Cayley Graphs of Semigroups and Applications to Hashing

Bianca Sosnovski

Graduate Center, City University of New York

Recommended Citation
CAYLEY GRAPHS OF SEMIGROUPS
AND APPLICATIONS TO HASHING

by

Bianca Sosnovski

A dissertation submitted to the Graduate Faculty in Mathematics in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

2016
This manuscript has been read and accepted for the Graduate Faculty in Mathematics in satisfaction of the dissertation requirements for the degree of Doctor of Philosophy.

Dr. Vladimir Shpilrain

Date

Chair of Examining Committee

Dr. Ara Basmajian

Date

Executive Officer

Dr. Vladimir Shpilrain

Dr. Delaram Kahrobaei

Dr. Melvyn Nathanson

Dr. Benjamin Steinberg

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK
Abstract

Cayley Graphs of Semigroups and Applications to Hashing

by

Bianca Sosnovski

Adviser: Vladimir Shpilrain

In 1994, Tillich and Zémor proposed a scheme for a family of hash functions that uses products of matrices in groups of the form $SL_2(\mathbb{F}_{2^n})$. In 2009, Grassl et al. developed an attack to obtain collisions for palindromic bit strings by exploring a connection between the Tillich-Zémor functions and maximal length chains in the Euclidean algorithm for polynomials over $\mathbb{F}_2$.

In this work, we present a new proposal for hash functions based on Cayley graphs of semigroups. In our proposed hash function, the noncommutative semigroup of linear functions under composition is considered as platform for the scheme. We will also discuss its efficiency, pseudorandomness and security features.

Furthermore, we generalized the Fit-Florea and Matula’s algorithm (2004) that finds the discrete logarithm in the multiplicative group of integers modulo $2^k$ by establishing a connection between semi-primitive roots modulo $2^k$ where $k \geq 3$ and the logarithmic base used in the algorithm.
Dedicated to the memory of
Raymundo Amoras, Sebastiana Amoras,
Arnold Presayzen and Sophia Lubensky.
Acknowledgments

I would like to especially thank Professor Vladimir Shpilrain for being my thesis adviser. I appreciate his availability, comments, research ideas and assistance, which made possible the completion of this thesis.

I also would like to express my gratitude to Professor Delaram Kahrobaei for her continued support, guidance and assistance over the years to me, as a student at the Graduate Center. Many thanks to Professors Melvyn Nathanson and Benjamin Steinberg for serving on my defense committee.

Last, but by no means least, I am eternally indebted to my family and especially to my husband, Alexander, for the invariable support and encouragement during the time I pursued my doctorate.

Bianca Sosnovski
Contents

List of Figures xi

List of Tables xii

1 Introduction 1

2 Cryptographic Hash Functions 6
  2.1 Definition ................................................................. 7
  2.2 Security properties ....................................................... 7
  2.3 Relations between properties ......................................... 9
  2.4 Traditional constructions of hash functions ...................... 11
  2.5 Mathematical hash functions .......................................... 14
  2.6 Practical Security ........................................................ 14

3 Cayley hash functions 16
  3.1 Cayley graphs .............................................................. 16
  3.2 Expander graphs .......................................................... 17
  3.3 Cayley hash functions .................................................... 21
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4</td>
<td>The Tillich-Zémor hash function</td>
<td>23</td>
</tr>
<tr>
<td>3.5</td>
<td>More examples of Cayley hash functions</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>Cryptanalysis of Cayley hash functions</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>Security of Cayley hash functions</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Attacks against Cayley hash functions</td>
<td>28</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Generic attacks</td>
<td>28</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Subgroup attacks</td>
<td>29</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Lifting attacks</td>
<td>29</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Finding elements of small order</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>The collision algorithm of Grassl et al.</td>
<td>32</td>
</tr>
<tr>
<td>4.4</td>
<td>Other considerations about the Tillich-Zémor hash functions</td>
<td>35</td>
</tr>
<tr>
<td>5</td>
<td>Hashing with compositions of linear functions</td>
<td>37</td>
</tr>
<tr>
<td>5.1</td>
<td>Basic results</td>
<td>37</td>
</tr>
<tr>
<td>5.2</td>
<td>Freeness of semigroups of linear functions</td>
<td>38</td>
</tr>
<tr>
<td>5.3</td>
<td>Proposed Cayley hash function</td>
<td>41</td>
</tr>
<tr>
<td>5.4</td>
<td>The semigroup generated by ( f_0(x) = 2x + 1 ) and ( f_1(x) = 3x + 1 ) modulo ( p )</td>
<td>43</td>
</tr>
<tr>
<td>5.5</td>
<td>Girth of the Cayley graph of the semigroup of linear functions</td>
<td>46</td>
</tr>
<tr>
<td>5.6</td>
<td>Efficiency of composition of linear functions modulo ( p )</td>
<td>48</td>
</tr>
<tr>
<td>5.7</td>
<td>Pseudorandomness</td>
<td>51</td>
</tr>
</tbody>
</table>
5.7.1 Χ²-goodness-of-fit test ........................................... 51
5.7.2 The NIST Statistical Test Suite ................................. 53
5.8 Security ................................................................. 57
5.8.1 Generic attacks ....................................................... 57
5.8.2 Subgroup attacks ...................................................... 57
5.8.3 Finding elements of small order ................................. 57
5.8.4 Lifting attacks ......................................................... 58

6  Semi-primitive roots and the discrete logarithm modulo 2^k 60
   6.1 The digit inheritance property .................................. 62
   6.2 Properties of the discrete logarithm modulo 2^k ............ 63
   6.3 The Digit-Serial Discrete Logarithm Algorithm ............. 65

7  Products of powers of semi-primitive roots equal to 1 in the ring of integers modulo 2^{127} 69

8  Conclusion 75

Appendices 77

A Cayley Graph for p = 5 78

B Python algorithm for the runtime of evaluating compositions of modular linear functions 80

C SAGE Algorithm for generating sequences modular linear hash values 83
D  Python code for the Chi-square test 85

E  SAGE code for the FitFlorea-Matula DLG Digit-serial Algorithm 89

Bibliography 95
List of Figures

2.1 Classification of hash functions .............................................. 12
2.2 General model of a hash function construction ............................... 13
3.1 Graph $C_4$ ........................................................................... 18
3.2 Possible subsets with at most 2 vertices of the graph $C_4$ ................. 18
3.3 The cycle graphs $C_3$, $C_4$, $C_5$ and $C_6$ .................................. 19
3.4 Graph $C_6$ with the bottom half subset of vertices and its boundary .... 19
A.1 Cayley graph of matrix semigroup generated by $f(x) = 2x + 1$ and $g(x) =$
 $3x + 1$ modulo $p = 5$ ................................................................. 79
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Average runtime to evaluate compositions with $f_0(x) = 2x + 1$ and $f_1(x) = 3x + 1$ modulo $p$</td>
<td>50</td>
</tr>
<tr>
<td>5.2</td>
<td>$\chi^2$-Uniformity Test</td>
<td>53</td>
</tr>
<tr>
<td>5.3</td>
<td>NIST Statistical Suite Results</td>
<td>56</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Hash functions are an important tool for cryptography. They are fundamental blocks in the construction of several cryptographic primitives such as digital signatures and key derivation systems.

Hash functions are easy-to-compute compression functions that take a variable-length input and convert it to a fixed-length output. Hash functions are used as compact representations, or digital fingerprints, of data and to provide message integrity. The following are some basic requirements for cryptographic hash functions:

- **Preimage resistance**: it should be computationally infeasible to find an input which hashes to a specified output;

- **Second-preimage resistance**: it should be computationally infeasible to find a second input that hashes to the same output as a specified input;

- **Collision resistance**: it should be computationally infeasible to find two different inputs that hash to the same output.

Most hash functions in use today have constructions that apply some sort of iterative
design involving a compression function and a transformation as the Merkle-Damgård [19]. But the vulnerability of such hash functions to attacks in recent years suggest investigating new designs. In contrast to the traditional model of hash functions, provably secure hash functions are hash functions whose security are based on the difficulty of solving a known “hard” problem.

Examples of provably secure hash functions are the Cayley hash functions that are based on the Cayley graphs of certain (semi)groups. The Cayley graphs of the underlying groups of the past proposals are expander graphs, thus presenting interesting properties such that of the rapid mixing of Markov chains [7]. These Cayley hash functions are designed so that their security would follow from the alleged hardness of a mathematical problem related to the expander graph of the (semi)group associated with it.

In 1991, Zémor introduced a family of hash functions whose values correspond to matrix products in groups of the form $SL_2(\mathbb{F}_p)$ for $p$ prime [38]. This first instance of a Cayley hash function was broken in 1994 by Tillich and Zémor. As replacement to increase the security of the scheme, they also provided the group $SL_2(\mathbb{F}_{2^n})$ where $\mathbb{F}_{2^n}$ is a field with $2^n$ elements [35, 34]. The work of Tillich and Zémor received significant interest and other proposals for Cayley hash functions were suggested based on the Ramanujan graphs constructed by Pizer and Lubotzky-Phillips-Sarnak (LPS) [7], and the Ramanujan graphs constructed by Morgenstern [28].

Unlike the SHA family of hash functions that hash blocks of input, the Tillich-Zémor function hashes each bit individually. More specifically, the “0” bit is hashed to a particular $2 \times 2$ matrix $A$, and the “1” bit is hashed to another $2 \times 2$ matrix $B$. Then any bit string is hashed simply to the product of matrices $A$ and $B$ corresponding to bits in this string. For example, the bit string 10010 is hashed to the matrix $BA^2BA$.

The matrices used in the Tillich-Zémor scheme are elements from the group $SL_2(\mathbb{F}_{2^n})$
with $\mathbb{F}_2^n \approx \mathbb{F}_2[x]/(p(x))$ where $\mathbb{F}_2$ is the field with two elements, $\mathbb{F}_2[x]$ is the ring of polynomials over $\mathbb{F}_2$, and $(p(x))$ is the ideal generated by an irreducible polynomial $p(x)$ in $\mathbb{F}_2[x]$ of degree $n$. These matrices are $A = \begin{pmatrix} \alpha & 1 \\ 1 & 0 \end{pmatrix}$ and $B = \begin{pmatrix} \alpha & \alpha + 1 \\ 1 & 1 \end{pmatrix}$, where $\alpha$ is a root of $p(x)$.

The Tillich-Zémor hash function sustained attacks until 2009 when Grassl et al. [11] established a connection between the Tillich-Zémor function and maximal length chains in the Euclidean algorithm for polynomials over the field with two elements. The connection makes possible to obtain collisions between distinct palindromic bit strings. Following this attack, Petit and Quisquater [29] offered a modification of the Grassl et al.’s algorithm to provide a second-preimage algorithm and also provided an extended form of the Grassl et al.’s algorithm to find preimages.

Such attacks can only be applied to the specific group generated by the matrices $A$ and $B$ above. A general attack for the Tillich-Zémor scheme with group $SL_2(\mathbb{F}_q)$ is proposed in [21]. The attack runs with super-polynomial time $\mathcal{O}(\sqrt{q})$ to find collisions for arbitrary $q$. This attack is infeasible for bit strings of length $n > 100$.

Petit and Quisquater [29, 30] suggested that security might be recovered by introducing new generators. The Cayley hash function design is still appealing and it deserves further interest in cryptography by showing that the factorization, representation and balance problem in noncommutative groups still are potentially hard problems for general parameters of Cayley hash functions. As an example, Bromberg, Shpilrain and Vdovina in a recent paper [4] suggested other pairs of matrices of the form $\begin{pmatrix} 1 & k \\ 0 & 1 \end{pmatrix}$ and $\begin{pmatrix} 1 & 0 \\ k & 1 \end{pmatrix}$, $k \geq 2$, with the idea that these matrices generate a free monoid over $\mathbb{Z}$ and that there cannot be any short
relations between them over \( \mathbb{F}_p \).

In this work, we offer a new semigroup platform for hashing, corresponding to a pair of two linear functions in one variable over \( \mathbb{F}_p \) under composition operation. The hash functions are \( f(x) = 2x + 1 \) and \( g(x) = 3x + 1 \) modulo a prime \( p > 3 \). The result is a very efficient hash function, which hashes a bit string of length \( n \) with time complexity of at most \( 2n \) multiplications and \( 2n \) additions in \( \mathbb{F}_p \).

The input string for the new hash function can have an arbitrary length, while the output is of size \( 2 \log p \). This is an advantage compared to the Tillich-Zémor proposal whose outputs have length \( 4 \log p \). With respect to the security of our hash function, we give explicit lower bound on the length of collisions and also discuss the possibility of known attacks be applied to our hash function.

Furthermore, we also present a generalization of the discrete logarithm algorithm introduced by Fit-Florea and Matula in [9]. They present an additive bit-serial algorithm where the integer 3 is used as logarithmic base modulo \( 2^k \) for \( k \geq 3 \).

Hardware support for applications where fast residue arithmetic is needed generally relies on residue addition and multiplication. Finding efficient algorithms to perform other residue operations was desirable. New applications in cryptography can benefit from higher precision hardware implementation of integer arithmetic. Fit-Florea-Matula algorithm represents an efficient way of implementing hardware to perform fast residue arithmetic for the discrete logarithm and multiplicative inverse with any precision of \( k \) bits.

We extend Fit-Florea and Matula’s results to any semi-primitive roots modulo \( 2^k \). An integer \( h \) is a semi-primitive root modulo \( n \) if its order is \( \phi(n)/2 = 2^{n-2} \). The connection between semi-primitive roots modulo \( 2^k \) and the logarithmic base is possible because the order of such elements.

In relation to the discrete logarithm modulo \( 2^k \), we also discuss the problem of finding
the minimum value for $n + m$ such that $5^n \cdot 3^m \equiv 1 \mod 2^{127}$. It turned out that finding the minimum value is not an easy task. But applying the results generalizing the Fit-Florea and Matula’s algorithm, we are at least able to find an upper bound to $n + m$. 
Chapter 2

Cryptographic Hash Functions

Cryptographic hash functions are fundamental building blocks of many computer security schemes and protocols. They are used to ensure data integrity, authentication, certification and confidentiality.

Similar to conventional hash functions commonly used in non-cryptographic computer applications, cryptographic hash functions map larger domains to smaller ranges (many-to-one mappings) but the latter differ in several important aspects from the former. We restrict our focus to cryptographic hash functions and from now on we refer to them simply as hash functions.

A hash function takes a message as input and returns an output referred to as hash code, hash value or simply hash. The hash value of such a function serves as a compact representative image (also known as digital fingerprint or message digest) of an input string, which can be used as if it were uniquely identifiable with that string. Because the existence of collisions (pairs of inputs with same hash value) in many-to-one functions is guaranteed, the unique identification between inputs and their hash values is in the computational sense. In practice, a hash value should be uniquely identifiable with a single input and collisions
should be computationally difficult to find, that is, “never” occurring in practice.

