EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

Eindhoven University of Technology

MASTER

Algebraic differential attacks on symmetric cryptography

Lukas, K.A.Y.

Award date:
2016

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mathematics and Computer Science

Master's thesis

Algebraic differential attacks on symmetric cryptography

by

K.A.Y. Lukas (0758084)

Supervisors:
prof. dr. J. Daemen,
prof. dr. T. Lange

Second reader:
dr. B. Škorić

Nijmegen, December 2016

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this thesis are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This thesis is the result of my own work and includes nothing that is the outcome of work done in collaboration, except where specifically indicated in the text.

K.A.Y. Lukas
December 2016

Abstract

This thesis studies the topic of algebraic differential attacks on symmetric cryptography. In particular, the effects of cube attacks by Dinur and Shamir [19] on sponge, duplex and farfalle constructions are researched. All three of these constructions utilize a permutation f to build cryptographic primitives. The impact of vulnerabilities of the used permutation in a duplex construction is investigated by performing a cube attack on the PRIMATE permutation family. The PRIMATE permutation family is used in the PRIMATEs family for authenticated encryption. Farfalle is a construction for building a pseudorandom function that uses, in addition to a permutation f, an algorithm to encode block indices. We will show that the encoding algorithm used in farfalle must be chosen carefully, and that the sequence of encodings generated should not contain large affine subspaces.

Our contribution in this thesis is a cube attack requiring 2^{32} input blocks against one of the PRIMATEs encryption schemes (`gibbon`), assuming that nonces are reused. By performing attacks on a reduced-round PRIMATE permutation, we make a prediction of the computational power required for the attack on `gibbon`. Furthermore, we present various types of algorithms that can be used to encode a sequence of block indices that will not contain large affine subspaces. These algorithms are specifically designed to be used in the farfalle construction to avoid the weakness we identified.

Contents

Declaration iii

Abstract iv

Contents vii

1 Introduction 1

2 Background and Definitions 5
- 2.1 Background 5
- 2.2 Definitions 8

3 Sponge and Duplex constructions 15
- 3.1 Sponge construction 16
  - 3.1.1 Applications 18
- 3.2 Duplex construction 19
  - 3.2.1 Applications 21
- 3.3 Permutation types 21
  - 3.3.1 ARX-based 22
  - 3.3.2 S-box-based 22

4 Higher-order differential cryptanalysis 25
- 4.1 Attack model 25
- 4.2 Differential analysis 26
  - 4.2.1 Differential along a vector space 29
- 4.3 Cube attacks 30
  - 4.3.1 An example attack 30
  - 4.3.2 Attack procedure 32
  - 4.3.3 Limitations of this attack 34

5 Cube attack on PRIMATES 35
- 5.1 PRIMATEs: an authenticated encryption family 35
  - 5.1.1 Schemes 36
  - 5.1.2 The permutation 40

5.2 Cube attack on PRIMATEs . . . 41
5.2.1 Basic cube attack . . . 42
5.2.2 Generating linear equations out of quadratic equations . . . 42
5.2.3 Skipping the first SubElements operation . . . 43
5.3 Results . . . 44
5.3.1 Impact on the PRIMATE-family . . . 45
5.3.2 Attack details . . . 46

6 Farfalle: a parallelizable PRF construction **49**
6.1 Sketch of the scheme . . . 49
6.2 Vulnerability: creating an all-zero intermediate state . . . 51
6.3 Countermeasure using the counter encoding . . . 53

7 Low affine dimensional encoding **55**
7.1 Calculating the affine dimension . . . 57
7.2 Basic multiplicative encodings . . . 60
7.2.1 Mathematical properties . . . 61
7.2.2 Experimental results . . . 66
7.3 Logarithmic encodings . . . 70
7.3.1 Dealing with even values of $\lfloor \log_2(i) + 1 \rfloor$ . . . 70
7.3.2 Experimental results . . . 72
7.4 Concatenated encodings . . . 73
7.4.1 Mathematical properties . . . 74
7.4.2 Experimental analysis . . . 78
7.5 Rotation-based encodings . . . 80
7.5.1 Exponential encoding . . . 80
7.5.2 Shifted encoding . . . 81
7.5.3 Smeer encoding . . . 83

8 Conclusion **93**
8.1 Future work . . . 94

References **95**

Chapter 1

Introduction

Cryptography is the science in which we study secret codes. These secret codes or *ciphers* transform unencrypted data (*plaintext*) into *ciphertext*. This ciphertext can not be read directly without decrypting it first, decryption transforms the ciphertext into the plaintext again. Cryptography is often divided into two groups: symmetric cryptography and asymmetric cryptography. Symmetric cryptography uses a single key for both encryption and decryption. Asymmetric cryptography uses one key for encryption and another key for decryption. As two different keys are used, a receiver can publish the key used for encryption (the *public key*), allowing anyone to send a message to the receiver, while keeping the key used for decryption private (the *private key*). This allows the receiver to be the only one able to decrypt messages. Symmetric cryptography is normally not able to do this. In this thesis we are concerned with symmetric cryptography only.

When a new symmetric-key cipher has just been designed, it is often not known if this cipher is secure. The security of these ciphers cannot always be proven mathematically. This makes one question how our current ciphers are derived and why we consider the mainstream ciphers as secure. Secret-key cryptography has a long standing tradition of selecting secure ciphers by organizing competitions. These secure ciphers are selected from a wide range of submissions. The scientific community puts these submitted ciphers under scrutiny by analysing and searching for vulnerabilities for each cipher. This allows the committee to select secure and well-performing ciphers. An example of such a competition is the AES competition. This competition was won by the Rijndael cipher (now called the AES cipher). The aim of this thesis is to provide a contribution to the CAESAR competition. The CAESAR competition is a competition for authenticated ciphers. The main difference between the AES competition and the CAESAR competition is that the CAESAR competition is about authenticated encryption schemes while the AES competition was purely for block ciphers. Authenticated ciphers provide encryption but also provide authenticity assurance on the data. Authenticity assurance means that the receiver can detect if unauthorized changes are made to the message. Often authenticated ciphers produce a ciphertext as well as an authentication tag. This authentication tag allows to the receiver to check the ciphertext for authenticity.

In this thesis we contribute to two different ciphers: the authenticated encryption family PRIMATEs and Keyak. Both ciphers were submitted as a candidate to the CAESAR competition. While this thesis was being written, the PRIMATEs family was deselected for the current round (round three) of the

CAESAR competition. Nonetheless, we hope that the result can still be helpful for anyone wanting to use the PRIMATEs family of authenticated algorithms and be helpful for the PRIMATEs designers.

To provide a contribution and to analyse these ciphers we will study differential attacks on symmetric cryptography. Differential attacks are attacks that exploit relations between differences induced in the input data and differences observed in the output data. This type of cryptanalysis was first introduced by Biham and Shamir in 1991 [9]. They used this type of cryptanalysis to break amongst others DES. Biham and Shamir recognized that this type of attack was not specific to DES and directly applied it to other DES-like system. Differential attacks are together with linear cryptanalysis the most significant type of attacks on symmetric cryptography [23]. Linear cryptanalysis is a type of attack that formulates linear relations between the input and output data that hold with a high probability. The exact process is out of the scope of this thesis, more information can be found in "Linear Cryptanalysis Method for DES Cipher" by Matsui [31].

By encrypting multiple sets of data we are able to perform a differential attack. This input data will each time be nearly the same, except for a unique difference from the original data. Afterwards, we combine the output data in such a way that linear relations between input data and output data are found. By solving these linear relations we can reconstruct the input data. In chapter 4 a closer look into differential attacks is given. Formalisms are taken from Seidlová [43].

Ciphers submitted to the CAESAR competition could be developed using various different techniques. Firstly, one can create a whole new cryptographic algorithm providing both encryption and authentication. This can give performance advantages because calculation of the encryption and the authentication tag can be combined in an efficient manner. An example of such a submission is AEGIS [47]. Secondly, a design can be based on a stream cipher. Stream ciphers generate pseudorandom data (called the keystream) using the cryptographic key and the input message. By combining this pseudorandom data with the input message a ciphertext can be created. This same keystream (which is dependent on the plaintext data) can be reused to generate an authentication tag. This gives performance advantages because such a tag is calculated from an already existing keystream. Examples of such submissions are Keyak and PRIMATEs. Lastly, one can use an already existing block cipher, which is treated as a black box, and provide a mode to add authentication to this cipher. Research on these techniques has already been started more than ten years ago [26, 47, p. 2]. An example of such a submission is AES-OTR [34].

The authenticated encryption family PRIMATEs and Keyak are both based on or very similar to the sponge and/or duplex construction [7, p.3][1, p.16]. The sponge and duplex construction are a relatively new construction introduced in 2011 by the Keccak team [5]. The SHA-3 algorithm (a subset of Keccak) uses a sponge construction and made them popular [36]. A sponge construction is a type of construction that allows someone to build an algorithm that accepts any input stream of finite length and can output an arbitrarily long stream of data. Internally, such a construction uses a finite state to store information about the input stream. This finite state is combined with a permutation algorithm to generate an arbitrarily long stream of data. The duplex construction extends this by allowing the interleaving of input data and output data. Duplex constructions can be used as stream ciphers. The heart of both of these constructions is the permutation algorithm. Therefore, we will focus on finding flaws in permutation algorithms. A view into sponge and duplex constructions is given in chapter 3.

The state of a duplex construction can be used to find information of the data that was output as well as the data that will be output. The security of a duplex construction relies on the adversary not knowing the state completely. Thus when there is knowledge about the state it is, in some cases, possible to find vulnerabilities in the algorithm. Therefore, our differential attack focuses on finding the state of these algorithms. This is done by performing a differential attack on the permutation of PRIMATEs. In chapter 4 the attack method is explained and in chapter 5, we document our attack on PRIMATEs.

After we have seen how such differential attacks are performed, some methods to defend against such an attack are given. We will develop a new defence mechanism by generating a set of block indices with special properties. This defence mechanism is designed to be used in a novel construction for parallelizable pseudorandom functions, called farfalle. As we have explained before, differential attacks work by inserting differences in the data that has to be encrypted/hashed/permuted. The effect of these differences on the output data are then analysed, which we will use to extract information of the input data. Our new defence mechanism works by limiting the possible differences that can be inserted, by allowing only certain elements to be used. More specifically, we limit the elements in such a way that the size of the largest affine subspace possible is small. This prevents an attacker from inserting all vectors from a certain large vector space into the data. Consequently, limiting the elements in our way is a countermeasure against higher order differential attacks. This is explained in chapter 7.

Chapter 2

Background and Definitions

In this chapter some background information and definitions used throughout this thesis are given. Definitions are given in this chapter to create a centralized chapter where all (important) definitions can be found.

2.1 Background

As described in the introduction, cryptography is the study of secret codes or ciphers. Ciphers are algorithms that perform encryption or decryption. Encryption allows someone to encode a message in such a way that only authorized parties can read this message. Encryption will output a ciphertext C of a message M given a key K. A decryption algorithm can output the original message M given the ciphertext C and a key $\overline{K}$. Thus using encryption, someone can encode a message in such a way, that only owners of the key $\overline{K}$ can read the message. In general, we distinguish two types of ciphers: symmetric and asymmetric ciphers.

In symmetric ciphers the keys used for encryption and decryption are equal, thus $K = \overline{K}$. Asymmetric ciphers use different keys for encryption in decryption, thus $K \neq \overline{K}$. This means that anyone that has access to K and thus can encrypt messages, cannot read all messages that are encrypted with K. A receiver can choose to publish the key K while keeping $\overline{K}$ secret. This allows anyone to send a message to the receiver while the receiver is the only one able to read this message. Such a usage poses some security questions: when K is public, $\overline{K}$ should not be derivable from K. Due to these extra requirements, asymmetric ciphers require more computational power than symmetric ciphers. In a communication protocol, asymmetric ciphers can be used to first negotiate a key to be used in a symmetric cipher, which is then used to encrypt all messages from then on [17, p. 57]. We are only concerned about symmetric ciphers in this thesis.

Symmetric ciphers can be divided into various different types. The most common types of symmetric ciphers are block ciphers and stream ciphers. Block ciphers are keyed permutations that map an n-bit block to another n-bit block, in such a way that it is difficult to recover the input from the output without knowing the encryption key. A block cipher will always encrypt the same block given the same key to the same output block. A consequence of this is that block cipher can only encrypt a single block per key securely. If multiple blocks are encrypted separately with the same

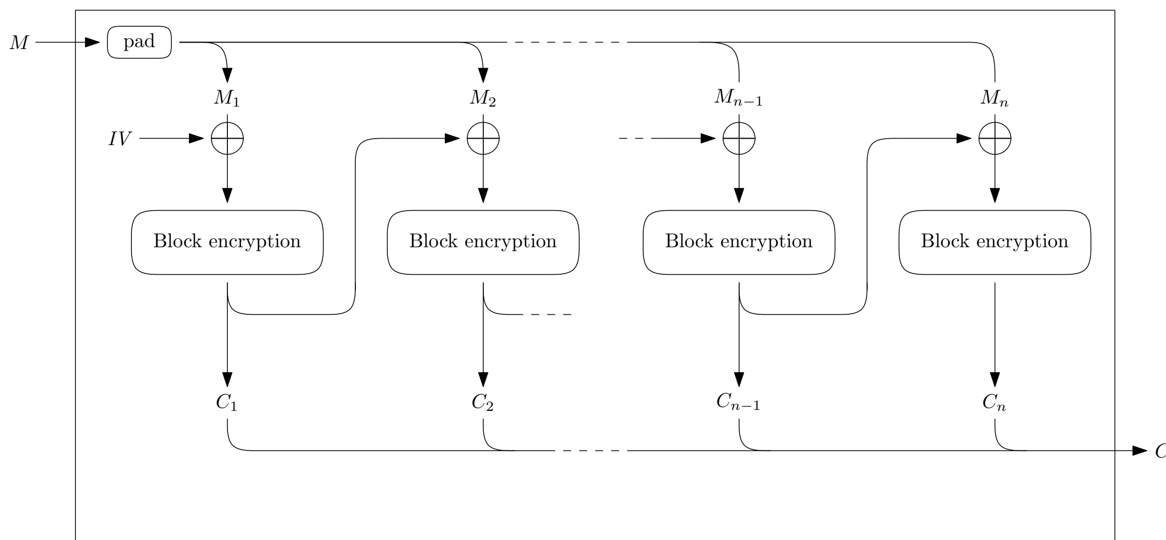

Figure 2.1 A diagram of the cipher block chaining mode of operation. Each block of M is exclusive-OR'ed with the previous ciphertext block before encrypting it using a given block cipher.

key, a codebook attack can be performed [11]. To allow the usage of block ciphers for longer messages consisting of multiple blocks, modes of operation are developed.

Modes of operation are algorithms that use a block cipher to encode messages that are longer than the block size of a block cipher. A mode essentially ensures, among other things, that whenever two blocks in the input message are the same the blocks in the ciphertext corresponding to these input blocks are not the same. An example of such a mode is the cipher block chaining (CBC) mode. This mode was invented in 1976 and patented by IBM [21]. CBC encrypts a message M by dividing the message into blocks $M_1, M_2, M_3, \ldots$ where the length of each block is equal to the block size of the cipher. A bitwise exclusive-or operation with an initialization vector IV is then done on the first block before M_1 is encrypted. This gives C_1. For any other block M_i with $i > 1$ an exclusive-or operation is done with the previous ciphertext C_{i-1} before encryption. An illustration of this principle is given in figure 2.1. By making the encryption of a block dependent on the previous ciphertext, blocks with the same value will no longer be encrypted to the same ciphertext. Many other different modes of operation exist.

In contrast to block ciphers, stream ciphers encrypt bits individually [37]. This means that stream ciphers can encrypt messages of arbitrary length. A stream cipher can encrypt bits individually by producing a keystream. Each bit is encrypted by combining it with a bit from the keystream using an exclusive-or operation. It should be noted that modes like cipher feedback (CFB), output feedback (OFB) and counter (CTR) mode can turn a block cipher into a stream cipher [38]. Therefore, block ciphers can be used as a building block to build a stream cipher.

Another important concept in cryptography next to encryption, is authentication. While encryption prevents intermediaries to read the message sent, it does not prevent them from modifying this message. Thus, the receiver does not know if the data has been modified and who sent the data. In other words, encryption does not provide *integrity* and *authenticity*. Methods that provide authenticity can be distinguished into two groups: MAC algorithms and digital signatures [41].

MAC algorithms were first referenced in a patent in 1972, but were already used for a long time in the banking community at that moment. MAC algorithms allow the sender to generate a MAC value using the input message and a shared key K. Both the receiver and the sender must have access to this shared key K. The receiver can than verify the authenticity of a message by using K and the given MAC value. For example, the receiver can generate the MAC value of the message itself and check if this generated MAC value is equal to the MAC value received. MAC algorithms attempt to make it difficult to change both the message and change the MAC value accordingly without having access to the key K [41].

Because the sender and the receiver both use the same key to generate and verify the MAC, it is impossible to prove who generated a certain message. This is because the sender as well as the receiver can generate correct MACs . In other words, a MAC algorithm does not provide *non-repudiation*. Digital signatures on the other hand, do provide this property. Digital signatures were introduced by Diffie and Hellman in 1976 [18]. Digital signatures use asymmetric encryption to provide authenticity and non-repudiation. When using digital signatures the sender of a message has a private key that he or she keeps secret, but it publishes a public key. Using the private key it can generate a signature of a message. Anyone that has access the public key can then verify the authenticity of the message. However, to generate a signature of a message one needs to have access to the private key. As the sender is the only one with access to this private key, it can be proven that the message was sent by the sender [42]. As with symmetric and asymmetric encryption, MAC algorithms require less computation than digital signatures and are often preferred when non-repudiation is not required [41].

MAC values can be generated using *hash functions*. Hash functions are a class of functions that maps data of arbitrary size into a bit string of fixed length, often called the digest or the hash value. Hash functions can be dependent on a key (*keyed hash function*) or only dependent on the input message (*unkeyed hash function*). For usage in MAC algorithms, these hash functions are often dependent on a key.

Hash functions should also be hard to invert: the input data cannot be reconstructed using the output data. More formally, this means that given a digest h of a message M, it is infeasible to find any message M' such that the digest of M' is h. This property is called *preimage resistant*. Moreover, it is also required that given M and its digest h, it is hard to find an M' different from M such that the digest of M' is also h. This is called *second preimage resistant* [40]. Besides second preimage resistant, we also want that it must be hard to find two messages M and M' that give the same hash values. Note that in this case M is not chosen beforehand. This is called *collision resistance*.

Historically, cryptographic hash functions were built from one-way compression functions [39, p. 3]. One-way compression functions are functions that transform two fixed-length inputs into an output of the same size as one of the inputs [32, p. 328]. This means that a compression function will lose information about the two fixed-length inputs. By losing information, compression functions aim to make it difficult to find the (or any) preimage of the output. Such a compression function is then used in other constructions, such as the Merkle–Damgård construction, to construct a cryptographic hash function. This construction is an iterative design and divides the input into blocks equal to the input length of the compression function (padding the last block if necessary), and then compresses each block sequentially, combining the output of the previous round with the next block.

Another application for hash functions is password hashing. Password hashing is a method of storing passwords used in a system without keeping track of the actual passwords by using a cryptographic hash function. If the database of a system is compromised, this prevents attackers from being able to login on the system or other systems, as users often reuse the same password (sometimes with slight modifications). To provide security, it must be hard to generate the preimage of a hash value. As password are often simple/short and can be brute forced by trying all possible values or a list of commonly used passwords (dictionary attack), such methods require that generating a hash value takes a considerable amount of computation. By requiring such a computation, it becomes difficult to try all possible values.

Hash functions can also be used for other applications than MAC algorithms or password hashing. For instance, cryptographic hash functions can be used to create a cryptographic key derivation function. Cryptographic key derivation is used to generate secure keys from several data sources. In various cryptographic algorithms, a key must be of a fixed size. Often, the user of this cryptographic algorithm has a secret that is not of the size that is needed, either more or less data. To provide security, a cryptographic hash function used in cryptographic key derivation should generate a uniformly random key derived from the secret.

Lastly, cryptographic key derivation functions can be generalized by defining pseudorandom number generators (PRNG). PRNGs are algorithms generating a sequence of numbers approximating a sequence of truly random numbers. These generators are used in many applications: gambling, gaming, computer simulations and many more. A cryptographically secure pseudorandom number generator (CSPRNG) is a pseudorandom number generator that can be used in cryptography. This means that it has more requirements to its randomness. Often such a CSPRNG is slower than a PRNG used in computer simulations, because of these requirements. CSPRNG can be used to generate uniform keys from a truly random (but not uniform) source of data, and are in that functionality often similar or equal to a key derivation function. The difference with a key derivation function, however, is that a CSPRNG is able to generate an arbitrary number of random numbers.

In this thesis, we will study symmetric encryption schemes that also provide authentication.

2.2 Definitions

Our defence mechanism that will be introduced in chapter 7, is based on an encoding that encodes a number as a binary string. To be able to define these encodings we need to be able to transform from elements in the set of natural numbers ($\mathbb{N}$) to binary strings. Binary strings are strings of single bits. Bits are defined as elements in $\mathbb{F}_2$. We write $\mathbb{F}_2 = \{\mathbb{0}, \mathbb{1}\}$. We use $\mathbb{0}$ and $\mathbb{1}$ instead of 0 and 1 to be able to distinguish between elements of $\mathbb{F}_2$ and $\mathbb{N}$. This is important as addition in $\mathbb{N}$ is different from addition in $\mathbb{F}_2$. We define a function `i2bsp` that transforms a number into a binary string. `bs2ip` transforms binary strings back to numbers.

Definition 1. `i2bp` *(Integer to binary primitive) converts an integer to a binary bit b ($b \in \mathbb{F}_2$):*

$$\mathtt{i2bp}(i) = \begin{cases} \mathbb{0} & \textit{if } i \textit{ is even,} \\ \mathbb{1} & \textit{if } i \textit{ is odd.} \end{cases}$$

One can see that i2bp gives the least significant bit of the binary representation of an integer.

Definition 2. i2bsp *converts an integer* $i \in \mathbb{N}$ *to a binary string* $s \in \mathbb{F}_2^d$ *of length* d.

$$\mathtt{i2bsp}(i, d) = \begin{cases} (s_1, \ldots, s_j, \ldots, s_d) & \text{if } i < 2^d, \\ \mathit{error} & \text{otherwise,} \end{cases}$$

with:

$$s_j = \mathtt{i2bp}(\lfloor \frac{i}{2^{d-j}} \rfloor).$$

Definition 3. $\overline{\mathit{i2bsp}}$ *converts an integer* $i \in \mathbb{N}$ *to a binary string* $s \in \mathbb{F}_2^d$ *of length* d. *In contrast to* i2bsp, $\overline{\mathit{i2bsp}}$, *discards the most significant bits, in case* $i \geq 2^d$.

$$\overline{\mathit{i2bsp}}(i, d) = (s_1, \ldots, s_j, \ldots, s_d)$$

with:

$$s_j = \mathtt{i2bp}(\lfloor \frac{i}{2^{d-j}} \rfloor).$$

Definition 4. b2ip *converts a binary element* $b \in \mathbb{F}_2$ *to an integer* $i \in \{0, 1\}$.

$$\mathtt{b2ip}(b) = \begin{cases} 1 & \text{if } b = \mathbb{1}, \\ 0 & \text{otherwise.} \end{cases}$$

Definition 5. bs2ip *converts a binary string* $s \in \mathbb{F}_2^d$ *of length* d *to an integer* $i \in \mathbb{N}$.

$$\mathtt{bs2ip}(s) = \sum_{j=1}^{d} \mathtt{b2ip}(s_j) \cdot 2^{d-j}.$$

To be able to design encodings, we need to be able to concatenate binary strings. Concatenation is denoted by $|$ and is defined as follows:

Definition 6. *Concatenation on two binary strings* $s \in \mathbb{F}_2^d, s' \in \mathbb{F}_2^{d'}$ *is defined as follows:*

$$s \mid s' = t \in \mathbb{F}_2^{d+d'},$$

with:

$$t_j = \begin{cases} s'_{j-d} & \text{if } j > d \\ s_j & \text{otherwise.} \end{cases}$$

Differential cryptanalysis analyses relations between differences induced in the input and differences observed in the output strings. Differences between two strings can be found when subtracting

(which is equal to our addition operation in $\mathbb{F}_2^d$) one string from another. Therefore, we need to be able to perform addition of two field elements in $\mathbb{F}_2^d$.

Definition 7. *Addition of two equal-length binary strings $s, s' \in \mathbb{F}_2^d$ is defined as follows:*

$$s + s' = t \in \mathbb{F}_2^d,$$

with:

$$t_j = s_j + s'_j.$$

In other words, addition on binary strings is the bitwise exclusive-or operation.

In differential cryptanalysis we often use $\mathcal{O}$ as the null vector. Furthermore, we use the unit vectors e_i that consists of only one $\mathbb{1}$. This is because encrypting $\mathcal{O}$ allows us to analyse certain properties of the cipher. Because these are repeatedly used in this thesis, we will define them below.

Definition 8. *$\mathcal{O}^d$ is the null element of $\mathbb{F}_2^d$ with respect to addition $(+)$ and is equal to:*

$$\mathcal{O}^d = (\mathbb{0}, \dots, \mathbb{0}) = \texttt{i2bsp}(0, d).$$

When d is clear from the context, we sometimes write $\mathcal{O}^d$ as $\mathcal{O}$. By e_i we denote the ith unit vector of $\mathbb{F}_2^d$ and it is defined as:

$$\begin{aligned} e_i &= (\mathbb{0}, \dots, \mathbb{0}, \overbrace{\mathbb{1}}^{i^{th}\ element}, \mathbb{0}, \dots, \mathbb{0}) \\ &= \texttt{i2bsp}(2^{d-i}, d). \end{aligned}$$

Definition 9. *Multiplication of a bit $b \in \mathbb{F}_2$ with a binary string $s \in \mathbb{F}_2^d$ is defined as follows:*

$$b \cdot s = \begin{cases} s & \text{if } b = \mathbb{1} \\ \mathcal{O}^d & \text{otherwise.} \end{cases}$$

Definition 10. *Multiplication of two binary strings $s, s' \in \mathbb{F}_2^d$ is defined as follows:*

$$s \cdot s' = t \in \mathbb{F}_2^d,$$

with:

$$t_j = s_j \cdot {s'}_j.$$

In other words, multiplication on binary strings is the bitwise and operation. Of course, addition of two elements in $\mathbb{Z}$ remains the usual addition. Integer multiplication also remains unchanged. Mathematically, $\mathbb{F}_2^d$ is the d-dim vectorspace over the field $\mathbb{F}_2$ with $\cdot$ the componentwise multiplication.

Lastly, we use the rotation operation `rol` in several encodings.

Definition 11. $\mathtt{rol}$ *(Rotate Left) is an operation on bit strings:* $\mathbb{F}_2^d \to \mathbb{F}_2^d$*. Let an* $s = (s_1, \ldots, s_d) \in \mathbb{F}_2^d$ *be given, and let an* $i \in \mathbb{N}$ *be given.*
Then:

$$\mathtt{rol}(s, i) = (s_{1+i}, s_{2+i}, \ldots, s_d, s_1, s_2, \ldots, s_i).$$

The following part concerns affine subspaces in $\mathbb{F}_2^d$. Affine spaces are an extension of linear subspaces. To understand affine spaces, we will take a look at some spaces in $\mathbb{R}^3$. First, consider the subspace S of $\mathbb{R}^3 = \{(x, y, z) : x, y, z \in \mathbb{R}\}$ defined by the solutions to the linear equation $y = 0$. This gives $S = \{(x, 0, z) : x, z \in \mathbb{R}\}$. One can see that S is a subspace of $\mathbb{R}^3$, as S contains the zero vector $(0, 0, 0)$; the sum of two elements $(x_1, 0, z_1), (x_2, 0, z_1) \in S$ yields $(x_1 + x_2, 0, z_1 + z_2) \in S$ and for any $(x, 0, z) \in S$ and $c \in \mathbb{R}$ scalar multiplication yields $c \cdot (x, 0, z) = (c \cdot x, 0, c \cdot z) \in S$. In figure 2.2a S has been graphed.

While the linear equation $y = 0$ induces a subspace, $y = 1$ does not. If T is the set of solutions of this linear equation, we have $T = \{(x, 1, z) : x, z \in \mathbb{R}\}$. Obviously, $(0, 0, 0) \notin T$. A careful reader might note that T is the translation of all elements by $(0, 1, 0)$ in S: $T = \{\vec{s} + (0, 1, 0) : \vec{s} \in S\}$. This is illustrated in figure 2.2b. To extend the definition of linear subspaces, we could define $(0, 1, 0)$ as the origin of T. However, this translation vector $(0, 1, 0)$ is not unique. For instance, translating S by $(1, 1, 0)$ would also yield T:

$$\begin{aligned}
T &= \{(x, 1, z) : x, z \in \mathbb{R}\} \\
&= \{(x + 1, 1, z) : x + 1, z \in \mathbb{R}\} \\
&= \{(x + 1, 1, z) : x, z \in \mathbb{R}\} \\
&= \{(1, 1, 0) + (x, 0, z) : x, z \in \mathbb{R}\} \\
&= \{(1, 1, 0) + \vec{s} : \vec{s} \in \mathbb{S}\}.
\end{aligned}$$

This is illustrated in figure 2.2c. Likewise, if we take any $\vec{t} \in T$ then $\vec{t} = (t_x, t_y, t_z)$ can translate S to T:

$$\begin{aligned}
T &= \{(x, 1, z) : x, z \in \mathbb{R}\} \\
&= \{(x + t_x, t_y, z + t_z) : x + t_x, z + t_z \in \mathbb{R}\} && (t_y = 1) \\
&= \{(x + t_x, t_y, z + t_z) : x, z \in \mathbb{R}\} \\
&= \{\vec{t} + (x, 0, z) : x, z \in \mathbb{R}\} \\
&= \{\vec{t} + \vec{s} : \vec{s} \in \mathbb{S}\}.
\end{aligned}$$

This means that the origin is not translated to any specific point. In fact, S can be translated to T in so many ways that the origin can become any arbitrary point in T.

In this example, T is an affine space. As with T, in all affine spaces there are no distinguished

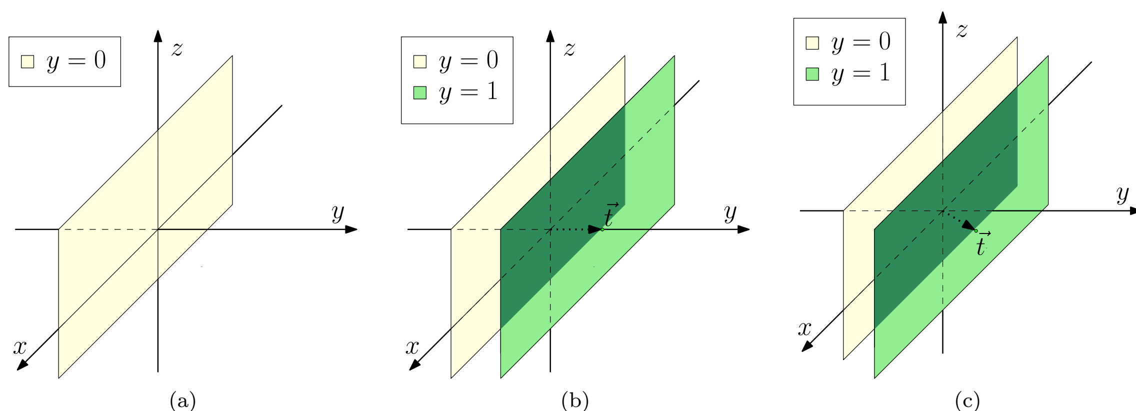

Figure 2.2 An illustration of affine spaces. In figure 2.2a the subspace S induced by the linear equation $y = 0$ is shown. In figure 2.2b and figure 2.2c S is translated with two different translation vectors $\vec{t}$, both yielding T (the set of solutions for the linear equation $y = 1$). T is an affine space.

origin vectors. Affine spaces are of the form:

$$W = \vec{a} + V = \{\vec{a} + \vec{v} : \vec{v} \in V\},$$

for V a vector space.

Because an affine space does not have an origin, addition does not make sense. However, in our case we will define our own kind of addition on affine subspaces. This allows us to perform various proofs in chapter 7. For more information, the reader can read Dodson and Poston [20]. To be able to define addition of two elements in an affine subspace, we fix a unique element (in fact, any unique element in the affine subspace is enough). To this end, we introduce an ordering on $\mathbb{F}_2^d$ using `bs2ip`, and select the smallest element.

Definition 12. *The numerically minimal element o of an affine subspace $A \subseteq \mathbb{F}_2^d$ is the unique element satisfying:*

$$o \in A$$
$$\forall a \in A : \texttt{bs2ip}(o) \leq \texttt{bs2ip}(a)$$

Definition 13. *Let an affine subspace $A \subseteq \mathbb{F}_2^d$ be given and its numerically minimal element be o. Then the addition of two strings $s, s' \in A$ with regard to the affine subspace A, denoted as $\oplus_A$ is defined as:*

$$s \oplus_A s' = s + s' + o$$

o is also called the origin with respect to $\oplus_A$.

We will use some functions in this thesis and will need to reason about the image of these functions. Therefore, we need to define the image of a function:

Definition 14. *Let a function $f : X \to Y$ be given. The image $\mathrm{im}(f) \subseteq Y$ of a function f is defined by:*

$$\mathrm{im}(f) = \{f(x) : x \in X\}.$$

Chapter 3

Sponge and Duplex constructions

Sponge constructions are a type of construction, originally used to construct hash functions from permutations. As stated in section 2.1 originally hash function were built from one-way compression functions using constructions like the Merkle–Damgård construction. Sponge constructions offer an alternative, by allowing usage of one-way compression functions as well as permutations.

