750	0.015625	262143	262144	0	0.433	0.000000	258047	266240	0	4096.250	0.003906
	3	108544	145407	0	12457.354	0.011880	130816	131328	0	255.875	0.000244	126975	135168	0	2896.486	0.002762
	4	48128	78847	0	8597.845	0.008200	62368	68448	0	2136.422	0.002037	62511	68656	0	1774.376	0.001692
R10	2	245760	278528	0	16383.750	0.015625	262143	262144	0	0.433	0.000000	262143	262144	0	0.433	0.000000
	3	108544	145407	0	12457.354	0.011880	131071	131072	0	0.331	0.000000	131071	131072	0	0.331	0.000000
	4	48128	78847	0	8597.845	0.008200	65312	65760	0	160.013	0.000153	65471	65600	0	64.063	0.000061

Table 9: Distribution table for *F3* (2,3,4 bit tuples)

LFSR feedback function	m-tuple	CONS					NON-CONS									
							T1					*T2*				
		Min	Max	No. non-occurring tuples	S.D.	S.D (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)
R1	5	176	356	0	42.977	0.005247	199	304	0	19.773	0.002414	217	287	0	17.793	0.002172
	6	84	200	0	25.952	0.003168	96	165	0	13.255	0.001618	94	155	0	12.511	0.001527
	7	37	107	0	15.090	0.001842	42	94	0	9.329	0.001139	38	85	0	8.552	0.001044
R2	5	176	356	0	42.977	0.005247	222	311	0	19.819	0.002420	225	302	0	18.612	0.002272
	6	84	200	0	25.952	0.003168	99	166	0	13.349	0.001630	102	156	0	11.851	0.001447
	7	37	107	0	15.090	0.001842	46	87	0	8.655	0.001057	48	86	0	8.013	0.000978
R3	5	704	1424	0	172.081	0.005252	982	1081	0	25.816	0.000788	981	1076	0	26.990	0.000824
	6	336	800	0	103.897	0.003171	480	559	0	18.383	0.000561	478	575	0	19.895	0.000607
	7	148	428	0	60.408	0.001844	220	290	0	14.565	0.000444	228	301	0	14.887	0.000454
R4	5	704	1424	0	172.081	0.005252	937	1103	0	41.745	0.001274	989	1062	0	21.262	0.000649
	6	336	800	0	103.894	0.003171	445	565	0	26.610	0.000812	471	543	0	17.379	0.000530
	7	148	428	0	60.408	0.001844	202	301	0	17.555	0.000536	212	295	0	15.558	0.000475
R5	5	1408	2848	0	334.221	0.005252	1926	2203	0	62.643	0.000956	1946	2166	0	55.637	0.000849
	6	672	1600	0	207.817	0.003171	932	1134	0	42.299	0.000645	954	1098	0	36.066	0.000550
	7	296	856	0	120.831	0.001844	443	579	0	29.533	0.000451	442	572	0	25.480	0.000389
R6	5	1408	2848	0	344.221	0.005252	1908	2143	0	59.277	0.000905	1922	2207	0	69.419	0.001059
	6	672	1600	0	207.817	0.003171	913	1100	0	39.210	0.000598	938	1128	0	41.864	0.000639
	7	296	856	0	120.831	0.001844	442	572	0	25.577	0.000390	455	603	0	26.019	0.000397
R7	5	5632	11392	0	1377.058	0.005253	8111	8285	0	47.568	0.000181	8063	8340	0	65.300	0.000249
	6	2688	6400	0	831.356	0.003171	3965	4183	0	46.534	0.000178	3968	4273	0	50.730	0.000194
	7	1184	3424	0	483.372	0.001844	1960	2158	0	39.926	0.000152	1940	2184	0	40.198	0.000153
R8	5	5632	11392	0	1377.058	0.005253	8122	8262	0	40.403	0.000154	7965	8415	0	108.488	0.000414
	6	2688	6400	0	831.356	0.003171	3980	4176	0	40.491	0.000154	3867	4263	0	71.141	0.000271
	7	1184	3424	0	483.372	0.001844	1964	2138	0	32.524	0.000124	1897	2176	0	44.984	0.000172
R9	5	22528	45568	0	5508.405	0.005253	29947	35580	0	1517.746	0.001447	30783	34896	0	1027.430	0.000980
	6	10752	25600	0	3325.509	0.003171	14539	18756	0	943.309	0.000900	14832	18143	0	660.416	0.000630
	7	4736	13696	0	1933.536	0.001844	6824	9730	0	555.135	0.000529	7067	9345	0	398.233	0.000380
R10	5	22528	45568	0	5508.405	0.005253	32489	33063	0	123.809	0.000118	32665	32887	0	54.749	0.000052
	6	10752	25600	0	3325.509	0.003171	16121	16620	0	105.154	0.000100	16031	16730	0	260.381	0.000248
	7	4736	13696	0	1933.536	0.001844	8025	8400	0	74.757	0.000071	7858	8566	0	189.267	0.000180

Table 10: Distribution table for *F3* (5,6,7 bit tuples)