2.1 Definition

Hash functions can be classified as \textit{keyed} and \textit{unkeyed} hash functions. The specification of keyed hash functions dictates two distinct inputs, a message and a secret key, while unkeyed hash functions have only a single input, the message.

\textbf{Definition 2.1.} A (Unkeyed) hash function is a function $h$ which satisfies, as a minimum, the following two conditions:

1. \textit{Compression}: $h$ maps an input $x$ of arbitrary finite bit length to an output $h(x)$ of fixed bit length $n$.
   \begin{equation}
   h : \{0, 1\}^* \longrightarrow \{0, 1\}^n
   \end{equation}

2. \textit{Easy of computation}: given $h$ and an input $x$, $h(x)$ is easy to compute\footnote{“Computationally feasible” (or “easy”) means polynomial time and space or, in practice, with a certain number of machine operations to time units \cite{19}.}.

2.2 Security properties

Since hash functions are uniquely identifiable with input strings, one of their applications is to serve as \textit{manipulation detection codes} (MDCs) whose hash values are such that an accidental or intentional change of the input string will change the hash value. For that purpose (together with other mechanisms), it is desirable that hash function also satisfies at least one of the properties below. Let $h$ be a hash function:
Chapter 2. Cryptographic Hash Functions

1. **Preimage resistance**: Given a hash value $y$ for which a corresponding input is not known, it is computationally infeasible \(^2\) to find any input $x$ such that $y = h(x)$. This property is also known as *one-wayness*.

2. **Second-preimage resistance**: Given an input $x_1$ it is computationally infeasible to find another input $x_2$ where $x_1 \neq x_2$ such that $h(x_1) = h(x_2)$. This property is also referred to as *weak collision resistance*.

3. **Collision resistance**: It is computationally infeasible to find any two inputs $x_1$ and $x_2$ where $x_1 \neq x_2$ such that $h(x_1) = h(x_2)$. This property is sometimes referred to as *strong collision resistance*.

**Definition 2.2.** A *one-way hash function* (OWHF) is a hash function that satisfies the properties of preimage resistance and second-preimage resistance.

**Definition 2.3.** A *collision resistant hash function* (CRHF) is a hash function that satisfies the properties of second-preimage resistance and collision resistance.

**Example 2.1.** Consider a simple modulo-32 checksum (different modulus may be used). This function takes an binary string and divides it into 32-bit blocks and computing the modular sum of those blocks. This function is easily computed and offers compression. But we can easily construct a binary string that shares the same checksum. That is, it is possible to work backwards from the checksum to an input satisfying the sum. Hence, this checksum function is not preimage resistant. It is neither second-preimage resistant nor collision resistant.

\(^2\)According to Petit and Quisquater [30], “computationally infeasible” (or “hard”) can be understood in two ways:
- Practically, it means that no big cluster of computers can perform the task.
- Theoretically, it means that no probabilistic algorithm running in time polynomial in $n$ succeeds in performing the task for large values of the parameter $n$ with probability larger than the inverse of some polynomial function of $n$. 
Example 2.2. Let \( g(x) = x^2 \mod n \) be a function for appropriate randomly chosen primes \( p \) and \( q \) where \( n = pq \) and the factorization of \( n \) is unknown. \( g \) is preimage resistant because finding a preimage corresponds to computing a square root, which is equivalent to factoring \( n \) and thus allegedly computationally hard [19, § 9.2.4]. However, \( x \) and \(-x\) yield the same output making trivial to find a second-preimage and a collision. Therefore, \( g \) is neither an OWHF nor a CRHF.

2.3 Relations between properties

Here we present some relationships between the hash function properties above.

**Fact 2.1.** Preimage resistance does not guarantee second-preimage resistance\(^3\)

*Justification:* See example 2.2 above.

**Fact 2.2.** A collision resistance hash function is also second-preimage resistant.

*Justification:* Assume that \( h \) is a collision resistant hash function. Fix an input \( x_1 \). If \( h \) is not second-preimage resistant then it is computationally feasible to find a distinct input \( x_2 \) such that \( h(x_1) = h(x_2) \). Thus, the pair \((x_1, x_2)\) forms a collision for \( h \), contradicting the collision resistance property of \( h \).

**Fact 2.3.** Second-preimage resistance does not guarantee preimage resistance.

*Justification:* Let \( g \) be a collision resistant hash function that maps arbitrary-length inputs to \( n \)-bit outputs. Then define a hash function by\(^4\)

\[
h(x) = \begin{cases} 
1 \| x, & \text{if } x \text{ has bit length } n \\
0 \| g(x), & \text{otherwise.} 
\end{cases}
\]

\(^3\)However, in practice, CRHF almost always has the additional property of preimage resistance [19].

\(^4\)\(\|\) denotes concatenation.
The function $h$ outputs $(n + 1)$-bit hash values and is collision resistant since finding a collision for it is equivalent to finding a collision for $g$. However, finding the preimage of any hash value beginning with the bit 1 is a trivial task. Thus, $h$ is not preimage resistant.

Keyed hash functions are specifically applied to message authentication, thus they are called *message authentication code algorithms* (MACs).

**Definition 2.4.** A *message authentication code algorithm* is a family of functions $h$ parameterized by a secret key $k \in \mathcal{K}$ where $\mathcal{K}$ denotes the key space.

$$h : \{0, 1\}^* \times \mathcal{K} \rightarrow \{0, 1\}^n$$

$$h(x, k) \mapsto h_k(x)$$

$h$ has the following properties:

1. **Easy of computation:** For a known function $h_k$, given an input $x$ and a value $k$, $h_k(x)$ is easy to compute. The result is called *MAC-value* or simply *MAC*.

2. **Compression:** $h_k$ maps an input $x$ of arbitrary finite bit length to an output $h_k(x)$ of fixed bit length $n$.

3. **Computation-resistance:** Given a description of the function family $h$, for every fixed allowable value of $k$ (unknown to an adversary) the following holds; given zero or more message-MAC pairs $(x_i, h_k(x_i))$, it is computationally infeasible to compute any pair $(x, h_k(x))$ for any new input $x \neq x_i$ (including possibly for $h_k(x) = h_k(x_i)$ for some $i$).

A *MAC forgery* occurs if computation-resistance does not hold.

Note that computation-resistance implies the property of *key non-recovery*, that is, it
is computationally hard to recover the secret key $k$, given one or more message-MAC pairs $(x_i, h_k(x_i))$ for that $k$. However, key non-recovery does not imply computation-resistance. To forge a new MAC, the key need not always to be recovered.

The computation-resistance property should hold whether the messages $x_i$’s with corresponding matching MAC-values that are available are given to the adversary (known-text attack) or freely chosen by the adversary (chosen-text attack). Thus, if $h_k$ is a MAC algorithm then $h_k$ is against a chosen-text attack without the knowledge of the key $k$.

**Fact 2.4.** Let $h_k$ be a MAC algorithm. For an adversary not knowing the key $k$, $h_k$ is both second-preimage resistant and collision resistant.

*Justification:* Since $h_k$ is a MAC, the computation-resistance property holds implying that the hash values should not even be computable by parties without the secret key $k$.

**Fact 2.5.** Let $h_k$ be a MAC algorithm. For an adversary not knowing the key $k$, $h_k$ is preimage resistant with respect to hash input.

*Justification:* Suppose that $h_k$ is not preimage resistant. Then, given a randomly-selected hash value $y$ it is possible to recover the preimage $x$. But this violates computation-resistance property.

### 2.4 Traditional constructions of hash functions

Modern (unkeyed) hash functions are traditionally constructed using an iterative design. The idea behind the general model is that hash functions mapping variable bit length messages into fixed bit length can be constructed by using compression functions that map bit strings of fixed length to bit strings of shorter length.

**Definition 2.5.** A *compression function* is a function that maps elements from $\{0, 1\}^{n+r}$ to elements of $\{0, 1\}^n$ where $n$ and $r$ are positive integers.
To construct a hash function, let $f : \{0, 1\}^{n+r} \mapsto \{0, 1\}^n$ be a compression function. A message $x$ of arbitrary finite length is divided into fixed-length blocks $x_i$ each of bit length $r$, padding the last block with 0-bits if necessary. Denote the padded message by $x = x_1 x_2 \cdots x_t$.

After preprocessing the message as above, we use each block $x_i$, one at a time, as part of the input in the iterative process involving $f$, which computes a new intermediate result of bit length $n$. Let $H_i$ denote the partial result after stage $i$. Define a fixed constant $H_0$ with bit length $n$ as the initial value. The iterative construction of a hash function $h$ can be modeled as:

$$
H_0 = IV; \quad H_i = f(H_{i-1} \parallel x_i), 1 \leq i \leq t; \quad h(x) = g(H_t).
$$

$g$ is an optional output transformation used in the final step to map the $n$-bit output to an
Chapter 2. Cryptographic Hash Functions

$m$-bit result, and is often the identity mapping $g(H_t) = H_t$. The diagram below describe the construction [19].

![Diagram of hash function construction]

Figure 2.2: General model of a hash function construction

For security reasons, Merkle and Damgård proposed to include an additional block $x_{t+1}$ that contains the bit length of the original message in the preprocessing step of the construction above. This is known Merkle-Damgård strengthening. The Merkle-Damgård transform is the iterative process with the application of $H_0 = 0^n$ as initialization value and the Merkle-Damgård strengthening.

Merkle and Damgård, independently, showed that if the underlying compression function $f$ is collision resistant then the Merkle-Damgård transform gives a hash function that is also collision resistant.
2.5 Mathematical hash functions

Most hash functions in use today are constructed using the iterative design describe in the previous section. These traditionally constructed hash functions are based on performing several rounds of complex bit operations in sequence with the hope that it is difficult to reverse. Serious attacks were developed against the two most commonly of these hash functions, the SHA-1 and MD5. Efforts by the research community have been made to replace such type of hash functions [13, § 4.6.5].

Some of the options proposed is to create hash functions using algebraic structures. One example of a mathematical function is the modular arithmetic secure hash algorithm (MASH-1), which somewhat follows the Merkle-Damgård transform but it uses a compression function based on modular arithmetic. MASH-1 involves the use of an RSA-like modulus \( N \) that should be difficult to factor. Its security is based partially on the difficulty of extracting modular roots.

Other examples of mathematical functions is the so called provably secure hash functions. The security of hash functions are based on some hard mathematical problem and finding collisions of the hash functions is as hard as breaking the underlying problem. Their security is more than just relying on complex mixing of bits as in the classical approach.

We are interested in hash functions whose constructions are related to the Cayley graphs of (semi)groups that underly the hard mathematical problem. More about them will be discussed in the next chapter.

2.6 Practical Security

We conclude this chapter by including some requirements necessary, as a minimum, about the size of the output of hash functions in order to avoid some known attacks [19].
Assume that $2^{80}$ but not fewer operations is considered beyond computational hardness. Let a hash function outputs hash values of length $n$ bits then the following statements can be made about $n$:

- For a OWHF, $n \geq 80$ is required.
- For a CRHF, $n \geq 160$ is required.
- For a MAC, $n \geq 64$ along with secret key of 64-80 bits is sufficient for most applications.
Chapter 3

Cayley hash functions

In this chapter, we present a class of hash functions based on Cayley graphs that are expander graphs. We start with basic definitions, followed by a general scheme on how to construct such functions and some examples.

3.1 Cayley graphs

Definition 3.1. A graph is an ordered pair $G = (V, E)$ composed of a vertex set $V$ together with a edge multiset $E$. The vertex set $V$ can be any set and the edge multiset $E$ is a multiset whose elements are of the form $\{v, w\}$ or $\{v\}$ where $v$ and $w$ are distinct vertices. An edge of the form $\{v\}$ is called a loop.

Definition 3.2. The diameter of $G$ is defined by $\text{diam}(G) = \max_{v, w \in V} \text{dist}(v, w)$, that is, the maximal length of the shortest path between any two vertices $v$ and $w$. If there is no path connecting two vertices then conventionally the distance is defined as infinite.

Now let $G$ be a multiplicative (semi)group and $S = \{s_1, s_2, \ldots, s_k\}$ be a subset of $G$.

Definition 3.3. A Cayley graph $C_{G, S} = (V, E)$ is a $k$-regular graph constructed from the (semi)group $G$ with respect to $S \subset G$ as follows: For each element $g \in G$, $V$ contains a
vertex $v_g$ associated to $g$. $E$ contains the directed edge $(v_{g_1}, v_{g_2})$ if and only if there is $s_i \in S$ for some $i$ such that $g_2 = g_1 s_i$. The elements of $S$ are called the graph generators.

If $S$ is stable under inversion ($S = S^{-1}$) then the graph $C_{G,S}$ is undirected.

$C_{G,S}$ is a connected graph if and only if $S$ generates the whole group $G$.

### 3.2 Expander graphs

To define the expansion (or isoperimetric, or Cheeger) constant of a graph, let’s consider a regular undirected graph $G = (V, E)$ with $|V| = n$.

For $U, S \subset V$, the set of all edges between $U$ and $S$ is denoted by $E(U, S) = \{(u, s) \mid u \in U, s \in S \text{ and } (u, s) \in E\}$. The edge boundary of $U \subset V$ is $\partial U = E(U, \bar{U})$.

**Definition 3.4.** The expansion constant of a graph $G = (V, E)$ is defined by:

$$h(G) = \min_{0 < |S| \leq \frac{n}{2}} \frac{|\partial S|}{|S|}$$

**Remark 3.1.** By definition of $h(G)$, there exists at least one subset $S$ of vertices such that $h(G) = \frac{|\partial S|}{|S|}$. This subset measures the worst case scenario for the graph $G$, that is, every other subset of vertices of the graph has a larger boundary relatively to its size.

**Example 3.1.** Consider the cycle graph $C_4$ shown in Figure 3.1 [15]. To compute its expansion constant we consider the subsets of size at most $n/2 = 2$.

Figure 3.2 shows all possible subsets under consideration represented by the white vertices with corresponding boundary set as dashed lines. Therefore, $h(C_4) = 1$. 
Chapter 3. Cayley hash functions

Figure 3.1: Graph $C_4$

![Graph $C_4$](image)

Figure 3.2: Possible subsets with at most 2 vertices of the graph $C_4$

(a) $|\partial S| = 2$

(b) $|\partial S| = 1$

(c) $|\partial S| = 2$

Definition 3.5. A family of expander graphs $\{G_i\}_{i \in \mathbb{N}}$ is a collection of graphs such that:

1. $G_i$ is a $d$-regular graph of size $n_i$ for all $i \in \mathbb{N}$ ($d$ is a constant for the family).
2. $\{n_i\}$ is a monotone growing sequence that doesn’t grow too fast.
3. For all $i \in \mathbb{N}$, $h(G_i) \geq \epsilon > 0$.