Lately, several complications to constructions such as the Merkle–Damgård construction are found, as described below. Because hash functions combine arbitrarily large input data into a fixed size digest, collisions will always occur. Sponge constructions are not different in this matter. However, in a simple Merkle-Damgård construction it is easy to generate multiple collisions, when a single collision has been found. For instance, if a certain message pair M and M' (where the length of M and M' are both a multiple of the block length) generate a collision, then the messages $M \mid X$ and $M' \mid X$ will generate the same digest for any message X. This is because the blocks are digested sequentially, and when the Merkle-Damgård construction has processed either M or M' it is in the same state. Secondly, a need has risen to generate arbitrarily long digests. For instance, this allows someone to use one algorithm to generate keys of different sizes. Because Merkle-Damgård constructions only output a fixed length block, it is not straightforward to generate an arbitrarily long digest. Sponge constructions are designed, however, to output arbitrary long digests. It is even possible to output an infinite string of data, which allows sponge constructions to be used as a PRNG [5].

In contrast to the Merkle-Damgård construction, sponge constructions take a different approach to building such a hash function: instead of using a compression function, they allow the usage of a permutation function. In contrast to compression functions, permutations are invertible. To avoid generating multiple collisions whenever a collision between two input messages is found, sponge constructions use a state. The state is used to generate the digest. It is desirable that there are many more possible states than possible digests. If correctly designed, multiple states can generate the same digest and make it difficult to determine the state from the digest. Under that assumption, a collision in the digest does not directly imply a collision in the state. Moreover, the size of the state can be chosen much larger that the size of the digest. This allows the designer to make it very difficult to find collisions in the state. Furthermore, an arbitrarily long output string can be generated by repeatedly permuting the state and extracting a part of the state in between the permutations. Again,

because the state can be chosen large enough, the first part of the output string does not determine the remaining part of the output string. In section 3.1 the sponge construction is explained.

One should take into account that to solve the problem where one collision can be used to generate multiple collisions more advanced Merkle-Damgård constructions have also been developed. These more advanced constructions use a finalisation part to compress a larger internal state at the end of the construction to the required length, simulating the same principle that a sponge construction uses.

Duplex constructions are an extension upon sponge constructions. In sponge constructions, input data is first processed before an arbitrarily long data string is outputted. Duplex constructions allow the input data and the output data to be interleaved. In section 3.2 the duplex construction is explained. Further uses of the sponge and duplex construction are explained in section 3.1.1 and 3.2.1. This chapter explains sponge constructions and duplex constructions by summarizing [5].

3.1 Sponge construction

A sponge function $h(M, l)$ is a function that, given an arbitrary but finite bit string M and a length l, outputs a bit string of length l only dependent on M. It is often required that $h(M, l)$ should be one-way: it must be hard to reconstruct M using $h(M, l)$. Moreover, we often also require that it should be hard to find collisions for h.

A sponge construction $\texttt{sponge}[f, \texttt{pad}, r]$ constructs a sponge function given a permutation f, a padding rule pad and an integer r. Here f is a permutation, i.e. a one-to-one map from bit strings of length b to bit strings of length b. This length b is implicitly given to the sponge construction, and is also called the *width*. Security of the sponge construction is determined by the properties of f. Informally, the goal is to have the permutation behave as a random oracle. A random oracle responds to every unique query with a completely random, uniform, response [2]. A random oracle will, however, always respond to the same query with the same response. As described before, a sponge construction uses a state. This state is also of length b. pad is a padding rule. A padding rule $\texttt{pad}[x](|M|)$ outputs a bit string, such that the length of $M \mid \texttt{pad}[x](|M|)$ is a multiple of x and given $M \mid \texttt{pad}[x](|M|)$ the padding portion of the bit string can be identified and removed, so that M can be found. The parameter r describes the *bit rate* of the sponge construction. The bit rate is the number of bits the sponge construction can absorb in its state before the permutation function is called. Because the bits are absorbed into the state, the bit rate must be smaller than the width b: $r < b$.

A sponge construction $\texttt{sponge}[f, \texttt{pad}, r]$ constructs a sponge function from a permutation as follows:

- An input M and output length l is given.
- The sponge construction uses a state of size b. This state is initialized to the zero bit string: $s_1 = \mathscr{O}^b \in \mathbb{F}_2^b$.
- The message M is padded, such that its length is a multiple of r: $M' = M \mid \texttt{pad}[r](|M|)$.
- M' is split into a sequence of n blocks of length r: $M' = M'_1 \mid M'_2 \mid \ldots \mid M'_n$.

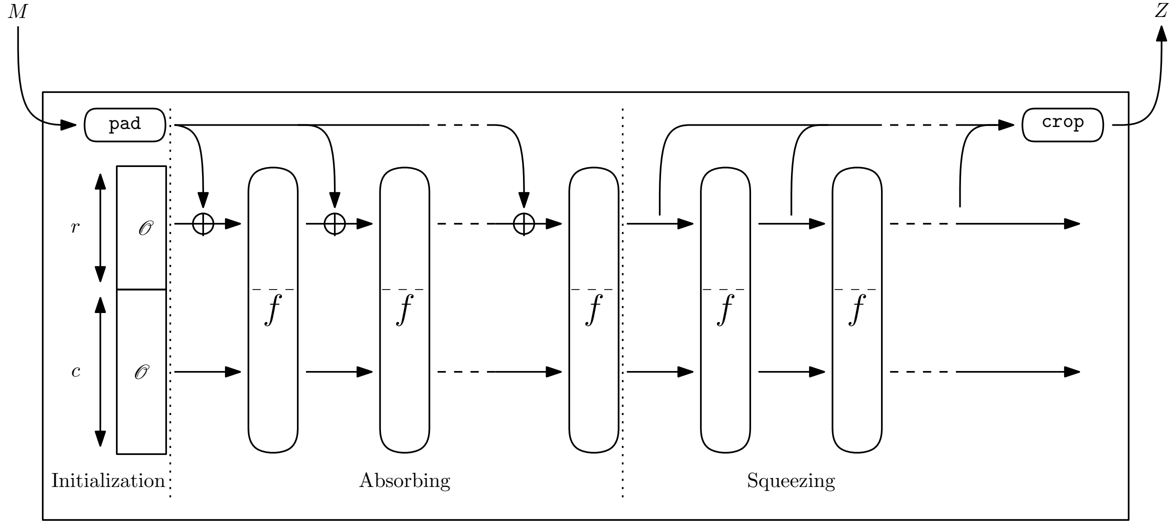

Figure 3.1 A diagram of the sponge construction

- Then the **absorbing phase** absorbs the blocks into the state. Each block is processed sequentially, by first performing an exclusive or of the block with the state, and afterwards permuting this state with f. Formally: $s_i = f(s_{i-1} + (M_{i-1} \mid \mathscr{O}^{b-r}))$ for $1 < i \leq n+1$.

- In the **squeezing phase** the output blocks are generated from the state. This output is generated from the first r bits of the state. More than r bits can be generated by permuting the state until enough bits are generated. Formally we have $s_i = f(s_{i-1})$ for $i > n+1$. The output string is denoted by $Z = Z_1 \mid \ldots \mid Z_m$, with $m = \lceil \frac{l}{r} \rceil$. Each output block Z_i can be derived from the states as follows:

$$Z_i = \begin{cases} \texttt{crop}(s_{i+n}, r) & \text{if } i < m, \\ \texttt{crop}(s_{m+n}, l - (m-1) \cdot r) & \text{otherwise,} \end{cases}$$

  where

$$\texttt{crop}((v_1, \ldots, v_j, \ldots, v_{\hat{n}}), j) = (v_1, \ldots, v_j) \quad \text{for } v \in \mathbb{F}_2^{\hat{n}}, j, \hat{n} \in \mathbb{N}, j < \hat{n}.$$

Note that it is not necessary to remember previous states, and the implementation can repeatedly calculate the next state, generating the output of each block whenever the state required for a certain output block is reached. Figure 3.1 illustrates the construction with a diagram. A more detailed and formal definition of the sponge construction can be found in [5, algorithm 1].

As can be seen in figure 3.1 the last $b - r$ bits of the state cannot be read from or written to by the user. This part of the state is referred to as the inner state, and its width is denoted by $c = b - r$, also called the *capacity*. The first r bits of the state are public and can be read from or written to by the user, depending on the phase. This part is called the outer state, and its width is the rate r. The inner state cannot be modified by the user. As the user can only affect the outer state, the user can only generate a collision if he or she can generate a collision in the inner state. Creating a collision in

the inner state costs in the order of $2^{\frac{c}{2}}$ queries when using a general attack. Therefore, to achieve a security level s, we need to have $c \geq 2 \cdot s$. As such, c determines the security of the construction [4].

3.1.1 Applications

Sponge functions can be used in wide variety of applications. Some of them are briefly explained in this section.

Cryptographic hash functions

As described before, hash functions are functions that map bit strings of arbitrary size to data of fixed size (called *hash value*). These can be very simple functions and are used in a wide variety of applications. For instance, hash functions can be used in hash tables for quickly comparing data for equality. In our case, we are interested in cryptographic hash functions. These functions are made for usage in cryptography. Sponge constructions can be used in cryptographic key derivation, message authentication codes and password hashing.

These different applications of cryptographic hash functions have different and possibly conflicting requirements, as described below.

- For cryptographic key derivation and MAC algorithms calculating the hash should be fast: this puts less strain on the system using the algorithms.
- In password hashing, passwords are often simple/short and can be brute forced by trying all possible values or using a list of commonly used passwords. In order to slow down such an attack, generating a hash value should take a considerable amount of time.

  Moreover, it should be hard to generate the input message (in this case, the password) from the hash value. Furthermore, it should be hard to generate collisions, as this would implicate that the system would accept multiple passwords.
- In MAC functions the secret key should not be derivable from the MAC. Furthermore, it should not be possible to modify the input message and use the original hash value to generate a correct MAC value without having access to the secret key.
- In cryptographic key derivation, the hash value should be generated as uniformly as possible over the key space. Moreover, the length of the hash value generated should match the key length required.

More requirements can also be formulated, but that is outside of the scope of this thesis. Many hash functions can only output a hash of a fixed size. Therefore, it can be difficult to match the output size of a hash function with the key size of another cryptographic algorithm, when using the hash function in cryptographic key derivation. Sponge functions do not have this limitation: a hash of any arbitrary length can be generated.

Furthermore, we have seen that different applications require a computationally efficient or slow hash function. Often a hash function is only designed to be computationally efficient. To build a computationally difficult hash function from a fast hash function several methods have been published, such as PBKDF2 [24]. Moreover, some hash functions are specialized to handle this case: such

as bcrypt. Sponge functions have this functionality automatically: the algorithm can be used to generate a hash quickly or slowly. By using a sponge function in the regular way a hash is generated relatively quickly. To generate a hash in a more computationally difficult manner, a very long hash can be generated discarding all but the last bits. Because the permutation function must be applied each time r bits are output, this can be used to increase the number of permutations, and thus the computational difficulty.

Pseudorandom number generator

Sponge functions can be used to generate random numbers for usage in cryptographic protocols as a CSPRNG. This requires that the sponge function produces uniformly distributed bits and passes several statistical tests.

Stream cipher

Stream ciphers generate a keystream that is combined with the input message. As a sponge can generate an arbitrarily long output, this can be seen as a keystream. Sponge functions can be used as such a stream cipher by providing the symmetric key as input.

3.2 Duplex construction

As just shown, sponge functions have a great flexibility in their purposes: they can be used as a PRNG, hash functions, et cetera. Two interesting uses of sponge functions are their application in stream ciphers and MACs. In some cases we both need a symmetric-key cipher and a MAC. This combination creates a cipher with authentication, also called an authenticated encryption scheme. This scheme simultaneously provides confidentiality, integrity, and authenticity. Building such an authentication scheme directly from a sponge function is impractical. Building a MAC function requires absorbing all blocks first into the inner state before outputting the MAC. If we use such an approach to build an authenticated encryption scheme, encrypting a bitstream requires two passes: first to absorb all blocks into the inner state to generate the MAC, and then to encrypt the blocks by generating the encryption stream using the symmetric key. This requires twice the amount of computation required for a stream cipher. The duplex construction solves this problem by allowing both absorption of blocks into the state as squeezing data out of the state at the same time.

A duplex object D is an interface with a state and one operation: $D.\texttt{duplexing}(M, l)$. Just as a sponge function, a duplex object has a bit rate r. A duplex object is also dependent on k: the maximum length of the block that can be absorbed. After creation a duplex objects always starts in the same state. $D.\texttt{duplexing}(M, l)$ with $l \leq r$ is a function that, given a finite bit string M of maximum length k outputs a bit string of length l only dependent on M and the current state of D. This means that the output is only dependent on all previous M's that were passed to the duplex object, including the order. Again, $D.\texttt{duplexing}(M, l)$ should be one-way: it must be hard to reconstruct M, or any of the other previous blocks passed to D. Moreover, it should be hard to find collisions between outputs. In practice $D.\texttt{duplexing}$ will output a pseudorandom bit string.

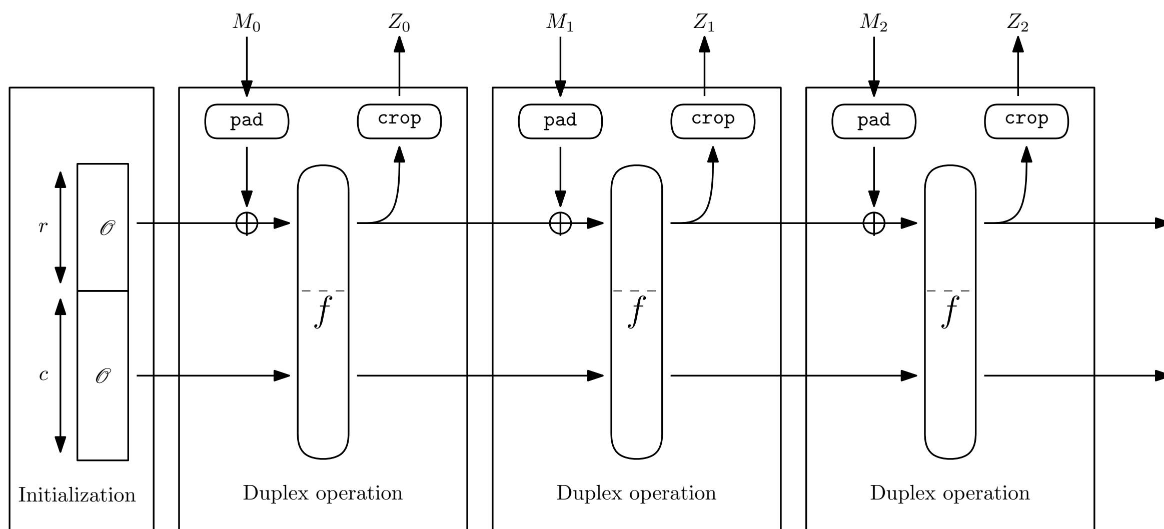

Figure 3.2 A diagram of the duplex construction

A duplex construction $\texttt{duplex}[f, \texttt{pad}, r]$ constructs a duplex object given a permutation f, a padding rule pad and an integer r. The meaning of each of these parameters is the same as in the sponge construction. Moreover, the meaning of inner state and outer state has not changed: the constructed duplex object has a state size of $b = r + c$.

A duplex construction constructs and defines the operation on a duplex object as follows:

- A duplex object is constructed with an empty state $s = \mathscr{O}^b \in \mathbb{F}_2^b$.
- The variable k is set to the maximum length of the block that can be absorbed. k is dependent on pad, it must be chosen such that $M_i \mid \texttt{pad}[r](|M_i|)$ with $|M_i| \leq k$ will always result in a bit string of length r. Note that pad must be applied to each block M_i separately as we allow any length per block less than or equal to k.
- Whenever a duplexing operation has to be performed on the duplexing object, the following operations will be performed:
  - An input M_i and output length l is given, with $|M_i| \leq k$ and $l < r$;
  - The state s is updated: $s \leftarrow f(s + (M_i \mid \texttt{pad}[r](|M_i|) \mid \mathscr{O}^c))$;
  - The first l bits of s are output to the user.

In figure 3.2 the construction is illustrated with a diagram. A more detailed and formal definition of the sponge construction can be found in [5, algorithm 2].

It is interesting to note that a duplex construction is in various ways similar to a sponge construction, that is, most security properties applicable to sponge constructions are also applicable to duplex constructions. Lemma 1 formalizes some of these similarities.

Lemma 1 (Duplexing-sponge lemma). *If we denote the input to the ith call to a duplex object (M_i, l_i) and the corresponding output by Z_i we have:*

$$Z_i = D.\mathtt{duplexing}(M_i, l_i) = \mathtt{sponge}(M_1 \mid \mathtt{pad}_1 \mid M_2 \mid \mathtt{pad}_2 \mid \ldots \mid M_i, l_i)$$

with $\mathtt{pad}_i$ a shortcut notation for $\mathtt{pad}[r](|M_i|)$ *[5, p. 3].*

Please note that M_i is not padded in lemma 1 when used as an input to the sponge construction. This is because the sponge construction itself will pad M_i and if we would pad M_i ourselves, then it would be padded twice.

3.2.1 Applications

Duplex objects can be used in place of sponge functions, as they can perform at least the same functionality as sponge functions. Most of the extra functionalities provided by a duplex function are more advanced versions of the functionalities provided by a sponge function.

Reseedable pseudorandom number generator

As described earlier, a sponge function can generate cryptographically secure random numbers. Often, a PRNG is seeded with data generated by a truly random source. Often, such a truly random source can only generate a limited amount of data in a certain time frame. However, after using the PRNG for a while, this truly random source may be able to generate more truly random data. By allowing input messages while simultaneously generating output, a duplex function can act as a reseedable pseudorandom number generator: new entropy can be added to the state while in use.

Authenticated encryption

As mentioned at the start of section 3.2, a duplex construction can be used to provide authentication and encryption at the same time. That is, it can be used to create an efficient authenticated encryption scheme [33].

3.3 Permutation types

For most cryptographically secure sponge or duplex constructions, f is often chosen to be a permutation. A permutation is a bijective function, in this case mapping bit strings of some length to bit strings of the same length. The design of the permutation function determines the security of the sponge or duplex construction.

It should be considered, that f does not need to be a permutation. In fact, the best known general attack to find secondary preimages has a lower order when f is a permutation than when f is a non-injective transformation. Therefore, sponge constructions with f a non-injective transformation might be preferred when designing a hash function and do exist [3]. On the other hand, other general attacks have a higher cost when f is a permutation. Nonetheless, for general purpose usages both

permutations as non-injective transformation can be equally secure, as the strongest general attack against both types has the same order [22].

In 1945, Claude Shannon reported two important properties a cryptographic function should have: confusion and diffusion. Confusion seeks to make the relationship between the statistics of the ciphertext and the value of the input as complex as possible. Diffusion seeks to make a single input digit affect, in general, a lot (statistically a half) of the output bits [45]. It might be possible that a single input affects only some bits or even only one bit, but this should be a rare occurence.

Another important property that the permutation function should have is non-linearity. This means that each output bit cannot be defined by only performing an exclusive-or on some of the input bits. This is important because a linear permutation can be attacked by generating a linear system from its inputs and outputs and performing Gaussian elimination. The attacks described in this thesis are roughly based on such an approach.

3.3.1 ARX-based

ARX stands for algorithms that rely on modular Addition, Rotation and eXclusive-or. ARX algorithms are fairly attractive in the area of lightweight cryptography and anywhere where simplicity and efficiency is required. Furthermore, ARX provides some defences against hardware attacks such as cache-timing and side-channel attacks [12].

In ARX, modular addition provides non-linearity [27]. Unfortunately, while ARX algorithms can provide a good level of security, their methods of analysis are limited. This makes it hard to design such an algorithm without relying on a lot of heuristics [12].

Several general attacks have been formulated for ARX-based algorithms. The branch of cryptography concerning these attacks is called Rotational Cryptanalysis. Techniques that are universally applicable to any ARX algorithm exist, however the complexity of such an attack depends on the number of ARX operations. This means that such an attack is not feasible for all types of ARX algorithms. As a side note, any cryptographic algorithm can actually be implemented using modular addition, rotation and exclusive-or operations, but this attack is not feasible for all cryptographic algorithms [27].

3.3.2 S-box-based

S-box-based(*Substitution-box-based*) algorithms rely on S-boxes to provide non-linearity. S-boxes transform a relatively small number of bits (relative to the input size). When S-boxes are used to build a permutation they are bijective mappings and are designed in such a way that the relation between the input and output is complicated. A designer of a cryptographic primitive has a lot of freedom by being able to specify any bijective operation, as the designer can specify the output for each specific input. Designing a good S-box with a lot of input bits is difficult, however. Moreover, an S-box is often implemented using a lookup-table to avoid using too many operations. This limits the number of input/output bits an S-box uses [35]. For instance, the well known AES S-box uses only 8 input bits. Such an S-box operation is then commonly applied side-by-side on all input bits, such that bits 0-7 are in the same S-box, the bits 8-15 in the same S-box, et cetera. Applying the S-box side-by-side is often called the SubBytes operation, the SubElements operation or similar. When

an S-box is implemented using a lookup-table, the algorithm using this S-box can be susceptible to cache-timing and side-channel attacks.

Therefore, an S-box does provide confusion, but the input to the S-box only affects a limited number of output bits (that is, at most the output size). In this case, the other operations should mix the output bits so that a certain part of the input can affect any output bit. These operations often provide no non-linearity. Because an S-box provides non-linearity on a limited number of input bits, an S-box is often applied multiple times after a diffusion layer is applied. This is commonly done using rounds.

This thesis is mostly interested in S-box-based permutations.

Mathematical view on S-box

Because exclusive-or (with a variable or a constant) and logical and operations are together functionally complete, any mapping in $\mathbb{F}_2^d$ can be built using addition and multiplication. If each output bit of an S-box is seen as a combination of exclusive-or and logical and operations, these functions can be seen as polynomials in $\mathbb{F}_2$. Because the degree of a polynomial in $\mathbb{F}_2$ is limited by the number of input bits, the degree of these S-box is limited.

Because the other operations are linear, the degree of these S-box-based permutations is dependent only on the number of SubElements operations in the permutation. In this thesis, we will exploit the low degree of an S-box-based algorithm and formulate attacks on its permutations. Not all S-box-based algorithms are vulnerable to our attack: this depends on the polynomial degree of the S-boxes and the number of rounds in the algorithm.

In contrast, generally modular addition operates on a larger number of bits. While modular additions can also be modelled by a polynomial in $\mathbb{F}_2$ the degree is most often sufficiently high. On the one hand, this makes analysis harder because this produces high degree polynomials with a certain structure. Attacks on these operations exist because of the limited possible polynomials using modular addition. On the other hand, attacks based on a low degree when modelling these polynomials in $\mathbb{F}_2$ are infeasible.

Chapter 4

Higher-order differential cryptanalysis

As mentioned in chapter 1, differential attacks are together with linear cryptanalysis the most significant type of attacks on symmetric cryptography [23]. Linear cryptanalysis is a type of attack, that formulates linear relations between the input and output data that hold with a high probability. The exact process is out of the scope of this thesis, more information can be found in [31]. Differential attacks are attacks that exploit relations between differences induced in the input data on the output [8].

As we have seen in chapter 3 on page 23 all functions can be represented as polynomials in $\mathbb{F}_2$. Moreover, the degree of permutations based on S-boxes are often low. In this chapter we formulate a general attack on S-box-based permutations used in duplexes. The attack is based on the work of Lai [30] and Seidlová [43]. In this chapter we will formulate our attacker model, briefly summarize the findings by Seidlová and specify the attack. In chapter 5 the attack is applied to the PRIMATEs family of permutation-based authenticated encryption algorithms for lightweight applications: PRIMATEs is a submission to the CAESAR competition. While this thesis was being written the PRIMATEs family was rejected from round three of the CAESAR competition.

4.1 Attack model

In chapter 3 duplex and sponge function were explained. In this chapter, a general attack on duplex functions using permutations will be formulated. Whether the formulated attack is successful on the duplex function depends on properties of the permutation: the degree as will be explained below, the capacity and the rate of the duplex function.

Our attack follows the chosen plaintext attack (CPA) model. Because CPA models are often formulated for block ciphers, we will formulate below under which assumptions our attack is made.

Firstly, a duplex object is attacked. In our setting, the duplex object has a certain secret inner state. The goal is to determine this inner state. We assume that we have no information about this inner state, and thus, cannot read from the inner state. We can, however, read the outer state. Furthermore, we can write to the outer state by performing duplexing calls to the duplex object. By performing duplexing calls, we can also read the outer state after a duplex call has been made. We assume that we can set the duplex state to a certain state multiple times, for instance by controlling

all input that is given to the duplex function. Lastly, it is assumed that the attacker knows the inner workings of the permutation function and can perform offline calculations while having complete control over the permutation function.

To illustrate why these capabilities are needed, an overview of our attack is given below. Our attack can be divided in three phases:

- **Offline phase**, in this phase the attack is prepared. This phase is not dependent on the internal state about which we need to find information. Therefore, this attack can be run on a local computation system. The output of the offline phase are linear equations that generate information about the inner state. This phase is the most computationally-complex phase. In this phase the permutation is analysed and thus requires an implementation of the permutation that can be run locally on a computer.
- **Online phase**, in this phase duplexing calls are made to the initial duplex object (that is, the duplex object with the secret inner state). The input that is given to the duplexing calls will be chosen in the offline phase. The outputs of the duplexing calls are saved.
- **Solving phase**, in this phase the outputs of the duplexing calls made in the online phase are combined with the linear equations of the offline phase. The offline phase specifies linear equations between combinations of inputs and combinations of outputs produced by the duplex object. This linear system is solved, giving the inner state.

4.2 Differential analysis

The state of a duplex function with rate r and capacity c is a sequence of $b = c + r$ bits. Mathematically, a duplex function D has a state $s \in \mathbb{F}_2^b$, and uses a permutation $f : \mathbb{F}_2^b \to \mathbb{F}_2^b$. The state s can be written as $s = (s_1, \ldots, s_b) = (\alpha_1, \ldots, \alpha_r, \beta_1, \ldots, \beta_c)$. Then $\alpha = (\alpha_1, \ldots, \alpha_r)$ is the outer state of s and $\beta = (\beta_1, \ldots, \beta_c)$ is the inner state.

An attacker can only read and write to the α_i values of s and calculate other states by performing $f(s)$. The permutation $f(s)$ generates a new state. In our analysis we split $f(s)$ per bit:

$$f(s) = (f_1(s), \ldots, f_i(s), \ldots, f_b(s)),$$

where $f_i : \mathbb{F}_2^b \to \mathbb{F}_2$.

As we mentioned in chapter 3, all functions can be constructed using exclusive-or and logical and operations. As these operation are equivalent to addition and multiplication in $\mathbb{F}_2$, we can write f_i as a polynomial. Such a polynomial f_i consists of a summation of multiple monomials. A monomial is the product of a number of input variables. In $\mathbb{F}_2$ this means that a monomial is a logical and operation applied to a series of input bits. In $\mathbb{F}_2$ we will write a monomial $m(x) : \mathbb{F}_2^b \to \mathbb{F}_2$ as:

$$m(s) = s_{i_1} \cdot s_{i_2} \cdot \ldots s_{i_j} = s^H,$$

where $H = \{i_1, \ldots i_j\}$ is the index set. Each index must be between 1 and b. As a shorthand, we write $\mathcal{I}_b^+$ for all indices between 1 and b. Thus, $\mathcal{I}_b^+ = \{1, \ldots, b\}$ and $H \subseteq \mathcal{I}_b^+$. Any polynomial can

be written as the sum of monomials. The form in which we write our polynomials is referred to as the algebraic normal form [13, p. 79]. In algebraic normal form we write f_i as:

$$f_i(s) = \sum_{H \subseteq \mathcal{I}_b^+} t_H s^H,$$

with $t_H \in \mathbb{F}_2$. Essentially, t_H is a boolean constant that determines if s^H is a term of f_i. Please note that f_i can also have a constant value, this is encoded as $t_\varnothing$.

By performing summation while varying various input variables, we can reduce the degree of f_i. For instance, write f_i as follows:

$$\begin{aligned} f_i(s) &= \sum_{H \subseteq \mathcal{I}_b^+} t_H s^H \\ &= \sum_{H \subseteq \mathcal{I}_b^+ \setminus \{1\}} t_{(H \cup \{1\})} s^{(H \cup \{1\})} + \sum_{H \subseteq \mathcal{I}_b^+ \setminus \{1\}} t_H s^H \\ &= s_1 \cdot \sum_{H \subseteq \mathcal{I}_b^+ \setminus \{1\}} t_{(H \cup \{1\})} s^H + \sum_{H \subseteq \mathcal{I}_b^+ \setminus \{1\}} t_H s^H \\ &= s_1 \cdot p(s) + q(s), \end{aligned}$$

with $p(s)$ and $q(s)$ independent of s_1. We have split f_i into two parts: one dependent on s_1 and one independent of s_1.

To simplify the notation we repeat the definition of the zero vector $\mathscr{O}$ and the unit vectors e_j.

$$\begin{aligned} \mathscr{O} &= (\mathbb{0}, \ldots, \mathbb{0}) \\ e_j &= (\mathbb{0}, \ldots, \mathbb{0}, \overbrace{\mathbb{1}}^{j^{\text{th}} \text{ element}}, \mathbb{0}, \ldots, \mathbb{0}) \end{aligned}$$

We can extract $p(s)$ by changing only the first bit of s and performing an addition of the evaluations:

$$\begin{aligned} f_i(\mathbb{0}, s_2, \ldots, s_n) + & f_i(\mathbb{1}, s_2, \ldots, s_n) \\ &= \mathbb{0} \cdot p(\mathbb{0}, s_2, \ldots, s_n) + q(\mathbb{0}, s_2, \ldots, s_n) + \\ &\quad \mathbb{1} \cdot p(\mathbb{1}, s_2, \ldots, s_n) + q(1, s_2, \ldots, s_n) \\ &= \mathbb{0} \cdot p(s) + q(s) + \mathbb{1} \cdot p(s) + q(s) && (p(s), q(s) \text{ independent of } s_1) \\ &= p(s) + (\mathbb{1} + \mathbb{1}) \cdot q(s) = p(s). \end{aligned}$$

By extracting $p(s)$ from $f_i(s)$, we are effectively reducing the degree: if the degree of $f_i(s) = d$ then the degree of $p(s)$ must be equal to or smaller than $d - 1$.

This can be done in the same manner for other bits. Take $p(s)$ and $q(s)$ independent of bit j and such that:

$$f_i(s) = s_j \cdot p(s) + q(s).$$

For simplicity we introduce the differential notation δ_j:

$$\begin{aligned}
\delta_j(f_i)(s) &= f_i(s + \mathbb{0} \cdot e_j) + f_i(s + \mathbb{1} \cdot e_j) \\
&= f_i(s_1, \ldots, s_{j-1}, s_j + \mathbb{0}, s_{j+1}, \ldots, s_n) + f_i(s_1, \ldots, s_{j-1}, s_j + \mathbb{1}, s_{j+1}, \ldots, s_n) \\
&= f_i(s_1, \ldots, s_{j-1}, \mathbb{0}, s_{j+1}, \ldots, s_n) + f_i(s_1, \ldots, s_{j-1}, \mathbb{1}, s_{j+1}, \ldots, s_n) \\
&= \mathbb{0} \cdot p(s) + q(s) + \mathbb{1} \cdot p(s) + q(s) \\
&= p(s) \\
&= \sum_{H \subseteq (\mathcal{I}_b^+ \setminus \{j\})} t_{(H \cup \{j\})} s^H. \qquad (4.1)
\end{aligned}$$

The differential operator δ_j can also be applied to the function $f = (f_1, \ldots, f_b)$. The same properties then hold, but t_H will be in the set $\mathbb{F}_2^b$ instead of $\mathbb{F}_2$. Obviously, we need to apply the differential over multiple bits to generate a linear equation. To this end, let a set $G = \{g_1, \ldots, g_i, \ldots, g_m\} \subseteq \mathbb{N}$, $1 \leq g_i \leq n$ be given and we define:

$$\begin{aligned}
\delta_G(f_i)(s) &= \delta_{g_1}(\delta_{g_2}(\ldots \delta_{g_m}(f_i) \ldots))(s) \\
&= \sum_{H \subseteq (\mathcal{I}_b^+ \setminus G)} t_{(H \cup G)} s^H. \qquad (4.2)
\end{aligned}$$

By choosing G appropriately, $\delta_G(f_i)$ can be linear. However, to formulate an attack, we need to be able to determine the representation of $\delta_G(f_i)$. Essentially, this means that we need to determine the values of t_H. Firstly, we should note that $\mathscr{O} = (\mathbb{0}, \mathbb{0}, \ldots, \mathbb{0})^H$ is $\mathbb{0}$ except when $H = \varnothing$. When H is $\varnothing$, s^H becomes the empty product, which is $\mathbb{1}$. This gives a way to retrieve $t_\varnothing$:

$$f_i(\mathscr{O}) = t_\varnothing.$$

To retrieve other t_H's we can combine this approach together with applying δ_G:

$$\begin{aligned}
\delta_G(f_i)(\mathscr{O}) &= \sum_{H \subseteq (\mathcal{I}_b^+ \setminus G)} t_{(H \cup G)} \mathscr{O}^H \\
&= t_G. \qquad (4.3)
\end{aligned}$$

One can also calculate $t_{(G \cup \{j\})}$ when t_G is known by using the following observation:

$$\delta_G(f_i)(e_j) = t_G + t_{G \cup \{j\}}. \qquad (4.4)$$

Because the computation difficulty of δ_G doubles when the number of elements in G increases by one, this trick can halve the computation time. We did not use this trick in our implementation, as we discovered this observation in a late stage.