LFSR feedback function	m-tuple	CONS					NON-CONS									
							T1					*T2*				
		Min	Max	No. non-occurring tuples	S.D.	S.D (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)
R1	8	14	59	0	8.926	0.001090	18	54	0	6.277	0.000766	13	49	0	5.885	0.000719
	9	5	33	0	5.347	0.000653	6	32	0	4.346	0.000531	3	28	0	4.008	0.000489
	10	1	23	0	3.389	0.000414	0	18	1	2.964	0.000362	1	17	0	2.782	0.000340
R2	8	12	63	0	8.765	0.001070	18	47	0	5.793	0.000707	19	50	0	5.663	0.000691
	9	6	37	0	5.279	0.000644	5	30	0	4.027	0.000492	5	33	0	3.999	0.000488
	10	0	23	1	3.355	0.000410	1	17	0	2.817	0.000344	1	22	0	2.846	0.000347
R3	8	58	228	0	34.430	0.001051	90	154	0	11.039	0.000337	102	167	0	10.849	0.000331
	9	22	121	0	19.253	0.000588	40	87	0	7.933	0.000242	45	92	0	7.742	0.000236
	10	7	72	0	10.811	0.000330	14	49	0	5.662	0.000173	16	52	0	5.592	0.000171
R4	8	58	228	0	34.430	0.001051	81	158	0	11.761	0.000359	95	157	0	11.646	0.000355
	9	22	121	0	19.253	0.000588	31	89	0	8.124	0.000248	45	93	0	8.344	0.000255
	10	9	75	0	10.866	0.000332	13	50	0	5.719	0.000175	16	50	0	5.865	0.000179
R5	8	116	456	0	68.868	0.001051	206	304	0	20.166	0.000308	209	298	0	17.707	0.000270
	9	44	242	0	38.510	0.000588	90	167	0	13.464	0.000205	96	162	0	12.166	0.000186
	10	17	144	0	21.282	0.000325	40	100	0	8.997	0.000137	38	88	0	8.422	0.000129
R6	8	116	456	0	68.868	0.001051	213	300	0	16.853	0.000257	202	322	0	17.163	0.000262
	9	44	242	0	38.510	0.000588	98	165	0	11.274	0.000172	92	173	0	11.676	0.000178
	10	17	144	0	21.282	0.000325	42	90	0	8.028	0.000122	41	93	0	8.068	0.000123
R7	8	464	1824	0	275.498	0.001051	943	1109	0	31.966	0.000122	950	1134	0	31.871	0.000122
	9	176	968	0	154.054	0.000588	441	586	0	25.601	0.000098	431	595	0	23.351	0.000089
	10	68	576	0	85.135	0.000325	201	318	0	18.480	0.000070	204	320	0	16.483	0.000063
R8	8	464	1824	0	275.498	0.001051	950	1139	0	28.871	0.000110	927	1105	0	32.782	0.000125
	9	176	968	0	154.054	0.000588	451	581	0	22.730	0.000087	427	588	0	23.031	0.000088
	10	68	576	0	85.135	0.000325	201	306	0	16.333	0.000062	196	308	0	16.002	0.000061
R9	8	1856	7296	0	1102.019	0.001051	3226	5052	0	315.350	0.000301	3388	4906	0	230.709	0.000220
	9	704	3872	0	616.230	0.000588	1521	2589	0	176.842	0.000169	1617	2483	0	130.778	0.000125
	10	272	2304	0	340.548	0.000325	689	1352	0	100.163	0.000096	768	1298	0	74.992	0.000072
R10	8	1856	7296	0	1102.019	0.001051	3916	4272	0	58.125	0.000056	3830	4383	0	120.261	0.000115
	9	704	3872	0	616.230	0.000588	1935	2201	0	42.025	0.000040	1841	2256	0	74.240	0.000071
	10	272	2304	0	340.548	0.000325	913	1133	0	30.240	0.000029	865	1159	0	45.913	0.000044

Table 11: Distribution table for *F3* (8,9,10 bit tuples)

LFSR feedback function	m-tuple	CONS					NON-CONS									
							T1					*T2*				
		Min	Max	No. non-occurring tuples	S.D.	S.D (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)	Min	Max	No.non-occurring tuples	S.D	S.D. (Ratio)
R1	11	0	14	37	2.206	0.000269	0	14	32	2.037	0.000249	0	11	42	1.985	0.000242
	12	0	10	581	1.504	0.000184	0	9	564	1.427	0.000174	0	8	570	1.403	0.000171
	13	0	8	3090	1.038	0.000127	0	6	3037	1.005	0.000123	0	5	3009	0.991	0.000121
R2	11	0	15	53	2.225	0.000272	0	13	41	1.991	0.000243	0	14	34	2.000	0.000244
	12	0	10	638	1.514	0.000185	0	8	564	1.407	0.000172	0	8	541	1.401	0.000171
	13	0	9	3152	1.043	0.000127	0	6	3051	1.002	0.000122	0	6	2990	0.989	0.000121
R3	11	3	41	0	6.218	0.000190	4	35	0	4.148	0.000127	4	32	0	3.976	0.000121
	12	0	28	7	3.739	0.000114	0	25	2	2.934	0.000090	0	20	2	2.785	0.000085
	13	0	16	261	2.369	0.000072	0	20	164	2.053	0.000063	0	14	144	1.974	0.000060
R4	11	2	45	0	6.295	0.000192	5	31	0	4.087	0.000125	1	31	0	4.132	0.000126
	12	0	30	6	3.788	0.000116	0	20	1	2.901	0.000089	0	19	4	2.902	0.000089
	13	0	17	244	2.375	0.000072	0	13	161	2.040	0.000062	0	14	171	2.044	0.000062
R5	11	6	83	0	11.846	0.000181	15	55	0	6.028	0.000092	13	53	0	5.863	0.000089
	12	1	51	0	6.782	0.000103	5	32	0	4.150	0.000063	3	31	0	4.087	0.000062
	13	0	32	25	4.019	0.000061	0	22	4	2.906	0.000044	0	20	3	2.868	0.000044
R6	11	6	80	0	11.876	0.000181	16	52	0	5.661	0.000086	17	54	0	5.649	0.000086
	12	0	46	2	6.770	0.000103	4	31	0	4.038	0.000062	4	32	0	3.977	0.000061
	13	0	28	31	4.038	0.000062	0	21	1	2.843	0.000043	0	21	1	2.811	0.000043
R7	11	30	320	0	46.566	0.000178	84	175	0	13.072	0.000050	88	175	0	11.632	0.000044
	12	8	188	0	25.258	0.000096	34	98	0	8.966	0.000034	34	98	0	8.213	0.000031
	13	1	114	0	13.766	0.000053	12	56	0	6.076	0.000023	12	55	0	5.753	0.000022
R8	11	30	320	0	46.566	0.000178	92	163	0	11.429	0.000044	86	166	0	11.389	0.000043
	12	8	188	0	25.258	0.000096	37	94	0	8.059	0.000031	37	96	0	8.062	0.000031
	13	3	119	0	13.766	0.000053	12	60	0	5.694	0.000022	14	57	0	5.679	0.000022
R9	11	120	1280	0	186.265	0.000178	305	733	0	57.276	0.000055	375	674	0	43.603	0.000042
	12	32	752	0	101.033	0.000096	139	411	0	33.462	0.000032	168	352	0	25.984	0.000025
	13	14	448	0	54.409	0.000052	61	239	0	19.903	0.000019	79	191	0	15.866	0.000015
R10	11	120	1280	0	186.265	0.000178	445	609	0	21.915	0.000021	408	630	0	28.819	0.000027
	12	32	752	0	101.033	0.000096	201	309	0	15.680	0.000015	193	335	0	18.564	0.000018
	13	14	448	0	54.409	0.000052	87	169	0	11.113	0.000011	90	181	0	12.372	0.000012

Table 12: Distribution table for *F3* (11, 12, 13 bit tuple)

Appendix C

Experimental Results for Section 4.1

Appendix C lists the results from the analysis of the m-tuple distribution, for the linearly filtered nonlinear feedback shift registers in Section 4.1