In the definition above, we have arbitrarily large graphs but they don’t have a lot of edges. Each vertex of any graph in the family has $d$ edges. Expander graphs are sparse graphs that are highly connected. Intuitively, in an expander family of graphs, each graph is such that every subset $S \subset V$ of vertices expands quickly in the sense it is connected to many vertices outside $S$.

Example 3.2. Let $C_n$ represent an infinite family of 2-regular cycle graphs as shown in Figure 3.3. Take the subset $S$ to be the bottom half vertices of $C_n$. We assume that $n$ is even. For example, Figure 3.4 shows that $S$ is the set of white vertices for the graph $C_6$.

\[\text{Example 3.2.} \quad \text{Let } C_n \text{ represent an infinite family of 2-regular cycle graphs as shown in Figure 3.3. Take the subset } S \text{ to be the bottom half vertices of } C_n. \text{ We assume that } n \text{ is even. For example, Figure 3.4 shows that } S \text{ is the set of white vertices for the graph } C_6.\]

\[\text{[Footnote]} \quad \text{The sequence } \{h(G_i)\} \text{ is said to be bounded away from zero.}\]
Thus, $|S| = \frac{n}{2}$ and $|\partial S| = 2$ and consequently

$$h(C_n) = \min_{0 < |S| \leq \frac{n}{2}} \frac{|\partial S|}{|S|} \leq \frac{|\partial S|}{|S|} = \frac{2}{\frac{n}{2}} = \frac{4}{n}$$

If $n$ is odd, then $h(C_n) = \frac{4}{n-1}$. Therefore, $h(C_n) \to 0$ as $n \to \infty$, and so $\{C_n\}$ is not a family of expander graphs.

![Figure 3.3: The cycle graphs $C_3$, $C_4$, $C_5$ and $C_6$](image)

![Figure 3.4: Graph $C_6$ with the bottom half subset of vertices and its boundary](image)

**Example 3.3.** Let $G_n$ be a random $d$-regular graph with $n$ vertices. Random graphs are constructed by connecting each vertex to $d$ randomly chosen vertices. Let $S$ be a subset of vertices of $G_n$ such that $|S| \leq n/2$. Then a typical vertex in $S$ will be connected to $\approx \frac{d|S|}{n}$. Because there are $|S|$ such vertices in $S$, then

$$|\partial S| \approx |S| \cdot \frac{d|S|}{n} \Rightarrow \frac{|\partial S|}{|S|} \approx \frac{d|S|}{n}$$

$$\min_{0 < |S| \leq \frac{n}{2}} \frac{|\partial S|}{|S|} \approx \min_{0 < |S| \leq \frac{n}{2}} \frac{d|S|}{n}.$$
The quantity \( d \frac{|S|}{n} \) is minimal when \( |\bar{S}| \) is minimal. This happens when \( |S| \) is maximal, that is, \( |S| = \frac{n}{2} \). Thus, \( |\bar{S}| = n - |S| = n - \frac{n}{2} = \frac{n}{2} \).

We can now conclude that

\[
\min_{0 < |S| \leq \frac{n}{2}} \frac{|\partial S|}{|S|} \approx d \frac{n}{2} \frac{n}{n} = \frac{d}{2}.
\]

Since \( h(G_n) \) does not depend on \( n \), a family of \( d \)-regular random graphs is an expander family.

**Definition 3.6.** Let \( G = (V, E) \) with \( |V| = n \) be a graph. The **adjacency matrix** of \( G \), denoted \( A(G) \), is a \( n \times n \) matrix whose entries correspond to the number of edges between \((v, w) \in E\). If there are \( t \) edges from vertex \( i \) to vertex \( j \), then we put \( t \) as the entry on row \( i \), column \( j \) of the matrix \( A(G) \).

Unless stated otherwise, the following results are for undirected graphs.

The set of eigenvalues of the adjacency matrix of a graph is called the **spectrum** of the matrix. For an undirected graph, the adjacency matrix is symmetric and thus its eigenvalues\(^2\) are real.

Let \( \lambda_i \) be the eigenvalues of the adjacency matrix of a \( d \)-regular graph \( G \) with \( n \) vertices satisfy the following properties [12, 15]:

1. \( \lambda_1 = d \geq \lambda_2 \geq \ldots \geq \lambda_n \geq -d \)

2. \( \frac{d - \lambda_2}{2} \leq h(G) \leq \sqrt{2d(d - \lambda_2)} \)

If \( G_n \) is a sequence of \( d \)-regular graphs with \( |G_n| \to \infty \) and \( n \to \infty \) then the following gives a lower bound on \( \lambda = \max\{|\lambda_2|, |\lambda_n|\} \):

\(^2\)For directed graphs, we can consider the singular values of the adjacency matrix \( A = A(G) \), which are equal to the square roots of the eigenvalues of the symmetric matrix \( A^T A \).
Chapter 3. Cayley hash functions

- \( \lambda \geq 2 \sqrt{d - 1} - O_n(1) \) where \( O_n(1) \to 0 \) for every fixed \( d \) as \( n \to \infty \).

In property 2 above, the quantity \( \Delta(G) = \lambda_1 - \lambda_2 = d - \lambda_2 \) is called the spectral gap of the graph \( G \) and gives a good estimation for the expansion constant \( h(G) \). It is easier to compute the spectral gap than enumerating exponentially many subsets of vertices of the graph and computing the ratios \( \frac{|\partial S|}{|S|} \).

Property 2 also implies that a graph is an expander \( (h(G) \geq \epsilon) \) if and only if \( \Delta(G) \) is bounded away from zero.

Ramanujan graphs have asymptotically optimal expansion, that is, \( \lambda \leq 2 \sqrt{d - 1} \).

**Lemma 3.1** (Expander Mixing Lemma, Alon-Chung). Let \( G = (V, E) \) be a \( d \)-regular undirected graph with \( |V| = n \), and \( \lambda = \max\{|\lambda_2|, |\lambda_n|\} \).

For all \( S, T \subset V \),

\[
\left| |E(S, T)| - \frac{d|S||T|}{n} \right| \leq \lambda \sqrt{|S||T|}
\]

**Proof.** The proof is shown in \([12]\). \( \square \)

\( \lambda \) is the largest absolute value of an eigenvalue other than \( \lambda_1 = d \). The lemma relates \( \lambda \) with the “randomness” of the graph \( G \). The difference \( |E(S, T)| - \frac{d|S||T|}{n} \) represents the difference between the actual number of edges connecting the subsets \( S \) and \( T \) in \( G \), and the expected number of edges in a random graph. This difference is small when \( \lambda \) is small, that is, \( \Delta(G) \) is large. Lemma 3.1 shows that a random walk on an expander graph produces a walk that looks like a walk on a random graph. Another important characteristic of expander graphs is that they have logarithmic diameters.

### 3.3 Cayley hash functions

It is well known that expander family of graphs are used to produce pseudorandom behavior. This pseudorandom behavior is due to the rapid mixing of Markov chains on expander graphs.
The idea is to use multiplicative (semi)groups whose Cayley graph are expander graphs to produce hash functions that are collision-resistant.

In the construction of hash functions from expander Cayley graphs, the input to the hash function gives directions for walking around the graph (without backtracking), and the output of the hash is the ending vertex of the walk. Because of the expander mixing propriety of expander graphs, a random walk on such graphs mixes very fast so the output of a Cayley hash function will be uniform provided the input was uniformly random.

The following general scheme is designed to construct a hash function from an expander Cayley graph [35].

**Defining parameters:**

Let $G$ be a finite (semi)group with a set of generators $S$ that has the same size as the text alphabet $A$. Choose a function: $\pi : A \to S$ such that defines an one-to-one correspondence between $A$ and $S$.

**Algorithm:**

The hash value of the text $x_1 x_2 \ldots x_k$ is the (semi)group element $\pi(x_1)\pi(x_2)\ldots\pi(x_k)$.

One of the advantages of this design is that the computation of the hash value can be easily parallelized due to the associativity property $\pi(xy) = \pi(x)\pi(y)$ for any $x$ and $y$ in the (semi)group. Moreover, considering groups with expander Cayley graphs guarantees equidistribution of the hash values and if the graph has a large girth (the length of the smallest cycle in the graph), the hash function is protected against small modifications of the message input.
3.4 The Tillich-Zémor hash function

J. Tillich and G. Zémor in their 1994 paper [35] proposed a family of hash functions that uses the group of matrices $SL_2$ over a finite field of $2^n$ elements as platform for their design.

Let $n$ be a positive integer and let $p(x)$ be a irreducible polynomial of degree $n$ over $\mathbb{F}_2$. Let $A_0$ and $A_1$ be defined as follow:

$$A_0 = \begin{pmatrix} x & 1 \\ 1 & 0 \end{pmatrix}, \quad A_1 = \begin{pmatrix} x & x+1 \\ 1 & 1 \end{pmatrix}$$

Both $A_0$ and $A_1$ have determinant 1 over $\mathbb{F}_2$. These matrices are the generators of the Tillich-Zémor hash function.

Let $m = m_1m_2\ldots m_k \in \{0,1\}^*$ be a bit string representation of a message and $K := \mathbb{F}_2[x]/(p(x)) \approx \mathbb{F}_{2^n}$. The Tillich-Zémor family of functions $H : \{0,1\}^* \rightarrow SL_2(K)$ is determined by the parameter $n$, the degree of $p(x)$. The hash value of $m$ is just the matrix product

$$H(m_1m_2\ldots m_k) = A_{m_1}A_{m_2}\cdots A_{m_k} \mod p(x).$$

The image of a Tillich-Zémor hash function are elements of the special linear group over the field $K$. This group consists of $2 \times 2$ matrices that have determinant 1 and whose entries are polynomials with coefficients in $\mathbb{F}_2$ and reduced modulo $p(x)$. The operations in this group are ordinary matrix multiplication and matrix inversion.

The Tillich-Zémor hash function, unlike functions in the SHA family, is not a block hash function, i.e., each bit is hashed individually. More specifically, the “0” bit is hashed to the matrix $A_0$, and the “1” bit is hashed to the matrix $A_1$. Then a bit string is hashed simply to the product of matrices $A_0$ and $A_1$ corresponding to bits in this string. For example, the bit string 11100110 is hashed to the matrix $A_1^3A_0^2A_1^2A_0$. 
Since this function only uses basic operations such as addition in a finite field of characteristic 2 with $2^n$ elements, where the parameter $n$ is in the range of 130-170\(^3\), it can be easily implemented allowing fast computations.

The Tillich-Zémor hash function satisfy the homomorphic property, that is, if $x$ and $y$ are two inputs then $H(xy) = H(x)H(y)$. This property allows parallelization and even pre-computations when parts of the message is known in advance.

The construction of Tillich-Zémor function is based on the Cayley graph associated with $SL_2(K)$. The Cayley graph of the group has a large girth, the function is then protect against local modifications of the input. The choice of groups such $SL_2(\mathbb{F}_q)$ by Tillich and Zémor, in particular when $q = 2^n$, is because the matrix generators make it easy to obtain fast hash functions. Moreover, it is relatively easy to obtain Cayley graphs over these groups that have large girth.

To find collisions for the Tillich-Zémor hash function one needs to find two distinct sequences of matrix generators such that the corresponding products coincide in the group $SL_2(K)$. The Tillich-Zémor hash function was proven secure against early attacks. However, Grassl et al. \cite{11} introduced a new and very elegant algorithm that finds collisions for this hash functions. They discovered a particular structure in the hash values of palindromic messages (messages such that their representation in bit strings are the same backward as forward). The attack showed that the Tillich-Zémor function was not collision resistant and consequently not even preimage or second preimage resistant.

\(^3\)later research suggest even larger values for $n$, see \cite{11}
3.5 More examples of Cayley hash functions

The Zémor hash function was the first instance of Cayley hash functions proposed [38]. This hash function has as generators the matrices
\[
\begin{pmatrix}
1 & 1 \\
0 & 1 \\
\end{pmatrix}
\quad \text{and} \quad
\begin{pmatrix}
1 & 0 \\
1 & 1 \\
\end{pmatrix}
\]
and its hash values are elements in $SL_2(\mathbb{F}_p)$ for $p$ prime. The Zémor hash function was broken by Tillich and Zémor using lifting attacks [34].

The LPS hash function was proposed by Charles, Goren and Lauter [7] and is based on the Ramanujan expander graphs of Lubotzky, Philips and Sarnak (1988).

Let $p$ and $l$ be primes, $l$ small and $p$ large, both $p$ and $l$ equal to $1 \mod 4$, and $l$ being a quadratic residue modulo $p$. An LPS graph $X_{l,p}$ is associated to $p$ and $l$ as follows. Let $i$ be an integer such that $i^2 = -1 \mod p$ and the vertices of $X_{l,p}$ are matrices in $G = PSL_2(\mathbb{F}_p)$. The generating set is taken as $S = \{G_j, j = 1, \ldots, l+1\}$, where

\[
G_j = \begin{pmatrix}
g_{0,j} + ig_{1,j} & g_{2,j} + ig_{3,j} \\
g_{2,j} + ig_{3,j} & g_{0,j} - ig_{1,j} \\
\end{pmatrix}, \quad j = 1, \ldots, l+1;
\]

where $(g_{0,j}, g_{1,j}, g_{2,j}, g_{3,j})$ are all the solutions of $g_0^2 + g_1^2 + g_2^2 + g_3^2 = l$, with $g_0 > 0$ and $g_1, g_2, g_3$ even. Note that $S$ is stable under inversion, so the Cayley graph $X_{l,p} = C(G, S)$ is undirected. The LPS hash function is the Cayley hash function associated to $C(G, S)$, starting at the identity. In [36], Tillich and Zémor provide an attack that find collisions for the LPS hash function. The attack is somewhat reminiscent of the density attack (see section 4.2).
Chapter 3. Cayley hash functions

The Morgenstern hash function is a generalization of the LPS hash function from an odd prime \( p \equiv 1 \mod 4 \) to a power of prime \( q \). More specifically, the parameter \( q = 2^k \) is suggested [28].