By generating linear equations and determining the representation of these equations, we can build a linear system. Building this linear system will be explained in section 4.3.2. Elaborate proofs of these statements are omitted in this paper and one can read [30] or [43] for more information.

Lastly, to simplify the notation we define for $f = (f_1, \ldots, f_b)$:

$$\delta_G(f) = (\delta_G(f_1), \ldots, \delta_G(f_b)).$$

4.2.1 Differential along a vector space

Up to now, we only considered differentials along a set of input variables: the input variables specified can be varied independently. For completeness, we will have to note that this theory can also be extended to differentials along a vector space. This allows some more advanced attacks. In our differential attack we do not need this more generalized approach. To be able to translate our δ function to a function that does calculate the differentials along vector spaces instead of input variables, we define the vector space W_H generated by varying the bits indexed by the set H:

$$W_H = \text{span}(\{e_h : h \in H\}).$$

In *'Higher Order Differential Analysis of NORX'* [16] the derivative along a vector space is defined as follows:

Definition 15. *The derivative of function $f : \mathbb{F}_2^b \to \mathbb{F}_2$ with respect to a linear subspace V of $\mathbb{F}_2^b$ is defined as*

$$\Delta_V f(s) = \sum_{v \in V} f(s + v).$$

There exists an equivalence between $\Delta_V f(s)$ and $\delta_G(f)(s)$ for $V = W_G$:

$$\delta_G(f)(s) = \Delta_{W_G} f(s).$$

Moreover, $\Delta_V f$ reduces the degree of f in the same manner as the δ operator:

Theorem 1. *Let V be a vector space in $\mathbb{F}_2^b$ and $f : \mathbb{F}_2^b \to \mathbb{F}_2$ a function. Then:*

$$\deg(\Delta_V f) \leq \deg(f) - \dim(V).$$

[43, Theorem 19].

Calculating the polynomial representation of $\Delta_V f(s)$ is done in the same manner as for $\delta_G f(s)$:

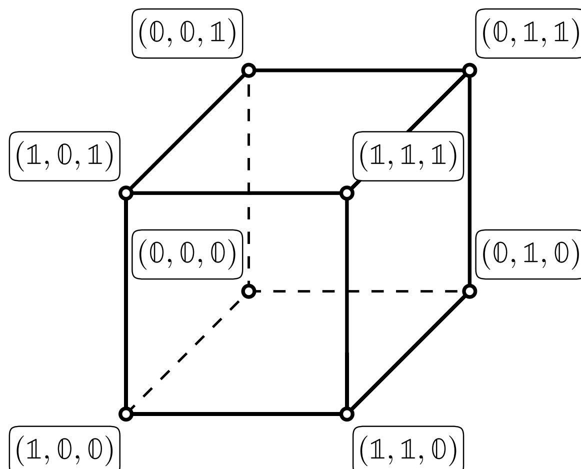

Figure 4.1 The cube represented by $W_G \in \mathbb{F}_2^3$ with $G = \{1, 2, 3\}$.

Theorem 2. *Let $f : \mathbb{F}_2^v \to \mathbb{F}_2$ be a function and*

$$\Delta_V f(s) = \sum_{H \subseteq \mathcal{I}_b^+} t_H s^H$$

its derivative along the linear space V. For any $H \subseteq \mathcal{I}_b^+$, it holds that

$$t_H = \begin{cases} \Delta_{W_H}(\Delta_V f)(\mathscr{O}) = \delta_H(\Delta_V f)(\mathscr{O}) & \text{if } V \cap W_H = \{\mathscr{O}\} \\ \mathbb{0} & \text{otherwise.} \end{cases}$$

[43, Theorem 17].

Proving these properties of Δ_V is more difficult than proving the properties of δ_G, but the idea is the same. These theorems are proven in [43].

4.3 Cube attacks

As seen above, we can generate linear equations, by applying the δ_G function multiple times for different values of G. If we take a look at the values in W_G, we see that W_G represents all points in $|G|$-dimensional hypercube. The attack then uses the sum of the result of each vertex in the hypercube [19, p. 3]. An illustration of this can be seen in figure 4.1. Therefore, following Dinur and Shamir we call attacks using the δ_G function *cube attacks.*

We will illustrate the attack by applying it to an example and then specify the steps required to perform such an attack.

4.3.1 An example attack

Let $f = (f_1, f_2, f_3, f_4)$ be a permutation on $\mathbb{F}_2^4$. Let f be some permutation of degree 2. Let $\tilde{f} = (f_1, f_2)$. By applying the techniques described in this chapter, we will find information about

the inner state. We will attack a duplex construction D using f, with parameters $r = 2$ and $c = 2$. The first step is to generate linear equations. This is done by choosing a set G (in our case $|G| = 1$) and calculating the representation using equation (4.3). We write the polynomial representation of $\tilde{f} = (f_1, f_2)$ as $\sum_{H \subseteq \mathcal{I}_b^+} t_H s^H$ and the representation of f_i as $\sum_{H \subseteq \mathcal{I}_b^+} t_H^{(i)} s^H$ where $t_H = (t_H^{(1)}, t_H^{(2)})$.

In our attack, we can choose $G = \{1\}$ and calculate the representation of $\delta_G(f_1)$ and $\delta_G(f_2)$. This is done by using equation (4.3). In our case, suppose we find the following results:

$$\begin{aligned} t_{\{1\}} &= (t_{\{1\}}^{(1)}, t_{\{1\}}^{(2)}) = \delta_G(\tilde{f})(\mathscr{O}) = \tilde{f}(\mathscr{O}) + \tilde{f}(e_1) = (\mathbb{1}, \mathbb{1}) \\ t_{\{1,2\}} &= (t_{\{1,2\}}^{(1)}, t_{\{1,2\}}^{(2)}) = \delta_{G\cup\{2\}}(\tilde{f})(\mathscr{O}) = \tilde{f}(\mathscr{O}) + \tilde{f}(e_1) + \tilde{f}(e_2) + \tilde{f}(e_1 + e_2) = (\mathbb{1}, \mathbb{1}) \\ t_{\{1,3\}} &= (t_{\{1,3\}}^{(1)}, t_{\{1,3\}}^{(2)}) = \delta_{G\cup\{3\}}(\tilde{f})(\mathscr{O}) = \tilde{f}(\mathscr{O}) + \tilde{f}(e_1) + \tilde{f}(e_3) + \tilde{f}(e_1 + e_3) = (\mathbb{1}, \mathbb{0}) \\ t_{\{1,4\}} &= (t_{\{1,4\}}^{(1)}, t_{\{1,4\}}^{(2)}) = \delta_{G\cup\{4\}}(\tilde{f})(\mathscr{O}) = \tilde{f}(\mathscr{O}) + \tilde{f}(e_1) + \tilde{f}(e_4) + \tilde{f}(e_1 + e_4) = (\mathbb{0}, \mathbb{0}). \end{aligned}$$

This gives us the following linear equations:

$$\begin{aligned} \delta_1(f_1)(s) &= \delta_G(f_1)(\mathscr{O}) + \delta_1(\delta_G(f_1))(\mathscr{O}) \cdot s_1 + \delta_2(\delta_G(f_1))(\mathscr{O}) \cdot s_2 + \\ &\quad \delta_3(\delta_G(f_1))(\mathscr{O}) \cdot s_3 + \delta_4(\delta_G(f_1))(\mathscr{O}) \cdot s_4 \\ &= t_{\{1\}}^{(1)} + \mathbb{0} \cdot s_1 + t_{\{1,2\}}^{(1)} \cdot s_2 + t_{\{1,3\}}^{(1)} \cdot s_3 + t_{\{1,4\}}^{(1)} \cdot s_4 \\ &= \mathbb{1} + \mathbb{1} \cdot s_2 + \mathbb{1} \cdot s_3 + \mathbb{0} \cdot s_4 \\ &= \mathbb{1} + s_2 + s_3, \\ \delta_1(f_2)(s) &= \delta_G(f_2)(\mathscr{O}) + \delta_1(\delta_G(f_2))(\mathscr{O}) \cdot s_1 + \delta_2(\delta_G(f_2))(\mathscr{O}) \cdot s_2 + \\ &\quad \delta_3(\delta_G(f_2))(\mathscr{O}) \cdot s_3 + \delta_4(\delta_G(f_2))(\mathscr{O}) \cdot s_4 \\ &= t_{\{1\}}^{(2)} + \mathbb{0} \cdot s_1 + t_{\{1,2\}}^{(2)} \cdot s_2 + t_{\{1,3\}}^{(2)} \cdot s_3 + t_{\{1,4\}}^{(2)} \cdot s_4 \\ &= \mathbb{1} + \mathbb{1} \cdot s_2 + \mathbb{0} \cdot s_3 + \mathbb{0} \cdot s_4 \\ &= \mathbb{1} + s_2. \end{aligned}$$

Note that $\delta_1(\delta_G(\tilde{f}_1)))$ and $\delta_1(\delta_G(\tilde{f}_1)))$ are both $\mathbb{0}$ as $\delta_G(\tilde{f})$ is independent of s_1. This can be seen in equation (4.1) and theorem 2. Unfortunately, $\delta_1(f_2)$ does not provide any information about the inner state because s_3 and s_4 are not involved. Therefore we try another G, to find another linear equation. Choose $G = \{2\}$ and suppose we find:

$$\begin{aligned} t_{\{2\}} &= \delta_G(\tilde{f})(\mathscr{O}) = \tilde{f}(\mathscr{O}) + \tilde{f}(e_2) = (\mathbb{0}, \mathbb{0}) \\ t_{\{2,1\}} &= \delta_{G\cup\{1\}}(\tilde{f})(\mathscr{O}) = \tilde{f}(\mathscr{O}) + \tilde{f}(e_2) + \tilde{f}(e_1) + \tilde{f}(e_2 + e_1) = (\mathbb{1}, \mathbb{1}) \\ t_{\{2,3\}} &= \delta_{G\cup\{3\}}(\tilde{f})(\mathscr{O}) = \tilde{f}(\mathscr{O}) + \tilde{f}(e_2) + \tilde{f}(e_3) + \tilde{f}(e_2 + e_3) = (\mathbb{0}, \mathbb{0}) \\ t_{\{2,4\}} &= \delta_{G\cup\{4\}}(\tilde{f})(\mathscr{O}) = \tilde{f}(\mathscr{O}) + \tilde{f}(e_2) + \tilde{f}(e_4) + \tilde{f}(e_2 + e_4) = (\mathbb{0}, \mathbb{1}). \end{aligned}$$

This gives us the following linear equations:

$$\begin{aligned} \delta_2(f_1)(s) &= s_1 + \mathbb{0} \\ \delta_2(f_2)(s) &= s_1 + s_4 + \mathbb{0}. \end{aligned}$$

Again, $\delta_2(f_1)$ does not provide any information about the inner state. We choose to drop this equation as it is not useful for our purposes. In the following steps, we could still use the equations of $\delta_2(f_1)$ and $\delta_1(f_2)$, there is just no use for it. In our implementation of this attack, we do not drop these equations, as calculating the values of these equations in the next step is essentially free, as we must calculate $\delta_1(\tilde{f})$ and $\delta_2(\tilde{f})$ to find $\delta_1(f_1)$ and $\delta_2(f_2)$. Just keeping these values around is easier from a programming perspective. In this example, however, we drop them and we get the following linear equations:

$$\begin{aligned} \delta_1(f_1)(s) &= s_2 + s_3 + \mathbb{1} \\ \delta_2(f_2)(s) &= s_1 + s_4 + \mathbb{0}. \end{aligned}$$

Note that we can only read from the first 2 bits, so we can only consider linear equations found by analysing f_1 or f_2. The output of f_3 and f_4 cannot be read directly. Furthermore, we cannot directly write to the last 2 bits, so δ_3 or δ_4 cannot be calculated, because we cannot control the value of s_3 or s_4. Moreover, s_1 and s_2 are known. To use these linear equations to generate an attack, we formulate the linear system in the form $M \cdot x = y$. This gives:

$$\begin{pmatrix} \mathbb{1} & \mathbb{0} \\ \mathbb{0} & \mathbb{1} \end{pmatrix} \cdot \begin{pmatrix} s_3 \\ s_4 \end{pmatrix} + \begin{pmatrix} \mathbb{1} \\ \mathbb{0} \end{pmatrix} + \begin{pmatrix} s_2 \\ s_1 \end{pmatrix} = \begin{pmatrix} \delta_1(f_1)(s) \\ \delta_2(f_2)(s) \end{pmatrix}$$

$$\Rightarrow$$

$$\begin{pmatrix} \mathbb{1} & \mathbb{0} \\ \mathbb{0} & \mathbb{1} \end{pmatrix} \cdot \begin{pmatrix} s_3 \\ s_4 \end{pmatrix} = \begin{pmatrix} \delta_1(f_1)(s) + \mathbb{1} + s_2 \\ \delta_2(f_2)(s) + \mathbb{0} + s_1 \end{pmatrix}.$$

So we have:

$$\begin{aligned} M &= \begin{pmatrix} \mathbb{1} & \mathbb{0} \\ \mathbb{0} & \mathbb{1} \end{pmatrix} \\ x &= \begin{pmatrix} s_3 \\ s_4 \end{pmatrix} \\ y &= \begin{pmatrix} \delta_1(f_1)(s) + \mathbb{1} + s_2 \\ \delta_2(f_2)(s) + \mathbb{0} + s_1 \end{pmatrix}. \end{aligned}$$

So to find s_3 and s_4 of an arbitrary state, we just need to:

- calculate $\delta_1(f_1)(s)$, $\delta_2(f_2)(s)$;
- calculate y. Note that s_1 and s_2 are known;
- solve the linear system $M \cdot x = y$ to find x;
- x contains the values for s_3 and s_4.

4.3.2 Attack procedure

Suppose we attack a duplex construction produced by the permutation f, with rate r, capacity c and width b. As we mentioned above, our attack procedure has three phases:

Offline phase, here we generate linear equations and calculate their polynomial representation. This phase does not require to usage of the actual state, and can be reused for every attack. Various sets $G_1, G_2, \ldots, G_m$ must be chosen, such that $\delta_{G_j}(f_i)(s)$ for $i \leq r$ are linear functions and reveal information about $\beta = (s_{r+1}, s_{r+2}, \ldots, s_b)$. Because $\delta_{G_j}(f_i)(s)$ for each $i \leq r$ can be calculated together, we generate r linear equations per set G_j. Please note that all values in G_j must be at most r, because we cannot set the values of the inner state. Write these linear equations as:

$$M \cdot x = o + d + M_\alpha \cdot \alpha.$$

Here x is a vector of unknowns:

$$x = \begin{pmatrix} s_{r+1} \\ \vdots \\ s_{r+c} \end{pmatrix},$$

o is the output of the $\delta_G(f_i)(s)$ functions and will be determined in the next phase, depending on the state.

$$o = \begin{pmatrix} \delta_{G_1}(f_1)(s) \\ \vdots \\ \delta_{G_1}(f_r)(s) \\ \delta_{G_2}(f_1)(s) \\ \vdots \\ \delta_{G_m}(f_r)(s) \end{pmatrix},$$

d are the constant values of each linear equations, α are the values that will be known when attacking the state (the outer state):

$$\alpha = \begin{pmatrix} s_1 \\ \vdots \\ s_r \end{pmatrix}.$$

The linear terms in the equations are split into the known and unknown terms. The known terms will be written to M_α, the unknown terms will be written to M.

Lastly, the sets $G_1, G_2, \ldots, G_m$ must be chosen such that rank of M is c, otherwise not all inner state values can be correctly determined. We will select the sets adaptively.

Online phase, in this phase we are given the duplex construction with some state s. The first step is to manipulate the outer state values and calculate $\delta_{G_j}(f_i)$ by applying the definition given in equation (4.2). Furthermore, by reading the outer state of s, we can derive the value of α. In summary, we calculate o and α in this phase.

Solving phase, this phase uses the information collected in the previous phases to derive the inner state. Firstly, we can calculate $M_\alpha \cdot \alpha$. Combining this with the vector o (the $\delta_{G_j}(f_i)$ calculated in the online phase) and c gives the constant vector $b = o + d + M_\alpha \cdot \alpha$. Then apply a Gaussian elimination

method to solve the equation $M \cdot x = y$ for x. The value of x is the inner state $\beta = (s_{r+1}, \dots, s_b)$.

4.3.3 Limitations of this attack

Fortunately for permutation-based duplex constructions, this type of attack will not always work. Firstly, theorem 1 gives a good indication about how the δ_G function affects the degree of a boolean function f. If the degree of f is d, then the degree of $\delta_G(f)$ can be at most $d - |G|$. Because G can only contain bits in the outer state, $|G|$ has a maximum value of r. Thus if the degree d of f is bigger than r, there is a high probability no linear equations can be found.

A second factor that affects the success of this attack is the actual polynomial representation. As seen in theorem 1, the maximum degree of $\delta_G(f)$ is $\deg(f) - |G|$. However, there is no reason that there exists a G such that $\deg(f) - |G|$ is 1, so we might not be able to find a G that makes $\delta_G(f)$ linear. To illustrate this, let $r = 5$ and $c = 5$. Define the following boolean function f:

$$f(s) = s_1 \cdot s_2 \cdot s_6 \cdot s_7 + s_3 \cdot s_4 \cdot s_8 \cdot s_9 + s_5 \cdot s_6 \cdot s_{10} \cdot s_1.$$

Because each monomial in f is dependent on 2 bits in the inner state of s, we cannot generate linear equations with $G \subseteq \{1, 2, 3, 4, 5\}$. In our research presented in chapter 5, this problem actually occurred while attacking the PRIMATEs family of permutations. However, using some clever tricks, we could circumvent this problem. Therefore, it is not advised to rely solely on such a defence against this attack. Applying similar tricks might allow attackers to bypass this defence.

Chapter 5

Cube attack on PRIMATEs

In this chapter, we will describe a cube attack performed on the family of authenticated encryption algorithms called PRIMATEs.

PRIMATEs uses duplexes to provide authenticated encryption. Firstly, we will describe how a duplex object is used to provide authenticated encryption in PRIMATEs. Afterwards, we describe the permutation used in the duplex object. Then we will describe our attacks performed on the permutation and the results of these attacks. Lastly, the impact of this attack on the PRIMATEs algorithm is given. The PRIMATEs family is defined in [1].

5.1 PRIMATEs: an authenticated encryption family

The PRIMATEs family of authenticated encryption defines several algorithms to provide authenticated encryption. The algorithms defined are generated by a *scheme* and a security level $\hat{s}$. The scheme is a mode of operation that defines how the duplex function is used to provide encryption. This scheme can be `APE`, `HANUMAN` or `GIBBON`. The security level defines the size of the permutation, or more specifically the capacity of the permutation (the rate of the permutation stays the same). The security level can be either 80 or 120 bits. The permutations used in the schemes are called PRIMATE. PRIMATE has four different variants called p_1, p_2, p_3 and p_4. The exact definition of those four variants will be given in section 5.1.2. Each variant has a 80 bit version, which is called PRIMATE-80, and a 120 bit version, which is called PRIMATE-120. Thus in total eight versions of the PRIMATE permutation are defined. The inverse of p_i will be denoted with p_i^{-1}. A full authentication scheme is notated as *scheme*-$\hat{s}$ where $\hat{s}$ is the security level, for instance `APE-120`, denotes the `APE` scheme used with a security level of 120 bits.

The size, rate and capacity of the different duplexes used by PRIMATEs is given by table 5.1. A security level of 80 bits does not mean a state size or capacity size of 80 bits, instead it means a capacity size of $2 \cdot 80 = 160$ bits. As stated in chapter 3, this is because sponge constructions are resistant against generic attacks of complexity $2^{\frac{c}{2}}$ (assuming the permutation behaves as a random oracle). Therefore, a capacity of c provides a security level of $\frac{c}{2}$ [22, p. 9].

In our attack, we are only concerned with the security level of 80 bits. However, increasing the capacity of the duplex construction does not provide extra defence against our attack. When the

	$\hat{s} = 80$ bits	$\hat{s} = 120$ bits
family name	PRIMATE-80	PRIMATE-120
b (state size)	200 bits	280 bits
c (capacity size)	160 bits	240 bits
r (rate size)	40 bits	40 bits

Table 5.1 The security levels for the PRIMATE permutations.

capacity size is larger, we just need to find more linear equations: once some linear equations are found, finding more is often not harder, and does not provide a significant (that is non-linear) increase in computational costs.

5.1.1 Schemes

As explained before, the PRIMATEs family provides three different encryption schemes: `APE`, `HANUMAN` and `GIBBON`. Our attack only has a direct impact on the `GIBBON` encryption scheme, as this scheme uses a PRIMATE permutation with fewer rounds, and thus has a lower polynomial degree. For completeness we will briefly discuss all three algorithms. For a more elaborate explanation, please read [1].

Each scheme provides an encryption algorithm $\mathcal{E}_K(N, A, M)$ and a decryption algorithm $\mathcal{D}_K(N, A, C, T)$. Both algorithms are parametrized by the input key K. A different nonce N should be given every time an encryption is performed. The actual value of N does not matter. A is the associated data. This parameter A allows one to use the encryption scheme to also authenticate A, while not having to keep A confidential. In contrast, M is authenticated but also encrypted. For instance, A can be a header that is sent in plaintext together with ciphertext. The decryption will verify that A corresponds to the ciphertext and is not changed. M is the message that is to be encrypted. The encryption algorithm $\mathcal{E}_K(N, A, M)$ outputs a tuple (C, T) where C is the ciphertext of M and T is a tag that proves the authenticity of the ciphertext C and the associated data A. Inserting these parameters into the decryption algorithm will output the message M again. If the tag T cannot authenticate the ciphertext C and the associated data A, the decryption algorithm will output the empty string $\perp$.

APE

`APE` is the most robust scheme defined by the PRIMATEs family. It uses the more secure versions of their permutation algorithm (i.e. it uses more rounds) and is resistant to nonce reuse. This means that reusing the nonce does not provide vulnerabilities directly. The nonce is actually handled the same way that the associated data is handled. The encryption and decryption algorithms can be seen in figure 5.1.

In practice, they use a duplex construction, where V is the duplex state. V_r is the outer state, and V_c is the inner state of the duplex construction.

It is interesting to see that the ciphertext blocks are produced by duplexing the message block $M[i]$ into the duplex state and then taking the resulting outer state (line 18,19 of the encryption algorithm). In contrast, ciphertexts are often produced by applying an exclusive-or with the outer

Algorithm 1: $\mathcal{E}_K(N, A, M)$

Input: $K \in \mathbf{C}$, $N \in \mathbf{C}^{\frac{1}{2}}$, $A \in \{0,1\}^*$, $M \in \{0,1\}^*$
Output: $C \in \{0,1\}^*$, $T \in \mathbf{C}$

1 $V \leftarrow 0^r \parallel K$
2 $N[1]N[2]\cdots N[y] \leftarrow N$
3 **for** $i = 1$ **to** y **do**
4   $V \leftarrow p_1\big(N[i] \oplus V_r \parallel V_c\big)$
5 **end**
6 **if** $A \neq \varnothing$ **then**
7   $A[1]A[2]\cdots A[u] \leftarrow A$
8   $A[u] \leftarrow A[u] \parallel 10^*$
9   **for** $i = 1$ **to** u **do**
10     $V \leftarrow p_1\big(A[i] \oplus V_r \parallel V_c\big)$
11   **end**
12 **end**
13 $V \leftarrow V \oplus (0^{b-1} \parallel 1)$
14 $M[1]M[2]\cdots M[w] \leftarrow M$
15 $l \leftarrow |M[w]|$
16 $M[w] \leftarrow M[w] \parallel 10^*$
17 **for** $i = 1$ **to** w **do**
18   $V \leftarrow p_1\big(M[i] \oplus V_r \parallel V_c\big)$
19   $C[i] \leftarrow V_r$
20 **end**
21 $C \leftarrow C[1]C[2]\cdots C[w-2]$
22 $C \leftarrow C \parallel \lfloor C[w-1] \rfloor_l$
23 $C \leftarrow C \parallel C[w]$
24 $T \leftarrow V_c \oplus K$
25 **return** (C, T)

Algorithm 2: $\mathcal{D}_K(N, A, C, T)$

Input: $K \in \mathbf{C}$, $N \in \mathbf{C}^{\frac{1}{2}}$, $A \in \{0,1\}^*$, $C \in \{0,1\}^*$, $T \in \mathbf{C}$
Output: $M \in \{0,1\}^*$ or $\perp$

1 $IV \leftarrow 0^r \parallel K$
2 $N[1]N[2]\cdots N[y] \leftarrow N$
3 **for** $i = 1$ **to** y **do**
4   $IV \leftarrow p_1\big(N[i] \oplus IV_r \parallel IV_c\big)$
5 **end**
6 **if** $A = \varnothing$ **then**
7   $A[1]A[2]\cdots A[u] \leftarrow A$
8   $A[u] \leftarrow A[u] \parallel 10^*$
9   **for** $i = 1$ **to** u **do**
10     $IV \leftarrow p_1\big(A[i] \oplus IV_r \parallel IV_c\big)$
11   **end**
12 **end**
13 $C[1]C[2]\cdots C[w] \leftarrow C$
14 $l \leftarrow |C[w]|$
15 $C[w] \leftarrow \lceil C[w-1] \rceil_{r-l} \parallel C[w]$
16 $C[w-1] \leftarrow \lfloor C[w-1] \rfloor_l$
17 $C[0] \leftarrow IV_r$
18 $V \leftarrow p_1^{-1}\big(C[w] \parallel K \oplus T\big)$
19 $M[w] \leftarrow \lfloor V_r \rfloor_l \oplus C[w-1]$
20 $V \leftarrow V \oplus M[w]10^* \parallel 0^c$
21 **for** $i = w-1$ **to** 1 **do**
22   $V \leftarrow p_1^{-1}(V)$
23   $M[i] \leftarrow C[i-1] \oplus V_r$
24   $V \leftarrow C[i-1] \parallel V_c$
25 **end**
26 $M \leftarrow M[1]M[2]\cdots M[w]$
27 **if** $IV_c = V_c \oplus (0^{c-1} \parallel 1)$ **then**
28   **return** M
29 **else**
30   **return** $\perp$
31 **end**

Figure 5.1 The APE encryption $\mathcal{E}_K(N, A, M)$ and decryption $\mathcal{D}_K(A, C, T)$ algorithms. The pseudocode is taken directly from [1, p. 5] and therefore the notation is unchanged: $\oplus$ is the exclusive-or operator, $||$ is the concatenation operator and $\mathbf{C} = \mathbb{F}_2^c, \mathbf{C}^{\frac{1}{2}} = \mathbb{F}_2^{\frac{c}{2}}$, $1 = \mathbb{1}$, $0 = \mathbb{0}$.

state of the current duplex state. The consequence of this method is that APE must use the inverse of p_1 to decrypt the message.

HANUMAN

HANUMAN provides authenticated encryption and decryption, while being more lightweight than APE but more robust then GIBBON. HANUMAN uses the same number of rounds in its permutation as APE does, but does not provide resistance against nonce reuse.

As with APE, the encryption algorithm uses a duplex construction to encrypt the data, but instead of using the same permutation every time, the duplex constructions switches between permutation p_1 and p_4. Because the ciphertext is created by applying an exclusive-or with the outer state of the current duplex state, the inverses of p_1 and p_4 are not needed to decrypt the data. The algorithm can be seen in figure 5.2.

Algorithm 3: $\mathcal{E}_K(N, A, M)$

```
   Input: K ∈ C^{1/2}, N ∈ C^{1/2}, A ∈ {0,1}*,
          M ∈ {0,1}*
   Output: C ∈ {0,1}*, T ∈ C^{1/2}
 1 V ← p1(0^r || K || N)
 2 if A ≠ ∅ then
 3   A[1]A[2]···A[u] ← A
 4   A[u] ← A[u] || 10*
 5   for i = 1 to u − 1 do
 6     V ← p4(A[i] ⊕ V_r || V_c)
 7   end
 8   V ← p1(A[u] ⊕ V_r || V_c)
 9 end
10 M[1]M[2]···M[w] ← M
11 ℓ ← |M[w]|
12 M[w] ← M[w] || 10*
13 for i = 1 to w do
14   C[i] ← M[i] ⊕ V_r
15   V ← p1(C[i] || V_c)
16 end
17 C ← C[1]C[2]···C[w − 1]⌊C[w]⌋_ℓ
18 T ← ⌊V_c⌋_{c/2} ⊕ K
19 return (C, T)
```

Algorithm 4: $\mathcal{D}_K(N, A, C, T)$

```
   Input: K ∈ C^{1/2}, N ∈ C^{1/2}, A ∈ {0,1}*,
          C ∈ {0,1}*, T ∈ C^{1/2}
   Output: M ∈ {0,1}* or ⊥
 1 V ← p1(0^r || K || N)
 2 if A ≠ ∅ then
 3   A[1]A[2]···A[u] ← A
 4   A[u] ← A[u] || 10*
 5   for i = 1 to u − 1 do
 6     V ← p4(A[i] ⊕ V_r || V_c)
 7   end
 8   V ← p1(A[u] ⊕ V_r || V_c)
 9 end
10 C[1]C[2]···C[w] ← C
11 ℓ ← |C[w]|
12 for i = 1 to w − 1 do
13   M[i] ← C[i] ⊕ V_r
14   V ← p1(C[i] || V_c)
15 end
16 M[w] ← ⌊V_r⌋_ℓ ⊕ C[w]
17 V ← p1((M[w] || 10* ⊕ V_r) || V_c)
18 M ← M[1]M[2]···M[w − 1]M[w]
19 T' ← ⌊V_c⌋_{c/2} ⊕ K
20 return T = T' ? M : ⊥
```

Figure 5.2 The HANUMAN encryption $\mathcal{E}_K(N, A, M)$ and decryption $\mathcal{D}_K(N, A, C, T)$ algorithms. The pseudocode is taken directly from [1, p. 6] and therefore the notation is unchanged: $\oplus$ is the exclusive-or operator, $||$ is the concatenation operator and $\mathbf{C} = \mathbb{F}_2^c, \mathbf{C}^{\frac{1}{2}} = \mathbb{F}_2^{\frac{c}{2}}$, $1 = \mathbb{1}$, $0 = \mathbb{0}$.

GIBBON

`GIBBON` is the most efficient, but is less secure: it does not provide resistance against nonce reuse and uses fewer permutation rounds. Because `GIBBON` uses fewer permutation rounds, the polynomial degree of its permutation is lower. This allows us to perform a differential attack on this algorithm.

Just like `HANUMAN`, the encryption algorithm uses a duplex construction to encrypt the data. Just like `HANUMAN` it switches between the permutations used in the duplex construction. In this case, p_1, p_2, p_3 are used. Furthermore, the ciphertext is created by applying an exclusive-or with the outer state of the current duplex state and thus we do not need the inverses of these permutations to decrypt the ciphertext. The algorithm can be seen in figure 5.3.

Algorithm 5: $\mathcal{E}_K(N, A, M)$

```
   Input: K ∈ C^(1/2), N ∈ C^(1/2), A ∈ {0,1}*,
          M ∈ {0,1}*
   Output: C ∈ {0,1}*, T ∈ C^(1/2)
 1 V ← p1(0^r || K || N)
 2 V ← V_r || (K || 0^(c/2)) ⊕ V_c
 3 if A ≠ ∅ then
 4 |  V ← p2(V)
 5 |  A[1]A[2]···A[u] ← A
 6 |  A[u] ← A[u] || 10*
 7 |  for i = 1 to u − 1 do
 8 |  |  V ← p2(A[i] ⊕ V_r || V_c)
 9 |  end
10 |  V ← A[u] ⊕ V_r || V_c
11 end
12 M[1]M[2]···M[w] ← M
13 ℓ ← |M[w]|
14 M[w] ← M[w] || 10*
15 V ← p3(V)
16 for i = 1 to w do
17 |  C[i] ← M[i] ⊕ V_r
18 |  V ← p3(C[i] || V_c)
19 end
20 V ← p1(V_r || (K || 0^(c/2)) ⊕ V_c)
21 C ← C[1]C[2]···C[w − 1]⌊C[w]⌋_ℓ
22 T ← ⌊V_c⌋_(c/2) ⊕ K
23 return (C, T)
```

Algorithm 6: $\mathcal{D}_K(N, A, C, T)$

```
   Input: K ∈ C^(1/2), N ∈ C^(1/2), A ∈ {0,1}*,
          C ∈ {0,1}*, T ∈ C^(1/2)
   Output: M ∈ {0,1}* or ⊥
 1 V ← p1(0^r || K || N)
 2 V ← V_r || (K || 0^(c/2)) ⊕ V_c
 3 if A ≠ ∅ then
 4 |  V ← p2(V)
 5 |  A[1]A[2]···A[u] ← A
 6 |  A[u] ← A[u] || 10*
 7 |  for i = 1 to u − 1 do
 8 |  |  V ← p2(A[i] ⊕ V_r || V_c)
 9 |  end
10 |  V ← A[u] ⊕ V_r || V_c
11 end
12 C[1]C[2]···C[w] ← C
13 ℓ ← |C[w]|
14 V ← p3(V)
15 for i = 1 to w − 1 do
16 |  M[i] ← C[i] ⊕ V_r
17 |  V ← p3(C[i] || V_c)
18 end
19 M[w] ← ⌊V_r⌋_ℓ ⊕ C[w]
20 V ← p3((M[w] || 10* ⊕ V_r) || V_c)
21 M ← M[1]M[2]···M[w − 1]M[w]
22 V ← p1(V_r || (K || 0^(c/2)) ⊕ V_c)
23 T' ← ⌊V_c⌋_(c/2) ⊕ K
24 return T = T' ? M : ⊥
```

Figure 5.3 The `GIBBON` encryption $\mathcal{E}_K(N, A, M)$ and decryption $\mathcal{D}_K(N, A, C, T)$ algorithm. The pseudocode is taken directly from [1, p. 6] and therefore the notation is unchanged: $\oplus$ is the exclusive-or operator, $||$ is the concatenation operator and $\mathbf{C} = \mathbb{F}_2^c, \mathbf{C}^{\frac{1}{2}} = \mathbb{F}_2^{\frac{c}{2}}$, $1 = \mathbb{1}$, $0 = \mathbb{0}$.