NLFSR	*M*-tuple	Uneven taps								Consecutive taps							
		T2				*T3*				*T4*				*T5*			
		Min	Max	N.O	S.D	Min	Max	N.O	S.D	Min	Max	N.O	S.D	Min	Max	N.O	S.D
R1	2	1048575	1048576	0	0.50000	1048575	1048576	0	0.50000	1048575	1048576	0	0.50000	1048575	1048576	0	0.50000
	3	524287	524288	0	0.35355	524287	524288	0	0.35355	524287	524288	0	0.35355	524287	524288	0	0.35355
	4	262143	262144	0	0.25000	262143	262144	0	0.25000	262143	262144	0	0.25000	262143	262144	0	0.25000
	5	131071	131072	0	0.17678	131071	131072	0	0.17678	131071	131072	0	0.17678	131071	131072	0	0.17678
	6	65535	65536	0	0.12500	65535	65536	0	0.12500	65535	65536	0	0.12500	65535	65536	0	0.12500
	7	32735	32800	0	32.00793	32767	32768	0	0.08839	32767	32768	0	0.08839	32767	32768	0	0.08839
	8	16244	16496	0	54.91645	16367	16400	0	16.00403	16383	16384	0	0.06250	16383	16384	0	0.06250
	9	8017	8372	0	67.94548	8062	8322	0	67.91167	8191	8192	0	0.04419	8191	8192	0	0.04419
	10	3932	4307	0	58.69262	3962	4220	0	54.77169	4095	4096	0	0.03125	4095	4096	0	0.03125
	11	1919	2199	0	43.13745	1932	2137	0	39.58642	2047	2048	0	0.02210	2047	2048	0	0.02210
	12	923	1147	0	30.81128	927	1114	0	29.37208	1023	1024	0	0.01562	1023	1024	0	0.01562
	13	435	602	0	22.14368	429	584	0	21.42993	511	512	0	0.01105	511	512	0	0.01105
	14	198	317	0	15.71891	190	315	0	15.90579	255	256	0	0.00781	255	256	0	0.00781
	15	86	176	0	11.19263	82	177	0	11.43048	127	128	0	0.00552	127	128	0	0.00552
	16	33	101	0	7.94320	34	103	0	8.08616	63	64	0	0.00391	63	64	0	0.00391
	17	10	62	0	5.62797	11	58	0	5.70148	31	32	0	0.00276	31	32	0	0.00276
	18	2	36	0	3.98779	1	36	0	4.01524	15	16	0	0.00195	10	22	0	2.44949
	19	0	25	182	2.82584	0	23	149	2.83454	7	8	0	0.00138	2	14	0	2.23607
	20	0	18	19108	1.99936	0	16	19363	2.00218	3	4	0	0.00098	0	11	12416	1.77218
	21	0	13	283720	1.41404	0	12	283783	1.41462	0	4	196608	1.00000	0	9	252000	1.32435
	22	0	9	1542758	0.99974	0	9	1543500	1.00004	0	4	1347585	0.86603	0	7	1499568	0.96935

NLFSR	*M*-tuple	Uneven Taps								Consecutive taps							
		T2				*T3*				*T4*				*T5*			
		Min	Max	N.O	S.D	Min	Max	N.O	S.D	Min	Max	N.O	S.D	Min	Max	N.O	S.D
R2	2	8388607	8388608	0	0.50000	8388607	8388608	0	0.50000	8388607	8388608	0	0.50000	8388607	8388608	0	0.50000
	3	4194303	4194304	0	0.35355	4194303	4194304	0	0.35355	4194303	4194304	0	0.35355	4194303	4194304	0	0.35355
	4	2097151	2097152	0	0.25000	2097151	2097152	0	0.25000	2097151	2097152	0	0.25000	2097151	2097152	0	0.25000
	5	1048575	1048576	0	0.17678	1048320	1048832	0	255.96881	1048575	1048576	0	0.17678	1048575	1048576	0	0.17678
	6	523967	524608	0	320.01566	523776	524800	0	313.53470	524287	524288	0	0.12500	524287	524288	0	0.12500
	7	261567	262592	0	303.59351	261152	263136	0	590.92511	262143	262144	0	0.08839	262143	262144	0	0.08839
	8	130687	131520	0	217.04147	130136	131864	0	419.60553	131071	131072	0	0.06250	131071	131072	0	0.06250
	9	65184	66008	0	159.52739	64776	66136	0	275.98889	65535	65536	0	0.04419	65535	65536	0	0.04419
	10	32428	33256	0	131.91829	32177	33231	0	183.57896	32767	32768	0	0.03125	32767	32768	0	0.03125
	11	16105	16800	0	103.42072	15950	16745	0	122.72352	16383	16384	0	0.02210	16383	16384	0	0.02210
	12	7957	8465	0	79.89017	7913	8527	0	86.62540	8191	8192	0	0.01562	8191	8192	0	0.01562
	13	3896	4331	0	59.09464	3882	4293	0	61.41571	4095	4096	0	0.01105	4095	4096	0	0.01105
	14	1892	2205	0	42.88934	1879	2204	0	44.01164	2047	2048	0	0.00781	2047	2048	0	0.00781
	15	906	1144	0	30.99203	900	1136	0	31.44050	1023	1024	0	0.00552	1023	1024	0	0.00552
	16	418	612	0	22.19957	406	604	0	22.37626	511	512	0	0.00391	511	512	0	0.00391
	17	192	330	0	15.82143	183	339	0	15.86420	255	256	0	0.00276	255	256	0	0.00276
	18	74	182	0	11.23199	80	181	0	11.24529	127	128	0	0.00195	127	128	0	0.00195
	19	31	108	0	7.95460	30	104	0	7.95780	63	64	0	0.00138	63	64	0	0.00138
	20	8	63	0	5.62347	9	63	0	5.62258	31	32	0	0.00098	31	32	0	0.00098
	21	1	40	0	3.96456	1	38	0	3.96313	15	16	0	0.00069	10	22	0	2.69258
	22	0	28	1385	2.78251	0	25	1453	2.78089	7	8	0	0.00049	0	16	256	2.34206
	23	0	18	153092	1.93635	0	18	153819	1.93552	3	4	0	0.00035	0	12	101632	1.78432
	24	0	15	2268916	1.32465	0	15	2269525	1.32426	0	4	1310720	1.00000	0	9	2070336	1.27090
	25	0	13	12343244	0.84624	0	11	12341336	0.84613	0	4	10702848	0.75215	0	8	12075936	0.83063