Let \( q \) be a power of 2 and \( f(x) = x^2 + x + \epsilon \) an irreducible in \( \mathbb{F}_q[x] \). Let \( g(x) \in \mathbb{F}_q[x] \) be irreducible of even degree \( n = 2d \) and let \( \mathbb{F}_{q^n}[x] \approx \mathbb{F}_q[x]/(g(x)) \). The vertices of the Morgenstern graph \( \Gamma_q \) are the elements of \( G = PSL_2(\mathbb{F}_{q^n}) \). Let \( i \in \mathbb{F}_{q^n} \) be a root of the polynomial \( f(x) \). The generating set is defined as \( S = \{ G_j, j = 1, \ldots, q + 1 \} \), where

\[
G_j = \begin{pmatrix} 1 & \gamma_j + \delta_j i \\ (\gamma_j + \delta_j i + \delta_j i)x & 1 \end{pmatrix}, \quad j = 1, \ldots, q + 1;
\]

where \( \gamma_j, \delta_j \in \mathbb{F}_q \) are all the \( q + 1 \) solutions in \( \mathbb{F}_q \) for \( \gamma_j^2 + \gamma_j \delta_j + \delta_j^2 \epsilon = 1 \). Each of the elements \( G_j \) has order 2 and the Cayley graph \( C(G, S) \) is an undirected graph. The Morgenstern hash function is associated to the Cayley Graphs \( \Gamma_q = C(G, S) \), starting at the identity. A cryptanalysis of this hash function can be found in [27].
Chapter 4

Cryptanalysis of Cayley hash functions

4.1 Security of Cayley hash functions

Cayley hash functions have their security properties strongly related to mathematical problems.

Let $G$ be a (semi)group and $S = \{s_1, \ldots, s_k\} \subset G$ be a generating set of $G$. Let $L$ be of polylogarithmic (small) in the size of $G$.

- **Balance problem**: Find an efficient algorithm that returns two words $m_1 \ldots m_l$ and $m'_1 \ldots m'_{l'}$ with $l, l' < L$, $m_i, m'_i \in \{1, \ldots, k\}$ that yield equal products in $G$, that is, $\prod_{i=1}^{l} s_{m_i} = \prod_{i=1}^{l'} s_{m'_i}$

- **Representation problem**: Find an efficient algorithm that returns a word $m_1 \ldots m_l$ with $l < L$, $m_i \in \{1, \ldots, k\}$ such that $\prod_{i=1}^{l} s_{m_i} = 1$.

- **Factorization problem**: Find an efficient algorithm that given any element $g \in G$ returns a word $m_1 \ldots m_l$ with $l < L$, $m_i \in \{1, \ldots, k\}$ such that $\prod_{i=1}^{l} s_{m_i} = g$. 

27
A Cayley hash function is collision resistant if and only if the balance problem is hard in the underlying (semi)group. If the representation problem is hard in the (semi)group, the associated Cayley hash is second preimage resistant and, it is preimage resistant if and only if the corresponding factorization problem is hard in (semi)group [26, 30].

4.2 Attacks against Cayley hash functions

The following represent some possible attacks against Cayley hash functions [11, 26, 30, 34].

4.2.1 Generic attacks

As for the case of any hash function, Cayley hash functions can be vulnerable to exhaustive search attacks solving the factorization problem in time roughly $|G|$ and to birthday attacks solving the balance problem in time roughly $|G|^{1/2}$.

Cayley hash functions are a particular case of the Merkle-Damgård hash functions with compression function $H : G \times \{1, \ldots, k\} \rightarrow G$ sending an intermediary product and a $k$-digit to the next intermediary product. Since this compression function can be efficiently inverted by exhaustive search, the factorization problem can be solved in time roughly $|G|^{1/2}$ with a meet-in-the-middle approach. To avoid this type of attacks is sufficient to choose (semi)groups of sufficiently large size.

Differential cryptanalysis is unlike to work in this category of hash functions, especially if the girth of the Cayley graph is large. In Cayley hash functions, differential attacks are best replaced by subgroup attacks [26].
4.2.2 Subgroup attacks

Let $G$ be a group with subgroup sequence $G = G_0 \supset G_1 \supset G_2 \supset \ldots \supset G_N = \{1\}$ with $|G_i|/|G_{i+1}|$ small for all $i$. Given $S = \{s_1, \ldots, s_k\}$, we can use the subgroup sequence to possibly reach the identity by successively going from $G_i$ to $G_{i+1}$. If successful, this approach solves the representation problem in the group.

The procedure is as follows. Generate random products of the $s_i$’s until we get an element $s_1^{(1)} \in G_1$. Repeat the operations until a set $S_1^{(1)} = \{s_1^{(1)}, \ldots, s_k^{(1)}\}$ that generates all the elements of $G_2$ is obtained. Then recursively repeat the procedure starting from the group $G_1$ and the set $S_1^{(1)}$, and so on. If we use substitutions, we can obtain a representation with elements of $S$. This attack has complexity roughly $\max_i |G_i|/|G_{i+1}|$. Using a meet-in-the-middle strategy, one can reduce its complexity to $\max_i (|G_i|/|G_{i+1}|)^{1/2}$. We can obtain $s_1^{(1)} \in G_1$ more efficiently if random products $g_j$ of the $s_i$ and random products $h_j$ of the $s_i^{-1}$ until getting a couple $(g_j, h_j)$ such that $s_1^{(1)} := g_j h_j^{-1} \in G_1$. These attacks can be extended to solve the factorization problem, as well [30].

4.2.3 Lifting attacks

This powerful technique was exploited by Tillich and Zémor against the first Cayley hash function proposed, the Zémor hash function [34]. This attack used the fact that any matrix of $SL_2(\mathbb{Z}_+)$ is a product of

\[
\begin{pmatrix}
1 & 1 \\
0 & 1
\end{pmatrix}
\quad \text{and} \quad
\begin{pmatrix}
1 & 0 \\
1 & 1
\end{pmatrix}
\]

that is, these matrices generate a dense subset in $SL_2(\mathbb{Z}_+)$. The attack is also enabled by the fact that each $i$-th step of the
Euclidean algorithm $r_{i-2} = q_{i-1}r_{i-1} + r_i$ can be expressed in matrix form as

$$
\begin{pmatrix}
    r_{i-2} \\
    r_{i-1}
\end{pmatrix} =
\begin{pmatrix}
    1 & q_{i-1} \\
    0 & 1
\end{pmatrix}
\begin{pmatrix}
    1 & 0 \\
    q_i & 1
\end{pmatrix}
\begin{pmatrix}
    r_i \\
    r_{i+1}
\end{pmatrix}.
$$

Furthermore,

$$
\begin{pmatrix}
    1 & q \\
    0 & 1
\end{pmatrix} =
\begin{pmatrix}
    1 & 1 \\
    0 & 1
\end{pmatrix}^q
$$

and

$$
\begin{pmatrix}
    1 & 0 \\
    q & 1
\end{pmatrix} =
\begin{pmatrix}
    1 & 0 \\
    1 & 1
\end{pmatrix}^q.
$$

Given any matrix $M = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in SL_2(\mathbb{F}_p)$, the factorization problem can be solved by selecting a matrix $M' = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \in SL_2(\mathbb{Z}_+)$ that reduces to $M$ modulo $p$. To factor the matrix $M'$ we use the Euclidean algorithm is applied to $(A, B)$ in the case that $A \leq B$, else it is applied to $(C, D)$. Since the set of matrices generated by

$$
\begin{pmatrix}
    1 & 1 \\
    0 & 1
\end{pmatrix}
$$

and

$$
\begin{pmatrix}
    1 & 0 \\
    1 & 1
\end{pmatrix}
$$

is dense in the semigroup $SL_2(\mathbb{Z}_+)$, it makes the cryptanalysis of the Zémor hash function simple. This represents an efficient algorithm since there is a high probability of success of obtaining relatively small factorizations of a matrix $M$ in the group [34].

To prevent this type of attack, one should choose generators that generate a very slim (not dense) subset $SL_2(\mathbb{Z})$. The Tillich-Zémor hash function, which was proposed to replace
the Zémor hash function and whose generators are \( \begin{pmatrix} x & 1 \\ 1 & 0 \end{pmatrix} \) and \( \begin{pmatrix} x & x + 1 \\ 1 & 1 \end{pmatrix} \), has density about \( 2^{-n} \).

Using similar technique, an attack was also provided in [36] to find collisions for the LPS hash function, where the group unit element is first lifted into a dense subset of \( SL_2(\mathbb{Z}) \) and then a factorization algorithm is applied in \( SL_2(\mathbb{Z}) \). Moreover, a more elaborated version of the lifting attack is used to break the Tillich-Zémor hash function. The attack against the Tillich-Zémor hash function will be describe in section 4.3.

### 4.2.4 Finding elements of small order

The group \( SL_2(\mathbb{F}_p) \), used in the Zémor hash function, contains some elements of relatively small order in the group. Trying to find messages that hash to one of such elements of small order is the goal of the attack that follows.

First, one must compute the hash values of random messages until obtaining a hash value, which is represented by diagonal matrix. According to Tillich and Zémor in [34], on average about \( \mathcal{O}(p^{0.6}) \) messages are computed in this search. A matrix is diagonalizable over \( \mathbb{F}_p \) if its eigenvalues are in \( \mathbb{F}_p \) and its minimal polynomial has no repeated roots.

If \( m \) is a bit string that hashes to a diagonal matrix \( M \), then \( M \) is similar to \( D = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \), where \( \lambda_1 \) and \( \lambda_2 \) are the eigenvalues of \( M \). Note that \( D^k = \begin{pmatrix} \lambda_1^k & 0 \\ 0 & \lambda_2^k \end{pmatrix} \) and that \( \text{ord}(M) = \text{ord}(D) = \text{lcm}(\text{ord}(\lambda_1), \text{ord}(\lambda_2)) \).

Because \( \lambda_1 \) and \( \lambda_2 \) are in \( \mathbb{F}_p^* \), their orders will divide \( p - 1 \). So with the factorization of \( p - 1 \) one can easily compute the orders of the eigenvalues. Thus factoring \( p - 1 \) must be
feasible in order to use this attack, say, something of order less than $2^{60}$. 

If the $\text{ord}(M) = t$, concatenating $t$ copies of the message $m$ will result in a message that hashes to the identity matrix. Then this concatenation can be inserted into any other message without changing the hash value of the message and thus providing a collision.

The practicality of this attack depends on the length of the concatenated message being short. As described in [34], in the case of $SL_2(\mathbb{F}_p)$ with a randomly chosen $p$, on average this attack yields a message whose hash value is of order less than $O(p^{0.4})$.

This category of attacks can be prevented by choosing a modulo $p$ such that the greatest non-trivial divisor of $p - 1$ is sufficiently large.

4.3 The collision algorithm of Grassl et al.

Despite of resisting attacks for 15 years, Tillich-Zémor family of hash functions have been broken by an attack designed by Grassl et al. [11]. We briefly describe the idea behind the Grassl et al.’s attack.

Let $v = m_1m_2\ldots m_k \in \{0,1\}^k$ be a bit string of length $k$. We denote by $v^r = m_km_{k-1}\ldots m_1$ the reversal bit string of $v$. In Grassl et al.’s attack, to find collisions the authors considered palindromes, that is, bit strings $v \in \{0,1\}^*$ such that $v = v^r$.

Consider the two new matrices: $S_0 = \begin{pmatrix} x & 1 \\ 1 & 0 \end{pmatrix}$ and $S_1 = \begin{pmatrix} x+1 & 1 \\ 1 & 0 \end{pmatrix}$.

Note that both matrices are in $SL_2(K)$ where $K := \mathbb{F}_{2^n}$. Also $S_0 = A_0$ and $S_1 = A_0^{-1}A_1A_0$ where $A_0$ and $A_1$ are the original generators of the Tillich-Zémor function. These matrices are the generators considered for the modified function $H' : \{0,1\}^* \rightarrow SL_2(K)$ defined by

$$H'(m_1m_2\ldots m_k) = S_{m_1}S_{m_2}\cdots S_{m_k} \mod p(x).$$ (4.3.1)
Grassl et al. showed that finding collisions for the Tillich-Zémor scheme is equivalent to finding collisions for the modified function $H'$.

Consider now the function (4.3.1) without reduction $h'$ such that

$$h'(m_1 m_2 \ldots m_k) = S_{m_1} S_{m_2} \cdots S_{m_k}.$$ 

For palindromes of even length, $h'$ gives hash values that are symmetric matrices and using the algorithm presented by Mesirov and Sweet in [20] it is possible to construct $m = m_1 \ldots m_k m_k \ldots m_1 \in \{0, 1\}^{2k}$ such that $h'(0m0) = h'(1m1) + \begin{pmatrix} p^2(x) & p^2(x) \\ p^2(x) & 0 \end{pmatrix}$ (see [11]).

For palindromes of even length, $h'$ gives hash values that are symmetric matrices with entries described by the following proposition.

**Proposition 4.1.** Let $m = m_1 \ldots m_k m_k \ldots m_1 \in \{0, 1\}^{2k}$ be a palindrome of even length. Let $a(0), \ldots, a(k)$ be the following polynomials

$$a(i) = \begin{cases} 1, & \text{if } i = 0; \\ x + m_1 + 1, & \text{if } i = 1; \\ (x + m_i)a(i-1) + a(i-2) & \text{if } 1 < i \leq k. \end{cases}$$

Then $h'(m) = \begin{pmatrix} a^2 & b \\ b & d^2 \end{pmatrix}$ for $a = a(k), d = a(k-1)$ and some $b \in \mathbb{F}_2[x]$. Moreover,
For the given irreducible polynomial \( p(x) \in \mathbb{F}_2[x] \) used to define \( K := \mathbb{F}_{2^n} \), one wants to find a palindrome \( m \) of length \( 2n \) such that \( h'(0m0) + h'(1m1) \) is the zero matrix of dimension 2 over \( K \). From proposition 4.1, the square root of the upper left entry of \( h'(m) \) where \( m \) has even length satisfies a reversed order Euclidean algorithm sequence whose quotients are either \( x \) or \( x + 1 \). These type of sequences are called \textit{maximal length Euclidean sequences}. Mesirov and Sweet [20] showed that when \( a \in \mathbb{F}_2[x] \) is a irreducible polynomial there are exactly two polynomials \( d \) such that \( a \) and \( d \) are the first terms of a maximal length Euclidean sequence and also provided an algorithm to find such polynomials \( d \).

In the Grassl et al.’s algorithm to find collisions, Mesirov and Sweet’s algorithm is applied to the irreducible polynomial \( a = p(x) \) to compute the polynomial \( d \). The corresponding bit sequence \( m_1 \ldots m_n \) can be computed by applying the Euclidean algorithm to \( a \) and \( d \) which will compose the palindrome \( m = m_1 \ldots m_n m_n \ldots m_1 \) and, by proposition 4.1, we have

\[
h'(0m0) = h'(1m1) + \begin{pmatrix} a^2 & a^2 \\ a^2 & 0 \end{pmatrix}.
\]

Reducing this equation modulo \( p(x) \), we obtain \( H'(0m0) \equiv H'(1m1) \), which is a collision of two distinct bit strings under the original Tillich-Zémor hash function.