5.1.2 The permutation

The permutation PRIMATE uses in the PRIMATEs authenticated encryption schemes is inspired by Fides [10] and its structure resembles the RIJNDAEL block cipher [15][1, p. 7]. The permutation is defined for two different sizes, a 200-bit version (PRIMATE-80) and a 280-bit version (PRIMATE-120). The permutation itself is an S-box-based permutation. The input and output states are divided into a grid of 5-bit elements, corresponding to the input and output size of the S-box used. In PRIMATE-80, the state is divided into 5×8 elements of 5 bits and in PRIMATE-120, the state is divided into 7×8 elements of 5 bits. The PRIMATE permutation updates the internal state in rounds, where each round is a sequence of permutations:

$$\texttt{PRIMATE-round} = \texttt{CA} \circ \texttt{MC} \circ \texttt{SR} \circ \texttt{SE}.$$

These rounds are then applied multiple times, which gives the permutations p_1, p_2, p_3 and p_4. The operations applied to the state in one of the `PRIMATE-rounds` are:

- `SE`, the SubElements operation. This is the only non-linear operation in the PRIMATE permutation. This operation applies a 5-bit S-box to each element of the state. The polynomial degree of this operation is 2.
- `SR`, the ShiftRows step. This step is a linear operation, and circularly shifts the rows of the state. A circular shift is an operation that rearranges the entries in tuple. Each row is circularly shifted by a different number (otherwise, all elements would stay in the same position relative to each other). The shift offset is determined by the sequence $h = (h_1, h_2, \ldots)$: row i is shifted left by h_i positions. In PRIMATE-80 we have $h = (0, 1, 2, 4, 7)$ and for PRIMATE-120 we have $h = (0, 1, 2, 3, 4, 5, 7)$.
- `MC`, the MixColumns operation. This step is a linear operation. The MixColumns operation multiplies each column with a 5×5 matrix in PRIMATE-80 or a 7×7 matrix in PRIMATE-120. This multiplication is done in the field $\mathbb{F}_2[x]/\langle x^5 + x^2 + 1\rangle$. The exact matrix is not important for our analysis and is left out.
- `CA`, the ConstantAddition step. This step is again a linear operation. It combines the second element of the second row with a constant rc using an exclusive-or operation. The purpose of `CA` is to make each round different and to break the symmetry of the other operations. The constant changes every round and is generated using a *Linear Feedback Shift Register* (LFSR). An LFSR is a register with a feedback loop that generates a sequence of numbers by performing a right shift. The new least significant bit is set to a combination of the previous bits, combined by an exclusive or. In our case, the LFSR is defined as follows:

  $$\text{LFSR}(rc) = (rc_5 + rc_1, rc_1, rc_2, rc_3, rc_4).$$

  The ConstantAddition step can be used to generate several different permutations, by setting the initial value of the LFSR to different values. This is used, for instance, to define p_1, p_2, p_3 and p_4.

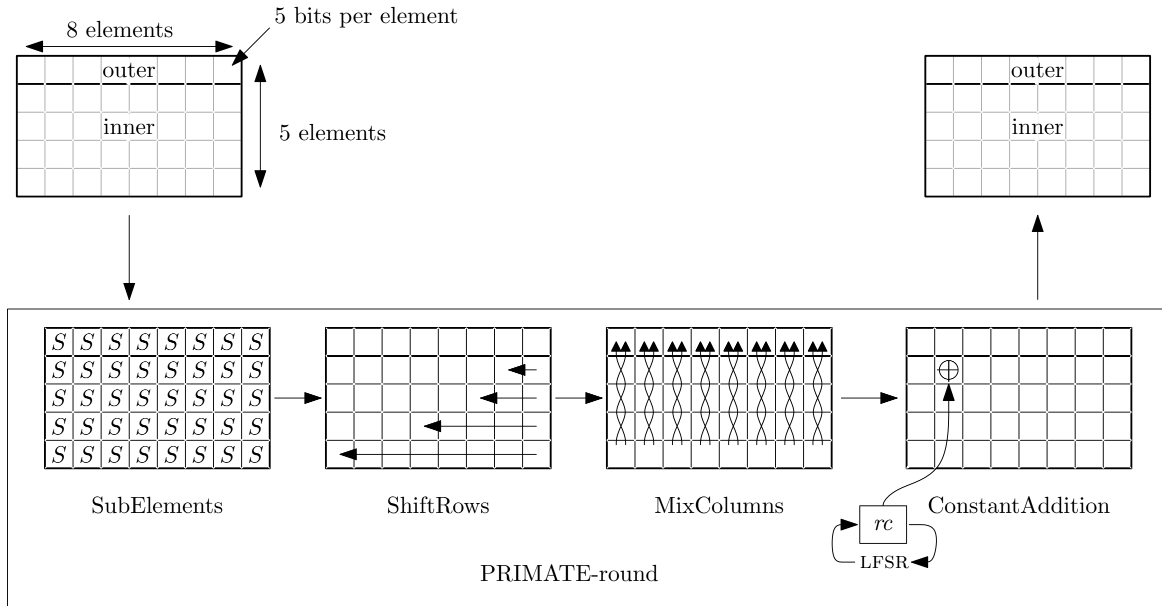

Figure 5.4 An diagram illustrating one of the `PRIMATE-rounds` in PRIMATE-80. The constant *rc* is updated and reused every round.

A visualization of the `PRIMATE-round` is given in figure 5.4.

As we have seen, there is only one non-linear operation in a `PRIMATE-round`, and its degree is 2. Therefore, the degree of one round of PRIMATE is 2. In general i rounds of PRIMATE have a polynomial degree of 2^i, but the degree is limited by the size of the permutation state. By varying the number of rounds and the initial value of rc, the designers of PRIMATE defined several permutations. These are given in table 5.2.

5.2 Cube attack on PRIMATEs

To attack the PRIMATEs scheme for authenticated encryption with associated data, we will use the cube attack defined in section 4.3. Note that we need to vary $i-1$ bits to reduce an i bit polynomial to a linear equation. Unfortunately, all permutations defined have a higher polynomial degree than the number of bits in the outer state. But with a clever trick, we are still able to attack p_2 and p_3. In this section, we will only consider the permutations with security level 80, this means the permutation has an input and output size of 200 bits. From now on, we will refer to one of the

Permutation-name	p_1	p_2	p_3	p_4
Number of rounds	12	6	6	12
Initial value of rc	1	24	30	24
Polynomial degree	200/280	64	64	200/280

Table 5.2 The permutations p_1, p_2, p_3 and p_4. The polynomial degree of p_1 and p_4 is 200 for the 200-bit variant and 280 for the 280-bit variant as the maximum polynomial degree is limited by the number of input bits.

PRIMATE permutations as f. Until section 5.2.3, f is a permutation by applying `PRIMATE-round` multiple times with $rc = 0$. The number of rounds f has should be clear from the context. In section section 5.2.3 we apply the attack for each rc value as described in table 5.2. We abstract from this rc value initially (by taking $rc = 0$), as it does not make a difference for our type of attack. Only the 200-bit variant is attacked.

5.2.1 Basic cube attack

The initial approach is to basically apply the attack defined in section 4.3.2. As a start, we tried to attack the first one and two rounds of f.

However, after analysing the polynomial of the one and two-round permutation, we see that no linear function can be extracted. For instance, the polynomial representation of the first bit of one round of PRIMATEs is of the form:

$$
\begin{aligned}
f_1(\alpha_1, \dots \alpha_{40}, \beta_1, \dots, \beta_{160}) = {} & \alpha_1\alpha_3 + \alpha_2\alpha_5 + \alpha_1 + \alpha_4 + \\
& \beta_6\beta_7 + \beta_6\beta_{10} + \beta_8\beta_9 + \beta_8\beta_{10} + \beta_{10} + \\
& \beta_{51}\beta_{54} + \beta_{52}\beta_{53} + \beta_{52}\beta_{55} + \beta_{53}\beta_{55} + \beta_{52} + \beta_{53} + \beta_{54} + \\
& \beta_{101}\beta_{104} + \beta_{102}\beta_{103} + \beta_{102}\beta_{105} + \beta_{103}\beta_{105} + \beta_{102} + \beta_{103} + \beta_{104} + \\
& \beta_{156}\beta_{157} + \beta_{156}\beta_{160} + \beta_{158}\beta_{159} + \beta_{158}\beta_{160} + \beta_{160} + \mathbb{1}.
\end{aligned}
$$

In this equation, the α variables are the outer part of the state, β variables are the inner part of the state. If we would differentiate towards an α-term, no β variables are left in the linear polynomial. This means that deriving an equation from this would not give any information about the inner part, which is the part where we need information about.

In the two-round permutation, something similar happens. The two-round permutation is of degree 4. So to derive a linear polynomial from this function, we need to differentiate over a vector space in a of dimension 3. This linear polynomial needs to be dependent on β-variables to allow us to derive information about the inner part of the state. This means that terms of the form $\alpha_i \cdot \alpha_j \cdot \alpha_k \cdot \beta_l$ should be contained in the two-round permutation. However, this is not the case.

These two approaches fail because the S-box is the only non-linear operation in the permutation. The S-box is only applied to groups of 5 bits at a time. This means that the S-box does not create mix terms of a high degree between α and β, because the SubElements step is the first step in the `PRIMATE-round`. When more rounds are applied, the highest degree terms will always have either at least two β-variables, or none.

It can, thus, be seen that increasing the number of rounds, will not create better attack opportunities.

5.2.2 Generating linear equations out of quadratic equations

The second attempt is focussed on the two-round permutation. The idea is to create mix terms between α and β of a high enough degree having only one β-variable in the mix term, by combining

the output of several bits. If we view the polynomials obtained by differentiating the two-round permutation, some interesting properties come to light.

The polynomials $f_1, \dots, f_5$ are quite similar: the same variables are used. $f_1, \dots, f_5$ seem to depend only on $\alpha_1, \dots \alpha_5$, $\beta_6 \dots \beta_{10}$, $\beta_{51}, \dots \beta_{55}$, $\beta_{101} \dots, \beta_{105}$, $\beta_{156}, \dots, \beta_{160}$. Furthermore, terms of the form $\alpha_i \cdot \alpha_j \cdot \beta_k$ (not be confused with terms of the form $\alpha_i \cdot \alpha_j \cdot \alpha_k \cdot \beta_l$) do exist in the polynomials $f_1, \dots, f_5$. Note the degree of $f_1, \dots, f_5$ is 4. As such, if we differentiate $f_1, \dots, f_5$ over a vector space of dimension 3 to make $f_1, \dots, f_5$ linear, the terms of the form $\alpha_i \cdot \alpha_j \cdot \beta_k$ are reduced to a constant.

The idea is to differentiate the polynomials $f_1, \dots, f_5$ multiple times over different vector spaces V of dimension 2, with $V \subseteq \langle e_1, \dots e_5 \rangle$. This means that we differentiate f two times towards $\alpha_1, \dots \alpha_5$ and reduce the polynomial to a quadratic polynomial. This gives us a set of polynomials $g_1, \dots, g_m$. Because the quadratic polynomials are quite similar, we try to find combinations of $\sum g_i$ such that $\sum g_i$ is at most linear. This is done using linear algebra.

If we put the quadratic terms of each function into a matrix where each mixed term is a specific column, the kernel space of this matrix gives the combination of functions without quadratic terms.

However after implementing this, it seems that every function found this way is the zero function: this means that there is a dependency between the linear terms in g and the quadratic terms in g, which means that whenever we cancel out quadratic terms, the linear terms also cancel out.

5.2.3 Skipping the first SubElements operation

The third attempt is based on the observation that by applying SubElements first, mix terms are created with only outer state variables or only inner state variables. If we would be able to skip this step, mix terms between the inner state and outer state variables should be created. Another advantage would be that the polynomial degree would be halved, making the degree of p_2 and p_3 32, which would allow us to attack it by varying bits in the outer state.

Fortunately, it is possible to skip the first SubElements step. The `PRIMATE-round` was defined as:

$$\texttt{PRIMATE-round} = \texttt{CA} \circ \texttt{MC} \circ \texttt{SR} \circ \texttt{SE}.$$

Write the input state s as $(\alpha_1, \dots, \alpha_{40}, \beta_1, \dots, \beta_{160})$, write the S-box used as $\texttt{S}$ and its inverse $\texttt{S}^{-1}$ and we can derive:

$$\begin{aligned}\texttt{PRIMATE-round}(s) &= \texttt{CA}(\texttt{MC}(\texttt{SR}(\texttt{SE}(s)))) \\ &= \texttt{CA}(\texttt{MC}(\texttt{SR}(\texttt{SE}(\alpha_1, \dots, \alpha_{40}, \beta_1, \dots, \beta_{160})))) \\ &= \texttt{CA}(\texttt{MC}(\texttt{SR}(\texttt{S}(\alpha_1, \dots \alpha_5) \mid \dots \mid \texttt{S}(\alpha_{36}, \dots, \alpha_{40}) \mid \\ &\qquad \texttt{S}(\beta_1, \dots, \beta_5) \mid \dots \mid \texttt{S}(\beta_{156}, \dots, \beta_{160})))).\end{aligned}$$

Now instead of inputting $\alpha_1, \ldots, \alpha_{40}$ we first apply $\mathtt{S}^{-1}$ to each 5 bit element:

$$\begin{aligned}
\mathtt{PRIMATE\text{-}round}(s) &= \mathtt{CA}(\mathtt{MC}(\mathtt{SR}(\mathtt{S}(\mathtt{S}^{-1}(\alpha_1, \ldots \alpha_5)) \mid \ldots \mid \mathtt{S}(\mathtt{S}^{-1}(\alpha_{36}, \ldots, \alpha_{40})) \mid \\
&\qquad \mathtt{S}(\beta_1, \ldots, \beta_5) \mid \ldots \mid \mathtt{S}(\beta_{156}, \ldots, \beta_{160})))) \\
&= \mathtt{CA}(\mathtt{MC}(\mathtt{SR}((\alpha_1, \ldots \alpha_5) \mid \ldots \mid (\alpha_{36}, \ldots, \alpha_{40}) \mid \\
&\qquad \mathtt{S}(\beta_1, \ldots, \beta_5) \mid \ldots \mid \mathtt{S}(\beta_{156}, \ldots, \beta_{160})))) \\
&= \mathtt{CA}(\mathtt{MC}(\mathtt{SR}((\alpha_1, \ldots, \alpha_{40}) \mid \mathtt{S}(\beta_1, \ldots, \beta_5) \mid \ldots \mid \mathtt{S}(\beta_{156}, \ldots, \beta_{160})))).
\end{aligned}$$

Now set $\mathtt{S}(\beta_1, \ldots, \beta_5) \mid \ldots \mid \mathtt{S}(\beta_{156}, \ldots, \beta_{160})$ as our unknown variables. If we apply our attack this way, we will find $\mathtt{S}(\beta_1, \ldots, \beta_5) \mid \ldots \mid \mathtt{S}(\beta_{156}, \ldots, \beta_{160})$ and we just need to apply $\mathtt{S}^{-1}$ to find the real values of $\beta_1, \ldots, \beta_{160}$.

This method is able to find linear equations and the results are discussed in the following section.

5.3 Results

The third attack was implemented by first attacking two rounds, then three rounds and so on. Because the SubElements step was skipped, the polynomial representation of a single round instantly gives 40 linear equations. Unfortunately, we cannot generate more than 40 equations as we cannot apply δ_G with different values for G. Please note, that G is the set of indices of the bits that we apply the differentiation function δ on. Because there are more than 40 unknowns, we cannot use these 40 equations to find the inner state.

As described in section 4.3.2, we must choose sets $G_1, \ldots, G_n$ (for some n) for each number of rounds greater than 1. These sets are used to calculate $\delta_{G_i}(f)$, which should result in linear equations. The sets $G_1, \ldots, G_n$ should be chosen in such a way, that the linear equations produced by $\delta_{G_i}(f)$ produce a linear system of rank 160, and thus we are able to derive the inner state using these equations. Furthermore, to reduce the complexity of the attack, the number of sets n should be kept as low as possible. In our attack, we have experimented with different values $G_1, \ldots, G_n$, and the smallest number of sets giving a fully determined system are written down here. We aim to find $G_1, \ldots, G_n$ with n as small as possible, as this reduces the computation time. The exact results, and the $G_1, \ldots, G_n$ sets chosen for each round are documented in section 5.3.2, including the exact specifications on which the tests are done.

In table 5.3 some statistics of the attack applied on each round are given. We see that most rounds can be attacked with 8 $\delta_{G_i}(f)$ computations. It can be seen that the computation time of the offline phase and the computation time of the online and solving phase increases exponentially. Furthermore, when a small number of rounds is attacked, most of the computation time of the solving phase is caused by starting the MATLAB engine. Please note, as said before, that we omit the first SubElements step in f, this halves the degree of f.

For six rounds of the PRIMATE-permutation, the polynomial degree of f_i is 32. Therefore, our sets $G_0, \ldots, G_n$ should be of size 31. When calculating the linear equations for five rounds of the

Rounds	$\deg(f)$	$\|\mathbf{G}\|$	$t_{offline}$	$t_{online} + t_{solv}$
2	2	8	< 1 s	19 s
3	4	16	4 s	19 s
4	8	8	20 s	20 s
5	16	8	44 m	29 s

Table 5.3 Various statistics about the implemented cube attack on PRIMATE. The attack was performed for different rounds, which are shown in this table. $|\mathbf{G}|$ indicates the number of sets in $\mathbf{G} = \{G_1, \ldots, G_n\}$ we require to make the linear system generated by $\delta_{G_i}(f)$ fully determined. $t_{offline}$ is the time required to compute the offline phase, $t_{online} + t_{solv}$ is the time required to perform the online phase and the solving phase.

PRIMATE-permutation already takes a considerable amount of time. Because the calculation time will probably grow exponentially with the size of G, it is reasonable to first determine the expected calculation time of the representation of $\delta_G(f)$ with $|G| = 31$ before attempting to calculate it. This gives us the calculation time of one $\delta_G(f)$ with $|G| = 31$. It is likely that we at least need to calculate it 8 times to generate a fully determined system. Furthermore, finding somewhat optimal $G_1, \ldots, G_n$ that generate a fully determined system also requires some trial-and-error: not all sets G with $|G| = 31$ will result in useful linear equations.

To determine the calculation time of $\delta_G(f)$ with $|G| = 31$, we have calculated $\delta_G(f)$ with $1 \leq |G| \leq 21$. As expected for $|G|$ large enough, the calculation time will approximately double if $|G|$ increases by 1. We have extrapolated the results found and determined the approximate time to calculate $\delta_G(f)$ with $|G| = 31$: this will approximately take 234 days on our computer. This can be seen in figure 5.5. Please note, that we did not use the speed up of equation (4.4) on page 28, as we only discovered this afterwards. If we would implement this speed up, the calculation time would be reduced to approximately 116 days. While 116 days per $\delta_G(f)$ calculation is a long time, it is certainly not infeasible. Furthermore, the attack is highly parallelizable: a small to medium company could rent 116 servers of equal computing power to the used laptop to perform such a calculation in one day. Another approach would be to implement the attack using a graphics card: as the attack is highly parallelizable, a good speed-up might be achieved. As the attack only needs to compute the sum of various permuted input blocks, which can be calculated on demand, the amount of memory and the throughput required is fairly limited. This makes an implementation on a graphics card feasible.

For more than six rounds of the PRIMATE-permutation, it is impossible to apply this attack: we cannot do a cube attack using more than 40 bits, as the outer state of the duplexes used in PRIMATEs is 40 bits.

5.3.1 Impact on the PRIMATE-family

As we have noted before, only `GIBBON` uses the PRIMATE permutation with 6 rounds. Therefore, we could try to attack the `GIBBON` scheme by finding the inner state of the permutation. Such an attack would require that the same inner state can be reproduced multiple times. Figure 5.3 shows that this requires all inputs to the duplex to be the same until line 17 of the encryption algorithm. The only way to achieve this, would be to reuse the nonce multiple times.

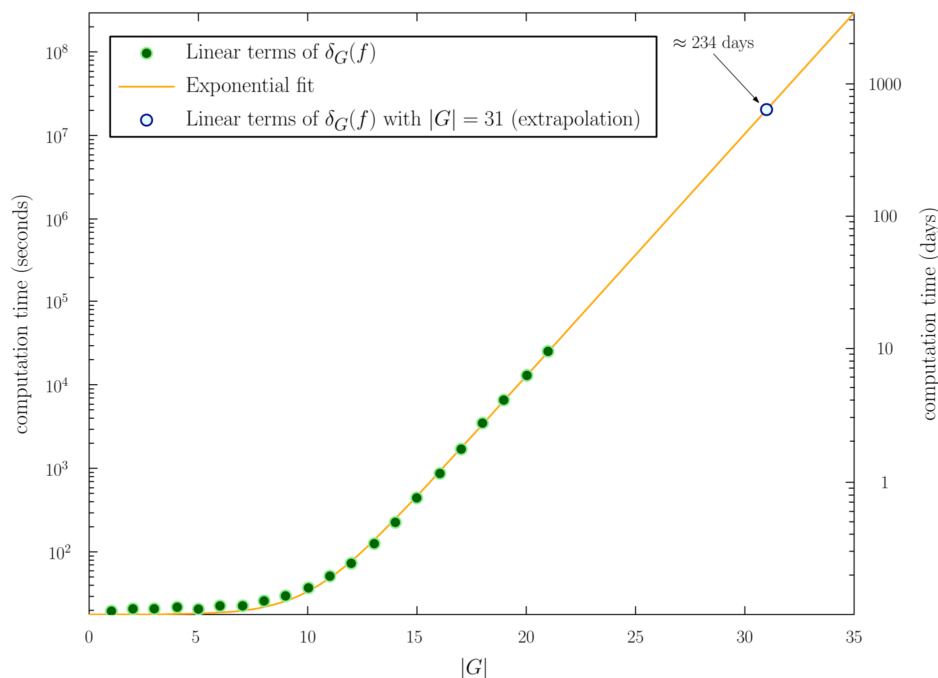

Figure 5.5 The approximate calculation time of $\delta_G(f)$ with $|G| = 31$ was calculated by extrapolating the calculation time of $\delta_G(f)$ with $1 \leq |G| \leq 21$.

So if the nonce would be reused multiple times we can find the inner state at line 18. Because we use a chosen plaintext attack, we know the associated data A and the input message M. Using this, we can calculate the inner state backwards until line 2. We cannot calculate the inner state at line 1, as we cannot reverse the operation at line 2 because we do not know K. Therefore, we cannot find K but we can deduce the inner state at line 2. This inner state is always the same if the same key K and nonce N are used. Therefore, this attack reveals a vulnerability in GIBBON if one would reuse the nonces.

It must be noted that in the specification of GIBBON [1, p. 4], it was stated that GIBBON was not resistant against nonce reuse. However, this vulnerability can be avoided easily at the cost of adding a single round to p_2 and p_3.

5.3.2 Attack details

This section describes the details of the PRIMATE attack. We attacked two rounds of the permutation, and then continued with three, four and so on. Because the SubElements step was skipped, the polynomial representation of a single round instantly gives 40 linear equations. Unfortunately, we cannot generate more than 40 equations as we cannot apply δ_G with different values for G. Please note, that G is the set of indices of the bits that we apply the differentiation function δ on. Because there are more than 40 unknowns, we cannot use these 40 equations to find the inner state.

As described in section 4.3.2, we must choose the sets $G_1, \ldots, G_n$ for each round. These sets are used to calculate $\delta_{G_i}(f)$, which should result in linear equations. The sets $G_1, \ldots, G_n$ should be chosen in such a way, that the linear equations produced by $\delta_{G_i}(f)$ produce a linear system of rank

160, and thus we are able to derive the inner state using these equations. Furthermore, to reduce the complexity of the attack, the number of sets n should be kept as low as possible. In our attack, we have experimented with different values $G_1, \ldots, G_n$, and the smallest number of sets giving a fully determined system are written down here.

The computations documented here have been implemented in C++. Solving the linear system in the last phase is performed using MATLAB. The computations have been carried out on a laptop with the following specs:

- **Type:** Asus K501LX-DM104T.
- **Memory:** 8GB of RAM.
- **CPU:** Intel Core i7 5500U.
- **Operating system:** Linux.

The implementation is according to the method described in section 4.3.2 and uses only a single thread. The polynomial representation of the linear equations has been calculated using equation 4.3 on page 28. Please note, that we did not apply the speed-up described in equation 4.4. Therefore, in theory the computation time of the offline phase can be halved. The speed-up was not applied, as it would not make a difference between what is computable and what is not. Furthermore, this speed-up was discovered after the attack was implemented. We kept track of the computation by recording the real time and the system time, the computation time is the sum of those two. Please note, as said before, that we skip the first SubElements step: by skipping this step, we half the polynomial degree of f_i for any round that we attack.

For two rounds of the PRIMATE-permutation, the polynomial degree of f_i is 2. Therefore, our sets $G_1, \ldots, G_n$ should be of size 1. After experimentation, we found that a fully determined system is generated by the sets $G_1, \ldots, G_n$ only if for each 5-bit element of the outer state a set G_i contains the index of one bit that is inside that 5-bit element. In other words, this means that for each integer set $K = \{5 \cdot j + 1, \ldots, 5 \cdot j + 5\}$ with $0 \leq j < 8$, there must be a set G_i such that $G_i = \{g_1\}$ with $g_1 \in K$. In our case, we have chosen the sets: $\{1\}$, $\{5\}$, $\{6\}$, $\{10\}$, $\{11\}$, $\{15\}$, $\{16\}$, $\{20\}$, $\{21\}$, $\{25\}$, $\{26\}$, $\{30\}$, $\{31\}$, $\{35\}$, $\{36\}$, $\{40\}$. So we needed to perform $\delta_{G_i}(f)$ for 8 different sets in total to generate a fully determined system.

For three rounds of the PRIMATE-permutation, the polynomial degree of f_i is 4. Therefore, our sets $G_1, \ldots, G_n$ should be of size 3. After experimentation, several manners to generate a fully determined system have been found. These different manners all use the same number of sets. The first manner is to systematically choose sets G_i with all indices in the G_i pointing to the same 5-bit element in the state. After a linear system of rank 20 has been generated with all indices in G_is all pointing to bits of the same 5-bit element, this method can be extended to generate a fully determined linear system. In this manner we have found that the sets $\{1,2,3\}$, $\{2,4,5\}$, $\{3,4,5\}$, $\{1,2,4\}$, $\{6,7,8\}$, $\{7,9,10\}$, $\{8,9,10\}$, $\{6,7,9\}$, $\{11,12,13\}$, $\{12,14,15\}$, $\{13,14,15\}$, $\{11,12,14\}$, $\{16,17,18\}$, $\{17,19,20\}$, $\{18,19,20\}$, $\{16,17,19\}$, $\{21,22,23\}$, $\{22,24,25\}$, $\{23,24,25\}$, $\{21,22,24\}$, $\{26,27,28\}$, $\{27,29,30\}$, $\{28,29,30\}$, $\{26,27,29\}$, $\{31,32,33\}$, $\{32,34,35\}$, $\{33,34,35\}$, $\{31,32,34\}$, $\{36,37,38\}$, $\{37,39,40\}$, $\{38,39,40\}$, $\{36,37,39\}$ are probably optimal. However, each set G_i could be chosen such that the indices in G_i point to bits of three different 5-bit elements. This is, however, more

tricky to choose correctly, as there are more combinations possible. Just as with the first approach, this approach requires us to perform $\delta_{G_i}(f)$ for 16 sets in total to generate a fully determined system.

For four rounds of the PRIMATE-permutation, the polynomial degree of f_i is 8. Therefore, our sets $G_1, \ldots, G_n$ should be of size 7. After experimentation, we found that a fully determined system is generated by the sets $\{1, \ldots, 7\}$, $\{6, \ldots, 12\}$, $\{11, \ldots, 17\}$, $\{16, \ldots, 22\}$, $\{21, \ldots, 27\}$, $\{26, \ldots, 32\}$, $\{31, \ldots, 37\}$, $\{36, \ldots, 40, 1, 2\}$. So we needed to perform $\delta_{G_i}(f)$ for 8 sets in total to generate a fully determined system.

For five rounds of the PRIMATE-permutation, the polynomial degree of f_i is 16. Therefore, our sets $G_1, \ldots, G_n$ should be of size 15. After experimentation, we found that a fully determined system is generated by the sets $\{1, \ldots, 9, 11, \ldots, 14, 16, 17\}$, $\{6, \ldots, 14, 16, \ldots, 19, 21, 22\}$, $\{11, \ldots, 19, 21, \ldots, 24, 26, 27\}$, $\{16, \ldots, 24, 26, \ldots, 29, 31, 32\}$, $\{21, \ldots, 29, 31, \ldots, 34, 36, 37\}$, $\{26, \ldots, 34, 36, \ldots, 39, 1, 2\}$, $\{31, \ldots, 39, 1, \ldots, 4, 6, 7\}$, $\{36, \ldots, 40, 1, \ldots, 4, 6, \ldots, 9, 11, 12\}$. So we need to perform $\delta_{G_i}(f)$ for 8 sets in total to generate a fully determined system. The computation time of the offline phase for these three rounds takes around 44 minutes. Performing the online attack and solving this system takes around 29 seconds.

For six rounds of the PRIMATE-permutation, the polynomial degree of f_i is 32. We did not find a set of G_i's such that a fully determined system is formed by our attack. Instead we calculated, how long it would take to calculate the representation of one $\delta_G(f)$ with $|G| = 31$. This turned out to take 234 days on our computer, and with some extra speed-ups, 116 seemed to be achievable.

Chapter 6

Farfalle: a parallelizable PRF construction

We have thus far seen the basic concepts of differential analysis, and how almost any duplex construction can be attacked. In this section, we will take a look at a novel cryptographic construction called *farfalle* (Italian for 'butterfly') introduced by the Keyak team in 'Farfalle: Parallel permutation-based cryptography' [6]. This construction is an attempt to find an alternative to the inherently sequential sponge construction. The alternative must be highly parallelizable, but still offer the same benefits a sponge construction has for keyed applications.

We will briefly introduce this construction in this chapter. After this construction has been introduced, we will discuss a vulnerability that is related to a cube attack. After this vulnerability is introduced, we will discuss a possible countermeasure to prevent such an attack exploiting this vulnerability.

6.1 Sketch of the scheme

As we have seen in section 3.2 we can use either a duplex construction or a sponge construction to encrypt and authenticate a message. When we get a message of ℓ blocks $M_1, \ldots, M_\ell$, we first initialize an empty duplex construction and provide the key K as input to the duplex construction, giving state s_2. Then each block will be given as input to the duplex construction, giving states $s_3, \ldots, s_\ell$. When each state s_i is found, it is used to generate the ciphertext C_{i-1} of each block M_{i-1}. So to calculate C_i, we calculate $C_i = M_i \oplus \text{outer}(s_{i-1})$. Because each state is dependent on the previous state, we must calculate each block sequentially.

Making a duplex construction parallelizable is hard: the state must absorb all input blocks $M_1, \ldots, M_\ell$ into the same state to provide authentication of all blocks.

A sponge construction can also be used to provide encryption and authentication: instead of using the outer state to encrypt a block and then immediately absorb the block into the state, we first absorb the key and generate an encryption stream. Then we absorb the key and all blocks into a zero state and generate a MAC. Note that we do not simply input the message M and use that

to encrypt M. In such a construction, the receiving side would need to know M to decrypt the encryption of M. Instead, we should then provide only the key K as input.

In farfalle, introduced by the Keyak team, they attempt to modify the sponge construction in such a way, that it becomes parallelizable. Just as with a normal sponge function, first all input blocks $M_1, \ldots, M_\ell$ are absorbed to give an intermediate state. This intermediate state output is squeezed to generate the outputs $Z_1, \ldots, Z_m$ for a chosen m. This output can then be used to generate a MAC or to encrypt a message. However, instead of absorbing all blocks one at a time, the absorbing process is designed to be done in parallel: that is, each block is absorbed together with key K into a state, and the intermediate state is computed by combining all these separate states. Furthermore, the squeezing phase is also designed to be done in parallel: each block Z_i is directly computed from the intermediate state, so that we do not need the state used to generate Z_{i-1} to generate block Z_i. Below, we will sketch the idea of this construction. We have omitted the lengths of each input and output block for simplicity.