NLFSR	*M*-tuple	Uneven taps								Consecutive taps							
		T2				*T3*				*T4*				*T5*			
		Min	Max	N.O	S.D	Min	Max	N.O	S.D	Min	Max	N.O	S.D	Min	Max	N.O	S.D
R3	2	67108863	67108864	0	0.00000	67108863	67108864	0	0.00000	67108863	67108864	0	0.00000	67108863	67108864	0	0.00000
	3	33554431	33554432	0	0.00000	33554431	33554432	0	0.00000	33554431	33554432	0	0.00000	33554431	33554432	0	0.00000
	4	16777215	16777216	0	0.25000	16777215	16777216	0	0.25000	16777215	16777216	0	0.25000	16777215	16777216	0	0.25000
	5	8388607	8388608	0	0.17678	8388607	8388608	0	0.17678	8388607	8388608	0	0.17678	8388607	8388608	0	0.17678
	6	4194303	4194304	0	0.12500	4194303	4194304	0	0.12500	4194303	4194304	0	0.12500	4194303	4194304	0	0.12500
	7	2097151	2097152	0	0.08839	2097151	2097152	0	0.08839	2097151	2097152	0	0.08839	2097151	2097152	0	0.08839
	8	1048575	1048576	0	0.06250	1048575	1048576	0	0.06250	1048575	1048576	0	0.06250	1048575	1048576	0	0.06250
	9	523984	524592	0	192.66762	524128	524448	0	115.37711	524287	524288	0	0.04419	524287	524288	0	0.04419
	10	261616	262656	0	226.06218	261648	262480	0	218.28458	262143	262144	0	0.03125	262143	262144	0	0.03125
	11	130584	131600	0	223.63388	130494	131594	0	182.58170	131071	131072	0	0.02210	131071	131072	0	0.02210
	12	64918	66220	0	235.64700	64870	66156	0	190.80722	65535	65536	0	0.01562	65535	65536	0	0.01562
	13	32237	33410	0	180.09283	32264	33246	0	156.43924	32767	32768	0	0.01105	32767	32768	0	0.01105
	14	15910	16829	0	129.92024	15934	16803	0	115.99477	16383	16384	0	0.00781	16383	16384	0	0.00781
	15	7799	8560	0	91.59654	7844	8502	0	85.19328	8191	8192	0	0.00552	8191	8192	0	0.00552
	16	3821	4370	0	64.47517	3847	4365	0	61.54081	4095	4096	0	0.00391	4095	4096	0	0.00391
	17	1862	2237	0	45.34310	1857	2231	0	44.31337	2047	2048	0	0.00276	2047	2048	0	0.00276
	18	896	1173	0	31.93825	890	1184	0	31.60216	1023	1024	0	0.00195	1023	1024	0	0.00195
	19	417	622	0	22.57020	395	634	0	22.43503	511	512	0	0.00138	511	512	0	0.00138
	20	185	343	0	15.97071	183	332	0	15.90561	255	256	0	0.00098	255	256	0	0.00098
	21	76	187	0	11.29915	68	193	0	11.26741	127	128	0	0.00069	127	128	0	0.00069
	22	29	111	0	7.99192	27	110	0	7.97446	63	64	0	0.00049	63	64	0	0.00049
	23	7	67	0	5.64855	8	65	0	5.63963	31	32	0	0.00035	24	40	0	4.10961
	24	0	43	3	3.98937	0	41	3	3.98437	15	16	0	0.00024	6	27	0	3.69015
	25	0	28	11308	2.79025	0	27	11268	2.78799	7	8	0	0.00017	0	19	11264	2.74970

NLFSR	*M*-tuple	Uneven taps								Consecutive taps							
		T2				*T3*				*T4*				*T5*			
		Min	Max	N.O	S.D	Min	Max	N.O	S.D	Min	Max	N.O	S.D	Min	Max	N.O	S.D
R3	26	0	19	122997	1.85203	0	20	1228000	1.85106	3	4	0	0.00012	0	15	1332225	1.86495
	27	0	14	18163580	1.04143	0	15	18156695	1.04124	0	4	8388608	0.70711	0	12	18697985	1.04792
	28	0	11	98745333	0.62132	0	12	98741925	0.62130	0	4	85131265	0.50153	0	9	99815169	0.62388

NLFSR	*M*-tuple	Uneven taps								Consecutive taps							
		T2				*T3*				*T4*				*T5*			
		Min	Max	N.O	S.D	Min	Max	N.O	S.D	Min	Max	N.O	S.D	Min	Max	N.O	S.D
R4	2	134217727	134217728	0	0.00000	134217727	134217728	0	0.00000	134217727	134217728	0	0.00000	134217727	134217728	0	0.00000
	3	67108863	67108864	0	0.00000	67108863	67108864	0	0.00000	67108863	67108864	0	0.00000	67108863	67108864	0	0.00000
	4	33554431	33554432	0	0.00000	33554431	33554432	0	0.00000	33554431	33554432	0	0.00000	33554431	33554432	0	0.00000
	5	16777215	16777216	0	0.17678	16777215	16777216	0	0.17678	16777215	16777216	0	0.17678	16777215	16777216	0	0.17678
	6	8388607	8388608	0	0.12500	8388607	8388608	0	0.12500	8388607	8388608	0	0.12500	8388607	8388608	0	0.12500
	7	4194303	4194304	0	0.08839	4194303	4194304	0	0.08839	4194303	4194304	0	0.08839	4194303	4194304	0	0.08839
	8	2097151	2097152	0	0.06250	2097151	2097152	0	0.06250	2097151	2097152	0	0.06250	2097151	2097152	0	0.06250
	9	1048575	1048576	0	0.04419	1048575	1048576	0	0.04419	1048575	1048576	0	0.04419	1048575	1048576	0	0.04419
	10	524287	524288	0	0.03125	524287	524288	0	0.03125	524287	524288	0	0.03125	524287	524288	0	0.03125
	11	262143	262144	0	0.02210	262143	262144	0	0.02210	262143	262144	0	0.02210	262143	262144	0	0.02210
	12	131007	131136	0	64.00024	130880	131264	0	143.10846	131071	131072	0	0.01562	131071	131072	0	0.01562
	13	65312	65824	0	121.85253	65280	65792	0	110.85125	65535	65536	0	0.01105	65535	65536	0	0.01105
	14	32472	33032	0	121.32613	32484	32924	0	84.75847	32767	32768	0	0.00781	32767	32768	0	0.00781
	15	16070	16710	0	106.77424	16134	16618	0	72.83916	16383	16384	0	0.00552	16383	16384	0	0.00552
	16	7893	8465	0	84.95203	8012	8392	0	61.94783	8191	8192	0	0.00391	8191	8192	0	0.00391
	17	3886	4327	0	65.07114	3918	4308	0	50.08730	4095	4096	0	0.00276	4095	4096	0	0.00276
	18	1876	2241	0	46.33252	1878	2207	0	38.51984	2047	2048	0	0.00195	2047	2048	0	0.00195
	19	894	1170	0	32.56383	897	1155	0	28.61381	1023	1024	0	0.00138	1023	1024	0	0.00138
	20	407	620	0	22.86748	416	613	0	21.00669	511	512	0	0.00098	511	512	0	0.00098
	21	178	338	0	16.09214	185	336	0	15.30773	255	256	0	0.00069	255	256	0	0.00069
	22	76	190	0	11.34546	72	186	0	11.03585	127	128	0	0.00049	127	128	0	0.00049
	23	26	110	0	8.01013	27	110	0	7.88894	63	64	0	0.00035	63	64	0	0.00035
	24	6	66	0	5.64077	7	65	0	5.59462	31	32	0	0.00024	31	32	0	0.00024
	25	0	42	6	3.89895	0	42	5	3.88132	15	16	0	0.00017	12	19	0	1.47667