Not only the attack above provide a way of obtaining collisions for the Tillich-Zémor hash function but also it helped Petit and Quisquater [29] to develop an algorithm to compute preimages for the function.
4.4 Other considerations about the Tillich-Zémor hash functions

Despite the security issue with Tillich-Zémor hash function, it is important to point out that the Grassl et al.’s attack does not invalidate the general scheme proposed by Tillich and Zémor. The key feature in the cryptanalysis of Tillich-Zémor hash function is Mesirov and Sweet’s algorithm [11], which can only be applied to the specific group of matrices used, which is specific to quotients $x$ and $x+1$ in the Euclidean algorithm. Collision and pre-image resistances are obtained if the matrices $A_0$ and $A_1$ are replaced by the matrices $S_0$ and $S_1$.

Petit and Quisquater [29] suggested that security might be recovered by introducing some simple redundancy in the messages and it might even be sufficient to replace $A_0$ by $A_0^2$ or $A_0^3$ and modifying the underlying group. Similar hash functions can be constructed from other noncommutative groups and generators.

In a survey about Cayley hash functions [30], Petit and Quisquater also demonstrated that the Cayley hash function design is still appealing and that it deserves further interest by the cryptography community by showing that the factorization, representation and balance problem in non-Abelian groups still are potentially hard problems for general parameters of Cayley hash functions.

In a recent paper [4], the authors suggested other pairs of matrices, of the form

$$\begin{pmatrix} 1 & k \\ 0 & 1 \end{pmatrix}$$

and

$$\begin{pmatrix} 1 & 0 \\ k & 1 \end{pmatrix}, \quad k \geq 2,$$

with the idea that since these matrices generate a free monoid over $\mathbb{Z}$, there cannot be any short relations over $\mathbb{F}_p$. 
We also offer hashing with a pair of $2 \times 2$ matrices, but these matrices generate a semigroup isomorphic to the semigroup (with respect to composition) of linear functions of one variable over $\mathbb{F}_p$. The corresponding hash functions are very efficient; the time complexity of hashing a bit string of length $n$ with our method is determined by performing at most $2n$ multiplications and about $2n$ additions in $\mathbb{F}_p$. 
Chapter 5

Hashing with compositions of linear functions

5.1 Basic results

We want to show that certain linear functions generate a free semigroup under function composition.

Let \( f_0 = p_0x + q_0 \) and \( f_1 = p_1x + q_1 \) be functions over \( \mathbb{Z} \) where \( p_0, q_0, p_1 \) and \( q_1 \) are elements of \( \mathbb{Z} \).

For a finite bit string \( B = (\epsilon_i)_{i=1}^n \), we denote \( f_B = f_{\epsilon_n} \circ f_{\epsilon_{n-1}} \circ \cdots \circ f_{\epsilon_2} \circ f_{\epsilon_1} \). For the empty bit string of length 0, we define \( f_{\emptyset}(x) = x \). For simplicity, we writing \( f_B = f_{\epsilon_n} f_{\epsilon_{n-1}} \cdots f_{\epsilon_2} f_{\epsilon_1} \) for the composition where the rightmost function is applied first.

**Lemma 5.1.** For every finite bit string \( B \) of length \( n \geq 1 \), \( f_B(x) = \left( \prod_{i=1}^n p_{\epsilon_i} \right) x + \sum_{i=1}^n \left( q_{\epsilon_i} \prod_{k=i+1}^n p_{\epsilon_k} \right) \).

**Proof.** We prove the result by induction on the length \( n \) of the bit string \( B = (\epsilon_i)_{i=1}^n \).

For \( n = 1 \), the result holds because \( f_B(x) = f_0(x) \) or \( f_B(x) = f_1(x) \). Suppose that the result holds for some \( B = (\epsilon_1, \epsilon_2, \ldots, \epsilon_n) \) and consider \( B' = (\epsilon_1, \epsilon_2, \ldots, \epsilon_n, \epsilon_{n+1}) \). Then
Chapter 5. Hashing with compositions of linear functions

\[ f_{B'}(x) = (f_{\epsilon_{n+1}} f_B)(x) = f_{\epsilon_{n+1}} (f_B(x)). \]  

Thus,

\[
f_{B'}(x) = p_{\epsilon_{n+1}} \left[ (\prod_{i=1}^{n} p_{\epsilon_i}) x + \sum_{i=1}^{n} q_{\epsilon_i} \left( \prod_{k=i+1}^{n} p_{\epsilon_k} \right) \right] + q_{\epsilon_{n+1}}
\]

\[
= p_{\epsilon_{n+1}} \left( \prod_{i=1}^{n} p_{\epsilon_i} \right) x + p_{\epsilon_{n+1}} \left( \sum_{i=1}^{n} q_{\epsilon_i} \prod_{k=i+1}^{n} p_{\epsilon_k} \right) + q_{\epsilon_{n+1}}
\]

\[
= \left( \prod_{i=1}^{n+1} p_{\epsilon_i} \right) x + \sum_{i=1}^{n+1} \left( q_{\epsilon_i} \prod_{k=i+1}^{n+1} p_{\epsilon_k} \right) + q_{\epsilon_{n+1}}
\]

\[
= \left( \prod_{i=1}^{n+1} p_{\epsilon_i} \right) x + \sum_{i=1}^{n+1} \left( q_{\epsilon_i} \prod_{k=i+1}^{n+1} p_{\epsilon_k} \right)
\]

If \( f_0 \) and \( f_1 \) are two functions that commute then there are distinct bit strings that yield the same resulting composition. Assume that neither \( f_0 \) nor \( f_1 \) is the identity function. Thus, we investigate what conditions make possible for \( f_0 f_1 \neq f_1 f_0 \):

\[
(f_0 f_1)(x) = (f_1 f_0)(x) \implies p_0 (p_1 x + q_1) + q_0 = p_1 (p_0 x + q_0) + q_1 \implies
\]

\[
p_0 p_1 x + p_0 q_1 + q_0 = p_1 p_0 x + p_1 q_0 + q_1 \implies p_0 q_1 + q_0 = p_1 q_0 + q_1 \implies
\]

\[
q_1 (p_0 - 1) = q_0 (p_1 - 1) \implies \frac{q_1}{q_0} = \frac{p_1 - 1}{p_0 - 1}
\]

Thus, even in case that \( q_0 = q_1 \), the functions \( f_0 \) and \( f_1 \) don’t commute if \( p_0 \neq p_1 \).

5.2 Freeness of semigroups of linear functions

The following is a brief discussion of the results presented by Cassaigne, Harju and Karhumäki in [6] that the finitely generated semigroup of \( 2 \times 2 \) upper matrices over nonnegative integers
under certain conditions is a free semigroup.

The problem (P) of deciding when two upper-triangular matrices $A$ and $B$ with rational entries, that is, the matrices of the form

$$A = \begin{pmatrix} a_1 & a_2 \\ 0 & a_3 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} b_1 & b_2 \\ 0 & b_3 \end{pmatrix}$$

generate a free semigroup is studied.

The authors first noted that in this case one can assume that both the matrices are invertible, otherwise, they satisfy either $A^2BA = ABA^2$ or $B^2AB = BAB^2$, and therefore the semigroup generated by $A$ and $B$ is not free. Also, if one of the matrices is a power of the second then the generated semigroup can be free, but with one free generator. Thus, the term “free semigroup” refers to “free with two generators”.

Furthermore, $A$ and $B$ generate a free semigroup if and only if the semigroup generated by $\lambda A$ and $\mu B$ is free for any nonzero rational $\lambda$ and $\mu$.

In proposition 1, they showed that the problem (P) is decidable if and only if the restricted problem (P’) where

$$A = \begin{pmatrix} a & 0 \\ 0 & 1 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} b & 1 \\ 0 & 1 \end{pmatrix}$$

where $a, b \in \mathbb{Q} \setminus \{-1, 0, 1\}$ is decidable. Thus, the instance of the restricted problem (P’) may be encoded by a pair of rational numbers $(a, b)$. They also showed that the instances $(a, b), (b, a), (\frac{1}{a}, \frac{1}{b})$ and $(\frac{1}{b}, \frac{1}{a})$ are equivalent, that is, they have the same answer. In par-
ticular, each instance of the problem with rational entries is equivalent to one with integer entries if one chooses suitable rational number $\lambda$ and $\mu$ and then consider the semigroup generated by $\lambda A$ and $\mu B$.

Let $A$ and $B$ be as in the restricted problem $(P')$, and let $\nu_p(x)$ denotes the $p$-adic valuation of $x$ defined by $\nu_p\left(p^n \frac{y}{z}\right) = n$ for a prime $p$ and integers $n, y, z$ such that $y$ and $z$ are not divisible by $p$.

**Theorem 5.2** (Cassaigne, Harju and Karhumäki, 1999). Each of following two conditions is sufficient for the semigroup generated by $A$ and $B$ to be free:

1. there is a prime $p$ such that $\nu_p(a) > 0$ and $\nu_p(b) > 0$;
2. $|a| + |b| \leq 1$.

The results proved in [6] show that two different words in the semigroup generated by matrices $A = \begin{pmatrix} a_1 & a_2 \\ 0 & a_3 \end{pmatrix}$ and $B = \begin{pmatrix} b_1 & b_2 \\ 0 & b_3 \end{pmatrix}$ are different if $A$ and $B$ do not commute and $a_1, b_1 \geq 2$.

Let $f(x) = ax + b$ and $g(x) = cx + d$ be two linear functions with integer coefficients. The semigroup generated by these two functions under composition operation is isomorphic to the semigroup of matrices generated by the following $2 \times 2$ matrices:

$$A = \begin{pmatrix} a & b \\ 0 & 1 \end{pmatrix}, \quad C = \begin{pmatrix} c & d \\ 0 & 1 \end{pmatrix}.$$  

Then the semigroup generated by $f(x)$ and $g(x)$ is free if $f(x)$ and $g(x)$ do not commute and $a, c \geq 2$. 
Proposition 5.3. Let \( f(x) = 2x + 1 \) and \( g(x) = 3x + 1 \) be linear functions over \( \mathbb{Z} \). Then \( f \) and \( g \) generate a free semigroup.

Proof. \( f \) and \( g \) don’t commute since \( f(x)g(x) = 6x + 3 \) while \( g(x)f(x) = 6x + 4 \). Applying the inversion \( (a, c) \rightarrow \left( \frac{1}{a}, \frac{1}{c} \right) \), we see that the pair of functions satisfies \( \left| \frac{1}{2} \right| + \left| \frac{1}{3} \right| \leq 1 \). □

As described above, the elements of \( S = \{ f, g \} \) generate a free semigroup, thus the hash values would grow indefinitely with the length of the input message and they would be in a subset of a matrix ring with unique factorization. Therefore, the message could be recovered digit by digit from right to left. Applying modular reductions, some information is lost in the products and the semigroup generated by \( S \) is no longer free and factorization is no longer trivial [30].

If the coefficients of linear functions are now considered as elements of the field \( \mathbb{Z}_p \) for some prime \( p \), the computations become reasonably fast and sizable. Moreover, there cannot be an equality of two different semigroup words in \( f(x) \) and \( g(x) \) unless at least one of the coefficients in at least one of the two words is \( \geq p \).

5.3 Proposed Cayley hash function

We now consider the linear functions \( f_0(x) = 2x + 1 \) and \( f_1(x) = 3x + 1 \) over the field \( \mathbb{Z}_p \) where \( p > 3 \). We define the new hash function by the following.

**Parameter:** \( p > 3 \) is a prime number.

**Algorithm:** Consider the matrices \( A_0 = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix} \) and \( A_1 = \begin{pmatrix} 3 & 1 \\ 0 & 1 \end{pmatrix} \) and the bit assignment \( 0 \mapsto A_0 \) and \( 1 \mapsto A_1 \).

Then compute \( h(b_1b_2\cdots b_k) = A_{b_1}A_{b_2}\cdots A_{b_k} \pmod{p} \).
The corresponding linear function of the product above is of the form \( L(x) = rx + s \in \mathbb{Z}_p \rtimes \mathbb{Z}_p^* \). The hash value \( H(b_1b_2\cdots b_k) = (r + s, s) \).

The definition of the hash value as the pair \((r + s, s)\) instead of simply \((r, s)\) is because \( r \) does not have a uniform distribution on \([0, p)\). Not all integers on \([0, p)\) can be represented as a product of form \(2^n3^m \) modulo \( p \). But \( r + s \) presents a uniform distribution since \( s \) has a uniform distribution. More about the distribution of the pair \((r + s, s)\) is presented in section 5.7.

In this instance of a Cayley hash function, a binary text can be associated to a directed path in the Cayley graph of the semigroup generated by \( \mathcal{S} = \{f_0, f_1\} \), with the identity vertex (identity function) as starting point, and the resulting composition function as its endpoint, which is then associated to the hash value \((r + s, s)\). Appendix A shows an example of the Cayley graph of such matrix semigroup with modulo \( p = 5 \).

This define a family of hash functions depending on \( p \) and they present the following features:

- Variable size input and fixed output size
- The Cayley graphs of the semigroup have relatively large girth
- Efficient computation
- Pseudorandom
- Collision Resistant

Cryptographic hash functions like SHA-256 and SHA-512 have a maximum input message size of \(2^{64} - 1 \) and \(2^{128} - 1 \) bits, respectively \([25]\). Theoretically, our family of hash functions outputs hash values for inputs of any size.

The hash values output by our linear hash function can be encoded as bit strings of fixed length equal to the length of the parameter \( p \). Any initial and final transformations
of input or output of the function do not influence its security. Unless otherwise stated, we will consider our hash function as a function from \( \{0,1\}^* \) to the set of elements formed by the pair \((r+s,s)\) associated to functions generated by products of \(A_0\) and \(A_1\).

5.4 The semigroup generated by \( f_0(x) = 2x + 1 \) and \( f_1(x) = 3x + 1 \) modulo \( p \)

For \( f_0(x) = 2x + 1 \) and \( f_1(x) = 3x + 1 \) and by lemma 5.1, any composition has the form
\[
L_B(x) = 2^n 3^{n_1} x + \sum_{i=1}^{n} 1 \cdot \left( \prod_{k=i+1}^{n} p_{\epsilon_k} \right) \text{ where } p_{\epsilon_i} = 2 + \epsilon_i \text{ with } \epsilon_i \in \{0, 1\} \text{ and } n = n_0 + n_1.
\]

Proposition 5.4. Let \( G_2(\mathbb{Z}_p) = \langle A_0, A_1 \rangle \) over \( \mathbb{Z}_p \) where \( p > 3 \) is prime. Then \( G_2(\mathbb{Z}_p) \) is a group formed by matrices of the form
\[
\begin{pmatrix}
a & b \\
0 & 1
\end{pmatrix}
\]
for some \( a \in \mathbb{Z}_p^* \) and \( b \in \mathbb{Z}_p \).