Let $M = M_1 \mid \ldots \mid M_\ell \in \mathbb{F}_2^{\ell \cdot b}$ be the input message of appropriate length (i.e. the message has been padded to fit the required block size b and each block $M_i \in \mathbb{F}_2^b$). Let K be a symmetric key, m be the number of output blocks we want to generate. Finally, let f be the permutation function, just as in the sponge construction. Then:

- Let $i_1, \ldots, i_\ell$ be a sequence of bit strings generated by some kind of counter, where each i_j is unique. Then calculate the states $s_1, \ldots, s_\ell \in \mathbb{F}_2^b$ for each i as

  $$s_i = f((M_i \mid i_i) + K).$$

  As each s_i is only dependent on three input variables that are known at the start of the computation, s_i can be calculated in parallel.

- Combine the states $s_1, \ldots, s_\ell$ into one intermediate state I:

  $$I = \sum_{i=1}^{\ell} s_i.$$

  Note that summation over bit strings is equal to performing exclusive-or operations. Computing the exclusive-or operation in parallel is fairly straightforward and can be done in the same manner the parallel prefix sum is calculated [29].

- Next we squeeze the output blocks $Z_1, \ldots, Z_m$ from the intermediate state I. The output blocks $Z_1, \ldots, Z_m$ are specified as follows:

  $$Z_i = f(I + ((\mathbb{0}, \ldots, \mathbb{0}) \mid i_i)) + K.$$

  As each Z_i is only dependent on I, this can be computed in parallel as soon as I is known. Z is finally defined as:

  $$Z = Z_1 | \ldots | Z_m.$$

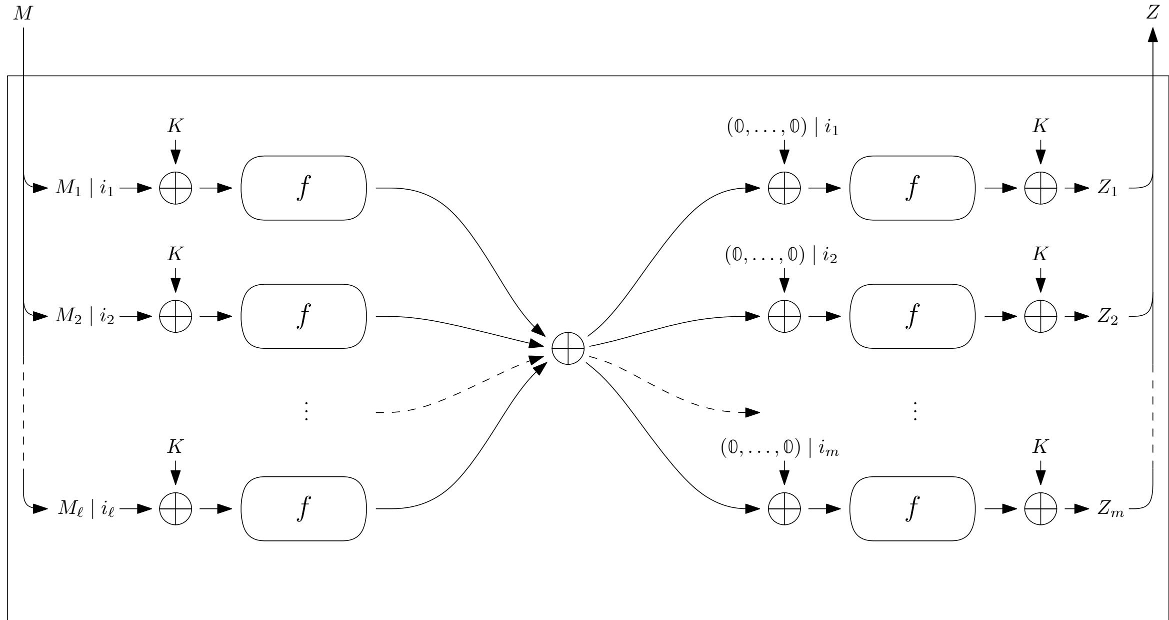

Figure 6.1 An illustration of the farfalle construction for PRFs. It can be seen that the shape of this diagram resembles a farfalle pasta shape.

As with a sponge construction, if the construction is used to encrypt or decrypt the message P, then M cannot be the plaintext message P. If the construction would be used in such a manner, then the receiving end should know P to decrypt the ciphertext of P. In other words, this construction can only be run in the forward direction, i.e. as a stream cipher. When used for encryption the input can for instance be a key and a nonce. The bitstream can then be combined using an exclusive-or operation with the plaintext message. An illustration of the farfalle construction is given in figure 6.1.

One interesting detail of the algorithms are the counters $i_1, \dots, i_\ell$. In the absorbing phase, these are included to let the output reflect the ordering of the blocks $M_1, \dots, M_\ell$. That is, if the counters $i_1, \dots, i_\ell$ are not included then the message $M = M_1 \mid \dots \mid M_{i-1} \mid M_i \mid \dots \mid M_\ell$ and the message $M' = M_1 \mid \dots \mid M_i \mid M_{i-1} \mid \dots \mid M_\ell$ would generate the same output string Z. These counters prevent such a vulnerability. Furthermore, these counters $i_1, \dots, i_\ell$ should also fulfil other properties to ensure a secure absorbing phase, but this will be shown later on. In the squeezing phase, the counters $i_1, \dots, i_\ell$ are used to generate different blocks from the same intermediate state.

6.2 Vulnerability: creating an all-zero intermediate state

Unfortunately, when f has a low polynomial degree a vulnerability exists with this construction in the current form. To explain this vulnerability, we take a look at the Δ function introduced in section 4.2.1. The definition of Δ stated that the derivative of a function $f : \mathbb{F}_2^b \to \mathbb{F}_2$ with respect to a linear subspace V of $\mathbb{F}_2^b$ is defined as

$$\Delta_V f(s) = \sum_{v \in V} f(s + v).$$

Furthermore, theorem 1 stated, that for a linear subspace V of $\mathbb{F}_2^b$ we have:

$$\deg \Delta_V f \leq \deg(f) - \dim(V).$$

Let c be the bitlength of i_j. Suppose that u is a sequence $u_1, \ldots, u_\ell$ in $\mathbb{F}_2^b$ with $\ell = 2^{\deg(f)}$. Furthermore, suppose there exists a linear subspace V of $\mathbb{F}_2^{b+c}$ generated as follows:

$$V = \{(u_h \mid i_h) : 1 \leq h \leq \ell\}.$$

By theorem 1, we have that $\deg(\Delta_V f) = 0$ and thus $\Delta_V f = d$ for some constant bit string d. Now, we attempt to absorb the sequence u using the proposed scheme, thus $M = u_1 \mid \ldots \mid u_\ell$. Then the intermediate state I becomes:

$$\begin{aligned} I &= \sum_{j=1}^{\ell} f((u_h \mid i_h) + K) \\ &= \Delta_V f(K) \\ &= d. \end{aligned}$$

Thus, the intermediate state becomes independent of K. Moreover, the intermediate state and also the output blocks become independent from the input message M. Even more problematic is that for $1 \leq h \leq \ell$:

$$\begin{aligned} Z_h &= K + f(I + ((\mathbb{0}, \ldots, \mathbb{0}) \mid i_h)) \\ &= K + f(d + (\mathbb{0}, \ldots, \mathbb{0}) \mid i_h) \\ \Rightarrow K &= Z_h + f(d + (\mathbb{0}, \ldots, \mathbb{0}) \mid i_h). \end{aligned}$$

Thus, we can derive K from a single output block. Moreover, when V is an affine subspace, that is V is of the form:

$$V = \{a + v' : v' \in V'\}$$

for some subspace $V' \subseteq \mathbb{F}_2^{b+c}$ and any string $a \in \mathbb{F}_2^{b+c}$, the intermediate state I also becomes d:

$$\begin{aligned} I &= \sum_{j=1}^{\ell} f((u_h \mid i_h) + K) \\ &= \Delta_{V'} f(a + K) \\ &= d. \end{aligned}$$

Please note that any linear space is also an affine space. This shows a major flaw in this construction.

6.3 Countermeasure using the counter encoding

To prevent such a vulnerability from forming, we take a look at the definition of the intermediate state I:

$$I = \sum_{j=1}^{\ell} f((u_h \mid i_h) + K).$$

To prevent the reduction of the degree of f, we must prevent that

$$V = \{(u_h \mid i_h) : 1 \leq h \leq \ell\}$$

is an affine subspace of $\mathbb{F}_2^{b+c}$. Because $u_1 \mid \ldots \mid u_\ell$ is the message M that is chosen by the attacker, we cannot formulate limitations on u_h. However, i_h is specific to the algorithm, and the only requirement up to now is that each i_h is unique. To prevent this vulnerability, we can add extra requirements to i_h. To formulate requirements, we analyse what properties i_h must fulfil if V is an affine subspace of $\mathbb{F}_2^{b+c}$. So let V be an affine subspace of the form:

$$V = \{(u_h \mid i_h) : 1 \leq h \leq \ell\}.$$

Let a be such that V is of the form

$$V = \{a + v' : v' \in V'\}$$

for some subspace V'. Because V is an affine subspace, we must have that for any h, j with $1 \leq h \leq l, 1 \leq j \leq \ell$:

$$\begin{aligned} a + (u_h \mid i_h) + a + (u_j \mid i_j) &= a + a + (u_h \mid i_h) + (u_j \mid i_j) \\ &= (u_h \mid i_h) + (u_j \mid i_j) \in V' \end{aligned}$$

and thus some k, $1 \leq k \leq \ell$ must exists, such that

$$a + (u_k \mid i_k) = (u_h \mid i_h) + (u_h \mid i_j).$$

We can conclude, that for any h, j,$1 \leq h \leq \ell, 1 \leq j \leq \ell$ there exists a k, $1 \leq k \leq \ell$ such that

$$i_h + i_j = i_k + a_{\text{counter}},$$

where a_{counter} are the last c bits of a.

Thus, if V is an affine space, then the counters $i_1, \ldots, i_\ell$ also form an affine space. As a defence to such a vulnerability, we formulate the requirement that the counters $i_1, \ldots, i_\ell$ do not form a large affine space. In this case, a V of the form

$$V = \{(u_h \mid i_h) : 1 \leq h \leq \ell\}$$

cannot be a large affine space, removing this vulnerability from the algorithm.

Furthermore, it is also important that $i_1, \ldots, i_\ell$ does not contain large enough affine spaces. For instance, suppose $i_1, \ldots, i_{\ell+1}$ does not form an affine space, but $i_2, \ldots, i_{\ell+1}$ does form an affine space with $\ell = 2^{\deg(f)}$. Then we could find a matching sequence $u_1, \ldots, u_{\ell+1}$ such that $V = \{u_2 \mid i_2, \ldots, u_{\ell+1} \mid i_{\ell+1}\}$ is an affine space. Again let a be such that V is of the form

$$V = \{a + v' : v' \in V'\}$$

for some subspace V'.

If we would then encrypt $M = u_1, \ldots, u_{\ell+1}$, the intermediate state I would be:

$$\begin{aligned} I &= \sum_{j=1}^{\ell+1} f((u_h \bigm| i_h) + K) \\ &= \Delta_V f(a + K) + f((u_1 \bigm| i_1) + K) \\ &= f((u_1 \bigm| i_1) + K). \end{aligned}$$

So I would only depend on the input messages outside the affine space. Therefore, the counter values $i_1, \ldots, i_\ell, \ldots$ should not only be unique but the largest affine subspace contained in $\{i_1, \ldots, i_\ell, \ldots\}$ should also be of a small enough dimension, at least smaller than the degree of f. Our research presented in chapter 7 will document how such a counter can be designed.

Chapter 7

Low affine dimensional encoding

As described in chapter 6 we need to define an encoding of the values $1, 2, 3, \ldots$ such that it is hard to find a large affine subspace in the set of encoded values. Thus we require that the largest affine subspace has a low dimension. The dimension of this largest affine subspace in the code words generated is called the maximal affine dimension of this encoding. In this chapter, we will first define what properties such an encoding should have and then try to construct an encoding fulfilling these properties. Besides limiting the size of the largest affine dimensional subspace, we also have practical limitations: the encoding must be easy to calculate and should not use a lot of bits. We call this type of encoding a low affine dimensional encoding.

This last property can be formulated using the concept entropy. Entropy, as a measure of randomness contained in a probability distribution, is a fundamental concept in information theory and cryptography [44]. When we speak of entropy, we mean Shannon entropy specifically. Roughly speaking the Shannon entropy of an encoding is the amount of randomness in each code word. In this thesis, we view entropy as the number of bits of information in each code word if we choose one of the code words uniformly. Thus, if an encoding can only represent two values and we choose one of these values uniformly, its Shannon entropy is 1 as we can describe our choice with 1 bit. This means that when we can encode more values in the same bit string length, the Shannon entropy increases. Another related concept is code rate. Code rate describes the amount of information per bit. This means that if the code rate of an encoding is $\frac{k}{n}$, for every k bits of information, the encoding generates a code word of length n, of which $n - k$ are redundant. In this chapter the functions `i2bsp` and `bs2ip` are used frequently. The function $\texttt{i2bsp}(i, d)$ converts an integer i to a bit string of length d. The function $\texttt{bs2ip}(s)$ converts a bit string s to an integer. Definition 2 and definition 5 define `i2bsp` and `bs2ip` respectively.

More formally, we require that a low affine dimensional encoding $\texttt{enc} : \mathcal{I}_n \rightarrow \mathbb{F}_2^m$, with $\mathcal{I}_n = [0, n] \subseteq \mathbb{N}$, must fulfil the following properties:

1. The maximal affine dimension has to be low, while having a low computational complexity. Moreover, if we limit the encoding to $\mathcal{I}_{n'} \rightarrow \mathbb{F}_2^m$ for any n' with $n' < n$, the maximal affine dimension over subsets should also be low. This means that the maximal affine dimension should slowly increase while the number of elements in the domain increases.

2. The entropy should be high: the encoding should be efficient in terms of length. In other

words, the ratio of the length of the code words to the ‘length’ of the largest integer that can be encoded must not be too big. With length of an integer, we mean in this case the fewest number of bits we can use to represent this integer as a bit string.

The first property directly defines the main property of a low affine dimensional encoding, meaning that the dimensions of the affine subspaces must be low. The encoding we propose will encode a simple increasing counter. The algorithms proposed will only encode a certain value i to use in farfalle if all values smaller than i have been encoded. Therefore, if the largest affine subspaces found in the encoding contain encodings of large numbers, the attacker must provide a very large input message to the parallelizable PRF to use this large affine subspace in an attack. Using such a large input message might not be feasible for the attacker. This is codified in the first property, by demanding that the affine dimension is relatively low when only considering the first n counters as input.

The second property is simply an efficiency property: anyone using the encoding should not have to use an excessive number of bytes to encode their counter. We do not state the requirement that when we limit the input space to $\mathcal{I}_{n'}$ (all natural numbers smaller than n') with $n' < n$ that the code rate should still be high. In other words, when encoding a small number, we do not try to use as few bits as possible. We only limit the total bits used when using the whole input space. This is because the encoding is often simply used as a fixed size encoding, so limiting the number of bits used would be an unnecessary restriction.

Below we define what constitutes an encoding.

Definition 16. *An encoding* `enc` *is a module with a state and functions. It contains the following functions:*

- ***Preparation*** *is an optional function that is called with a number of parameters determined by the encoding. This function is always executed before the encoding is used.*
- ***Encode***(i) *encodes the integer* $i, 0 \leq i < n$ *over* $\mathbb{F}_2^d$*, for some maximum range* n*. Some sub-encodings that are used in other encodings can be non-injective, but unless mentioned, all encodings must injective. The maximum value* n *of the input is often left implicit, and can be infinite.*
- ***Increment***(i, v) *encodes the integer* i *given the previous state* v *of* $i-1$*. The previous state is often the previous result, but in some cases more data is kept track of. The result is a tuple with as first element the value of* ***Encode***$(i-1)$*, possibly only one value. This makes it possible to optimize by remembering certain intermediate values. The other values in the tuple are together the state.*
- ***IncrementState***(i) *generates the state* v *that is used by* ***Increment****.*

Functions are called by using for example `enc.`***Preparation****,* `enc.`***Encode****, et cetera. In most cases only* `enc.`***Encode***(i) *is used for some* i*. We will use the shorthand notation* `enc`(i) *for this.*

	c_1	c_2	c_3	c_4	$\dots$	c_n
c_1	$c_1 + c_1$	$c_1 + c_2$	$c_1 + c_3$	$c_1 + c_4$	$\dots$	$c_1 + c_n$
c_2	$c_2 + c_1$	$c_2 + c_2$	$c_2 + c_3$	$c_2 + c_4$	$\dots$	$c_2 + c_n$
c_3	$c_3 + c_1$	$c_3 + c_2$	$c_3 + c_3$	$c_3 + c_4$	$\dots$	$c_3 + c_n$
c_4	$c_4 + c_1$	$c_4 + c_2$	$c_4 + c_3$	$c_4 + c_4$	$\dots$	$c_4 + c_n$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
c_n	$c_n + c_1$	$c_n + c_2$	$c_n + c_3$	$c_n + c_4$	$\dots$	$c_n + c_n$

Table 7.1 The table of differences of the set of code words C.

7.1 Calculating the affine dimension

The first step to design an encoding is to design and implement a program that can experimentally verify the maximum dimension of an affine subspace of the code words produced by an encoding. Unfortunately, it seems that such an algorithm will have a worst case exponential run time in the size of the input set. We try to achieve a good performance by implementing heuristics. To analyse an encoding that produced code words of length w, we generate the first n code words produced by this algorithm and try to find a subset with a maximum affine dimension. The algorithm is thus given as input the set C, $C = \{c_1, \dots, c_n\} \subseteq \mathbb{F}_2^w$.

Firstly, we propose a naive implementation by performing a DFS (depth-first search) on all possible affine subspaces of a given C. Each node in the tree on which we execute a DFS algorithm is then a subset S of C, where the root is the empty set. Each child of a node adds a single element to the subset S of its parent. To avoid considering the same subset twice, we have given each element in C an index. The children of a node can only add elements to the subset S of its parent that have a higher index then each element in S. Essentially, we generate subsets of C and check if these are affine. To check if a subset S of C is an affine subspace, we take all pairs of two elements $s_i, s_j \in S$ and a fixed element $s_1 \in S$. If S is affine, we have $s_i + s_j + s_1 \in S$. If this holds for every combination, we can conclude that S is an affine subspace. To increase the efficiency, we also add a pruning step. Essentially we know which elements the descendants will add to S. Moreover, we can check which elements at least need to be added to S to make S an affine subspace, these are elements can be found by calculating $s_i + s_j + s_1$ for $s_j, s_i \in S$. If the elements that need to be added are not in the set of elements that the descendants will add, we can just skip DFS on that branch.

The pseudocode of this algorithm can be seen in algorithm 1. By calculating the maximum size of all affine subspaces, we can calculate the maximum affine dimension. The maximum affine dimension is the dimension of the maximum affine subspace. The dimension of an affine subspace S can be found by calculating $\log_2(|S|)$.

Unfortunately, algorithm 1 becomes quickly too slow to check large C. For instance, if we define $C_n = \{\texttt{i2bsp}(9 \cdot i) : 0 \leq i < n\}$ (C_n is the set of binary representations of all multiples of 9) then our implementation calculates the affine dimension in 6 seconds for C_{128}, 122 seconds for C_{256} and 10 minutes for C_{376}. To find a quicker algorithm, we analyse the differences between all elements of C. The idea is to calculate the difference between each pair of elements in C. This will give table 7.1.

Suppose we want to know if C has an affine subspace A with N elements. We assume this affine subspace exists, and try to derive some properties of this table of differences. Then we can check if

Algorithm 1 `AffineSubspace`(C, E)

Input $C = \{c_1, \ldots, c_n\}$ is the set of elements;
$E = \{e_1, \ldots e_m\}$ are the indices of the current subset of C, initially $E = \varnothing$.
Output One-by-one all affine subspace of C are outputted.
Description Calculates all affine subsets by performing a DFS with pruning

▷ Check if the subset $\{c_{e_1}, \ldots, c_{e_m}\}$ is affine:
isAffine ← true
for $i \leftarrow 1, \ldots, m$ **do**
  for $j \leftarrow i, \ldots, m$ **do**
    ▷ Check if the sum is inside our subset
    $v \leftarrow c_{e_1} + c_{e_i} + c_{e_j}$
    if $v \notin \{c_{e_1}, \ldots, c_{e_m}, c_{e_m+1}, \ldots, c_n\}$ **then**
      ▷ This subset cannot be made into a subspace
       by adding elements from $\{c_{e_m}, c_{e_m+1}, \ldots, c_n\}$
      return
    else if $v \notin \{c_{e_1}, \ldots, c_{e_m}\}$ **then**
      isAffine ← false

if isAffine **then**
  output $\{c_{e_1}, \ldots, c_{e_m}\}$

▷ Visit all subsets with the first m indices given by E
for $i \leftarrow m + 1, \ldots, n$ **do**
  `AffineSubspace`$(C, E \cup \{i\})$

these properties hold: if this is not the case, C cannot possibly have an affine subspace A with N elements.

Firstly, the affine subspace A will have some numerically minimal element o. To perform addition inside the subspace A for some elements x and y inside the affine subspace, we first have to remove the numerically minimum element o from the subspace, and later add it again:

$$((x + o) + (y + o)) + o = x + y + o$$

For simplicity this is defined as $x \oplus_A y = x + y + o$. Now take an arbitrary pair of elements $x, y \in A$. It can be seen that we can find a vector $z \in A$ such that $x \oplus_A z = y$ for any x. Take $z = x \oplus_A y \in A$ and we get:

$$x \oplus_A z = x \oplus_A x \oplus_A y = x + x + y + o + o = y.$$

Thus for any $x, y \in A$ we can find a vector $z \in A$ such that $x + z = y + o$. Therefore, if the table of differences has an affine subspace A with N elements, it must contain N times the difference $y + o$. Because y was also arbitrary, there must also be N differences that are contained N times in the table of all differences. This gives us a way to calculate an upper bound on the maximum affine dimension of a coding. However, as this is a upper bound, there is no guarantee that an affine subspace with such an dimension is actually contained in the coding. Our algorithm will now use this upper bound to quickly find a subset that might itself be or does contain an affine subspace. After this subset has

been found we need to verify if the subset indeed contains such an affine subspace. If such a subspace cannot be found, this upper bound will not be tight.

The algorithm is given in algorithm 2. The idea is that we use the table of differences to determine which elements can be in an affine subspace of size 2^d, as each element must be found 2^d times in this table of differences once we add any vector of this affine subspace to all differences.

Algorithm 2 `MaximumAffineDimension`(C)

Input $C = \{c_1, \ldots, c_n\}$ is the set of elements.
Output The maximum dimension of the largest affine subspace of C.
Description Calculates all affine subsets by using the table of differences.
$T \leftarrow$ `table of differences` of C.
$d \leftarrow$ Largest number such that there are at least 2^d unique differences that are contained at least 2^d times in T.
while d > 0 **do**
  $T_{\text{outer}} \leftarrow T$
  $C_{\text{outer}} \leftarrow C$
  while $|C_{\text{outer}}| \geq 2^d$ **do**
    $T_{\text{inner}} \leftarrow T_{\text{outer}}$
    $C_{\text{inner}} \leftarrow C_{\text{outer}}$
    $v \leftarrow$ A vector that has at least 2^d differences $d_1, \ldots, d_m$ in its associated row with each d_i occurring at least 2^d times in T_{inner}.
    Add v to each difference in T_{inner}.
    $C' = C_{\text{inner}}$
    do
      $C_{\text{inner}} \leftarrow C'$
      $C' \leftarrow$ differences in T_{inner} that occur at least 2^d times.
      Remove all rows and columns associated with a vector that is not in C' from T.
    while $C_{\text{inner}} \neq C'$
    $S \leftarrow$ largest affine subspace in C_{inner} (Generated with `AffineSubspace`)
    if $|S| = 2^d$ **then**
      return d
    else
      Remove v from C_{outer}
      Remove the column and row associated with v from T_{outer}

The performance of this algorithm varies quite a bit. For some input sets our heuristics perform much better than algorithm 1. However, on some input sets we even see worse performance than algorithm 1. This is probably because the initial upper bound is not tight enough in all cases. Fortunately, algorithm 2 seems to perform better for encodings with lower maximum affine dimensions. As we aim to design such encodings, this means that for well designed encodings, algorithm 2 seems the best way to calculate that maximum affine dimension. Therefore, in the rest of the chapter we use algorithm 2 to calculate the affine subsets. In table 7.2 some results are summarized. The maximum affine dimension of the multiples of 3 and 9 behave almost similar (this will be seen further on), however the run time of the algorithms on the multiples of 3 is much longer than the run time on the multiples of 9. Moreover, algorithm 2 has a longer run time than algorithm 1 on the multiples of 3. This case seems to be unique, in other encountered cases algorithm 2 performs better than algorithm 1. As an example of a well performing encoding we have the multiples of 37. The properties of this encoding will be discussed later on. However, we can instantly see that both algorithm 2 and

Input set	Run time	
	Algorithm 1	Algorithm 2
Multiples of 3		
$\{\texttt{i2bsp}(3 \cdot i) : 0 \leq i < 128\}$	12.4 s	6.3 s
$\{\texttt{i2bsp}(3 \cdot i) : 0 \leq i < 256\}$	228 s	64 m
$\{\texttt{i2bsp}(3 \cdot i) : 0 \leq i < 384\}$	38 m	10 h
Multiples of 9		
$\{\texttt{i2bsp}(9 \cdot i) : 0 \leq i < 128\}$	6.0 s	0.03 s
$\{\texttt{i2bsp}(9 \cdot i) : 0 \leq i < 256\}$	122 s	0.06 s
$\{\texttt{i2bsp}(9 \cdot i) : 0 \leq i < 384\}$	10 m	3.8 s
Multiples of 37		
$\{\texttt{i2bsp}(37 \cdot i) : 0 \leq i < 128\}$	2.6 s	0.09 s
$\{\texttt{i2bsp}(37 \cdot i) : 0 \leq i < 256\}$	29 s	2.4 s
$\{\texttt{i2bsp}(37 \cdot i) : 0 \leq i < 384\}$	135 s	5.5 s

Table 7.2 A comparison of the run time of algorithm 1 and 2 on various input sets.

algorithm 1 have a shorter run time. Furthermore, algorithm 2 has a much shorter run time than algorithm 1 in this case. It can be seen that if the heuristics in algorithm 2 fail to find a correct upper bound, the affine subsets of the input set are calculated multiple times, this could explain the behaviour of the run time. Lastly, both algorithms were implemented in python.

The next step in our research is to formulate and experiment with new encodings. In this manner, we hope to get an idea as to what constitutes a good low affine dimensional encoding. Using algorithm 2 we can experimentally verify effectiveness of the proposed encodings.

7.2 Basic multiplicative encodings

The first encodings we consider are basic multiplicative encodings called $\texttt{mulenc}_c^d$. These encodings encode an integer i by multiplying it by the integer c. Finally, this integer is converted to a bit string using `i2bsp`. This is described in encoding 1.

Encoding 1 $\texttt{mulenc}_c^d$

Encode(i):
  return $\texttt{i2bsp}(c \cdot i, d)$

IncrementState(i):
  return $\texttt{i2bsp}(c \cdot i, d)$

Increment(i, v):
  return $\texttt{i2bsp}(\texttt{bs2ip}(v) + c, d)$

7.2.1 Mathematical properties

The most important property we are concerned about when analysing an encoding is the maximum affine dimension. As a shorthand, we write the maximum affine dimension of the image of an encoding `enc` as afdim (`enc`). In this section, we will make a mathematical analysis to determine how the afdim of `mulenc` behaves as we increase the number of elements in the domain.

The following properties apply to this encoding:

$$\begin{aligned} \texttt{mulenc}^{d}_{(b\cdot c)}(i) &= \texttt{i2bsp}(b \cdot c \cdot i) \\ &= \texttt{i2bsp}(b \cdot \texttt{bs2ip}(\texttt{mulenc}^{d}_{c}(i)), d) \\ \text{and} & \\ \texttt{mulenc}^{d+1}_{(2\cdot c)}(i) &= \texttt{i2bsp}(2 \cdot \texttt{bs2ip}(\texttt{mulenc}^{d}_{c}(i)), d+1) \\ &= \texttt{mulenc}^{d}_{c}(i) \mid (\mathbb{0}). \end{aligned}$$

In other words: when we encode an integer with an instantiation of $\texttt{mulenc}_c$ and multiply this output with an integer d (interpreting the output as a binary encoding of an integer), then this is the same as encoding with $\texttt{mulenc}_{c\cdot d}$. This also means that if we double the c value of some instantiation, the output will remain unchanged except for an extra trailing $\mathbb{0}$.

Lemma 2. *For any domain $\mathcal{I}_n$, the multiplicative encoding $\texttt{mulenc}^{d}_{c}$ with c an even number has the same affine dimension as $\texttt{mulenc}^{d-1}_{(\frac{c}{2})}$, but a lower code rate.*

Proof. Because $\frac{c}{2}$ is a whole number, we have:

$$\texttt{mulenc}^{d}_{c}(i) = 2 \cdot \texttt{mulenc}^{d-1}_{(\frac{c}{2})}(i) \mid (\mathbb{0}).$$

Any affine subspace found in $\texttt{mulenc}^{d-1}_{(\frac{c}{2})}$ is therefore also present in $\texttt{mulenc}^{d}_{c}$. Therefore, the affine dimension is the same. However, $\texttt{mulenc}^{d}_{c}$ uses an extra bit to encode the data, while still encoding the same number of inputs. Therefore, the code rate of $\texttt{mulenc}^{d}_{c}$ is lower than $\texttt{mulenc}^{d-1}_{(\frac{c}{2})}$. $\square$

Lemma 3. *Let an affine subspace $A = \{a_1, \ldots, a_n\}$ of $\texttt{mulenc}^{d}_{c}$ be given, with $a_i < a_j$ for $1 \leq i < j \leq n$. Then if $2^m > \texttt{bs2ip}(a_n)$, the set:*

$$\{a_i \mid a_j : i, j \in \{i, \ldots, n\}\}$$

is an affine subspace of $\texttt{mulenc}^{2d}_{c}$.

Proof. Let an arbitrary c be given, this will be used for our $\texttt{mulenc}_c$ encoding. Let $A = \{a_1, \ldots, a_n\}$ and m be given as described in the lemma. Let $B = \{b_1, \ldots, b_n\}$ be the preimages of the elements in A, thus $\texttt{mulenc}_c^d(b_i) = a_i$.

Then we define:

$$\begin{aligned}
A^* &= \{\texttt{mulenc}_c^{2d}(b_i \cdot 2^m + b_j) : i, j \in \{1, \ldots, n\}\} \\
&= \{\texttt{i2bsp}(c \cdot b_i \cdot 2^m + c \cdot b_j, 2d) : i, j \in \{1, \ldots, n\}\} \\
&= \{\texttt{i2bsp}(2^m \cdot \texttt{bs2ip}(\texttt{mulenc}_c^d(b_i)) \\
&\qquad + \texttt{bs2ip}(\texttt{mulenc}_c^d(b_j)), 2d) : i, j \in \{1, \ldots, n\}\} \\
&= \{\texttt{mulenc}_c^d(b_i) \mid \texttt{mulenc}_c^d(b_j) : i, j \in \{1, \ldots, n\}\} \qquad (\texttt{mulenc}_c(b_j) = a_j < 2^m) \\
&= \{a_i \mid a_j : i, j \in \{i, \ldots, n\}\}
\end{aligned}$$

This directly proves that $A^* \subseteq \mathrm{im}(\texttt{mulenc}_c^{2d})$. What remains to be proven, is that A^* is an affine space. The origin for A becomes a_1 and similarly the origin for A^* becomes $a_1 \mid a_1$. Take an arbitrary $i, j, k, l \in \{1, \ldots, n\}$, then:

$$\begin{aligned}
(a_i \mid a_j) \oplus_{A^*} (a_k \mid a_l) &= (a_i \mid a_j) + (a_k \mid a_l) + (a_1 \mid a_1) \\
&= (a_i + a_k + a_1) \mid (a_j + a_l + a_1) \\
&= (a_i \oplus_A a_k) \mid (a_j \oplus_A a_l) \\
&= b \mid c
\end{aligned}$$

where $b = (a_i \oplus_A a_k) \in A$ and $c = (a_j \oplus_A a_l) \in A$. By the definition of A^*, we can conclude $(b \mid c) \in A^*$. Because the chosen i, j and k, l both can be chosen such that every combination of elements in A^* is considered, we have shown that A^* is an affine space. □

From lemma 2 and 3 it follows that:

Corollary 1. *For any $c, n \in \mathbb{N}$ with $c, n > 0$, we have:*

- $\mathrm{afdim}(\texttt{mulenc}_c^{(n-1)}) = \mathrm{afdim}(\texttt{mulenc}_{2c}^n)$
- $\mathrm{afdim}(\texttt{mulenc}_c^{2n}) \geq 2 \cdot \mathrm{afdim}(\texttt{mulenc}_c^n)$.

Thus we see that for $\texttt{mulenc}_c$ the affine subspaces can be generated as described in lemma 3. This provides a means for attackers to generate affine subspaces. In reality we see that the linearity of the affine dimension of $\texttt{mulenc}_c$ generally holds: the affine dimension does not grow much faster than the lower bound given by the lemma.

One option to counter this property of $\texttt{mulenc}_c$ is to take a large c value. By choosing a large c value the starting length of the encoding will be large, an encoding that encodes the values $0 \ldots 2^n$ for some n will use much more than n bits, say l bits, with $l \gg n$. But when we encode $0 \ldots 2^{2n}$ the encoding will take up $l + n$ bits, while the dimension will double according to lemma 3 when the encoding uses $2l$ bits. Because $l \gg n$, and thus $2l \gg l + n$, lemma 3 has no significant impact. But

just taking any large c will not help. For instance, if c is even we have that the afdim of a range of indices is the same as $\frac{c}{2}$. But also, if $c = 2^d + 1$ for some $d > 1$, we have that for each $e < d$ that $\text{afdim}(\{\texttt{mulenc}_c^{2d+2}(0), \ldots \texttt{mulenc}_c^{2d+2}(2^e - 1)\}) = e$, thus the affine dimension is maximal for these sets.