NLFSR	*M*-tuple	Uneven taps								Consecutive taps							
		T2				*T3*				*T4*				*T5*			
		Min	Max	N.O	S.D	Min	Max	N.O	S.D	Min	Max	N.O	S.D	Min	Max	N.O	S.D
R4	26	0	28	22617	2.56072	0	29	22122	2.55436	7	8	0	0.00012	2	13	0	1.40098
	27	0	20	2457513	1.57248	0	19	2446993	1.57099	3	4	0	0.00009	0	9	196608	1.13687
	28	0	16	36324779	1.00031	0	15	36286321	1.00001	0	4	16777216	0.50000	0	7	21987329	0.73856
	29	0	11	197501555	0.58383	0	12	197444574	0.58323	0	4	169869312	0.38017	0	6	172785665	0.43752

Bibliography

[1] Sultan Al-Hinai. *Algebraic Attacks on Clock-Controlled Stream Ciphers*. PhD thesis, Queensland University of Technology, 2007.

[2] Ali Alhamdan, Harry Bartlett, Leonie Simpson, Ed Dawson, and Kenneth Koon-Ho Wong. State convergence in the initialisation of the Sfinks stream cipher. In Josef Pieprzyk and Clark Thomborson, editors, *Australasian Information Security Conference (AISC* 2012*)*, volume 125, pages 27–31. Australian Computer Society, 2012.

[3] Ali Alhamdan. A study of the initialization process of the A5/1 stream cipher. Master's thesis, Queensland University of Technology, 2008.

[4] Ross Anderson. Searching for the Optimum Correlation Attack. In Bart Preneel, editor, *Fast Software Encryption (FSE* 1994*)*, volume 1008 of *Lecture Notes in Computer Science*, pages 137–143. Springer, 1995.

[5] Anonymous. RC4 Source Code. Cypherpunks mailing list, 1994. Available from `http://web.archive.org/web/20080404222417/http://cypherpunks.venona.com/date/1994/09/msg00304.html`.

[6] François Arnault and Thierry P. Berger. F-FCSR: design of a new class of stream ciphers. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption (FSE* 2005*)*, volume 3557 of *Lecture Notes in Computer Science*, pages 83–97. Springer, 2005.

[7] S.H. Babbage. Improved "Exhaustive Search" Attacks on Stream Ciphers. In *European Convention on Security and Detection*, pages 161–166, 1995.

[8] Steve Babbage and Matthew Dodd. The stream cipher MICKEY (version 1). eSTREAM, ECRYPT Stream Cipher Project, Report 2005/015, 2005. Available from `http://www.ecrypt.eu.org/stream/ciphers/mickey/mickey.pdf`.

[9] Steve Babbage and Matthew Dodd. The stream cipher MICKEY 2.0, 2006. Available from http://www.ecrypt.eu.org/stream/p3ciphers/mickey/mickey_p3.pdf.

[10] Côme Berbain, Henri Gilbert, and Antoine Joux. Algebraic and Correlation Attacks against Linearly Filtered Non Linear Feedback Shift Registers. In Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, editors, *Selected Areas in Cryptography (SAC* 2008*)*, volume 5381 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2009.

[11] Daniel Bernstein. A reformulation of TRIVIUM. Submission to Phorum: ECRYPT forum, 2006. Available from http://www.ecrypt.eu.org/stream/phorum/read.php?1,448.

[12] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The Keccak sponge function family, 2012. Available from http://keccak.noekeon.org/.

[13] Eli Biham and Orr Dunkelman. Differential Cryptanalysis in Stream Ciphers. Cryptology ePrint Archive, Report 2007/218, June 2007. Available from http://eprint.iacr.org/2007/218.pdf.

[14] Eli Biham and Jennifer Seberry. Py (Roo) : A Fast and Secure Stream Cipher Using Rolling Arrays. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/023, 2005. Available from http://www.ecrypt.eu.org/stream/ciphers/py/py.ps.

[15] Eli Biham and Jennifer Seberry. Pypy: Another Version of Py. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/038, 2006. Available from http://www.ecrypt.eu.org/stream/papersdir/2006/038.pdf.

[16] Eli Biham and Adi Shamir. Differential Cryptanalysis of DES-like Cryptosystems. In Alfred Menezes and Scott A. Vanstone, editors, *Advances in Cryptology — CRYPTO* '90, volume 537 of *Lecture Notes in Computer Science*, pages 2–21. Springer, 1990.

[17] 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. Springer, 2000.

[18] 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. Springer, 2001.

[19] Bluetooth®. Specification of the Bluetooth System Version 1.1, 2001. Available from `http://www.tscm.com/BluetoothSpec.pdf`.

[20] Wieb Bosma, John J. Cannon, and Catherine Playoust. The Magma algebra system. I. The user language. *Journal of Symbolic Computation*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993).

[21] An Braeken, Joseph Lano, Nele Mentens, Bart Preneel, and Ingrid Verbauwhede. SFINKS : A Synchronous Stream Cipher for Restricted Hardware Environments. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/026, 2005. Available from `http://www.ecrypt.eu.org/stream/ciphers/sfinks/sfinks.ps`.

[22] Marc Briceno, Ian Goldberg, and David Wagner. A pedagogical implementation of A5/1, 1999. Available from `http://cryptome.org/jya/a51-pi.htm`.

[23] Bruno Buchberger. *An Algorithm for finding the Bases Elements of the Residue Class Modulo a Zero Dimensional Polynomial Ideal*. Phd thesis, University of Innsbruck, Austria, 1965.

[24] Bruno Buchberger. Gröbner bases: A short introduction for systems theorists. In Roberto Moreno-Díaz, Bruno Buchberger, and José Luis Freire, editors, *Computer Aided Systems Theory — EUROCAST* 2001, volume 2178 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001. Available from `http://people.reed.edu/~davidp/pcmi/buchberger.pdf`.

[25] Christophe De Cannière, Özgül Küçük, and Bart Preneel. Analysis of Grain's Initialization Algorithm. In Serge Vaudenay, editor, *Progress in Cryptology — AFRICACRYPT* 2008, volume 5023 of *Lecture Notes in Computer Science*, pages 276–289. Springer, 2008.

[26] Christophe De Canniére and Bart Preneel. Trivium. In Matthew J. B. Robshaw and Olivier Billet, editors, *New Stream Cipher Designs: The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 244–266. Springer, 2008.