**Proof.**

i) **Closure:** Let \( M_1 = \begin{pmatrix} a_1 & b_1 \\ 0 & 1 \end{pmatrix} \) and \( M_2 = \begin{pmatrix} a_2 & b_2 \\ 0 & 1 \end{pmatrix} \) be matrices in \( G_2(\mathbb{Z}_p) \).

Then multiplication is closed in \( G_2(\mathbb{Z}_p) \) since \( M_1 M_2 = \begin{pmatrix} a_1 a_2 & a_1 b_2 + b_1 \\ 0 & 1 \end{pmatrix} \) and clearly \( a_1 a_2 \in \mathbb{Z}_p^* \) and \( a_1 b_2 + b_1 \in \mathbb{Z}_p \).

ii) **Associativity:** It follows from the multiplication operation for matrices.

iii) **Identity:** Let \( A_0^n = \begin{pmatrix} a & b \\ 0 & 1 \end{pmatrix} \). By lemma 5.1, \( a \equiv 2^n \mod p \) and \( b \equiv 2^{n-1} + 2^{n-2} + \cdots + 2 + 1 \mod p \). If \( A_0^n = I \) then (I) \( 2^n \equiv 1 \mod p \) and (II) \( 2^{n-1} + 2^{n-2} + \cdots + 2 + 1 \equiv 0 \mod p \).
mod $p$. From (II), $\frac{1 - 2^n}{1 - 2} = 2^n - 1 \equiv 0 \mod p \implies 2^n \equiv 1 \mod p$. Thus, for $n = \text{ord}(2)$ in $\mathbb{Z}_p$, the identity is generated by a power of $A_0$.

iv) **Inverses:** Note that any matrix $M = \begin{pmatrix} a & b \\ 0 & 1 \end{pmatrix}$ generated by $A_0$ and $A_1$ has $a \equiv 2^n 3^m \mod p$ and thus $a \neq 0$ because $\gcd(2, p) = 1$ and $\gcd(3, p) = 1$. It follows that $a = \det(M) \neq 0$ so $M$ is invertible. Moreover, $M^{-1} = \frac{1}{\det M} \begin{pmatrix} 1 & -b \\ 0 & a \end{pmatrix} = \begin{pmatrix} a^{-1} & -a^{-1}b \\ 0 & 1 \end{pmatrix}$, which is of the required form.

Therefore, $G_2(\mathbb{Z}_p)$ is a group.

Not all elements of $\mathbb{Z}_p^*$ can be written as $2^n 3^m \mod p$ for an arbitrary prime $p$. We have that $G_2(\mathbb{Z}_p)$ is a subgroup of $G$ that contains all matrices of the form $\begin{pmatrix} a & b \\ 0 & 1 \end{pmatrix}$ for $a \in \mathbb{Z}_p^*$ and $b \in \mathbb{Z}_p$. The order of $G$ can be found by direct computation. A matrix in this group can have any of the $p - 1$ non-zero vectors $(a, 0) \in \mathbb{Z}_p^2$ as the first column. The second column can be any vector of the $p$ vectors $(b, 1) \in \mathbb{Z}_p^2$. This shows that $|G| = p(p - 1)$ and, thus, $|G_2(\mathbb{Z}_p)|$ divides $p(p - 1)$.

Elements of order 2 and 3 can be recognized as it is shown below.

**Proposition 5.5.** In the group $G_2(\mathbb{Z}_p) = \langle A_0, A_1 \rangle$:

1. The set of elements of order 2 is $\left\{ \begin{pmatrix} p - 1 & b \\ 0 & 1 \end{pmatrix}; b \in \mathbb{Z}_p \right\}$.
2. The elements of order 3 must have $a \in \mathbb{Z}_p^*$ whose order is $\text{ord}_{\mathbb{Z}_p^*}(a) = 3$ and be either of form:

\begin{align*}
    i) & \begin{pmatrix} a & 0 \\ 0 & 1 \end{pmatrix} \\
    ii) & \begin{pmatrix} a & b \\ 0 & 1 \end{pmatrix} \text{ with } b \neq 0 \text{ and } \det(M) \text{tr}(M) \equiv p - 1 \mod p.
\end{align*}

Proof. 1. For any matrix $M = \begin{pmatrix} a & b \\ 0 & 1 \end{pmatrix}$, we have $P_M(M) = 0$ where

\[ P_M(\lambda) = \lambda^2 - (a + 1)\lambda + a \] is the characteristic polynomial of $M$.

\[ \text{ord}(M) = 2 \iff I \neq M \text{ and } I = M^2 = (a + 1)M - aI \]
\[ \iff I \neq M \text{ and } (a + 1)I = (a + 1)M \]
\[ \iff I \neq M, a + 1 = (a + 1)a \text{ and } (a + 1)b = 0 \]
\[ \iff I \neq M, a = \pm 1 \text{ and } ab + b = 0 \]

\[ \iff M = \begin{pmatrix} -1 & b \\ 0 & 1 \end{pmatrix} ; b \in \mathbb{Z}_p \]

\[ \iff M = \begin{pmatrix} p - 1 & b \\ 0 & 1 \end{pmatrix} ; b \in \mathbb{Z}_p. \]
2. By direct computation, we have:

\[ \text{ord}(M) = 3 \iff I \neq M \text{ and } I = M^3 = \begin{pmatrix} a^3 & a^2b + ab + b \\ 0 & 1 \end{pmatrix} \]

\[ \iff I \neq M, a^3 \equiv 1 \mod p \text{ and } a^2b + ab + b \equiv 0 \mod p. \]

i) If \( b = 0 \), then \( M = \begin{pmatrix} a & 0 \\ 0 & 1 \end{pmatrix} \) where \( a \) is such that \( \text{ord}(a) = 3 \) in \( \mathbb{Z}_p^* \).

ii) If \( b \neq 0 \) then

\[ a^2b + ab + b \equiv 0 \mod p \]

\[ b^{-1}(a^2b + ab + b) \equiv b^{-1} \cdot 0 \mod p \]

\[ a^2 + a + 1 \equiv 0 \mod p \]

\[ a(a + 1) \equiv -1 \mod p \]

\[ \det(M) \text{tr}(M) \equiv p - 1 \mod p \]

\[ \square \]

5.5 Girth of the Cayley graph of the semigroup of linear functions

We first define the girth of a directed graph and then give a lower bound on the minimum length of bit strings where a collision may occur using the semigroup generated by the matrices in section 5.3.
Definition 5.1. The directed girth of a graph $G$ is the largest integer $\partial$ such that given any two $v$ and $w$, any pair of distinct directed paths joining $v$ and $w$ will be such that one of those paths has length (that is, the number of edges) $\partial$ or more.

Lemma 5.6. If $L(x) = rx + s \in \mathbb{Z}[x]$ is a word in $f_0(x) = 2x + 1$ and $f_1(x) = 3x + 1$ of length $n$, then $r$ and $s$ are less than or equal to $3^n$.

Proof. We prove by induction on the length $n$.

If $n = 1$, then word $L(x)$ is either equal to $f_0(x)$ or $f_1(x)$. In either case $r, s \leq 3$.

Assume the results holds for length $n > 1$. Then for a word $L'(x)$ of length $n + 1$ we have that it is a composition of a word $L(x) = rx + s$ of length $n$ with either $f_0(x)$ or $f_1(x)$. If $L'(x) = L(x)f_1(x)$ then $L'(x) = r(3x + 1) + s = 3rx + r + s$. Since $3r \leq 3 \cdot 3^n < 3^{n+1}$ and $r + s \leq 3^n + 3^n = 2 \cdot 3^n < 3^{n+1}$, the result of the lemma holds. Similar proof is obtained for the case that $L'(x) = L(x)f_0(x)$.

Proposition 5.7 (Shpilrain and Sosnovski [32]). Let the “0” be hashed to $f_0(x) = 2x + 1$ and the “1” bit be hashed to $f_1(x) = 3x + 1$. If two distinct bit strings $U$ and $V$ hash to the same value, then the length of either $U$ or $V$ is at least $\log_3 p$.

Proof. Suppose the length of $U$ is $n$, and the length of $V$ is $\leq n$. If $L(x) = rx + s$ is the hash of $U$, then by lemma 5.6 both coefficients $r$ and $s$ are less than or equal to $3^n$. Therefore, if $3^n < p$, the hashes of $U$ and $V$ cannot be equal because otherwise they would be equal also over $\mathbb{Z}$, which is impossible.

Thus, if the longer of the two bit strings has length $< \log_3 p$, their hashes cannot be equal over $\mathbb{Z}_p$.

For example, if $p \approx 2^{256}$, the above hash function cannot have collisions unless the length of at least one of the colliding bit strings is at least 162, which corresponds to a lower bound of the girth of the Cayley graph generated by $f_0(x) = 2x + 1$ and $f_1(x) = 3x + 1$ over $\mathbb{Z}_p$. 
The input bit string for our hash function can have an arbitrary length, while the output (with our suggested parameters) is a concatenation of two 256-bit numbers. If we compare this to the Tillich-Zémor hash function [35] and to the hash function in [4], we see that in these previous proposals, if one uses a field of size $2^{256}$, then the size of a hash will be 1024 bits, versus 512 bits in our case, which gives our hash function another advantage as far as performance is concerned.

Another desirable characteristic of a hash function is to be pseudorandom so that the hash function distribute uniformly over $[0, p)$, which minimizes the possibility of collisions. Other properties related to the hash function will be discussed in the following sections.

5.6 Efficiency of composition of linear functions modulo $p$

Let $B = \epsilon_1 \ldots \epsilon_n$ represent a bit string, and $f_0(x)$ and $f_1(x)$ be two linear functions modulo $p$ where $p$ is a prime. We define $L_B(x) = f_{\epsilon_1}(x) \cdots f_{\epsilon_n}(x) \mod p$ to be the function composition of $f_0(x)$ and $f_1(x)$ indexed by $B$. Then $L_B(x) = (mx + b) \mod p$ for some $m, b \in \mathbb{Z}_p$.

A linear function $L(x) = (mx + b) \mod p$ can be uniquely determined by two points. Say that $L(x_1) = y_1$ and $L(x_2) = y_2$ are known for $x_1 \neq x_2$. This gives a system of two equations in the unknowns $m$ and $b$.

$$\begin{cases} mx_1 + b \equiv y_1 \\ mx_2 + b \equiv y_2 \end{cases}$$

From the first equation $b \equiv y_1 - mx_1$ and by substitution in the second equation, we obtain $m(x_1 - x_2) \equiv y_1 - y_2$. Since $x_1 - x_2 \neq 0$ has an inverse in the field $\mathbb{Z}_p$ then $m$ can be recovered uniquely and $b$ as well.

We can use the values of $L_B(0) = b \mod p$ and $L_B(1) = m + b \mod p$ to construct the
whole hash function \( L_B(x) \). Each of the values \( L_B(0) \) and \( L_B(1) \) can be computed by using state transition values in the composition and in parallel. This way, to evaluate the composition performed for a bit string of length \( n \), it is necessary to perform \( 2n \) multiplications and \( 2n \) additions modulo \( p \). In \( \mathbb{Z}_p \), each addition requires \( O(\log p) \) and each multiplication requires \( O(\log^2 p) \) bit operations \([2]\). Thus, if \( p \approx 2^k \) for a fixed \( k \), each operation \( \text{mod } p \) is \( O(k^2) \), which is constant in \( n \). Thus, the number of bits operations needed to evaluate \( H_B(x) \) is \( O(k^2n) \).

Even though all computations are done in \( \mathbb{Z}_p \), we can altogether avoid multiplications during reductions modulo \( p \). This is because coefficients in our linear functions \( f_0(x) = 2x + 1 \) and \( f_1(x) = 3x + 1 \) are quite small, so when we multiply an integer \( x < p \) by 2 or 3 and it becomes greater than \( p \), all we have to do to reduce it modulo \( p \) is to subtract \( p \) or \( 2p \). Therefore, with coefficients at \( x \) as small as 2 and 3, multiplications (and inversions) can be avoided altogether because multiplication by 2 is the same as one addition, and multiplication by 3 amounts to two additions. Thus, with our suggested parameters, one needs to perform between \( 3n \) and \( 4n \) additions and no multiplications to hash a bit string of length \( n \).

It may seem that one inversion is still needed to recover a linear function from its values at two points, but since the choice of the two points \( x_1, x_2 \) is up to us, we can choose \( x_1, x_2 \) to have \( x_2 - x_1 = 1 \) (as above). Hence inversion is actually not needed.

Moreover, computing any Cayley hash function \( H \) can be easily parallelized due to the homomorphic property \( H(MN) = H(M)H(N) \) and the associativity property \( H(MNP) = H(MN)H(P) = H(M)H(NP) \) for any bit strings \( M, N, P \). (Here \( MN \) means a simple concatenation of \( M \) and \( N \).) Thus, a bit string can be split in several pieces, compute the hash of each piece separately, and then multiply out the hashes (in our situation, multiplication is composition of linear functions).
Experimental results of performance

We present below the average times obtained to evaluate the hash values $H_B(x)$ at $x = 0$ for 100 random bit strings $B$ of same bit length. In our experiments we used bit strings of 100, 200, 400, 800, 1000 and 2000 bits. The following average times only measure the composition work for the pair of functions $f_0(x) = 2x + 1$ and $f_1(x) = 3x + 1$. The tests were performed on an Intel Core i7 3.4GHz computer with 8 GB of RAM using Python 3.4.