But what if we would choose the opposite: choosing only $\mathbb{1}$'s in the binary representation: $c = \texttt{i2bsp}(\sum_{i=0}^{d-1} 2^i, d)$ for some $d > 1$. Or even, alternating $\mathbb{0}$'s and $\mathbb{1}$'s in the binary representation: $c = \sum_{i=0}^{d-1} 2^{2 \cdot i}$. In both cases the affine dimension is still far from optimal and close to maximal. In the next part of this section, this will be proven.

Lemma 4. *Let c be of the form $2^m - 1$, $m \in \mathbb{N}$ then for any $0 \leq i \leq c$ we have:*

$$\texttt{i2bsp}(c, m) + \texttt{i2bsp}(i, m) = \texttt{i2bsp}(c - i, m).$$

Proof. Let a c of the form $2^m - 1$ and an $0 \leq i \leq c$ be given. A bitwise exclusive-or with $\texttt{i2bsp}(c, m)$ means inverting all bits, as all elements of $\texttt{i2bsp}(c, m)$ are $\mathbb{1}$. We write:

$$i = \sum_{j=0}^{m-1} i_j \cdot 2^j,$$

where i_j is j^{th} element of the binary string $\texttt{i2bsp}(i, m)$. Then

$$\begin{aligned} c - i &= 2^m - 1 - \sum_{j=0}^{m-1} i_j \cdot 2^j \\ &= \sum_{j=0}^{m-1} 2^j - \sum_{j=0}^{m-1} i_j \cdot 2^j \\ &= \sum_{j=0}^{m-1} (1 - i_j) \cdot 2^j \end{aligned}$$

Thus $\texttt{i2bsp}(c - i, m)$ also inverts all bits, and thus $\texttt{i2bsp}(c, m) + \texttt{i2bsp}(i, m) = \texttt{i2bsp}(c - i, m)$. ☐

Lemma 5. *Let c be of the form $2^m - 1$, $m \geq 1$ then for $1 \leq i \leq 2^m$, we have:*

$$\texttt{i2bsp}(c, 2m) + \texttt{i2bsp}(c \cdot i, 2m) = \texttt{i2bsp}(i - 1, m) \mid \texttt{i2bsp}(i - 1, m).$$

Proof. By induction.

Base.

Take $i = 1$, then:

$$\begin{aligned}
\mathtt{i2bsp}(c, 2m) + \mathtt{i2bsp}(c \cdot i, 2m) &= \mathtt{i2bsp}(c, 2m) + \mathtt{i2bsp}(c, 2m) \\
&= (\mathbb{0}, \ldots, \mathbb{0}) \\
&= \mathtt{i2bsp}(0, m) \mid \mathtt{i2bsp}(0, m) \\
&= \mathtt{i2bsp}(i-1, m) \mid \mathtt{i2bsp}(i-1, m).
\end{aligned}$$

Step.

Let an i, $2^m \geq i > 1$ be given. First we derive:

$$\begin{aligned}
c \cdot i &= c \cdot (i-1) + c && (7.1)\\
&= \mathtt{bs2ip}(\mathtt{i2bsp}(c \cdot (i-1), 2m)) + c \\
&= \mathtt{bs2ip}(\mathtt{i2bsp}(c \cdot (i-1), 2m) + \mathtt{i2bsp}(c, 2m) + \mathtt{i2bsp}(c, 2m)) + c \\
&= \mathtt{bs2ip}((\mathtt{i2bsp}(i-2, m) \mid \mathtt{i2bsp}(i-2, m)) + \mathtt{i2bsp}(c, 2m)) + c && \text{(IH)} \\
&= \mathtt{bs2ip}(\mathtt{i2bsp}(i-2, m) \mid (\mathtt{i2bsp}(i-2, m) + \mathtt{i2bsp}(c, m))) + c && (c < 2^m) \\
&= \mathtt{bs2ip}((\mathtt{i2bsp}(i-2, m) \mid \mathtt{i2bsp}(c-(i-2), m)) + c && \text{(Lemma 4)} \\
&= \mathtt{bs2ip}(\mathtt{i2bsp}(i-2, m) \mid \mathtt{i2bsp}(c-(i-2), m)) + 2^m - 1 && (c = 2^m - 1) \\
&= \mathtt{bs2ip}(\mathtt{i2bsp}(i-2, m)) \cdot 2^m + \mathtt{bs2ip}(\mathtt{i2bsp}(c-(i-2), m)) + 2^m - 1 \\
&= (\mathtt{bs2ip}(\mathtt{i2bsp}(i-2, m)) + 1) \cdot 2^m + \mathtt{bs2ip}(\mathtt{i2bsp}(c-(i-2), m)) - 1 \\
&= \mathtt{bs2ip}(\mathtt{i2bsp}(i-1, m) \mid \mathtt{i2bsp}(c-(i-1), m)). && (0 \leq i-2 < c)
\end{aligned}$$

Then:

$$\begin{aligned}
\mathtt{i2bsp}(c, 2m) + \mathtt{i2bsp}(c \cdot i, 2m) &= \mathtt{i2bsp}(c, 2m) + \mathtt{i2bsp}(c \cdot (i-1) + c, 2m) \\
&= \mathtt{i2bsp}(c, 2m) + (\mathtt{i2bsp}(i-1, m) \mid \mathtt{i2bsp}(c-(i-1), m)) && \text{(Equation 7.1)} \\
&= \mathtt{i2bsp}(i-1, m) \mid (\mathtt{i2bsp}(c, m) + \mathtt{i2bsp}(c-(i-1), m)) && (c < 2^m) \\
&= \mathtt{i2bsp}(i-1, m) \mid \mathtt{i2bsp}(i-1, m). && \text{(Lemma 4)}
\end{aligned}$$

□

Lemma 6. *Let c be of the form $2^m - 1$, $m \in \mathbb{N}$ with $m > 0$ then the sets:*

$$\{\mathtt{mulenc}_c^{2m}(i) : i \in \mathbb{N}, 1 \leq i \leq 2^d\} \qquad d \in \mathbb{N}, 1 \leq d \leq m$$

are affine subspaces of $\mathtt{mulenc}_c^{2m}$.

Proof. Let c of the form $2^m - 1$, $m > 0$ and d, $1 \leq d \leq m$ be given.
Define:

$$A = \{\mathtt{mulenc}_c^{2m}(i) : i \in \mathbb{N}, 1 \leq i \leq 2^d\}.$$

Obviously, A is a subset of $\mathrm{im}(\mathtt{mulenc}_c^{2m})$. What remains to be proven is that A is an affine space. If A were an affine space, c would be its origin. Let i, j be arbitrary numbers fulfilling $1 \leq i, j \leq 2^d$. We define $k = \mathtt{bs2ip}(\mathtt{i2bsp}(i-1, m) + \mathtt{i2bsp}(j-1, m))$, $1 \leq k+1 \leq 2^d$. Using this, we have that:

$$\begin{aligned}
&\mathtt{mulenc}_c^{2m}(i) \oplus_A \mathtt{mulenc}_c^{2m}(j) \\
&\quad = \mathtt{i2bsp}(c, 2m) + \mathtt{i2bsp}(c \cdot i, 2m) + \mathtt{i2bsp}(c \cdot j, 2m) \\
&\quad = \mathtt{i2bsp}(c, 2m) + \mathtt{i2bsp}(c \cdot i, 2m) + \mathtt{i2bsp}(c, 2m) \\
&\qquad\quad + \mathtt{i2bsp}(c \cdot j, 2m) + \mathtt{i2bsp}(c, 2m) \\
&\quad = \mathtt{i2bsp}(c, 2m) + (\mathtt{i2bsp}(i-1, m) \mid \mathtt{i2bsp}(i-1, m)) \\
&\qquad\quad + (\mathtt{i2bsp}(j-1, m) \mid \mathtt{i2bsp}(j-1, m)) && \text{(Lemma 5)} \\
&\quad = \mathtt{i2bsp}(c, 2m) + (\ (\mathtt{i2bsp}(i-1, m) + \mathtt{i2bsp}(j-1, m)) \ \mid \\
&\qquad\qquad\qquad (\mathtt{i2bsp}(i-1, m) + \mathtt{i2bsp}(j-1, m))\ \ ) \\
&\quad = \mathtt{i2bsp}(c, 2m) + (\mathtt{i2bsp}(k, m) \mid \mathtt{i2bsp}(k, m)) \\
&\quad = \mathtt{i2bsp}(c \cdot (k+1), 2m) && \text{(Lemma 5)} \\
&\quad = \mathtt{mulenc}_c^{2m}(k+1)
\end{aligned}$$

This directly proves that A is an affine subspace. □

This gives more information about how we should choose c. From lemma 6 follows corollary 2:

Corollary 2. *Let $n, m \geq 1$, then:*

1. *If $c = 2^n - 1$, then for $k < n$:* $\mathrm{afdim}(\mathtt{mulenc}_c^{(n+k)}) = k - 1$.
2. *If $c = \sum_{i=0}^{n} 2^{m \cdot i}$, then for $k < n$:* $\mathrm{afdim}(\mathtt{mulenc}_c^{(n \cdot m+k)}) \geq k - 1$.

Why corollary 2.2 is true, can be seen by viewing the binary representation of c. This will be of the form: $(\mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{1}, \ldots, \mathbb{0}, \mathbb{1})$, $(\mathbb{1}, \mathbb{0}, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{0}, \mathbb{1}, \ldots, \mathbb{0}, \mathbb{0}, \mathbb{1})$, $(\mathbb{1}, \mathbb{0}, \mathbb{0}, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{0}, \mathbb{0}, \mathbb{1}, \ldots, \mathbb{0}, \mathbb{0}, \mathbb{0}, \mathbb{1})$, et cetera. But then we have that $\mathtt{mulenc}_c(3) = (\mathbb{1}, \mathbb{1}, \mathbb{1}, \mathbb{1}, \mathbb{1}, \ldots, \mathbb{1}, \mathbb{1})$, $\mathtt{mulenc}_c(7) = (\mathbb{1}, \mathbb{1}, \mathbb{1}, \mathbb{1}, \mathbb{1}, \ldots, \mathbb{1}, \mathbb{1})$, $\mathtt{mulenc}_c(15) = (\mathbb{1}, \mathbb{1}, \mathbb{1}, \mathbb{1}, \mathbb{1}, \ldots, \mathbb{1}, \mathbb{1})$ respectively. Combining this with lemma 6 directly gives the result.

So according to corollary 2, a repeating pattern should be avoided in the binary representation of c. Rather, $\mathtt{i2bsp}(c, d)$ should somewhere contain a string where two elements with value $\mathbb{1}$ are close to each other. This prevents that the encoding will just put the binary representations of i next to each other (thus preventing that the encoding of i looks like $\mathtt{i2bsp}(i, m) \mid (\mathbb{0}, \ldots, \mathbb{0}) \mid \mathtt{i2bsp}(i, m) \mid \ldots$. Furthermore, corollary 2 also gives us that $\mathtt{i2bsp}(c, d)$ should contain multiple consecutive $\mathbb{0}$'s to

prevent such a repeating pattern.

7.2.2 Experimental results

Using algorithm 2 we can verify the actual performance of $\texttt{mulenc}_c^d$ for different values of c. In our analysis, we limit ourselves to bit strings of 64 bits, this is the default integer size of our computer.

As a starting point the maximal affine dimension of the code words generated by $\texttt{mulenc}_3^{64}$, $\texttt{mulenc}_5^{64}$, $\texttt{mulenc}_7^{64}$ and $\texttt{mulenc}_9^{64}$ were calculated. To get a better understanding of how the maximal affine dimension increases when the encoding encodes more values, we vary the number of counters encoded. As a shorthand we define the function codewords($\texttt{enc}, n$) that maps the set of integers $0 \dots n-1$ to the set of the encodings using $\texttt{enc}$ of these same integers $0 \dots n-1$:

$$\text{codewords}(\texttt{enc}, n) = \{\texttt{enc}(i) : 0 \leq i < n\}.$$

Using the codewords function we calculate afdim(codewords($\texttt{enc}, n$)) to get an understanding of how the maximal affine dimension increases as a function of the number of code words. As calculating the maximum affine dimension of codewords($\texttt{enc}, n$) for each n takes a substantial amount of time, we use an adaptive stepsize to calculate the maximum affine dimension for different n. In our case, this means that the values of n are defined by the sequence n_i:

$$\begin{aligned} n_1 &= 1, \\ n_i &= n_{i-1} + 2^{\max\{\lfloor \log_2 n_{i-1} - 4 \rfloor, 0\}}. \end{aligned}$$

This means that the sequence n_i doubles its stepsize each time it reaches a power of 2, starting at 32:

$$n_i = (1, 2, 3, \dots, 32, 34, 36, \dots, 62, 64, 68, \dots, 124, 128, 136, 144, \dots).$$

Another property that we are concerned about is the entropy. In the ideal case, we can encode 2^d numbers in d bits: but this would also generate the worst possible maximum affine dimension. To gain an understanding how different encodings behave, we define the property $\texttt{Ipb}(C)$, information per bit:

$$\texttt{Ipb}(C) = \frac{\log_2(|C|)}{\hat{d}}.$$

Here $\hat{d}$ is the *minimal* number of bits used to represent C. With *minimal* number of bits, we mean that while we can use 128 bits to represent the code words $\texttt{i2bsp}(0, 128), \dots, \texttt{i2bsp}(15, 128)$ we can also use 4 bits to represent them. Therefore $\hat{d}$ will be 4 in this case.

In table 7.3 the minimal number of elements required to have a certain maximum affine dimension in each encoding $\texttt{mulenc}_3^{64}$, $\texttt{mulenc}_5^{64}$, $\texttt{mulenc}_7^{64}$ and $\texttt{mulenc}_9^{64}$ is given. One can see that the lower bound of afdim given in corollary 1 is pretty close to the actual behaviour of afdim: each time the number of bits used doubles, the afdim also doubles.

Minimal number of elements n and minimal number of bits $\hat{d}$ for various afdim values.

	Maximum affine dimension (afdim)											
	1		2		3		4		5		6	
Encoding	n	$\hat{d}$	n	$\hat{d}$	n	$\hat{d}$	n	$\hat{d}$	n	$\hat{d}$	n	$\hat{d}$
$\texttt{mulenc}_{(3)}^{(64)}$	2	3	5	5	18	6	72	8	288	10	>640	≥11
$\texttt{mulenc}_{(5)}^{(64)}$	2	4	4	5	19	7	50	8	304	11	>640	≥12
$\texttt{mulenc}_{(7)}^{(64)}$	2	4	5	6	9	6	68	9	272	11	544	12
$\texttt{mulenc}_{(9)}^{(64)}$	2	5	4	6	8	7	72	10	200	11	464	13

Table 7.3 The minimal number of elements n required such that codewords($\texttt{mulenc}_c^{(64)}, n$) for different c values has a certain maximum dimension of its affine subspaces. The n values can be up to $2^{\lfloor \log_2 n \rfloor - 4}$ too big. The $\hat{d}$ value gives the maximum word length in codewords($\texttt{enc}, n$). The c values are the first 4 odd numbers in $\mathbb{N}$ (excluding 1).

Table 7.3 gives us a baseline of compact encodings (i.e. encodings that can encode a lot of values in a certain string length). As we have seen in corollary 1 the maximum affine dimension can be reduced by increasing c. Corollary 2 tells us that the binary representation of c should not contain repetitions. To find new candidates for c, we have chosen four prime numbers between 32 and 1024, which have almost an equal number of $\mathbb{1}$'s and $\mathbb{0}$'s in their binary representation. The following four values for c have been chosen:

- 37: $(\mathbb{1}, \mathbb{0}, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{1})$. The binary representation has three $\mathbb{0}$'s and three $\mathbb{1}$'s. Also, no repetition can be found in the binary representation. The number 37 is also the number people are most likely to state when asked to give a random number between 0 and 100.
- 151: $(\mathbb{1}, \mathbb{0}, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{1}, \mathbb{1})$. The binary representation has three $\mathbb{0}$'s and five $\mathbb{1}$'s.
- 233: $(\mathbb{1}, \mathbb{1}, \mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{0}, \mathbb{1})$. The binary representation of 233 is the representation of 151 written backwards. Therefore, the binary representation has three $\mathbb{0}$'s and five $\mathbb{1}$'s. It will be interesting to see if 233 performs similar to 151.
- 821: $(\mathbb{1}, \mathbb{1}, \mathbb{0}, \mathbb{0}, \mathbb{1}, \mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{1})$. The binary representation has four $\mathbb{0}$'s and six $\mathbb{1}$'s. A hint of repetition can be found in its representation if one looks closely, as the representation starts with two $\mathbb{1}$'s than two $\mathbb{0}$'s and then again two $\mathbb{1}$'s. It will be interesting to see how this c value behaves.

Please note that the c values all have a different number of bits in its minimal representation. With minimal representation, we mean that the most significant bit would be $\mathbb{1}$ in that case. Therefore, if we just look at the number of bits 821 should perform better than 37. However, 37 has less repetition than 821. This experiment allows to find how much the number of bits and the amount of repetition affects the maximum affine dimension. Table 7.4 contains the minimal number of elements required to reach a certain afdim for each value of c. $\hat{d}$ is again the minimal number of bits for an afdim value.

According to table 7.4 we see that 821 performs relatively poorly while it is the largest c-value of the four. This shows that having little repetition in the binary representation can be more important

Minimal number of elements n and minimal number of bits $\hat{d}$ for various afdim values.

	Maximum affine dimension (afdim)											
	1		2		3		4		5		6	
Encoding	n	$\hat{d}$	n	$\hat{d}$	n	$\hat{d}$	n	$\hat{d}$	n	$\hat{d}$	n	$\hat{d}$
$\texttt{mulenc}^{(64)}_{(37)}$	2	7	4	8	52	11	288	14	1600	16	>3200	≥17
$\texttt{mulenc}^{(64)}_{(151)}$	2	9	12	11	50	13	108	14	224	16	3200	19
$\texttt{mulenc}^{(64)}_{(233)}$	2	9	18	13	104	15	864	18	>4096	≥20	>4096	≥20
$\texttt{mulenc}^{(64)}_{(821)}$	2	11	10	14	36	15	368	19	1024	20	2304	21

Table 7.4 The minimal number of elements n required such that codewords($\texttt{mulenc}^{(64)}_c, n$) for different c values has a certain maximum dimension of its affine subspaces. The n values can be up to $2^{\lfloor \log_2 n \rfloor - 4}$ too big. The $\hat{d}$ value gives the maximum word length in codewords($\texttt{enc}, n$). The c values have been chosen by hand picking various primes between 32 and 1024, which is motivated in the text.

than just having a large c-value. Furthermore, 233 outperforms 157 while the binary representation of 233 is the binary representation of 157 backwards. This shows that there is probably no correlation between the two. Furthermore, 37 performs relatively good having only a higher maximal affine dimension than 233. This is especially impressive, considering it uses fewer bits than the other three encodings. Furthermore, it seems that being a prime number does not provide a benefit to the maximum affine dimension. We found that by calculating the maximum affine dimension for several values in the $[32, 1024]$ range for different n, many composite numbers had similar maximum affine dimensions as the selected candidates.

Unfortunately, the properties we described do not completely describe the 'ideal' candidate. While they do provide some 'ideal' properties, many values fulfil such properties. Therefore, to find a good performing candidate we performed a more brute-force search: we calculated the maximum affine dimension for codewords($\texttt{mulenc}^{(64)}_c, 300$), codewords($\texttt{mulenc}^{(64)}_c, 400$), codewords($\texttt{mulenc}^{(64)}_c, 500$) and codewords($\texttt{mulenc}^{(64)}_c, 600$) for c in the range $[9000, 12\,000]$. The five best performing c candidates were selected. These c candidates are:

- 11 479: $(\mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{1}, \mathbb{0}, \mathbb{0}, \mathbb{1}, \mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{1}, \mathbb{1})$,
- 11 591: $(\mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{0}, \mathbb{0}, \mathbb{1}, \mathbb{1}, \mathbb{1})$,
- 11 641: $(\mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{1}, \mathbb{1}, \mathbb{1}, \mathbb{0}, \mathbb{0}, \mathbb{1})$,
- 11 727: $(\mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{1}, \mathbb{1}, \mathbb{0}, \mathbb{0}, \mathbb{1}, \mathbb{1}, \mathbb{1}, \mathbb{1})$,
- 11 873: $(\mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{1}, \mathbb{1}, \mathbb{0}, \mathbb{0}, \mathbb{1}, \mathbb{1}, \mathbb{0}, \mathbb{0}, \mathbb{0}, \mathbb{0}, \mathbb{1})$.

While the range from which the best candidates were selected was relatively large, the actual values of the candidates are quite close to each other. This results in the same most significant bits in all the selected candidates. Apparently, having $(\mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{1})$ as the most significant bits seems to be a good choice. Furthermore, having multiple consecutive $\mathbb{1}$'s seems to be a good choice too. This is interesting, as we have proven that c consisting of only consecutive $\mathbb{1}$'s is one of the worst combinations.

Minimal number of elements n and minimal number of bits $\hat{d}$ for various afdim values.

	Maximum affine dimension (afdim)							
	1		2		3		4	
Encoding	n	$\hat{d}$	n	$\hat{d}$	n	$\hat{d}$	n	$\hat{d}$
$\mathtt{mulenc}^{(64)}_{(11\,479)}$	2	15	21	18	496	23	>3584	≥26
$\mathtt{mulenc}^{(64)}_{(11\,591)}$	2	15	28	18	512	23	>3584	≥26
$\mathtt{mulenc}^{(64)}_{(11\,641)}$	2	15	20	18	448	23	2560	25
$\mathtt{mulenc}^{(64)}_{(11\,727)}$	2	15	22	18	480	23	2816	25
$\mathtt{mulenc}^{(64)}_{(11\,873)}$	2	15	30	19	512	23	2688	25

Table 7.5 The minimal number of elements n required such that codewords($\mathtt{mulenc}^{(64)}_c, n$) for different c values has a certain maximum dimension of its affine subspaces. The n values can be up to $2^{\lfloor \log_2 n \rfloor - 4}$ too big. The $\hat{d}$ value gives the maximum word length in codewords($\mathtt{enc}, n$). The c values have been chosen by selecting the best candidates from the range $[9000, 12\,000]$.

To explain why having multiple consecutive $\mathbb{1}$'s produces no bad results in this case, we can take a look at the lemma that allows us to easily produce affine spaces. An important property of c with multiple consecutive $\mathbb{1}$'s is lemma 5. This lemma states that

$$\mathtt{i2bsp}(c, 2m) + \mathtt{i2bsp}(c \cdot i, 2m) = \mathtt{i2bsp}(i-1, m) \mid \mathtt{i2bsp}(i-1, m).$$

If we have $i = 2$, this gives:

$$\mathtt{i2bsp}(c, 2m) + \mathtt{i2bsp}(c \cdot 2, 2m) = \mathtt{i2bsp}(1, m) \mid \mathtt{i2bsp}(1, m).$$

But in our case we get:

$$\mathtt{i2bsp}(11\,479 \cdot 2, 15) + \mathtt{i2bsp}(11\,479, 15) = (\mathbb{1}, \mathbb{1}, \mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{1}, \mathbb{1}, \mathbb{1}, \mathbb{0}, \mathbb{0}, \mathbb{1}).$$

Compare this with:

$$\mathtt{i2bsp}(11\,641, 15) = (\mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{1}, \mathbb{1}, \mathbb{1}, \mathbb{0}, \mathbb{0}, \mathbb{1}).$$

Thus even $\mathtt{i2bsp}(11\,479 \cdot 2, 15) + \mathtt{i2bsp}(11\,479, 15)$ produces a supposedly good candidate c value. Lemma 5 used this formulate to generate an affine subset. As this formula produces a good candidate, it makes it hard to generate affine subspaces based on the same trick we apply in lemma 5. It might thus be interesting to check for each potential c candidate whether $\mathtt{i2bsp}(c \cdot 2, 15) + \mathtt{i2bsp}(c, 15)$ would also be a good candidate.

In table 7.5 the behaviour of the maximum affine dimension is given. We see that 11 479 and 11 591 produces the best results.

7.3 Logarithmic encodings

The next encoding we propose will build upon the multiplicative encoding, or in general: improve another encoding. The most interesting and problematic result of the theoretical analysis of mulenc is lemma 3. As a reminder, lemma 3 allows us to construct a larger affine subspace from a smaller affine subspace of $\texttt{mulenc}_c$. This was done by using the property that $\texttt{mulenc}_c^{2d}(i \cdot 2^d) = \texttt{mulenc}_c^d(i) \mid (\mathbb{0}, \ldots, \mathbb{0})$. An idea to counter this, is to slowly change c when i increases, thus preventing such a construction.

A simple idea is to take multiples of c depending on the number of bits that i uses. This comes down to the following encoding, which is called the $\texttt{logenc}^d_{\texttt{mulenc}_c}$ encoding:

$$(c \cdot \lfloor \log_2(i) + 1 \rfloor) \cdot i$$

We see that this function wraps $\texttt{mulenc}_c$ in a way. In fact this method can be adapted to other encodings as well. Therefore, we define the encoding in the fashion as described by logenc in encoding 2. For a faster implementation of the **Increment** function, we use the PowerOfTwo function to check if an integer i is a power of two.

$$\texttt{PowerOfTwo}^d(i) = \begin{cases} \textbf{true} & \text{if } \texttt{i2bsp}(i, d) \cdot \texttt{i2bsp}(i-1, d) = 0, \\ \textbf{false} & \text{otherwise.} \end{cases} \tag{7.2}$$

Note that i must be greater than or equal to 1 for correctness of the PowerOfTwo function.

First, note that the length of the output binary string of the specified enc parameter must match d. Because these parameters are the same, the parameter is sometimes omitted. Secondly, the $\log_2$ operation is quite expensive. As we only need to calculate the integer values of $\log_2$, which is the minimum number of bits required to encode i, we can use functions designed to directly calculate this value. Albeit the implementation of these functions can be faster, it is still a more expensive operation than an addition or a multiplication. We aimed to remove this method in the **Increment** function: this is done by checking if i is a power of 2. For this we use the efficient $\texttt{PowerOfTwo}^d$ function.

7.3.1 Dealing with even values of $\lfloor \log_2(i) + 1 \rfloor$

A good observation that can be made is that $\lfloor \log_2(i) + 1 \rfloor$ will be even half of the time. As seen in corollary 11 even values of c used in $\texttt{mulenc}_c$ are useless. This is not necessarily true for the logarithmic encoding, as the value of $\lfloor \log_2(i) + 1 \rfloor$ will vary and still prevents the property proven in lemma 3. However, it can be conjectured that the encoding would be more efficient if $\lfloor \log_2(i) + 1 \rfloor$ would odd. A solution would be to change the logarithmic encoding to:

$$c \cdot (2\lfloor \log_2(i) + 1 \rfloor + 1) \cdot i$$

Encoding 2 $\texttt{logenc}^{d}_{\texttt{enc}}$

Encode(i):
  if $i > 1$ **then**
    return $\texttt{i2bsp}(\texttt{bs2ip}(\texttt{enc}.\textbf{Encode}(i), d) \cdot \lfloor \log_2(i) + 1 \rfloor, d)$
  else
    return $\texttt{enc}.\textbf{Encode}(i)$

IncrementState(i):
  $v_{\texttt{enc}} \leftarrow \texttt{enc}.\textbf{IncrementState}(i)$
  if $i > 1$ **then**
    return $(\texttt{i2bsp}(\texttt{bs2ip}((v_{\texttt{enc}})_0 \cdot \lfloor \log_2(i) + 1 \rfloor, d), \lfloor \log_2(i) + 1 \rfloor, v_{\texttt{enc}})$
  else
    return $((v_{\texttt{enc}})_0, 0, v_{\texttt{enc}})$

Increment($i, (v, v_{\log}, v_{\texttt{enc}})$):
  $v_{\texttt{enc}} \leftarrow \texttt{enc}.\textbf{Increment}(i, v_{\texttt{enc}})$
  if $i > 1$ **then**
    if $\texttt{PowerOfTwo}^{d}(i)$ **then**
      $v_{\log} \leftarrow v_{\log} + 1$
    $v \leftarrow \texttt{i2bsp}(\texttt{bs2ip}(v_{\texttt{enc}}) \cdot v_{\log}, d)$
    return $(v, v_{\log}, v_{\texttt{enc}})$
  else
    $v_{\texttt{enc}} \leftarrow \texttt{enc}.\textbf{Increment}(i, v_{\texttt{enc}})$
    return $((v_{\texttt{enc}})_0, 0, v_{\texttt{enc}})$

Minimal number of elements n and minimal number of bits $\hat{d}$ for various afdim values.

	Maximum affine dimension (afdim)											
	1		2		3		4		5		6	
Encoding	n	$\hat{d}$	n	$\hat{d}$	n	$\hat{d}$	n	$\hat{d}$	n	$\hat{d}$	n	$\hat{d}$
$\texttt{logenc}_{\texttt{mulenc}^{(64)}_{(3)}}$	2	4	7	6	32	10	124	12	800	15	2560	17
$\texttt{logenc}_{\texttt{mulenc}^{(64)}_{(5)}}$	2	5	7	7	48	11	184	13	992	16	3200	18
$\texttt{logenc}_{\texttt{mulenc}^{(64)}_{(7)}}$	2	5	7	8	62	12	200	14	320	15	2304	18
$\texttt{logenc}_{\texttt{mulenc}^{(64)}_{(9)}}$	2	6	11	9	80	13	96	13	128	14	3584	19
$\texttt{logenc}_{\texttt{mulenc}^{(64)}_{(37)}}$	2	8	7	10	88	15	1600	20	3968	21	>4096	≥21
$\texttt{logenc}_{\texttt{mulenc}^{(64)}_{(151)}}$	2	10	27	15	92	17	>4096	≥23	>4096	≥23	>4096	≥23
$\texttt{logenc}_{\texttt{mulenc}^{(64)}_{(233)}}$	2	10	29	16	208	19	1344	22	>4096	≥24	>4096	≥24
$\texttt{logenc}_{\texttt{mulenc}^{(64)}_{(821)}}$	2	12	20	17	176	21	608	23	864	23	>4096	≥26

Table 7.6 The minimal number of elements n required such that codewords($\texttt{logenc}_{\texttt{mulenc}^{64}_{c}}, n$) for different c values has a certain maximum dimension of its affine subspaces. The n values can be up to $2^{\lfloor \log_2 n \rfloor - 4}$ too big. The $\hat{d}$ value gives the maximum word length in codewords($\texttt{enc}, n$). The c values chosen are 3,5,7,9,37,151,233 and 821.

This encoding is called $\texttt{2logenc}^{d}_{\texttt{mulenc}_c}$ encoding. However, repeating multiple $\mathbb{1}$'s at the end of c should be avoided, and considering this, changing the logarithmic encoding to:

$$(c \cdot (4\lfloor \log_2(i) + 1 \rfloor + 1) \cdot i$$

seems better. This encoding is called the $\texttt{4logenc}^{d}_{enc}$ encoding. The logarithmic encodings `logenc` and `4logenc` are experimentally verified, and the results will be discussed below. The definition of both encodings is straightforward from encoding 2 and is left to the reader.

7.3.2 Experimental results

To verify the performance of `logenc` we calculate the behaviour of the maximum affine dimension of $\texttt{logenc}_{\texttt{mulenc}^{(64)}_{c}}$ for various c values. The results can be found in table 7.6. The behaviour seems quite random, while 9 requires the largest n to reach an affine dimension of 6 of the four c's smaller than 10, it can reach an affine dimension of 4 and 5 with less elements than any of the other encodings with c smaller than 10. A similar behaviour can be seen when c is 151. These two show that it might be hard to predict how the `logenc` performs when encoding larger sets of elements. An explanation can be that $\log_2$ will sometimes be an even number and sometimes an odd number with desirables properties. While `logenc` has unpredictable results, it does reduce the maximum affine dimension of each encoding. Moreover, `4logenc` might solve these problems, by making sure that the value we multiply with is odd and does not consist solely of only $\mathbb{1}$'s in its binary representation.

The next step is to verify the performance of `4logenc` in the same way as we did for `logenc`. To this end, the behaviour of the maximum affine dimension of $\texttt{4logenc}_{\texttt{mulenc}^{(64)}_{c}}$ is calculated for various c values. Some c values seen in table 7.6 have been omitted due to limitations in the resources available

Minimal number of elements n and minimal number of bits $\hat{d}$ for various afdim values.