[27] Anne Canteaut and Michaël Trabbia. Improved Fast Correlation Attacks Using Parity-Check Equations of Weight 4 and 5. In Bart Preneel, editor, *Advances in*

Cryptology — EUROCRYPT 2000, number 1807 in Lecture Notes in Computer Science, pages 573–588. Springer, 2000.

[28] Claude Carlet. Boolean functions for cryptography and error correcting codes, 2007. Available from `http://www-rocq.inria.fr/codes/Claude.Carlet/chap-fcts-Bool.pdf`.

[29] Andrew Clark, Ed Dawson, Joanne Fuller, Jovan Dj. Golić, Hoon Jae Lee, William Millan, Sang-Jae Moon, and Leonie Simpson. The LILI-II Keystream Generator. In Lynn Margaret Batten and Jennifer Seberry, editors, *Information Security and Privacy (ACISP* 2002), volume 2384 of *Lecture Notes in Computer Science*, pages 25–39. Springer, 2002.

[30] Don Coppersmith, Hugo Krawczyk, and Yishay Mansour. The shrinking generator. In Douglas R. Stinson, editor, *Advances in Cryptology — CRYPTO* '93, volume 773 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1994.

[31] Nicolas T. Courtois. Fast Algebraic Attacks on Stream Ciphers with Linear Feedback. In Dan Boneh, editor, *Advances in Cryptology — CRYPTO* 2003, volume 2729 of *Lecture Notes in Computer Science*, pages 176–194. Springer, 2003.

[32] Nicolas T. Courtois. Higher Order Correlation Attacks, XL Algorithm and Cryptanalysis of Toyocrypt. In Pil Joong Lee and Chae Hoon Lim, editors, *Information Security and Cryptology — ICISC* 2002, volume 2587 of *Lecture Notes in Computer Science*, pages 182–199. Springer, 2003.

[33] Nicolas T. Courtois. Cryptanalysis of Sfinks. In Dongho Won and Seungjoo Kim, editors, *Information Security and Cryptology — ICISC* 2005, volume 3935 of *Lecture Notes in Computer Science*, pages 261–269. Springer, 2006.

[34] Nicolas T. Courtois and Willi Meier. Algebraic Attacks on Stream Ciphers with Linear Feedback. In Eli Biham, editor, *Advances in Cryptology — EUROCRYPT* 2003, volume 2656 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2003.

[35] Joan Daemen, Joseph Lano, and Bart Preneel. Chosen Ciphertext Attack on SSS. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/045, July 2005. Available from `http://www.ecrypt.eu.org/stream/papersdir/045.pdf`.

[36] Data Assurance and Communication Security Research Center. ZUC 1.4 Specification, 2010. Available from http://www.gsmworld.com/documents/EEA3_EIA3_ZUC_v1_4.pdf.

[37] Ed Dawson and Lauren Nielsen. Automated Cryptanalysis of XOR Plaintext Strings. *Cryptologia*, 20(2):165–181, April 1996.

[38] Nicolaas Govert de Bruijn. A combinatorial problem. *Proc. Koninklijke Nederlandse Akademie v. Wetenschappen*, 49:758–764, 1946.

[39] 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. Springer, 2009.

[40] Elena Dubrova. A List of Maximum Period NLFSRs. Cryptology ePrint Archive, Report 2012/166, 2012. Available from http://eprint.iacr.org/2012/166.pdf.

[41] 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.

[42] Niklas Eén and Niklas Sörensson. MiniSat — A SAT Solver with Conflict-Clause Minimization. Poster presented at Theory and Applications of Satisfiability Testing Conference (SAT 2005), 2005. Available from http://minisat.se/Main.html.

[43] Patrik Ekdahl and Thomas Johansson. Snow — a new stream cipher, 2000. Available from https://www.cosic.esat.kuleuven.be/nessie/workshop/submissions/snow.zip.

[44] eSTREAM. eCRYPT NoE: Preliminary Call for Stream Cipher Primitives, 2005. Available from http://www.ecrypt.eu.org/stream/call/.

[45] European Network of Excellence for Cryptology. The eSTREAM Project. Available from http://www.ecrypt.eu.org/stream/index.html.

[46] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases F4. *Journal of Pure and Applied Algebra*, 139:61–88, 1999.

[47] Jean-Charles Faugère and Gwénolé Ars. An Algebraic Cryptanalysis of Nonlinear Filter Generators using Gröbner bases. Technical report, Institut National De Recherche En Informatique Et En Automatique, 2003. Available from `http://hal.inria.fr/docs/00/07/18/48/PDF/RR-4739.pdf`.

[48] Berndt M. Gammel and Rainer Göttfert. Combining Certain Nonlinear Feedback Shift Registers. Proceedings of SASC 2004, 2004. Available from `http://www.matpack.de/achterbahn/Gammel_Goettfert_SASC2004.pdf`.

[49] Berndt M. Gammel and Rainer Göttfert. Linear filtering of nonlinear shift-register sequences. In Øyvind Ytrehus, editor, *Coding and Cryptography, WCC 2005*, volume 3969 of *Lecture Notes in Computer Science*, pages 354–370. Springer, 2006.

[50] Berndt M. Gammel, Rainer Göttfert, and O. Kniffler. An NLFSR-based stream cipher. In *International Symposium on Circuits and Systems (ISCAS 2006)*, pages 2917–2920, 2006.

[51] J.D Golić. Cryptanalysis of Three Mutually Clock-Controlled Stop/Go Shift Registers. *IEEE Transactions on Information Theory*, 46(3):1081–1090, 2002.

[52] J.D Golić and Miodrag J. Mihaljevic. A Generalized Correlation Attack on a Class of Stream Ciphers Based on the Levenshtein Distance. *Journal of Cryptology*, 3(3):201–212, 1991.

[53] J.D Golić, Mahmoud Salmasizadeh, Leone Simpson, and Ed Dawson. Fast Correlation Attacks on Nonlinear Filter Generators. *Information Processing Letters*, 64(1):37–42, 1997.

[54] Jovan Dj. Golić. On the Security of Nonlinear Filter Generators. In Dieter Gollmann, editor, *Fast Software Encryption (FSE 1996)*, volume 1039 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 1996.

[55] Jovan Dj. Golić. Cryptanalysis of Alleged A5 Stream Cipher. In Walter Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, volume 1233 of *Lecture Notes in Computer Science*, pages 239–255. Springer, 1997.

[56] Jovan Dj. Golić, Vittorio Bagini, and Guglielmo Morgari. Linear Cryptanalysis of Bluetooth Stream Cipher. In Lars R. Knudsen, editor, *Advances in Cryptology*

— *EUROCRYPT* 2002, volume 2332 of *Lecture Notes in Computer Science*, pages 238–255. Springer, 2002.