Table 5.1: Average runtime to evaluate compositions with $f_0(x) = 2x + 1$ and $f_1(x) = 3x + 1$ modulo $p$

<table>
<thead>
<tr>
<th>$p$</th>
<th>Number of bits in $A$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
</tr>
<tr>
<td>$2^{127} - 1$</td>
<td>0.0s</td>
</tr>
<tr>
<td>$2^{137} - 555$</td>
<td>0.0s</td>
</tr>
<tr>
<td>$2^{147} - 387$</td>
<td>0.0s</td>
</tr>
<tr>
<td>$2^{157} - 213$</td>
<td>0.0s</td>
</tr>
<tr>
<td>$2^{167} - 771$</td>
<td>0.0002s</td>
</tr>
<tr>
<td>$2^{177} - 919$</td>
<td>0.0002s</td>
</tr>
<tr>
<td>$2^{187} - 477$</td>
<td>0.0s</td>
</tr>
<tr>
<td>$2^{197} - 775$</td>
<td>0.0s</td>
</tr>
<tr>
<td>$2^{207} - 429$</td>
<td>0.0s</td>
</tr>
<tr>
<td>$2^{217} - 675$</td>
<td>0.0s</td>
</tr>
<tr>
<td>$2^{227} - 721$</td>
<td>0.0s</td>
</tr>
<tr>
<td>$2^{237} - 949$</td>
<td>0.0s</td>
</tr>
<tr>
<td>$2^{247} - 309$</td>
<td>0.0s</td>
</tr>
<tr>
<td>$2^{256} - 1053$</td>
<td>0.0s</td>
</tr>
</tbody>
</table>

From table 5.1, we conclude that our proposed hash function with parameter $p = 2^{256} -$
1053 and without any optimization or parallelization hashes $10^6$ bits in 0.2 seconds in average.

## 5.7 Pseudorandomness

An ideal hash function should generate outputs as random as possible. We evaluate in two ways how pseudorandom this new family of hash functions is, we applied the $\chi^2$ Goodness-of-fit test (Pearson’s $\chi^2$-test) for testing that the distribution of the hash values as elements of $\mathbb{Z}_p$ is uniform [1]. In addition, we also applied the NIST Statistical Test Suite for randomness to a sequence of hash values in binary form [31][24].

### 5.7.1 $\chi^2$-goodness-of-fit test

A $\chi^2$-goodness-of-fit test is used to test whether a frequency distribution fits an expected distribution.

On average the distribution of the hash values on the input data obtained form our hash function should be uniformly distributed across its available range.

For the $\chi^2$-goodness-of-fit test to be used, the following conditions must be true:

1. The observed frequencies are obtained using a random sample.
2. Each expected frequency must be $\geq 5$.

If the conditions are satisfied, the sampling distribution for the test is approximated by a $\chi^2$ distribution with $k - 1$ degrees of freedom, where $k$ is the number of categories. The test statistic for the $\chi^2$-goodness-of-fit test is

$$x^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i}$$

where $O_i$ represents the observed frequency of each category and $E_i$ represents the expected frequency of each category.
If the null hypothesis $H_0$ is true, then by the definition of $\chi^2_\alpha (k - 1)$ the probability of rejecting $H_0$ is $P(x^2 > \chi^2_\alpha (k - 1)) = \alpha$. Thus, the significance level $\alpha$ is the probability of rejecting a true $H_0$ (type I error, as in all other hypothesis tests).

It is recommended in [31] to use a level of significance in the range [0.001,0.01]. We selected $\alpha = 0.01$ to perform a series of $\chi^2$-tests. The critical values $\chi^2_\alpha (k - 1)$ used in our tests were obtained by using the Chi-Square Calculator, a JavaScript developed by J. Walker available in [37]. The null hypothesis $H_0$ tested is that the distributions of $r + s$ and $s$ that form the hash values of the hash function are uniform.

To meet the conditions above for the $\chi^2$-test we used a number of bit strings randomly chosen from a uniform distribution as input in our hash function to generate distributions of $r + s$ and $s$ values. To satisfy the expected frequency for each bin (category) of at least 5, the number of hash values generated were adjusted based on the number of bins used in the test. For each prime used as hash function parameter, the test was applied 100 times and each time with a new distribution of output values. Table 5.2 gives the proportion of sequences of hash values that passed the test at significance level $\alpha = 0.01$. $k$ represents the number of values or intervals used in the tests.
We see that both distributions have an average passing rate of 99%, strongly indicating that they may be indeed uniform.

5.7.2 The NIST Statistical Test Suite

The NIST Statistical Test Suite is a package that include the 15 types of tests, each with a suitable metric needed to investigate the degree of randomness for binary sequences produced

* For larger primes in the test, we reduced the number of bins because computational issues such as memory and runtime in performing the tests. Instead of individual values as bins, we used intervals of integers.
by cryptographic random generators.

The following are deviations from randomness that each test in the NIST Suite detects in binary sequences:

- **Frequency test** - Too many zeroes or ones.
- **Block frequency test** - Too many zeros or ones within a block
- **Runs test** - Large (small) total number of runs indicates that the oscillation in the bit string is too fast (too slow).
- **Longest runs of ones test** - Deviation of the distribution of long runs of ones.
- **Rank test** - Deviation of the rank distribution from a corresponding random sequence, due to periodicity.
- **Discrete Fourier Transform (spectral) test** - Periodic features in the bit stream.
- **Non-overlapping template matchings test** - Too many occurrences of non-periodic templates.
- **Overlapping template matchings test** - Too many occurrences of \( m \)-bit runs of ones.
- **Universal statistical test** - Compressibility (regularity).
- **Linear complexity test** - Deviation from the distribution of the linear complexity for finite length (sub)strings.
- **Serial test** - Non-uniform distribution of \( m \)-length words. Similar to approximate entropy test.
- **Approximate entropy test** - Non-uniform distribution of \( m \)-length words. Small values of ApEn(\( m \)) imply strong regularity.
- **Cumulative sums test** - Too many zeroes or ones at the beginning of the sequence.
- **Random excursions test** - Deviation from the distribution of the number of visits of a random walk to a certain state.
- **Random excursion variant test** - Deviation from the distribution of the total number of visits (across many random walks) to a certain state.
Each of the tests above has a minimum length for the bits strings (hash values in binary) tested. If one wishes to apply all the tests in the suite, a minimum of $10^6$ in length is recommended for the bit strings tested. To meet the minimum requirement, the prime $p = 2 \cdot 191^{66971} + 1$ (see [5]) with bit length of 507,469 was used as parameter for the hash function to generate bit strings of length 1,014,938 that correspond to the concatenation of $r + s$ and $s$. We tested 100 of such bit strings. The level of significance applied used in the suite is $\alpha = 0.01$.

Table 5.3 presents the statistical properties of the hash values as reported by the NIST test suite. According to NIST documentation, a pass rate of 96% is acceptable.
Table 5.3: NIST Statistical Suite Results

<table>
<thead>
<tr>
<th>Number</th>
<th>Statistical test</th>
<th>p-value</th>
<th>Pass rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Frequency</td>
<td>0.779188</td>
<td>98/100</td>
</tr>
<tr>
<td>2</td>
<td>Block frequency</td>
<td>0.262249</td>
<td>97/100</td>
</tr>
<tr>
<td>3</td>
<td>Cumulative sums 1</td>
<td>0.090936</td>
<td>98/100</td>
</tr>
<tr>
<td>4</td>
<td>Cumulative sums 2</td>
<td>0.798139</td>
<td>98/100</td>
</tr>
<tr>
<td>5</td>
<td>Runs</td>
<td>0.739918</td>
<td>100/100</td>
</tr>
<tr>
<td>6</td>
<td>Longest runs of ones</td>
<td>0.816537</td>
<td>99/100</td>
</tr>
<tr>
<td>7</td>
<td>Rank</td>
<td>0.066882</td>
<td>98/100</td>
</tr>
<tr>
<td>8</td>
<td>FFT</td>
<td>0.366918</td>
<td>99/100</td>
</tr>
<tr>
<td>9..156</td>
<td>Non-overlapping templates (148 tests)</td>
<td>0.511260 (mean)</td>
<td>99/100 (mean)</td>
</tr>
<tr>
<td>157</td>
<td>Overlapping template</td>
<td>0.096578</td>
<td>98/100</td>
</tr>
<tr>
<td>158</td>
<td>Universal</td>
<td>0.678686</td>
<td>98/100</td>
</tr>
<tr>
<td>159</td>
<td>Approximate entropy</td>
<td>0.137282</td>
<td>100/100</td>
</tr>
<tr>
<td>160..167</td>
<td>Random excursions (8 tests)</td>
<td>0.313954 (mean)</td>
<td>64/65 (mean)</td>
</tr>
<tr>
<td>168..185</td>
<td>Random excursions (variant - 18 tests)</td>
<td>0.508271 (mean)</td>
<td>64/65 (mean)</td>
</tr>
<tr>
<td>186</td>
<td>Serial 1</td>
<td>0.834308</td>
<td>99/100</td>
</tr>
<tr>
<td>187</td>
<td>Serial 2</td>
<td>0.191687</td>
<td>99/100</td>
</tr>
<tr>
<td>188</td>
<td>Linear complexity</td>
<td>0.798139</td>
<td>97/100</td>
</tr>
</tbody>
</table>

We note that all but one of the tests passed according to the NIST acceptability rate. One of the 148 non-overlapping template tests had pass rate of 95/100 (just below of the recommended 96%) but overall it passed the non-overlapping tests with average 99/100. The bit strings output by our hash also passed each individual random excursions and random excursions variant tests performed, though we only present here the averages results for those tests.
5.8 Security

5.8.1 Generic attacks

Generic attacks use a “birthday paradox” kind of argument and involves a “brute force” search over all bit strings of a length depending on the size of the underlying field. In our situation, since we have a solid lower bound of 162 for the minimum length of colliding bit strings (if \( p \) is on the order of \( 2^{256} \)), a similar approach would involve a brute force search over bit strings of length about 80, which is considered computationally infeasible, so the “birthday paradox” reduction is simply not enough to make the attack feasible with our suggested size of \( p \).

5.8.2 Subgroup attacks

Subgroup attacks can be prevented by choosing the group \( G \) carefully. The minimal requirement is that the cardinality of \( G \) has a “large” factor [34, 30]. One possibility is to use a prime \( p = 2q + 1 \) for some large prime \( q \) (e.g. safe primes). It may be that additional requirements may be needed.

5.8.3 Finding elements of small order

Using a ground group as \( \mathbb{Z}_p \) where the prime \( p \) is such that \( p - 1 = 2q \) with a large prime \( q \) can also help preventing attacks using elements of small order. The goal is to find a bit string that hashes to one of such elements of small order, then insert a number (equal to the order) of copies of this bit string whose matrix is of small order into the hash of another bit string. This is equivalent to insert the identity matrix in the hash value and consequently obtaining a collision with the corresponding messages.
In the case of elements of order 2, which is of the form \( M = \begin{pmatrix} p-1 & b \\ 0 & 1 \end{pmatrix} \), where \( b \in \mathbb{Z}_p \), this means that we looking for a message of “large” length since the proposed hash function is generated by positive powers of \( A_0 \) and \( A_1 \) with the matrix coefficient very small and \( M \) has a large upper left element. The practicality of this attack depends on the resulting concatenating input message being short.

In general, let \( \text{ord}(M) = t \) and \( M = \begin{pmatrix} 2^{n_0}3^{n_1} & b \\ 0 & 1 \end{pmatrix} \), where \( b \in \mathbb{Z}_p \), \( n_0, n_1 > 1 \) are integers and corresponding message of length \( \ell = n_0 + n_1 \). Then \( M^t = I \) implies that at least we have must \( (2^{n_0}3^{n_1})^t \equiv 1 \mod p \implies (2^{n_0}3^{n_1})^t = 1 + kp \) with \( k \geq 1 \implies (2^{n_0}3^{n_1})^t > kp \implies (3^{n_0}3^{n_1})^t > kp \implies 3^{(n_0+n_1)t} > kp \implies 3^{lt} > kp \implies \log 3^{lt} > \log kp \implies \ell > \frac{\log p}{t \log 3} \).

If \( t = 2 \) and \( p \approx 2^{256} \) then \( \ell > 80 \). Hence, one must find a bit string of length at least 80 and insert two copies of it into another bit string to provide a collision. The exhaustive search of a bit string \( m \) of minimum length 80 bits such that its hash value is a matrix \( M \) of order 2 is considered out of reach today.

Also note that if the prime \( p \) is such that \( p - 1 = 2q \) with \( q \) prime and that the matrix group \( |G_2(\mathbb{Z}_p)| \) divides \( p(p-1) = 2pq \), this leaves only matrices of order 2 (small order) in \( G_2(\mathbb{Z}_p) \) to be used in this type of attack.

### 5.8.4 Lifting attacks

Probably the most powerful attack is the lifting attack [34]. The idea is to find a preimage of a given hash in the ambient free (semi)group by “splitting out” one (semi)group generator at a time, so that the “size” of the result would decrease at every step. In our context, where the hash is the pair \((r + s, s)\) corresponding to the function \( L_B(x) = rx + s \) over \( \mathbb{F}_p \), one
would lift $L_B(x)$ to a linear function $Rx + S$ over $\mathbb{Z}$ (i.e., $R = r + k_1p$ and $S = s + k_2p$ for some $k_1, k_2 \in \mathbb{Z}$) and try to multiply it by either $f_0^{-1}(x)$ or $f_1^{-1}(x)$ to decrease one or both coefficients (or, perhaps, to decrease their sum). However, there are two major obstacles that basically make this attack void. The main obstacle is that lifting itself in our situation is by no means unique, and there is no way to tell just by inspection which one is a “good” lifting. This is in sharp contrast with the situation considered in [34] where a “good” lifting can be detected by inspection. Since $L_B(x) = 2^n_03^n_1x + \sum_{i=1}^n 1 \cdot \left( \prod_{k=i+1}^n p_{\epsilon_k} \right)$ with $p_{\epsilon_i} \in \{2, 3\}$, the only necessary condition for a lifting to be “good” in our situation is that the coefficient at $x$ should be of the form $2^n_03^n_1$, but the only condition (visible by inspection) on the constant term $S$ is that $S - 1$ is either divisible by 2 or 3, which leaves a lot of possibilities for lifting.

The other obstacle is that splitting out a generator in this case is not unique either, because at every step of the procedure, one would have the constant term $S$ such that $S - 1$ is divisible by 2 or 3 with non-negligible probability, thus creating a tree of possible reduction sequences, where only one sequence is correct (assuming that the lifting was “good” to begin with, so that there is actually a correct sequence). Thus, even if the attacker was lucky to pick a “good” lifting, he will still have to search over exponentially many (in the length of an input bit string) possible reduction sequences. To be fair though, this problem is relatively insignificant compared to the problem of finding a “good” lifting for the constant term $s$. 
Chapter 6

Semi-primitive roots and the discrete logarithm modulo $2^k$

In this chapter, we establish a connection between semi-primitive roots of the multiplicative group of integers modulo $2^k$ where $k \geq 3$, and the logarithmic base in the algorithm introduced by Fit-Florea and Matula [9] for computing the discrete logarithm modulo $2^k$. Fit-Florea and Matula used properties of the semi-primitive root 3 modulo $2^k$ to obtain their results and provided a conversion formula for other possible bases, that is, other possible semi-primitive roots. We show that their results can be extended to any semi-primitive root modulo $2^k$ and also present a generalized version of their algorithm to find the discrete logarithm modulo $2^k$.