	Maximum affine dimension (afdim)											
	1		2		3		4		5		6	
Encoding	n	$\hat{d}$	n	$\hat{d}$	n	$\hat{d}$	n	$\hat{d}$	n	$\hat{d}$	n	$\hat{d}$
$\texttt{4logenc}_{\texttt{mulenc}^{(64)}_{(3)}}$	2	6	16	10	25	11	800	17	3584	20	>4096	≥20
$\texttt{4logenc}_{\texttt{mulenc}^{(64)}_{(5)}}$	2	7	8	10	64	14	608	17	736	18	960	18
$\texttt{4logenc}_{\texttt{mulenc}^{(64)}_{(7)}}$	2	7	13	11	176	16	352	17	1408	19	>4096	≥21
$\texttt{4logenc}_{\texttt{mulenc}^{(64)}_{(9)}}$	2	8	13	11	92	15	1984	20	3328	21	>4096	≥21
$\texttt{4logenc}_{\texttt{mulenc}^{(64)}_{(37)}}$	2	10	7	12	104	17	2048	22	>4096	≥23	>4096	≥23

Table 7.7 The minimal number of elements n required such that codewords($\texttt{4logenc}_{\texttt{mulenc}^{64}_{c}}, n$) for different c values has a certain maximum dimension of its affine subspaces. The n values can be up to $2^{\lfloor \log_2 n \rfloor - 4}$ too big. The $\hat{d}$ value gives the maximum word length in codewords($\texttt{enc}, n$). The c values chosen are 3,5,7,9 and 37.

to analyse the maximum affine dimensions. The results can be seen in table 7.7. Unfortunately, the behaviour still seems quite random and unpredictable. It still seems hard to predict which c will generate a set of code words with the smallest maximum affine dimension when we increase the number of code words. Moreover, if we compare table 7.7 with table 7.5 we see that the simple $\texttt{mulenc}_c$ with a large c value out performs $\texttt{4logenc}_{\texttt{mulenc}_c}$ with a smaller c value. Furthermore, the number of bits used ($\hat{d}$) is only a little bit more in the encodings in table 7.5. Therefore, `4logenc` and `logenc` do not provide an improvement in the maximum affine dimension.

7.4 Concatenated encodings

In some cases, the encoding is allowed to use a long bit string to encode relatively small numbers. In this case, the encodings presented so far struggle to use the whole encoding space. To make optimal use of all bits in our binary string, we could opt to combine multiple encodings and put them together. For instance, combining two multiplicative encodings with parameter c, d and c', d' can be done in the following manner:

$$\texttt{mulenc}^{d}_{c}(i) \mid \texttt{mulenc}^{d'}_{c'}(i)$$

We define `conenc` as an encoding allowing us to combine multiple encodings in encoding 3.

In most cases applying a `conenc` encoding at least doubles the required string length. In some cases, this is not feasible. In a way to reduce the required string length, one should observe, that we are only using injective encodings – until now. To make `conenc` injective, it is only required that one of the encodings used is injective. To this end, we define the following non-injective encoding, based on the $\texttt{mulenc}_c$ encoding in encoding 4. Note that $\overline{\texttt{mulenc}}^{d}_{c}(i) = \texttt{mulenc}^{d}_{c}(i)$ if $i < \lceil \frac{2^d}{c} \rceil$.

Encoding 3 $\texttt{conenc}_{\texttt{enc}_1,\texttt{enc}_2,\ldots,\texttt{enc}_n}$

Encode(i):
  return $\texttt{enc}_1$.**Encode**(i) | $\texttt{enc}_2$.**Encode**(i) | ... | $\texttt{enc}_n$.**Encode**(i)

IncrementState(i):
  return ($\texttt{enc}_1$.**IncrementState**(i), $\texttt{enc}_2$.**IncrementState**(i), ..., $\texttt{enc}_n$.**IncrementState**(i))

Increment($i, (u, v_1, v_2, \ldots, v_{n-1})$):
  for $j \leftarrow 1, \ldots, n$ **do**
    $v_j, \leftarrow \texttt{enc}_j$.**Increment**($i, v_j$)
    $u_j, w_j \leftarrow v_j$
  return $(u_1 \mid u_2 \mid \ldots \mid u_n), v_1, v_2, \ldots, v_n$

Encoding 4 $\overline{\texttt{mulenc}}_c^d$

Encode(i):
  return $\overline{\texttt{i2bsp}}(c \cdot i, d)$

IncrementState(i):
  return $\overline{\texttt{i2bsp}}(c \cdot i, d)$

Increment(i, v):
  return $\overline{\texttt{i2bsp}}(\texttt{bs2ip}(v) + c, d)$

7.4.1 Mathematical properties

As seen in lemma 3, it is possible to construct a larger affine subspace from a smaller affine subspace of $\texttt{mulenc}_c$. Unfortunately, the construction of a new encoding by concatenation does not prevent an similar version of lemma 3 applying to the new encoding.

First we need to prove some lemmas, to make it possible to prove a similar property as lemma 3.

Lemma 7. *Let* $f = \texttt{conenc}_{\texttt{enc}_1,\ldots,\texttt{enc}_n}$
with

$$\texttt{enc}_m = \overline{\texttt{mulenc}}_{c_m}^{d_m} \qquad 1 \leq m \leq n$$

for some c_m *and* $d_m \in \mathbb{N}$. *Let* $a, b, d \in \mathbb{N}$ *fulfilling* $b \cdot \max(c_1, \ldots, c_n) < 2^d$. *Then:*

$$f(a \cdot 2^d + b) = f(a \cdot 2^d) + f(b).$$

Proof. Let $f, \texttt{enc}_1, \ldots \texttt{enc}_n, a, b, d$ be given as stated in the lemma. Take an arbitrary m, $1 \leq m \leq n$. We are going to look at the bits produced by the mth encoding. Note that this is a fixed position/substring of the output of f. Projecting:

$$f(a \cdot 2^d + b)$$

on the mth encoding gives:

$$\overline{\texttt{mulenc}}_{c_m}^{d_m}(a \cdot 2^d + b).$$

We define the bit strings t, s such that:

$$(t_1, \ldots, t_{d_m}) = \overline{\texttt{mulenc}}_{c_m}^{d_m}(a),$$
$$(s_1, \ldots, s_{d_m}) = \overline{\texttt{mulenc}}_{c_m}^{d_m}(b).$$

Suppose $d_m > d$, then:

$$\begin{aligned}
\overline{\texttt{mulenc}}_{c_m}^{d_m}(a \cdot 2^d + b) &= \overline{\texttt{i2bsp}}(c_m \cdot a \cdot 2^d + c_m \cdot b, d_m) \\
&= \overline{\texttt{i2bsp}}(\texttt{bs2ip}(\overline{\texttt{i2bsp}}(c_m \cdot a \cdot 2^d, d_m)) + \texttt{bs2ip}(\overline{\texttt{i2bsp}}(c_m \cdot b, d_m)), d_m) \\
&= \overline{\texttt{i2bsp}}(\texttt{bs2ip}((t_{d_m-d}, \ldots, t_1, \overbrace{\mathbb{0}, \ldots, \mathbb{0}}^{d \text{ times}}))) + \texttt{bs2ip}((s_{d_m}, \ldots, s_0)), d_m) \\
&= \overline{\texttt{i2bsp}}(\texttt{bs2ip}((t_{d_m-d}, \ldots, t_1, \mathbb{0}, \ldots, \mathbb{0}) + \texttt{bs2ip}((\overbrace{\mathbb{0}, \ldots, \mathbb{0}}^{d_m - d \text{ times}}, s_d, \ldots, s_1)), d_m) \quad (c_m \cdot b < 2^d) \\
&= (t_{d_m-d}, \ldots, t_0, \mathbb{0}, \ldots, \mathbb{0}) + (\mathbb{0}, \ldots, \mathbb{0}, s_d, \ldots, s_1)) \\
&= \overline{\texttt{mulenc}}_{c_m}^{d_m}(a \cdot 2^d) + \overline{\texttt{mulenc}}_{c_m}^{d_m}(b).
\end{aligned}$$

If $d_m \leq d$, then we have $\overline{\texttt{mulenc}}_{c_m}^{d_m}(a \cdot 2^d) = (\mathbb{0}, \ldots, \mathbb{0})$ and so

$$\overline{\texttt{mulenc}}_{c_m}^{d_m}(a \cdot 2^d + b) = \overline{\texttt{mulenc}}_{c_m}^{d_m}(b) = \overline{\texttt{mulenc}}_{c_m}^{d_m}(a \cdot 2^d) + \overline{\texttt{mulenc}}_{c_m}^{d_m}(b).$$

In both cases, if we combine all $\texttt{enc}_m$ together into f, we get:

$$f(a \cdot 2^d + b) = f(a \cdot 2^d) + f(b).$$

□

Lemma 8. *Let* $f = \texttt{conenc}_{\texttt{enc}_1, \ldots, \texttt{enc}_n}$
with

$$enc_m = \overline{\texttt{mulenc}}_{c_m}^{d_m} \qquad 1 \leq m \leq n.$$

If

$$f(a) + f(b) + f(o) = f(e)$$

for some $a, b, o, e \in \mathbb{N}$*, then for any* $d \geq 0, d \in \mathbb{N}$*:*

$$f(a \cdot 2^d) + f(b \cdot 2^d) + f(o \cdot 2^d) = f(e \cdot 2^d).$$

Proof. Let f and the corresponding $\mathtt{enc}_1, \ldots, \mathtt{enc}_n$ be given. Suppose $a, b, o, e \in \mathbb{N}$ are given such that

$$f(a) + f(b) + f(o) = f(e).$$

Take an arbitrary m , $1 \leq m \leq n$. We are going to look at the bits produced by the mth encoding. Note that this is a fixed position/substring of the output of f. Projecting gives:

$$\mathtt{enc}_m(a) + \mathtt{enc}_m(b) + \mathtt{enc}_m(o) = \mathtt{enc}_m(e)$$

Now $\mathtt{enc}_m$ is of the form:

$$\overline{\mathtt{mulenc}}_{c_m}^{d_m}$$

for some $c_m, d_m \in \mathbb{N}$. This gives:

$$\overline{\mathtt{i2bsp}}(c_m \cdot a, d_m) + \overline{\mathtt{i2bsp}}(c_m \cdot b, d_m) + \overline{\mathtt{i2bsp}}(c_m \cdot o, d_m) = \overline{\mathtt{i2bsp}}(c_m \cdot e, d_m).$$

But if we look at the definition of $\overline{\mathtt{i2bsp}}(i, d_m)$, then we conclude that the following property holds: if $\overline{\mathtt{i2bsp}}(i, d_m) = (s_d, \ldots, s_1)$ then $\overline{\mathtt{i2bsp}}(2 \cdot i, d_m) = (s_{d-1}, \ldots, s_1, \mathbb{0})$. Now let an $n \in \mathbb{N}$ be given. If

$$\begin{aligned}
&\overline{\mathtt{i2bsp}}(c_m \cdot a, d_m) = (s_d, \ldots, s_1),\\
&\overline{\mathtt{i2bsp}}(c_m \cdot b, d_m) = (t_d \ldots, t_1),\\
&\overline{\mathtt{i2bsp}}(c_m \cdot o, d_m) = (u_d, \ldots, u_1) \text{ and}\\
&\overline{\mathtt{i2bsp}}(c_m \cdot e, d_m) = (v_d, \ldots, v_1)
\end{aligned}$$

then:

$$\begin{aligned}
\overline{\mathtt{i2bsp}}&(c_m \cdot a \cdot 2^d, d_m) + \overline{\mathtt{i2bsp}}(c_m \cdot b \cdot 2^d, d_m) + \overline{\mathtt{i2bsp}}(c_m \cdot o \cdot 2^d, d_m)\\
&= (s_{d_m - d}, \ldots, s_1, \mathbb{0}, \ldots, \mathbb{0}) + (t_{d_m - d}, \ldots, t_1, \mathbb{0}, \ldots, \mathbb{0}) + (u_{d_m - d}, \ldots, u_1, \mathbb{0}, \ldots, \mathbb{0})\\
&= (v_{d_m - d}, \ldots, v_1, \mathbb{0}, \ldots, \mathbb{0})\\
&= \overline{\mathtt{i2bsp}}(c_m \cdot e \cdot 2^d, d_m).
\end{aligned}$$

Thus

$$\mathtt{enc}_m(a \cdot 2^d) + \mathtt{enc}_m(b \cdot 2^d) + \mathtt{enc}_m(o \cdot 2^d) = \mathtt{enc}_m(e \cdot 2^d)$$

and so:

$$f(a \cdot 2^d) + f(b \cdot 2^d) + f(o \cdot 2^d) = f(e \cdot 2^d).$$

□

Using these lemmas, we can prove a similar property as lemma 3 for `conenc` when using as

sub-encoding the multiplicative encoding.

Lemma 9. *Let* $f = \texttt{conenc}_{\texttt{enc}_1,\ldots,\texttt{enc}_n}$ *with*

$$\texttt{enc}_m = \overline{\texttt{mulenc}}_{c_m}^{d_m} \qquad 1 \leq m \leq n.$$

Let B *be a set* $B \subseteq \mathbb{N}$*, such that*

$$A = \{f(i) : i \in B\}$$

is an affine subspace. If $\max(B) \cdot \max(c_1, \ldots, c_n) < 2^d$ *then*

$$\{f(i \cdot 2^d + j) : i, j \in B\}$$

is an affine subspace of f*.*

Proof. Let $f, \texttt{enc}_1, \ldots, \texttt{enc}_n, A$ and B be given as stated in the lemma. We write:

$$B^* = \{i \cdot 2^d + j : i, j \in B\}$$

and

$$\begin{aligned} A^* &= \{f(e) : e \in B^*\} \\ &= \{f(i \cdot 2^d + j) : i, j \in B\}. \end{aligned}$$

Obviously A^* is a subset of $\text{im}(f)$. What is left to prove is that A^* is an affine space. Let $o_H \cdot 2^d + o_L$ be the value in B^* for $o_H, o_L \in B$ such that

$$f(o_H \cdot 2^d + o_L)$$

is the origin of A^*. Take an arbitrary $u, v \in A^*$ and let $a_H, a_L, b_H, b_L \in B$ be such that $u = f(a_H \cdot 2^d + a_L)$ and $v = f(b_H \cdot 2^d + b_L)$. Because A is affine, a pair $e_H, e_L \in B$ must exist such that

$$f(a_H) + f(b_H) + f(o_H) = f(e_H)$$

and

$$f(a_L) + f(b_L) + f(o_L) = f(e_L).$$

Encoding ($\texttt{conenc}_{\texttt{enc}_1,\texttt{enc}_2}$)		Maximum affine dimension (afdim)					
$\texttt{enc}_1$	$\texttt{enc}_2$	1	2	3	4	5	6
$\texttt{mulenc}_3^{(32)}$	$\texttt{mulenc}_5^{(32)}$	2	10	52	248	832	3968
$\texttt{mulenc}_3^{(32)}$	$\texttt{mulenc}_7^{(32)}$	2	5	34	136	1088	>4096
$\texttt{mulenc}_5^{(32)}$	$\texttt{mulenc}_7^{(32)}$	2	10	76	608	3200	>4096
$\texttt{mulenc}_5^{(32)}$	$\texttt{mulenc}_9^{(32)}$	2	4	36	100	1152	3200
$\texttt{mulenc}_7^{(32)}$	$\texttt{mulenc}_9^{(32)}$	2	7	64	200	464	4096

Table 7.8 The minimal number of elements n required such that codewords($\texttt{conenc}_{\texttt{enc}_1,\texttt{enc}_2}, n$) has a certain maximum dimension of its affine subspaces. The n values can be up to $2^{\lfloor \log_2 n \rfloor - 4}$ too big. The sub-encodings $\texttt{enc}_1$ and $\texttt{enc}_2$ are chosen from the encodings used in table 7.3.

Note that $\max(c_1, \dots c_n) \cdot e_L < 2^d$. Then $e_H \cdot 2^d + e_L \in B^*$ and thus:

$$\begin{aligned}
u \oplus_{A*} v &= f(a_H \cdot 2^d + a_L) + f(b_H \cdot 2^d + b_L) + f(o_H \cdot 2^d + o_L) \\
&= f(a_H \cdot 2^d) + f(a_L) + f(b_H \cdot 2^d) + f(b_L) + f(o_H \cdot 2^d) + f(o_L) && \text{(Lemma 7)} \\
&= (f(a_H \cdot 2^d) + f(b_H \cdot 2^d) + f(o_H \cdot 2^d)) + (f(a_L) + f(b_L) + f(o_L)) \\
&= f(e_H \cdot 2^d) + f(e_L) && \text{(Lemma 8)} \\
&= f(e_H \cdot 2^d + e_L) \in A^*
\end{aligned}$$

Because u, v can be any element in A^* we have shown that A^* is an affine subspace of $\text{im}(f)$. □

7.4.2 Experimental analysis

To verify if the encoding construction provided by `conenc` yields useful results, we have combined several encodings using this construction. At first we started with combinations of `mulenc` encodings with small c values. This can be seen in table 7.8. Then we computed the maximum affine dimension of $\texttt{logenc}_{\texttt{mulenc}}$ with the same c values. This can be seen in table 7.9. Lastly, we combined `mulenc` encodings with a counter that counts from 0 to either 15 or 255 and then starts again at 0. These results can be seen in table 7.10.

If we compare table 7.8 to table 7.3 and table 7.9 to table 7.6, then we can see that the `conenc` reduces the affine dimension by 1 or 2. However, the encoding generated by selecting the ideal c values from 9000 to 12 000 (table 7.5) generated code words with a maximum affine dimension comparable to these `conenc` construction. One must also note that `conenc` increases the numbers of bits required to encode some values excessively. While it does provide lower maximum affine dimensions, it might be possible to find better encodings that can encode more values.

Encoding ($\texttt{conenc}_{\texttt{enc}_1,\texttt{enc}_2}$)		Maximum affine dimension (afdim)					
$\texttt{enc}_1$	$\texttt{enc}_2$	1	2	3	4	5	6
$\texttt{logenc}_{\texttt{mulenc}_3^{(32)}}$	$\texttt{logenc}_{\texttt{mulenc}_5^{(32)}}$	2	10	176	1728	3200	>4096
$\texttt{logenc}_{\texttt{mulenc}_3^{(32)}}$	$\texttt{logenc}_{\texttt{mulenc}_7^{(32)}}$	2	7	168	2304	>4096	>4096
$\texttt{logenc}_{\texttt{mulenc}_5^{(32)}}$	$\texttt{logenc}_{\texttt{mulenc}_7^{(32)}}$	2	10	208	3072	>4096	>4096
$\texttt{logenc}_{\texttt{mulenc}_5^{(32)}}$	$\texttt{logenc}_{\texttt{mulenc}_9^{(32)}}$	2	34	168	3072	>4096	>4096
$\texttt{logenc}_{\texttt{mulenc}_7^{(32)}}$	$\texttt{logenc}_{\texttt{mulenc}_9^{(32)}}$	2	28	112	3968	>4096	>4096

Table 7.9 The minimal number of elements n required such that codewords($\texttt{conenc}_{\texttt{enc}_1,\texttt{enc}_2}, n$) has a certain maximum dimension of its affine subspaces. The n values can be up to $2^{\lfloor \log_2 n \rfloor - 4}$ too big. The sub-encodings $\texttt{enc}_1$ and $\texttt{enc}_2$ are chosen from the encodings used in table 7.6.

Encoding ($\texttt{conenc}_{\texttt{enc}_1,\texttt{enc}_2}$)		Maximum affine dimension (afdim)					
$\texttt{enc}_1$	$\texttt{enc}_2$	1	2	3	4	5	6
$\texttt{mulenc}_3^{(60)}$	$\overline{\texttt{mulenc}}_1^{(4)}$	2	6	18	72	288	>640
$\texttt{mulenc}_3^{(56)}$	$\overline{\texttt{mulenc}}_1^{(8)}$	2	6	44	304	832	>3584
$\texttt{mulenc}_5^{(60)}$	$\overline{\texttt{mulenc}}_1^{(4)}$	2	4	20	50	304	>640
$\texttt{mulenc}_5^{(56)}$	$\overline{\texttt{mulenc}}_1^{(8)}$	2	4	48	208	832	3200
$\texttt{mulenc}_7^{(60)}$	$\overline{\texttt{mulenc}}_1^{(4)}$	2	7	34	68	400	>640
$\texttt{mulenc}_7^{(56)}$	$\overline{\texttt{mulenc}}_1^{(8)}$	2	7	50	448	1088	>3584
$\texttt{mulenc}_9^{(60)}$	$\overline{\texttt{mulenc}}_1^{(4)}$	2	4	8	72	200	464
$\texttt{mulenc}_9^{(56)}$	$\overline{\texttt{mulenc}}_1^{(8)}$	2	4	8	144	400	928

Table 7.10 The minimal number of elements n required such that codewords($\texttt{conenc}_{\texttt{enc}_1,\texttt{enc}_2}, n$) has a certain maximum dimension of its affine subspaces. The n values can be up to $2^{\lfloor \log_2 n \rfloor - 4}$ too big. The sub-encodings $\texttt{enc}_1$ are chosen from the encodings used in table 7.3. $\texttt{enc}_2$ is a simple 4 or 8 bits counter that overflows when the maximum value has been reached (i.e. $\overline{\texttt{mulenc}}_1^{(4)}$ or $\overline{\texttt{mulenc}}_1^{(8)}$)

7.5 Rotation-based encodings

The next encoding that will be introduced is the smeer encoding (named after the Dutch word for 'spread' in sandwich spread). The previous encodings were trivially injective, because all encodings were strictly monotonic when converted to an integer. The encodings that will be introduced now will not have this property. Injectivity will be proven for the more practical algorithms, the other algorithms merely serve as an illustration.

7.5.1 Exponential encoding

The idea of the smeer encoding comes from analysing the encoding, that can encode more than one value, with the lowest possible dimension: dimension 1. An example of such an encoding is the exponential-encoding $\texttt{expenc}_d$ as described in encoding 5.

Encoding 5 $\texttt{expenc}_d$

Encode(i):
  return $\texttt{i2bsp}(2^{i+1}, d)$

IncrementState(i):
  return $\texttt{i2bsp}(2^{i+1}, d)$

Increment(i, v):
  return $\texttt{i2bsp}(\texttt{bs2ip}(v) \cdot 2, d)$

In other words, the $\texttt{expenc}_d$ encodes the numbers $0, 1, 2, \ldots, d-1$ to $(\mathbb{0}, \ldots, \mathbb{0}, \mathbb{1})$, $(\mathbb{0}, \ldots, \mathbb{0}, \mathbb{1}, \mathbb{0})$, $(\mathbb{0}, \ldots, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{0})$, $(\mathbb{1}, \mathbb{0}, \ldots, \mathbb{0})$. While this is the best encoding to make sure no affine subspace with more than 2 elements will exist, it is horribly inefficient in terms of length and plainly not practical. The next step would be to extend this encoding, while keeping the maximal affine dimension low.

A logical method would be to keep the first d values the same as the first d (and only) values of $\texttt{expenc}_d$. The idea for the remaining part of the encoding, is to change the value that is being rotated over all bits. Logically, the next value after $\mathbb{1}$ would be $(\mathbb{1}, \mathbb{0})$, but this would result in the same encodings as the first d encoding. Thus $\mathbb{1}, \mathbb{1}$ is chosen. So the next values are chosen to be the values $(\mathbb{0}, \ldots, \mathbb{0}, \mathbb{1}, \mathbb{1})$, $(\mathbb{0}, \ldots, \mathbb{0}, \mathbb{1}, \mathbb{1}, \mathbb{0})$, $(\mathbb{0}, \ldots, \mathbb{0}, \mathbb{1}, \mathbb{1}, \mathbb{0}, \mathbb{0})$, $(\mathbb{1}, \mathbb{1}, \mathbb{0}, \ldots, \mathbb{0})$. and then the next values can then be $(\mathbb{0}, \ldots, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{1})$, $(\mathbb{0}, \ldots, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{0})$, $(\mathbb{0}, \ldots, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{0})$, $(\mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{0}, \ldots, \mathbb{0})$. The next values can then be found by shifting $(\mathbb{0}, \ldots, \mathbb{0}, \mathbb{1}, \mathbb{1}, \mathbb{1})$, thus we numerically increment the base number while skipping even numbers. If we continue this method, we get the shifted odd numbers encoding $\texttt{shiftedoddenc}^d$ (encoding 6).

Injectivity of the shifted odd numbers encoding, can be proven by interpreting the encoding as a number n and factorizing n in primes. Each factorization will be different: either the multiplicity of 2 will differ, or the odd number achieved by dividing n by 2 until it is an odd number is different.

In general, we can note several things: calculating the encoding of a specific number is very inefficient in this primitive implementation. Whereas all implementations of encodings seen so far were $\mathcal{O}(1)$, this implementation is $\mathcal{O}(n)$. This can probably be improved by investigating the properties of this encoding, but this is outside of the scope of this thesis.

Encoding 6 $\texttt{shiftedoddenc}^d$

```
Encode(i):
    v, w ← IncrementState(i)
    return v

IncrementState(i):
    v ← i2bsp(1, d)
    w ← i2bsp(1, d)
    for j ← 1, ..., i − 1 do
        v, w ← Increment(j, w, v)
    return v, w

Increment(i, v, j):
    if bs2ip(v) > 2^(d−1) then
        return i2bsp(j + 2, d), j + 2
    else
        return i2bsp(bs2ip(v) · 2, d), j
```

The second important consideration, is that this encoding will eventually reach every possible value in $\mathbb{F}_2^d$. It can be said that this encoding is equivalent to the $\texttt{mulenc}_1^d$ encoding, but the order is completely different and if the domain is limited, we actually get a decent encoding. These statements can be verified in the experimental analysis.

Experimental analysis

Table 7.11 gives the minimal number of elements to reach a certain maximum affine dimension. The table also includes the `Ipb` value, indicating how much information each bit gives about the encoded value. We see that the `Ipb` increases sublinearly with the maximum affine dimension. Furthermore, when the bit length is increased, the `Ipb` at a certain maximum affine dimension decreases. It can be seen that the encodings, especially $\texttt{shiftedoddenc}^{(64)}$, performs pretty well relative to the encodings seen so far when n is small. However, when n reaches a certain kind of boundary (for $\texttt{shiftedoddenc}^{(16)}$ around $n = 100$) the affine dimension rapidly increases. This is possibly be due to the fact that `shiftedoddenc` is simply a reordering of all the codewords generated by $\texttt{mulenc}_1$, which is the worst encoding possible. This means that when the encoding reaches a certain n, the combination of bit strings that make the $\texttt{mulenc}_1$ perform badly will all be used as a code word.

7.5.2 Shifted encoding

While the exponential encoding seems to be a decent encoding, we have seen better encodings. However, it is possible to generalize the developed encoding: instead of shifting odd values, we can shift the value of an existing encoding. This results in the `shiftedenc` encoding as described in encoding 7.

To make sure our version is injective, we assume that the existing encoding used does not contain both a value x with the last bit set to $\mathbb{0}$ and a value y equal to x but with the last bit set to $\mathbb{1}$. The encoding will force the last bit set to $\mathbb{1}$ and, in this manner, avoid an attack by using the fact that

Minimal number of elements n and information per bit `Ipb` for various afdim values.

	Maximum affine dimension (afdim)											
	1		2		3		4		5		6	
Encoding	n	`Ipb`	n	`Ipb`	n	`Ipb`	n	`Ipb`	n	`Ipb`	n	`Ipb`
$\texttt{shiftedoddenc}^{(16)}$	2	0.06	18	0.26	48	0.35	100	0.42	200	0.48	384	0.54
$\texttt{shiftedoddenc}^{(32)}$	2	0.03	34	0.16	96	0.21	216	0.24	448	0.28	896	0.31
$\texttt{shiftedoddenc}^{(64)}$	2	0.2	68	0.10	192	0.12	448	0.14	928	0.15	2048†	0.17

Table 7.11 The minimal number of elements n required such that codewords($\texttt{shiftedoddenc}^{(d)}, n$) for different d values (code word length) has a certain maximum dimension of its affine subspaces. The n values can be up to $2^{\lfloor \log_2 n \rfloor - 4}$ too big. The `Ipb` value gives the information per bit for the code word space, which is in this case equal to $\frac{\log_2(|C|)}{d}$. The d values chosen are 16, 32 and 64.
† The actual value is somewhere between 1728 and 2048.

Encoding 7 $\texttt{shiftedenc}^{d}_{\texttt{subenc}}$

```
Encode(i):
    v, w ← IncrementState(i)
    return v

IncrementState(i):
    v, subinc_state ← subenc.IncrementState(0)
    v_0 ← 𝟙
    w ← i2bsp(1, d)
    for j ← 1, ..., i − 1 do
        v, w ← Increment(j, w, v)
    return v, w

Increment(i, v, sub_v, j, subinc_state):
    if bs2ip(v) > 2^(d−1) then
        sub_v, subinc_state ← subenc.Increment(j + 1, sub_v, subinc_state)
        ▷ Set the least significant bit to 𝟙
        (sub_v)_0 ← 𝟙
        return i2bsp(sub_v, d), sub_v, j + 1, subinc_state
    else
        return i2bsp(bs2ip(v) · 2, d), sub_v, j, subinc_state
```

shifting a value is equal to doubling the input value in case of `mulenc`. In fact, without forcing the last bit set to 𝟙 the encoding presented is not injective.

Injectivity can again be proven in a similar way to shifted odd numbers encoding: the values achieved by encoding with `subenc` will always be forced to an odd number, and therefore, the number of shifts can be found by viewing the prime factorization.

Calculating the encoding of a specific number is still inefficient in this implementation. Moreover, finding a better implementation for this generalized encoding is almost impossible: one does not know the length of the encoded values before they are calculated. Moreover, these lengths do not even have to be monotonically increasing. This makes it, as far as we know, impossible to find a constant-time encoding, instead of this $\mathcal{O}(n)$ implementation.

Nonetheless, this encoding, gives some promising results as can be seen in the experimental analysis.

Experimental analyis

In table 7.12 some experimental results are given for `shiftedenc`. We used two different sub-encodings, $\texttt{mulenc}_{37}$ and $\texttt{4logenc}_{\texttt{mulenc}_{37}}$. Both encodings use the same c value, namely 37. This c value has been chosen because we have seen that $\texttt{mulenc}_{37}$ performs reasonable well, while 37 is also quite small, allowing us to observe the performance of `shiftedenc` for sub-encodings that perform well.

We can directly see, especially when the sub-encoding is $\texttt{mulenc}_{37}$ that the n values for when the maximum affine dimension is 2 are nearly the same as when `shiftoddenc` was used. As these n values for dimension 2 were already much higher than for the other encodings we have seen, this is a good sign. It shows that the 'optimal' behavior of `shiftedoddenc` for the first set of code words is kept when we use different sub-encodings.

Moreover, when we use the sub-encoding $\texttt{4logenc}_{\texttt{mulenc}_{37}}$, `shiftedenc` performs much better than when we use the sub-encoding $\texttt{mulenc}_{37}$, showing some synergy between `4logenc` and `shiftedenc`. In this case we see that the n values for when the maximum affine dimension is 2 are much higher then when `shiftedoddenc` is used.

7.5.3 Smeer encoding

As we have seen, `shiftedenc` gives promising results, but because the **Encode** operation will most likely always be inefficient, it is not a feasible encoding. Therefore, it is a good idea to modify the encoding in such a way that the **Encode** operation can be implemented in an efficient manner. One should note that the **Encode** operation is inefficient because per different `subenc`.**Encode** value, the number of shifts changes. To solve this problem, one wants to introduce a fixed number of shifts for every `subenc`.**Encode** value. Multiple options exist to construct such an encoding.

Firstly, one could limit the output length of the sub-encoding, such that we know a upper bound of the output. This upper bound can be used to shift the values over the whole range. This has multiple downsides: it is very costly, a lot of entropy is given up to make this work. Furthermore, for small values relative to the upper bound, the value is not shifted over the whole range.

Another, more constructive option, is to replace the shift operation used in the `shiftedenc` with

Minimal number of elements n and information per bit `Ipb` for various afdim values.

Encoding	Maximum affine dimension (afdim) 1		2		3		4		5	
	n	`Ipb`	n	`Ipb`	n	`Ipb`	n	`Ipb`	n	`Ipb`
$\texttt{shiftedenc}^{(16)}_{\texttt{mulenc}_{37}}$	2	0.06	17	0.26	116	0.43	640	0.58	2048†	0.69
$\texttt{shiftedenc}^{(32)}_{\texttt{mulenc}_{37}}$	2	0.03	34	0.16	320	0.26	2304	0.35	>2944	≥0.36
$\texttt{shiftedenc}^{(64)}_{\texttt{mulenc}_{37}}$	2	0.02	68	0.10	704	0.15	>2944	≥0.18	>2944	≥0.18
$\texttt{shiftedenc}^{(16)}_{\texttt{4logmulenc}_{37}}$	2	0.06	52	0.36	—‡	—	—‡	—	—‡	—
$\texttt{shiftedenc}^{(32)}_{\texttt{4logmulenc}_{37}}$	2	0.03	152	0.23	2048†	0.34	>2944	≥0.36	>2944	≥0.36
$\texttt{shiftedenc}^{(64)}_{\texttt{4logmulenc}_{37}}$	2	0.02	352	0.13	>4096	≥0.19	>4096	≥0.19	>4096	≥0.19

Table 7.12 The minimal number of elements n required such that codewords($\texttt{shiftedenc}^{(d)}, n$) for different d values (code word length) and encodings has a certain maximum dimension of its affine subspaces. The n values can be up to $2^{\lfloor \log_2 n \rfloor - 4}$ too big. The `Ipb` value gives the information per bit for the code word space, which is in this case equal to $\frac{\log_2(|C|)}{d}$. The d values chosen are 16, 32 and 64. The sub-encodings used are `mulenc` and $\texttt{4logmulenc}_c$ which is an abbreviation for $\texttt{4logenc}_{\texttt{mulenc}_c}$.
† The actual value is somewhere between 1728 and 2048. ‡ The $\texttt{shiftedenc}^{(16)}_{\texttt{4logmulenc}_{37}}$ encoding can only encode 164 elements, at which point the maximum affine dimension is 2 and the `Ipb` is 0.46.

a rotation (left-rotation in our case). If done correctly, this does not reduce the entropy of the sub-encoding and the value can be shifted over the whole range. One problem does exist, simply replacing shift with a rotation in `shiftedenc` does not result in a injective encoding: the sub-encoding could output the value $(\mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{0}, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{1})$ and output the value $(\mathbb{0}, \mathbb{0}, \mathbb{0}, \mathbb{1}, \mathbb{0}, \mathbb{1}, \mathbb{1}, \mathbb{0}, \mathbb{1})$, where the first value is the second value rotated to the right for 3 times. To solve this, we add a counter to the encoding that indicates the number of places the value has been rotated.