[57] Jovan Dj. Golic, Mahmoud Salmasizadeh, Andrew Clark, Abdollah Khodkar, and Ed Dawson. Discrete Optimisation and Fast Correlation Attacks. In Ed Dawson and Jovan Dj. Golic, editors, *Cryptography: Policy and Algorithms*, volume 1029 of *Lecture Notes in Computer Science*. Springer, 1995.

[58] Dieter Gollmann and Willam G. Chambers. Clock-Controlled Shift Registers: A Review. *IEEE Journal on Selected Areas in Communications*, 7(4):525–533, 1989.

[59] Solomon W. Golomb. *Shift Register Sequences*. Holden-Day, 1967.

[60] Edward J Gorth. Generation of Binary Sequences With Controllable Complexity. *IEEE Transactions on Information Theory*, IT-17(3):288–296, 1971.

[61] Helen Gustafson, Ed Dawson, Lauren Nielsen, and William J. Caelli. A computer package for measuring the strength of encryption algorithms. *Computers & Security*, 13(8):687–697, 1994.

[62] Richard Wesley Hamming. Error detecting and error correction codes. *Bell System Technical Journal*, 29(2):147–160, 1950.

[63] Philip Hawkes and Gregory G Rose. Guess-and-Determine Attacks on SNOW. In Kaisa Nyberg and Howard M. Heys, editors, *SAC* 2002, volume 2595 of *Lecture Notes in Computer Science*, pages 37–46. Springer, 2002.

[64] Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. A Stream Cipher Proposal: Grain-128. eSTREAM, ECRYPT Stream Cipher Project, 2006. Available from `http://www.ecrypt.eu.org/stream/p3ciphers/grain/Grain128_p3.pdf`.

[65] Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. The Grain Family of Stream Ciphers. In Matthew Robshaw and Olivier Billet, editors, *New Stream Cipher Designs: The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 191–209. Springer, 2008.

[66] Martin Hell, Thomas Johansson, and Willi Meier. Grain – A Stream Cipher for Constrained Environments. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/010, 2005. Available from `http://www.ecrypt.eu.org/stream/p3ciphers/grain/Grain_p3.pdf`.

[67] Martin E. Hellman. A Cryptanalytic Time-Memory Trade-Off. *IEEE Transactions on Information Theory*, 26(4):401–406, July 1980.

[68] Jin Hong. certain pairs of key-IV pairs for Trivium. eSTREAM Phorum, 2005. Available from `http://http://www.ecrypt.eu.org/stream/phorum/read.php?1,152,154`.

[69] Jin Hong and Woo-Hwan Kim. TMD-Tradeoff and State Entropy Loss Considerations of Streamcipher MICKEY. In Subhamoy Maitra, C. E. Veni Madhavan, and Ramarathnam Venkatesan, editors, *INDOCRYPT* 2005, volume 3797 of *Lecture Notes in Computer Science*, pages 169–182. Springer, 2005.

[70] Jin Hong and Palash Sarkar. New Applications of Time Memory Data Tradeoffs. In Bimal K. Roy, editor, *Advances in Cryptology — ASIACRYPT* 2005, volume 3788 of *Lecture Notes in Computer Science*, pages 353–372. Springer, 2005.

[71] Jin Hong and Palash Sarkar. Rediscovery of Time Memory Tradeoffs. Cryptology ePrint Archive, Report 2005/090, July 2008. Available from `http://eprint.iacr.org/2005/090.pdf`.

[72] Honggang Hu and Guang Gong. Periods on Two Kinds of Nonlinear Feedback Shift Registers with Time Varying Feedback Functions. *International Journal of Foundations of Computer Science*, 22(6):1317–1329, 2011.

[73] Takanori Isobe, Toshihiro Ohigashi, Hidenori Kuwakado, and Masakatu Morii. How to Break Py and Pypy by a Chosen-IV Attack. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/035, 2007. Available from `http://www.ecrypt.eu.org/stream/papersdir/2007/035.pdf`.

[74] Cees J.A. Jansen. Stream Cipher Design: Make your LFSRs jump! Presented at SASC 2004, 2004. Available from `http://www.ecrypt.eu.org/stvl/sasc/`.

[75] Cees J.A. Jansen, Tor Helleseth, and Alexander Kholosha. Cascade Jump Controlled Sequence Generator and Pomaranch Stream Cipher (Version 2). eSTREAM, ECRYPT Stream Cipher Project, Report 2006/006, 2006. Available from `http://www.ecrypt.eu.org/stream/papersdir/2006/006.pdf`.

[76] C.J.A. Jansen and D.E. Boekke. The Algebraic Normal Form of Arbitrary Functions of Finite Fields. In *Proceedings 8th Symposium on Information Theory in the Benelux*, pages 69–76, 1987.

[77] Éliane Jaulmes and Frédéric Muller. Cryptanalysis of the F-FCSR Stream Cipher Family. In Bart Preneel and Stafford E. Tavares, editor, *Selected Areas in Cryptography (SAC* 2005*)*, volume 3897 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2006.

[78] Fredrik Jönsson and Thomas Johansson. A fast correlation attack on LILI-128. *Information Processing Letters*, 81(3):127–132, February 2002.

[79] Ali A. Kanso. Mixer — A new stream cipher. *Journal of Discrete Mathematical Sciences and Cryptography*, 11(2):159–179, 2008.

[80] Itsik Mantin and Adi Shamir. A Practical Attack on Broadcast RC4. In Mitsuri Matsui, editor, *Fast Software Encryption (FSE* 2002*)*, volume 2355 of *Lecture Notes in Computer Science*, pages 152–164. Springer, 2002.

[81] George Marsaglia. The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness, 1995. Available from `http://www.stat.fsu.edu/pub/diehard/`.

[82] James L. Massey. Shift-Register Synthesis and BCH decoding. *IEEE Transactions on Information Theory*, 15(1):122–127, January 1969.

[83] Mitsuru Matsui. Linear Cryptanalysis Method for DES Cipher. In Tor Helleseth, editor, *Advances in Cryptology — EUROCRYPT* '93, volume 765 of *Lecture Notes in Computer Science*, pages 386–398. Springer, 1994.

[84] Alexander Maximov and Alex Biryukov. Two Trivial Attacks on Trivium. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *Selected Areas in Cryptography — SAC* 2007, volume 4876 of *Lecture Notes in Computer Science*, pages 36–55. Springer, 2007.

[85] Cameron McDonald, Chris Charnes, and Josef Pieprzyk. An Algebraic Analysis of Trivium Ciphers based on the Boolean Satisfiability Problem. Cryptology ePrint Archive, Report 2007/129, 2007. Available from `http://eprint.iacr.org/2007/129`.