This solution can be seen in encoding 8, which we call `smeerenc`. One must note that the sub-encoding used, `subenc` must have valid dimensions to fit the encoding. The dimension of the sub-encoding is either given or determined in the preparation phase. Often the dimension of `subenc` is not written down for convenience of notation.

It can be seen that the entropy of `smeerenc` of dimension d applied to a multiplicative encoding of dimension d' is almost the same as the entropy of the same multiplicative encoding on dimension d. This is because `smeerenc` generates almost $2^{d-d'}$ different rotations for each element, and increasing the dimension of a multiplicative encoding from d to d' generates approximately $2^{d-d'}$ times more elements. Thus in the optimal case `smeerenc` does not lower the entropy of the sub-encoding `mulenc`.

The counter has been added to the encoding to make sure the encoding is injective. As we have seen in section 7.4, concatenated encodings perform very poorly. This added counter can be seen as some sort of concatenated encoding. Therefore, removing this counter and thus allowing more rotations to occur could very well improve the performance of the encoding. Unfortunately, if the sub-encoding may produce full length outputs, injectivity of the encoding does not always hold. However, if we limit the sub-encoding to half the length of our encoding, the encoding is injective. This is proven in lemma 12. This allows us to introduce `uncountedsmeerenc` in encoding 8.

Encoding 8 $\texttt{smeerenc}^{(d,\texttt{maxrot})}_{\texttt{subenc}}$, $\texttt{smeerenc}^{d}_{\texttt{subenc}}$

Preparation:
  if maxrot is not given **then**
    $\texttt{maxrot} \leftarrow d - \lceil \log_2(d-1) \rceil$
  $\texttt{count_size} \leftarrow \lceil \log_2(\texttt{maxrot} - 1) \rceil$
  $\texttt{internal_size} \leftarrow d - \texttt{count_size}$
  Assert Dimension of subenc is internal_size

Encode(i):
  $j \leftarrow i \mod \texttt{maxrot}$
  $k \leftarrow \lfloor \frac{i}{\texttt{maxrot}} \rfloor$
  $v \leftarrow$ subenc.**Encode**(k)
  $v_0 \leftarrow \mathbb{1}$
  return $\texttt{i2bsp}(j, \texttt{count_size}) \mid \texttt{rol}(v, j)$

IncrementState(i):
  $w, \texttt{data} \leftarrow$ subenc.**IncrementState**(i)
  $j \leftarrow i \mod \texttt{maxrot}$
  $k \leftarrow \lfloor \frac{i}{\texttt{maxrot}} \rfloor$
  $v \leftarrow w$
  $v_0 \leftarrow \mathbb{1}$
  return $(\texttt{i2bsp}(j, \texttt{count_size}) \mid \texttt{rol}(v, j)), w, \texttt{data}$

Increment($i, v, w, \texttt{data}$):
  $j \leftarrow i \mod \texttt{maxrot}$
  $k \leftarrow \lfloor \frac{i}{\texttt{maxrot}} \rfloor$
  if $j = 0$ **then**
    $w, data \leftarrow$ subenc.**Increment**($k, w, \texttt{data}$)
    $v \leftarrow w$
    $v_0 \leftarrow \mathbb{1}$
    return $v, w, \texttt{data}$
  return $(\texttt{i2bsp}(j, \texttt{count_size}) \mid \texttt{rol}(v, 1)), w, \texttt{data}$

Encoding 9 $\texttt{uncountedsmeerenc}_{\texttt{subenc}}^{(d,\texttt{maxrot})}$, $\texttt{uncountedsmeerenc}_{\texttt{subenc}}^{(d)}$

Preparation:
  Assert Dimension of `subenc` is $\frac{d}{2}$
  if `maxrot` is not given **then**
    $\texttt{maxrot} \leftarrow d - \lceil \log_2(d-1) \rceil$
  $\texttt{count_size} \leftarrow \lceil \log_2(\texttt{maxrot} - 1) \rceil$
  $\texttt{internal_size} \leftarrow d - \texttt{count_size}$

Encode(i):
  $j \leftarrow i \mod \texttt{maxrot}$
  $k \leftarrow \lfloor \frac{i}{\texttt{maxrot}} \rfloor$
  $v \leftarrow \texttt{i2bsp}(0, \frac{d}{2}) \mid \texttt{subenc}.\textbf{Encode}(k)$
  $v_0 \leftarrow \mathbb{1}$
  return $\texttt{rol}(v, j)$

IncrementState(i):
  $w, \texttt{data} \leftarrow \texttt{subenc}.\textbf{IncrementState}(i)$
  $j \leftarrow i \mod \texttt{maxrot}$
  $k \leftarrow \lfloor \frac{i}{\texttt{maxrot}} \text{i}$
  $v \leftarrow \texttt{i2bsp}(0, \frac{d}{2}) \mid w$
  $v_0 \leftarrow \mathbb{1}$
  return $\texttt{rol}(v, j), w, \texttt{data}$

Increment$(i, w, \texttt{data})$:
  $j \leftarrow i \mod \texttt{maxrot}$
  $k \leftarrow \lfloor \frac{i}{\texttt{maxrot}}$
  if $j = 0$ **then**
    $w, \texttt{data} \leftarrow \texttt{subenc}.\textbf{Increment}(k, w, \texttt{data})$
    $v \leftarrow \texttt{i2bsp}(0, \frac{d}{2}) \mid w$
    $v_0 \leftarrow \mathbb{1}$
    return $v, w, \texttt{data}$
  return $\texttt{rol}(v, 1), w, \texttt{data}$

Theoretical analysis

Lemma 10. *Let* $d \in \mathbb{N}$ *and* $\texttt{maxrot} \leq d - \lceil \log_2(d-1) \rceil$ *be given. Let* subenc *be an arbitrary encoding of length* $d' = d - \lceil \log_2(\texttt{maxrot} - 1) \rceil$. *Let* D *be the domain of* subenc. *If for every* $i, j \in D, i \neq j$:

$$(\texttt{subenc}(i)_{d'}, \ldots, \texttt{subenc}(i)_2) \neq (\texttt{subenc}(i)_{d'}, \ldots, \texttt{subenc}(i)_2),$$

*(*subenc *is injective on the substring from position* 2 *to* d'*) then encoding 8 (*$\texttt{smeerenc}^{(d,\texttt{maxrot})}_{\texttt{subenc}}$*) using sub-encoding* subenc *is injective.*

Proof. Let an instantiation of the smeer encoding, $f = \texttt{smeerenc}^{(d,\texttt{maxrot})}_{\texttt{subenc}}$, be given with parameters subenc, d and maxrot. We have to prove that $f(i) = f(j) \Rightarrow i = j$. To this end, let i, j in the domain of f be given such that $f(i) = f(j)$, and let $v = f(i)$ and $w = f(i)$.

We define $e = \lceil \log_2(\texttt{maxrot} - 1) \rceil$, then define $h = \texttt{bs2ip}(v_d, \ldots, v_{d-e+1})$. This h must be the number of shifts that the original value was shifted by. Similarly, $g = \texttt{bs2ip}(w_d, \ldots, w_{d-e+1})$. Because $w = v$, we must have $h = g$.

As h is derived from $i \mod \texttt{maxrot}$ and g from $j \mod \texttt{maxrot}$ in encoding subenc, we have that $i \mod \texttt{maxrot} = j \mod \texttt{maxrot}$. Now we define:

$$q = \texttt{rol}((v_{d-e}, \ldots, v_2), -h),$$
$$p = \texttt{rol}((w_{d-e}, \ldots, v_2), -g).$$

Because $v = w$ and $h = g$, we must have $p = q$. Moreover, as h and g are the number of places that v and w were rotated respectively, q $(= p)$ will be equal to the encoding produced by subenc, without the least significant bit. And thus we have that:

$$q = (\texttt{subenc}(\lfloor \frac{i}{\texttt{maxrot}} \rfloor)_{d'}, \ldots, \texttt{subenc}(\lfloor \frac{i}{\texttt{maxrot}} \rfloor)_2)$$
$$q = (\texttt{subenc}(\lfloor \frac{j}{\texttt{maxrot}} \rfloor)_{d'}, \ldots, \texttt{subenc}(\lfloor \frac{j}{\texttt{maxrot}} \rfloor)_2)$$

But we also have that for any $k, l \in \mathbb{N}$:

$$(\texttt{subenc}(k)_{d'}, \ldots, \texttt{subenc}(k)_2) = (\texttt{subenc}(l)_{d'}, \ldots, \texttt{subenc}(l)_2) \Rightarrow k = l,$$

thus $\lfloor \frac{j}{\texttt{maxrot}} \rfloor = \lfloor \frac{i}{\texttt{maxrot}} \rfloor$, and also $i \mod \texttt{maxrot} = j \mod \texttt{maxrot}$. Combining these two equalities gives $i = j$. □

Lemma 11. *Let* $v = (\mathbb{0}, \ldots, \mathbb{0}, v_{\lfloor \frac{d}{2} \rfloor}, \ldots, v_2, \mathbb{1}) \in \mathbb{F}_2^d$. *Let* i *be given with* $i \mod d \neq 0$. *Then if* $w = \texttt{rol}(v, i)$*, then* w *is not of the form:*

$$(\mathbb{0}, \ldots, \mathbb{0}, w_{\lfloor \frac{d}{2} \rfloor}, \ldots, w_2, \mathbb{1}).$$

Proof. Let v, d, i and w be given as described in the lemma. Suppose $(i \mod d) \leq \lceil \frac{d}{2} \rceil$, then:

$$\begin{aligned} w_1 &= \texttt{rol}(v, i)_1 \\ &= v_{d+1-i} \\ &= \mathbb{0} && (i < \lceil \frac{d}{2} \rceil \Rightarrow d - i \geq \lfloor \frac{d}{2} \rfloor). \end{aligned}$$

Thus w is not of the form:

$$(\mathbb{0}, \ldots, \mathbb{0}, w_{\lfloor \frac{d}{2} \rfloor - 1}, \ldots, w_1, \mathbb{1}).$$

Suppose $(i \mod d) > \lceil \frac{d}{2} \rceil$, then:

$$\begin{aligned} w_i &= \texttt{rol}(v, i)_i \\ &= v_1 \\ &= \mathbb{1}. \end{aligned}$$

Thus w has in its first $\lceil \frac{d}{2} \rceil$ most-significant bits a $\mathbb{1}$. And thus w is not of the form:

$$(\mathbb{0}, \ldots, \mathbb{0}, w_{\lfloor \frac{d}{2} \rfloor}, \ldots, w_1, \mathbb{1}).$$

□

Lemma 12. *Let* $d \in \mathbb{N}$ *and* $\texttt{maxrot} \leq d$ *be given. Let* `subenc` *be an arbitrary encoding of length* $d' = \lfloor \frac{d}{2} \rfloor$*. Let* D *be the domain of* `subenc`*. If for every* $i, j \in D, i \neq j$*:*

$$(\texttt{subenc}(i)_{d'}, \ldots, \texttt{subenc}(i)_2) \neq (\texttt{subenc}(j)_{d'}, \ldots, \texttt{subenc}(j)_2),$$

(`subenc` *is injective on the substring from position* 1 *to* $\lfloor \frac{d}{2} \rfloor$*) then encoding 9 (*$\texttt{uncountedsmeerenc}_{\texttt{subenc}}^{(d,\texttt{maxrot})}$*) using sub-encoding* `subenc` *is injective.*

Proof. Let an instantiation of $\texttt{uncountedsmeerenc}^{(d,\texttt{maxrot})}_{\texttt{subenc}}$, be given with parameters $\texttt{subenc}$, d and $\texttt{maxrot}$. We have to prove that $f(i) = f(j) \Rightarrow i = j$. To this end, let i, j in the domain of f be given such that $f(i) = f(j)$, and let $v = f(i)$, $\hat{v} = f(j)$. By definition of the $\texttt{uncountedsmeerenc}$, there we have that for:

$$\begin{aligned} k &= \lfloor \frac{i}{\texttt{maxrot}} \rfloor, \\ \hat{k} &= \lfloor \frac{j}{\texttt{maxrot}} \rfloor, \\ l &= i \mod \texttt{maxrot}, \\ \hat{l} &= j \mod \texttt{maxrot} \end{aligned}$$

the following holds:

$$\begin{aligned} v &= \texttt{rol}(w, l), \\ \hat{v} &= \texttt{rol}(\hat{w}, \hat{l}) \end{aligned}$$

with

$$\begin{aligned} w &= \texttt{i2bsp}(0, \frac{d}{2}) \mid w', \\ \hat{w} &= \texttt{i2bsp}(0, \frac{d}{2}) \mid \hat{w}' \end{aligned}$$

and

$$\begin{aligned} w'_i &= (\texttt{subenc}(k)_{d'}, \ldots, \texttt{subenc}(k)_2, \mathbb{1}), \\ \hat{w}'_i &= (\texttt{subenc}(\hat{k})_{d'}, \ldots, \texttt{subenc}(\hat{k})_2, \mathbb{1}). \end{aligned}$$

Because:

$$\begin{aligned} w &= (\mathbb{0}, \ldots, \mathbb{0}, w_{\lfloor \frac{d}{2} \rfloor - 1}, \ldots, w_1, \mathbb{1}) \text{ and} \\ \hat{w} &= (\mathbb{0}, \ldots, \mathbb{0}, \hat{w}_{\lfloor \frac{d}{2} \rfloor - 1}, \ldots, \hat{w}_1, \mathbb{1}). \end{aligned}$$

By lemma 11, we know that $\texttt{rol}(v, m) = \texttt{rol}(w, m + l) = \texttt{rol}(\hat{w}, m + \hat{l})$ is of the form:

$$(\mathbb{0}, \ldots, \mathbb{0}, \texttt{rol}(v, m)_{\lfloor \frac{d}{2} \rfloor - 1}, \ldots, \texttt{rol}(v, m)_2, \mathbb{1})$$

if $m + l = 0 \mod d$ and $m + \hat{l} = 0 \mod d$. But w and $\hat{w}$ are both in this same form. Thus $\texttt{rol}(v, m) = w$ implies $m + l = m + \hat{l} \mod d$ and so $l = \hat{l}$, and also $w' = \hat{w}'$. Because for every $i, j \in D, i \neq j$:

$$(\texttt{subenc}(i)_{d'}, \ldots, \texttt{subenc}(i)_2) \neq (\texttt{subenc}(j)_{d'}, \ldots, subenc(j)_2),$$

we must have that $k = \hat{k}$ and so $i = j$. □

Minimal number of elements n and information per bit `Ipb` for various afdim values.

Encoding	Maximum affine dimension (afdim) 1		2		3	
	n	`Ipb`	n	`Ipb`	n	`Ipb`
$\texttt{smeerenc}^{(64)}_{\texttt{mulenc}_{37}}$	2	0.02	176	0.12	704	0.15
$\texttt{smeerenc}^{(64)}_{\texttt{mulenc}_{11\,479}}$	2	0.02	232	0.12	>3584	≥0.18
$\texttt{smeerenc}^{(64)}_{\texttt{mulenc}_{11\,591}}$	2	0.02	232	0.12	>3584	≥0.18
$\texttt{uncountedsmeerenc}^{(64)}_{\texttt{mulenc}_{37}}$	2	0.02	68	0.10	768	0.15
$\texttt{uncountedsmeerenc}^{(64)}_{\texttt{mulenc}_{11\,479}}$	2	0.02	256	0.13	2560	0.18
$\texttt{uncountedsmeerenc}^{(64)}_{\texttt{mulenc}_{11\,591}}$	2	0.02	256	0.13	3456	0.18

Table 7.13 The minimal number of elements n required such that codewords($\texttt{smeerenc}^{(d)}, n$) and codewords($\texttt{uncountedsmeerenc}^{(d)}, n$) for different encodings has a certain maximum dimension of its affine subspaces. The n values can be up to $2^{\lfloor \log_2 n \rfloor - 4}$ too big. The `Ipb` value gives the information per bit for the code word space, which is in this case equal to $\frac{\log_2(|C|)}{d}$. The sub-encodings used are $\texttt{mulenc}_c$ for $c \in \{37, 11\,479, 11\,591\}$.

Experimental analysis

To experimentally analyse `smeerenc` and `uncountedsmeerenc`, we calculated the minimal number of elements needed to reach a certain maximum affine dimension. We applied both `smeerenc` and `uncountedsmeerenc` on $\texttt{mulenc}_c$ with $c \in \{37, 11\,479, 11\,591\}$. The values 11 479, 11 591 have been chosen from table 7.5. The values 11 479, 11 591 were the best performing c values of this table. The values we found can be seen in table 7.13. We can see in this table that `uncountedsmeerenc` performs about as well as `smeerenc`. Using these values, it is impossible to tell if `uncountedsmeerenc` or `smeerenc` performs better, as at some values `smeerenc` is better and at some values `uncountedsmeerenc` is better. As `smeerenc` is just `uncountedsmeerenc` with an extra counter, which according to section 7.4 should not provide a lot of benefits, there should be not much of a difference between those two. It is interesting to note that there does not seem to be a lot of difference between the performance of `smeerenc` and `shiftedenc`, if we compare table 7.5 with table 7.12. Therefore, `smeerenc` seems to be a very useful alternative to `shiftedenc`.

Because we are rotating our code words, it can be seen that we use the whole string length early on. As we have seen, this lowers the maximum affine dimension when we are encoding up to 4096 numbers. However, we are also interested in what happens to the maximum affine dimension of the encoding when generating all possible code words. As generating all code words of length 64 by `smeerenc` is infeasible, we limited the number of bits used. Furthermore, we also analysed `4logenc` in this step. In table 7.14 this can be seen. Firstly, this table shows us that the affine dimension does not increase dramatically when all code words are used, assuming that a suitable sub-encoding is used. Note that this was not the case for `shiftedoddenc`. Secondly, we saw that `mulenc` for large c values did perform better than $\texttt{4logenc}_{\texttt{mulenc}_{37}}$ by table 7.7 and table 7.5. However, by table 7.14 and table 7.5, it seems that using `4logenc` gives extra benefits, when combined with `smeerenc`.

Minimal number of elements n and information per bit `Ipb` for various afdim values.

	Maximum affine dimension (afdim)					
	1		2		3	
Encoding	n	`Ipb`	n	`Ipb`	n	`Ipb`
$\texttt{smeerenc}^{(64)}_{\texttt{4logmulenc}_{37}}$	2	0.02	336	0.13	5888	0.20
$\texttt{smeerenc}^{(32)}_{\texttt{4logmulenc}_{37}}$	2	0.03	160	0.23	2688	0.36
$\texttt{smeerenc}^{(16)}_{\texttt{4logmulenc}_{37}}$	2	0.06	24	0.29	—†	—
$\texttt{smeerenc}^{(22)}_{\texttt{4logmulenc}_{37}}$ †	2	0.05	96	0.30	1664	0.49
$\texttt{smeerenc}^{(23)}_{\texttt{4logmulenc}_{37}}$ †	2	0.04	108	0.29	1856	0.47

Table 7.14 The minimal number of elements n required such that codewords($\texttt{smeerenc}^{(d)}, n$) for different d values (code word length) has a certain maximum dimension of its affine subspaces. The n values can be up to $2^{\lfloor \log_2 n \rfloor - 4}$ too big. The `Ipb` value gives the information per bit for the code word space, which is in this case equal to $\frac{\log_2(|C|)}{d}$. The d values chosen are 64, 32, 23, 22 and 16. The sub-encoding used is $\texttt{4logmulenc}_{37}$, which is an abbreviation for $\texttt{4logenc}_{\texttt{mulenc}_{37}}$.

† The $\texttt{smeerenc}^{(16)}_{\texttt{4logmulenc}_{37}}$ encoding can only encode 87 elements, at which point the maximum affine dimension is 2; the $\texttt{smeerenc}^{(22)}_{\texttt{4logmulenc}_{37}}$ can only encode 3439 elements, at which point the maximum affine dimension is 3; the $\texttt{smeerenc}^{(23)}_{\texttt{4logmulenc}_{37}}$ can only encode 3869 elements, at which point the maximum dimension is 3.

Chapter 8

Conclusion

To conclude, the impact of the research performed will be given. In section 8.1 various options to extend the research done in this master project will be given.

We have seen a vulnerability in the GIBBON algorithm of the PRIMATEs family of authenticated encryption algorithms. Furthermore, a proof of concept has been developed, showing how such a vulnerability could be exploited and showing the computational power required to perform such an attack. The GIBBON family uses a permutation that performs a particular round function six times. The vulnerability shown in this permutation was shown to only exist in permutations using the round function at most six times. If the round function would be applied seven times, the vulnerability would not exist. Furthermore, we have seen that this vulnerability can only be exploited if the nonce would be reused. Therefore, the attack is only applicable to implementations that deviate from the specification. However, when the same nonce is reused many times, the attacker can deduce the internal state at the start of the encryption. This internal state can be used to decrypt any message using the same nonce.

Secondly, farfalle using a symmetric key was discussed and its vulnerabilities were analysed. It was shown that this construction requires a counter, that would generate unique binary strings while also keeping the maximum affine dimension of any subset generated by this counter low. To implement such a counter we have introduced several low affine dimensional encodings and have proven some theoretical properties about these encodings. Using an algorithm we have experimentally verified the affine dimension of each of these encodings. Several suitable encodings have been found. In particular $\texttt{mulenc}_c$ for several specific c values has been found to generate a low affine dimensional encoding. However, the affine subspaces generated by $\texttt{mulenc}_c$ can be generated rather easily. To improve the $\texttt{mulenc}_c$ encoding, we have formulated encodings that wrap known encodings, and improve them by applying extra operations to their input or output. For instance, to prevent generation of affine subspaces, we have seen that we can use `logenc` to improve the $\texttt{mulenc}_c$ encoding. Unfortunately, `logenc` is not a computationally simple encoding, which has its drawbacks. Finally, we have seen that we could use `smeerenc` or `uncountedsmeerenc` to reduce the affine dimension of an encoding.

8.1 Future work

The attack performed on the PRIMATEs permutation did not require much knowledge about the internal workings of the PRIMATEs permutation. In a future work, one could develop a tool that automatically analyses such permutations and tries to perform attacks. This could be used by anyone designing a new permutation algorithm, such that they can instantly see problems with a certain design. The tool could also output an advice on the minimal number of rounds required in the algorithm. Moreover, in PRIMATEs the SubElements operation was executed at the start of the round, which allowed us to perform an attack on GIBBON. These kind of problems could be detected by this tool. This could improve all submissions entered into a competition and directly filter out some weaker algorithms.

Furthermore, more research can be done to find applications of low affine dimensional encodings. It might be interesting to check if using a low affine dimensional encoding in, for instance, a sponge construction would provide any benefit. Lastly, more research can be done in creating low affine dimensional encodings. The encodings presented in this thesis can be improved in three different properties. Firstly, the maximum affine dimension of the encodings can be lowered. Secondly, a smaller bit string length in relation to the total number of code words could be achieved. Lastly, the efficiency in computational cost on modern hardware can be improved.

References

[1] Andreeva, E., Bilgin, B., Bogdanov, A., Luykx, A., Mendel, F., Mennink, B., Mouha, N., Wang, Q., and Yasuda, K. Primates v1.1. Tech. rep., KU Leuven, July 2016.

[2] Bellare, M., and Rogaway, P. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security* (New York, NY, USA, 1993), CCS '93, ACM, pp. 62–73.

[3] Berger, T. P., D'Hayer, J., Marquet, K., Minier, M., and Thomas, G. The GLUON family: A lightweight hash function family based on fcsrs. In *Progress in Cryptology - AFRICACRYPT 2012: 5th International Conference on Cryptology in Africa, Ifrance, Morocco, July 10-12, 2012. Proceedings* (Berlin, Heidelberg, 2012), A. Mitrokotsa and S. Vaudenay, Eds., vol. 7374 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 306–323.

[4] Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. On the indifferentiability of the sponge construction. In *Advances in Cryptology – EUROCRYPT 2008: 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings* (Berlin, Heidelberg, 2008), N. Smart, Ed., vol. 4965 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 181–197.

[5] Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. Duplexing the sponge: Single-pass authenticated encryption and other applications. In *Selected Areas in Cryptography: 18th International Workshop, SAC 2011, Toronto, ON, Canada, August 11-12, 2011, Revised Selected Papers* (Berlin, Heidelberg, 2012), A. Miri and S. Vaudenay, Eds., vol. 7118 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 320–337.

[6] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., and Van Keer, R. Farfalle: Parallel permutation-based cryptography.

[7] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., and Van Keer, R. CAESAR submission: Keyak v2. Tech. rep., STMicroelectronics, Radboud University Nijmegen, September 2016.

[8] Biham, E. *Differential Cryptanalysis*. Encyclopedia of Cryptography and Security. Springer US, Boston, MA, 2005, pp. 147–152.

[9] Biham, E., and Shamir, A. Differential cryptanalysis of DES-like cryptosystems. In *Advances in Cryptology-CRYPT0' 90: Proceedings* (Berlin, Heidelberg, 1991), A. J. Menezes and S. A. Vanstone, Eds., vol. 537 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 2–21.

[10] Bilgin, B., Bogdanov, A., Knežević, M., Mendel, F., and Wang, Q. Fides: Lightweight authenticated cipher with side-channel resistance for constrained hardware. In *Cryptographic Hardware and Embedded Systems - CHES 2013: 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings* (Berlin, Heidelberg, 2013), G. Bertoni and J.-S. Coron, Eds., vol. 8086 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 142–158.

[11] Biryukov, A. *Codebook Attack*. Springer US, Boston, MA, 2011, pp. 216–216.

[12] Biryukov, A., Velichkov, V., and Le Corre, Y. Automatic search for the best trails in ARX: Application to block cipher speck. In *Fast Software Encryption: 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers* (Berlin, Heidelberg, 2016), T. Peyrin, Ed., vol. 9783 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 289–310.

[13] Çalik, Ç., and Doğanaksoy, A. Computing the weight of a boolean function from its algebraic normal form. In *Sequences and Their Applications – SETA 2012: 7th International Conference, Waterloo, ON, Canada, June 4-8, 2012. Proceedings* (Berlin, Heidelberg, 2012), T. Helleseth and J. Jedwab, Eds., vol. 7280 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 89–100.

[14] Daemen, J. Hashing and sponge functions part 1: What we have and what we need, May 2011. https://www.frisc.no/wp-content/uploads/2012/02/Sponge-Finse-p1.pdf.

[15] Daemen, J., and Rijmen, V. The block cipher rijndael. In *Smart Card Research and Applications: Third International Conference, CARDIS'98, Louvain-la-Neuve, Belgium, September 14-16, 1998. Proceedings* (Berlin, Heidelberg, 2000), J.-J. Quisquater and B. Schneier, Eds., vol. 1820 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 277–284.

[16] Das, S., Maitra, S., and Meier, W. Higher order differential analysis of NORX. Cryptology ePrint Archive, Report 2015/186, 2015. http://eprint.iacr.org/2015/186.

[17] Dierks, T., and Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, RFC Editor, August 2008. https://tools.ietf.org/rfc/rfc5246.txt.

[18] Diffie, W., and Hellman, M. New directions in cryptography. *IEEE Trans. Inf. Theor. 22*, 6 (Sept. 2006), 644–654.

[19] Dinur, I., and Shamir, A. Cube attacks on tweakable black box polynomials. In *Advances in Cryptology - EUROCRYPT 2009: 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings* (Berlin, Heidelberg, 2009), A. Joux, Ed., vol. 5749 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 278–299.

[20] Dodson, C. T. J., and Poston, T. *Affine Spaces*, vol. 130 of *Graduate Texts in Mathematics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991, ch. Tensor Geometry, pp. 43–56.

[21] Ehrsam, W., Meyer, C., Smith, J., and Tuchman, W. Message verification and transmission error detection by block chaining, Feb. 14 1978. US Patent 4,074,066.

[22] Guido, B., Joan, D., Michaël, P., and Gilles, V. Cryptographic sponge functions, 2011. http://sponge.noekeon.org/CSF-0.1.pdf.

[23] Heys, H. M. A tutorial on linear and differential cryptanalysis. *Cryptologia 26*, 3 (2002), 189–221. http://dx.doi.org/10.1080/0161-110291890885.

[24] Kaliski, B. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational), Sept. 2000. http://www.ietf.org/rfc/rfc2898.txt.

[25] Katz, J. *The Random Oracle Model*. Digital Signatures. Springer US, Boston, MA, 2010, pp. 135–142.

[26] Katz, J., and Yung, M. Unforgeable encryption and chosen ciphertext secure modes of operation. In *Fast Software Encryption: 7th International Workshop, FSE 2000 New York, NY, USA, April 10–12, 2000 Proceedings* (Berlin, Heidelberg, 2001), G. Goos, J. Hartmanis, J. van Leeuwen, and B. Schneier, Eds., vol. 1978 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 284–299.

[27] Khovratovich, D., and Nikolić, I. Rotational cryptanalysis of ARX. In *Fast Software Encryption: 17th International Workshop, FSE 2010, Seoul, Korea, February 7-10, 2010, Revised Selected Papers* (Berlin, Heidelberg, 2010), S. Hong and T. Iwata, Eds., vol. 6147 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 333–346.

[28] Koblitz, N., and Menezes, A. J. The random oracle model: a twenty-year retrospective. *Designs, Codes and Cryptography 77*, 2 (2015), 587–610.

[29] Ladner, R. E., and Fischer, M. J. Parallel prefix computation. *J. ACM 27*, 4 (Oct. 1980), 831–838.

[30] Lai, X. *Higher Order Derivatives and Differential Cryptanalysis*, vol. 276 of *The Springer International Series in Engineering and Computer Science*. Springer US, Boston, MA, 1994, ch. Communications and Cryptography, pp. 227–233.

[31] Matsui, M. Linear cryptanalysis method for DES cipher. In *Advances in Cryptology — EUROCRYPT '93: Workshop on the Theory and Application of Cryptographic Techniques Lofthus, Norway, May 23–27, 1993 Proceedings* (Berlin, Heidelberg, 1994), T. Helleseth, Ed., vol. 765 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 386–397.

[32] Menezes, A. J., van Oorschot, P. C., and Vanstone, S. A. *Handbook of Applied Cryptography*, fifth printing ed. CRC Press, August 2001.

[33] Mennink, B., Reyhanitabar, R., and Vizár, D. Security of full-state keyed sponge and duplex: Applications to authenticated encryption. Cryptology ePrint Archive, Report 2015/541, 2015. http://eprint.iacr.org/2015/541.

[34] Minematsu, K. AES-OTR v3.1. Tech. rep., NEC Corporation, Japan, September 2006.

[35] Mister, S., and Adams, C. Practical S-Box Design. In *SELECTED AREAS IN CRYPTOGRAPHY, 1996* (1996).

[36] NIST Computer Security Division. SHA-3 standard: Permutation-based hash and extendable-output functions. FIPS Publication 202, National Institute of Standards and Technology, U.S. Department of Commerce, May 2014. http://csrc.nist.gov/publications/drafts/fips-202/fips_202_draft.pdf.

[37] Paar, C., and Pelzl, J. *Understanding Cryptography*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, ch. Stream Ciphers, pp. 29–54.

[38] Paar, C., and Pelzl, J. *Understanding Cryptography.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, ch. More About Block Ciphers, pp. 123–148.

[39] Preneel, B. The first 30 years of cryptographic hash functions and the nist sha-3 competition. In *Topics in Cryptology - CT-RSA 2010: The Cryptographers' Track at the RSA Conference 2010, San Francisco, CA, USA, March 1-5, 2010. Proceedings* (Berlin, Heidelberg, 2010), J. Pieprzyk, Ed., vol. 5985 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 1–14.

[40] Preneel, B. Hash functions. In *Encyclopedia of Cryptography and Security*, H. C. A. van Tilborg and S. Jajodia, Eds. Springer US, Boston, MA, 2011, pp. 543–553.

[41] Preneel, B. Message authentication algorithm. In *Encyclopedia of Cryptography and Security*, H. C. A. van Tilborg and S. Jajodia, Eds. Springer US, Boston, MA, 2011, pp. 775–775.

[42] Sako, K. Digital signature schemes. In *Encyclopedia of Cryptography and Security*, H. C. A. van Tilborg and S. Jajodia, Eds. Springer US, Boston, MA, 2011, pp. 343–344.

[43] Seidlová, M. Algebraic-differential analysis of keccak. Master's thesis, Charles University, 2015.

[44] Skórski, M. Shannon entropy versus renyi entropy from a cryptographic viewpoint. In *Cryptography and Coding: 15th IMA International Conference, IMACC 2015, Oxford, UK, December 15-17, 2015. Proceedings* (Cham, 2015), J. Groth, Ed., vol. 9496 of *Lecture Notes in Computer Science*, Springer International Publishing, pp. 257–274.

[45] Stallings, W. *Cryptography and Network Security: Principles and Practice*, 3rd ed. Pearson Education, 2002.

[46] Stickler, B. A., and Schachinger, E. *Basic concepts in computational physics.* Springer, 2016.

[47] Wu, H., and Preneel, B. Aegis: A fast authenticated encryption algorithm. In *Selected Areas in Cryptography – SAC 2013: 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers* (Berlin, Heidelberg, 2014), T. Lange, K. Lauter, and P. Lisoněk, Eds., vol. 8282 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 185–201.