[86] Willi Meier, Enes Pasalic, and Claude Carlet. Algebraic Attacks and Decomposition of Boolean Functions. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology — EUROCRYPT* 2004, volume 3027 of *Lecture Notes in Computer Science*, pages 474–491. Springer, 2004.

[87] Willi Meier and Othmar Staffelbach. Fast Correlation Attacks on Certain Stream Ciphers. *Journal of Cryptology*, 1(3):159–176, October 1989.

[88] Alfred Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[89] Miodrag J. Mihaljevic and Jovan Dj. Golic. A Fast Iterative Algorithm For A Shift Register Initial State Reconstruction Given The Nosiy Output Sequence. In Jennifer Seberry and Josef Pieprzyk, editors, *Advances in Cryptology — AUSCRYPT '90*, volume 453 of *Lecture Notes in Computer Science*. Springer, 1990.

[90] William L. Millan. *Analysis and Design of Boolean Functions for Cryptographic Applications*. PhD thesis, Queensland University of Technology, 1997.

[91] National Institute of Standards and Technology. A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications, December 2008. Available from `http://csrc.nist.gov/publications/nistpubs/800-22-rev1/SP800-22rev1.pdf`.

[92] Deike Priemuth-Schmid and Alex Biryukov. Slid Pairs in Salsa20 and Trivium. In Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das, editors, *Progress in Cryptology — INDOCRYPT* 2008, volume 5365 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2008.

[93] Havard Raddum. Cryptanalytic Results on Trivium. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/039, 2006. Available from `http://www.ecrypt.eu.org/stream/papersdir/2006/039.ps`.

[94] Andrea Röck. Entropy of the Internal State of an FCSR in Galois Representation. In Kaisa Nyberg, editor, *Fast Software Encryption (FSE* 2005*)*, volume 5086 of *Lecture Notes in Computer Science*, pages 343–362. Springer, 2008.

[95] Rainer A. Rueppel. Correlation Immunity and the Summation Generator. In Hugh C. Williams, editor, *Advances in Cryptology — CRYPTO '85*, volume 218 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 1985.

[96] Rainer A. Rueppel. *Analysis and Design of Stream Ciphers*. Springer, 1986.

[97] S. S. Bedi and N. Rajesh Pillai. Cube Attacks on Trivium. Cryptology ePrint Archive, Report 2009/015, 2009. Available from `http://eprint.iacr.org/2009/015.pdf`.

[98] Mahmoud Salmasizadeh, Leonie Simpson, Jovan Dj. Golić, and Ed Dawson. Fast Correlation Attacks and Multiple Linear Approximations. In Vijay Varadharajan, Josef Pieprzyk, and Yi Mu, editors, *Information Security and Privacy (ACISP* 1997*)*, volume 1270 of *Lecture Notes in Computer Science*, pages 228–239. Springer, 1997.

[99] Claude Elwood Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[100] T Siegenthaler. Correlation-Immunity of Nonlinear Combining Functions for Cryptographic Applications. *IEEE Transactions on Information Theory*, IT-30(5):776–780, 1984.

[101] T Siegenthaler. Decrypting a Class of Stream Ciphers Using Ciphertext Only. *IEEE Transactions on Computers*, 34(1):81–85, January 1985.

[102] Thomas Siegenthaler, Amstein Walthert Kleiner, and Réjane Forré. Generation of Binary Sequences with Controllable Complexity and Ideal r-Tupel Distribution. In David Chaum and Wyn L. Price, editors, *Advances in Cryptology — EUROCRYPT '87*, volume 304 of *Lecture Notes in Computer Science*, pages 15–23. Springer, 1988.

[103] Ilaria Simonetti, Ludovic Perret, and Jean Charles Faugère. Algebraic Attack Against Trivium. In *First International Conference on Symbolic Computation and Cryptography, SCC* 2008, LMIB, pages 95–102, 2008. Available from `http://www-salsa.lip6.fr/~jcf/Papers/SCC08c.pdf`.

[104] Leone Simpson, Jovan Dj. Golić, and Ed Dawson. A Probabilistic Correlation Attack on the Shrinking Generator. In Colin Boyd and Ed Dawson, editors, *Information Security and Privacy (ACISP* 98*)*, volume 1438 of *Lecture Notes in Computer Science*, pages 147–158. Springer, 1998.

[105] Leonie Simpson. *Divide and Conquer Attacks on Shift Register Based Stream Ciphers*. PhD thesis, Queensland University of Technology, January 2000.

[106] Leonie Simpson and Serdar Boztas. State cycles, initialization and the Trivium stream cipher. *Cryptography and Communications*, 4(3-4):245–258, 2012.

[107] Leonie Simpson, Ed Dawson, Jovan Dj. Golić, and William Millan. LILI Keystream Generator. In Douglas R. Stinson and Stafford Tavares, editors, *SAC - Selected Areas in Cryptograpy (SAC* 2000*)*, volume 2012 of *Lecture Notes in Computer Science*, pages 248–261. Springer, 2000.

[108] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[109] Meltem Sönmez Turan and Orhun Kara. Linear approximations for 2-round trivium. In Atilla Elci, Siddika Berna Ors, and Bart Preneel, editors, *Proceedings of the First International Conference on Security of Information and Networks (SIN 2007)*, pages 96–105. Trafford Publishing, 2007.

[110] Hongjun Wu, Tao Huang, Phuong Ha Nguyen, Huaxiong Wang, and San Ling. Differential Attacks against Stream Cipher ZUC. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology — ASIACRYPT 2012*, Lecture Notes in Computer Science, pages 262–277. Springer, 2012.

[111] Hongjun Wu, Phuong-Ha Nguyen, Huaxiong Wang, and San Ling. Cryptanalysis of Stream Cipher ZUC in the 3GPP Confidentiality & Integrity Algorithms 128-EEA3 & 128-EIA3. Presented at the Rump Session of Asiacrypt 2010, 2010.

[112] Hongjun Wu and Bart Preneel. Attacking the IV Setup of Pypy and Pypy. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/050, 2006. Available from `http://www.ecrypt.eu.org/stream/papersdir/2006/050.pdf`.

[113] Hongjun Wu and Bart Preneel. Key Recovery Attack on Py and Pypy with Chosen IVs. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/052, 2006. Available from `http://www.ecrypt.eu.org/stream/papersdir/2006/052.pdf`.

[114] Hongjun Wu and Bart Preneel. Resynchronization Attacks on WG and LEX. In Matthew J. B. Robshaw, editor, *Fast Software Encryption (FSE 2006)*, volume 4047 of *Lecture Notes in Computer Science*, pages 422–432. Springer, 2006.

[115] Wen Zeng and Wenfeng Qi. Finding slid pairs in trivium with MiniSat. *Science China Information Sciences*, pages 1–8, 2012.