Let $\mathbb{Z}_n^*$ be the multiplicative group of integers modulo $n$. Lee, Kwon, Kang and Shin [17] define an integer $h$ as a semi-primitive root modulo $n$ if the order of $h$ in $\mathbb{Z}_n^*$ is equal to $\phi(n)/2$.

---

$^1\phi(n)$ is the Euler’s totient function
Chapter 6. Semi-primitive roots and the discrete logarithm modulo $2^k$


Suppose that $\mathbb{Z}_n^* \cong C_2 \times C_{\phi(n)/2}$. Then there exist a semi-primitive root $h \in \mathbb{Z}_n^*$ such that

$$\mathbb{Z}_n^* = \left\{ \pm h^i \mod n : i = 0, \ldots, \frac{\phi(n)}{2} - 1 \right\}.$$ 

Since $\mathbb{Z}_{2^k}^* \cong \mathbb{Z}_2 \times \mathbb{Z}_{2^{k-2}}$ for all $k \geq 3$, we can represent $\mathbb{Z}_{2^k}^*$ in terms of its semi-primitive roots as $\mathbb{Z}_{2^k}^* = \langle -1 \rangle \times \langle h \rangle$.

**Corollary 6.2.** For $k \geq 3$ and any semi-primitive root $h$ in $\mathbb{Z}_{2^k}^*$,

$$\mathbb{Z}_{2^k}^* = \{ \pm h^i \mod 2^k : i = 0, \ldots, 2^{k-2} - 1 \}.$$ 

Nathanson in [23, § 3.2] presented results showing that 5 is a semi-primitive root modulo $2^k$ for $k \geq 3$, that is,

$$\mathbb{Z}_{2^k}^* = \{ \pm 5^i \mod 2^k : i = 0, \ldots, 2^{k-2} - 1 \}.$$ 

Fit-Florea and Matula [9] used the semi-primitive root 3 modulo $2^k$ for $k \geq 3$ as the base for their discrete logarithm algorithm. Because of the algebraic properties of semi-primitive roots modulo $2^k$, we can extend their results to find the discrete logarithm modulo $2^k$ using any semi-primitive root in $\mathbb{Z}_{2^k}^*$ as the logarithmic base.

The notation $|m|_{2^k} = j$ represents the congruence relation $m \equiv j \mod 2^k$ where $k \geq 3$ and $0 \leq j \leq 2^k - 1$. The multiplicative inverse $|m^{-1}|_{2^k}$ exists for all odd $m$ with $0 < m \leq 2^k - 1$.

Half of the odd integers modulo $2^k$ can be expressed as positive powers of $h$, that is, as $|h^i|_{2^k}$ for some $i$ while the other half can be expressed as negative powers of $h$, that is, $|-h^i|_{2^k}$. 

Chapter 6. Semi-primitive roots and the discrete logarithm modulo $2^k$

Any $k$-bit integer $x = x_{k-1}x_{k-2} \ldots x_1x_0$ can be represented by a triple $(s, p, e)$ such that $x = |(-1)^s2^ph^e|_{2^k}$ with $s = \{0, 1\}$, $0 \leq e \leq 2^{k-2} - 1$ and $0 \leq p < 2^k$. Denote $\text{dlg}_{(h,k)}(x)$ the discrete logarithm $e$ modulo $2^k$ with respect to base $h$.

Similarly to what was suggested in [9], the discrete logarithm factorization $x = |(-1)^s2^ph^e|_{2^k}$ is uniquely determined by first factoring out the largest power $2^p$ dividing $x$ as the even part factor and employing the discrete logarithm factorization $|(-1)^sh^e|_{2^k-p}$ to provide the odd part factor.

### 6.1 The digit inheritance property

Given an integer with binary representation $x = x_{n-1}x_{n-2} \ldots x_1x_0$ then for $1 \leq k \leq n-1$,

$$|x|_{2^k} = |x_{n-1}x_{n-2} \ldots x_0|_{2^k} = x_{k-1}x_{k-2} \ldots x_0,$$

that is, reduction modulo $2^k$ is obtaining by simply truncating the leading portion of the bit string.

An integer operation $z = x \otimes y$ has the digit inheritance property if for all nonnegative integers $x$ and $y$,

$$|z|_{2^k} = |x|_{2^k} \otimes |y|_{2^k} |_{2^k} \text{ for all } k \geq 1.$$

An integer function $z = f(x)$ has the digit inheritance property if for all nonnegative integers $x$,

$$|z|_{2^k} = |f(|x|_{2^k}) |_{2^k} \text{ for all } k \geq 1.$$

The digit inheritance property states that for operations and functions with this property, the low order $k$ bits of the input arguments determine the low order $k$ bits of the output for all $k \geq 1$. 
Chapter 6. Semi-primitive roots and the discrete logarithm modulo $2^k$

Integer addition and multiplication operations, and the exponentiation and inverse functions satisfy the digit inheritance property.

6.2 Properties of the discrete logarithm modulo $2^k$

In this section, we present mathematical results that will be used to generalize the Fit-Florea and Matula’s discrete logarithm algorithm. These results were proved in [9] using the specific logarithmic base 3 and are provided here for the reader’s convenience. The generalization of the results is possible because depend on the order of the base and so can be adapted to any semi-primitive root.

Lemma 6.3. Let $h$ be a semi-primitive root modulo $2^k$, $k \geq 3$. For any odd residue $A$, either $A$ or its additive inverse $-A$ is congruent to some power of $h$ modulo $2^k$.

Proof. Apply Corollary 6.2.

From now on, let $A$ be an odd residue modulo $2^k$ that can be express as a positive power of a semi-primitive root $h$.

Lemma 6.4 (Generalized version – Fit-Florea and Matula, 2011). Let $B = |A^{-1}|_{2^k}$. Then $\text{dlg}_{(h,k)}(A) + \text{dlg}_{(h,k)}(B) = 2^{k-2}$ for $k \geq 3$.

Proof. If $a = \text{dlg}_{(h,k)}(A)$ and $b = \text{dlg}_{(h,k)}(B)$, then $|A|_{2^k} = |h^a|_{2^k}$ and $|B|_{2^k} = |h^b|_{2^k}$. Because $h$ is a semi-primitive root modulo $2^k$, we have that $|h^{2^{k-2}}|_{2^k} \equiv 1$.

Since $|AB|_{2^k} \equiv 1$, then $|AB|_{2^k} \equiv |h^{2^{k-2}}|_{2^k}$ and $|h^a h^b|_{2^k} \equiv |h^{a+b}|_{2^k}$. It follows that $a + b = 2^{k-2}$.

If the discrete logarithm mod $2^k$ is known for $B$, the multiplicative inverse of $A$, then we can compute $\text{dlg}_{(h,k)}(A) = 2^{k-2} - \text{dlg}_{(h,k)}(B)$. 

Lemma 6.5 (Generalized version – Fit-Florea and Matula, 2011). For \( k > 3 \) and \( h \) any semi-primitive root mod \( 2^k \), we have that \(|h^{2^{k-3}}|_{2^k} = |2^{k-1} + 1|_{2^k}|.

Proof. Let \( A = |2^{k-1} + 1|_{2^k} \). We have that \( A = A^{-1} \) since \(|A^2|_{2^k} \equiv |(2^{k-1} + 1)^2|_{2^k} \equiv |2^{2(k-1)} + 2 \cdot 2^{k-1} + 1|_{2^k} \equiv |1|_{2^k} \).

From Lemma 6.4, \( \text{dlg}_{(h,k)}(A) + \text{dlg}_{(h,k)}(A^{-1}) = 2^{k-2} \implies 2 \cdot \text{dlg}_{(h,k)}(A) = 2^{k-2} \). Therefore, \( \text{dlg}_{(h,k)}(A) = 2^{k-3} \).

\[ \text{Corollary 6.6.} \quad \text{dlg}_{(h,k)}(2^{k-1} + 1) = 2^{k-3} \]

\[ \text{Corollary 6.7 (Digit inheritance of the discrete logarithm).} \quad \text{The low order} \ (i-2) \ \text{bits of the discrete logarithm function} \ \text{dlg}_{(h,k)}(x) \ \text{depend only on the low order} \ i \ \text{bits of the argument} \ x \ \text{for} \ 3 \leq i \leq k. \]

We can apply Lemma 6.5 to compute the discrete logarithm modulo \( 2^i \) of residues \((2^{i-1} + 1) \mod 2^i \) for any \( i \).

Lemma 6.8 (Generalized version – Fit-Florea and Matula, 2011). For any \( k > 3 \),

\[ \text{dlg}_{(h,2^k)}(A) = \text{dlg}_{(h,2^{k-1})}(A) \]

or

\[ \text{dlg}_{(h,2^k)}(A) = \text{dlg}_{(h,2^{k-1})}(A) + 2^{k-3}. \]

Proof. Let \( a' = \text{dlg}_{(h,2^{k-1})}(A) \) and \( a = \text{dlg}_{(h,2^k)}(A) \). So \(|h^{a'}|_{2^{k-1}} = |A|_{2^{k-1}} \) and \(|h^a|_{2^k} = |A|_{2^k} \).

Because of the digit inheritance property, \( h^a \) and \( h^{a'} \) have the same digits whose binary weights are \( 2^{k-2}, \ldots, 2^1, 2^0 \).

If their digits with weight \( 2^{k-2} \) are the same then \(|h^a|_{2^k} = |h^{a'}|_{2^k} \). Therefore, \( a = a' \).
If not the same, we must have that $|h^a + 2^{k-1}|_{2^k} = |h^a|_{2^k}$.

Since $|h^a \times 2^{k-1}|_{2^k} = |2^{k-1}|_{2^k}$, we have that

$$|h^a (2^{k-1} + 1)|_{2^k} = |h^a 2^{k-1} + h^a|_{2^k} = |2^{k-1} + h^a|_{2^k} = |h^a|_{2^k}$$ \((6.2.1)\)

Applying Lemma 6.5,

$$|h^a (2^{k-1} + 1)|_{2^k} = |h^a 2^{k-3} + h^a|_{2^k} = |h^a + 2^{k-3}|_{2^k}$$ \((6.2.2)\)

Comparing \((6.2.1)\) and \((6.2.2)\), $|h^a|_{2^k} = |h^a + 2^{k-3}|_{2^k}$. Therefore, $a = a' + 2^{k-3}$.

This result allows for the computation of $\text{dlg}_{(h,k)}(A)$ one bit at a time.

### 6.3 The Digit-Serial Discrete Logarithm Algorithm

The following results are used to distinguish the positive powers of $h$ from the negative ones.

**Lemma 6.9.** Let $A$ be an odd positive integer with $|A|_{2^k} = a_{k-1}a_{k-2}...a_2a_1$ then:

i) If $A \equiv 1 \mod 4$ then $a_1 = 0$.

ii) If $A \equiv 3 \mod 4$ then $a_1 = 1$.

**Proof.** i) If $A \equiv 1 \mod 4$ then $A = 1 + 4q$ for some integer $q \geq 0$. Because of the digit
inheritance properties of the addition and multiplication modulo $2^k$,

$$|A|_{2^k} = |1|_{2^k} + |4|_{2^k} \cdot |q|_{2^k}_{2^k}$$

$$a_{k-1}a_{k-2} \ldots a_2a_1 = 0 \ldots 0001 + |0 \ldots 0100 \times q_{k-1}q_{k-2} \ldots q_2q_1q_0|_{2^k}$$

$$= 0 \ldots 0001 + q_{k-3}q_{k-4} \ldots q_2q_1q_000$$

$$= q_{k-3}q_{k-4} \ldots q_2q_1q_001$$

Thus, $a_1 = 0$.

ii) Similarly for the case $A \equiv 3 \mod 4$, we have that $|A|_{2^k} = |3|_{2^k} + |4|_{2^k} \cdot |q|_{2^k}_{2^k}$. It follows from $|3|_{2^k} = 0 \ldots 0011$ that $|A|_{2^k} = a_{k-3}a_{k-4} \ldots a_2a_1a_01$. Therefore, $a_1 = 1$. \hfill \Box

**Theorem 6.10.** Let $h$ be a semi-primitive root mod $2^k$, $k \geq 3$. For all positive powers of $h$ with bit string $|A|_{2^k} = a_{k-1}a_{k-2} \ldots a_2a_1$ we have that:

i) If $h \equiv 1 \mod 4$ then $a_1 = 0$.

ii) If $h \equiv 3 \mod 4$ then $a_2 = 0$.

**Proof.** i) If $h \equiv 1 \mod 4$ then $h^i \equiv 1 \mod 4$ for any $i \in \mathbb{N}$. By lemma (6.9), we must have $a_1 = 0$.

ii) If $h \equiv 3 \mod 4$ then $h^2 \equiv 1 \mod 4$. So the powers of $h$ are of the form $h^{2i} \equiv 1 \mod 4$ or $h^{2i+1} \equiv 3 \mod 4$ for all $i \in \mathbb{N}$. Both 1 and 3 have binary representation with $a_2 = 0$. \hfill \Box

The bit strings of the negative powers of $h$ are two’s complements of the positive powers, hence if $h \equiv 1 \mod 4$ then $a_1 = 1$, and if $h \equiv 3 \mod 4$ then $a_2 = 1$.

This was noted in [9] for the semi-primitive root 3 where positive powers of 3 mod $2^k$ have binary digit $a_2 = 0$, while the negative powers of 3 mod $2^k$ have binary digit $a_2 = 1$. 


Corollary 6.11. Let \(|h|_{2^k} = h_{k-1}h_{k-2} \ldots h_2h_1\) be a semi-primitive root modulo \(2^k\). For all positive powers of \(h\) modulo \(2^k\) whose binary representation is \(|A|_{2^k} = a_{k-1} \ldots a_2a_1\):

\[
\text{If } h_1 + 1 = \begin{cases} 
1 & \text{then } a_1 = 0 \\
2 & \text{then } a_2 = 0.
\end{cases}
\]

Proof. This follows from lemma (6.9) and theorem (6.10). \(\square\)

Therefore, one can identify if an integer \(A\) modulo \(2^k\) is a negative power of \(h\) by checking if it satisfies one of the following conditions:

i) \(h_1 = 0\) and \(a_1 = 1\)

ii) \(h_1 = 1\) and \(a_2 = 1\)

In algorithm 1 below, let \(|B|_{2^k} = |A^{-1}|_{2^k}\) and \(b = \text{dlg}_{(h,k)}(B)\).

Algorithm 1 finds the discrete logarithm modulo \(2^k\) whose base is a semi-primitive root. With the present results there is no need for conversion to find the discrete logarithm for bases other than 3 and the algorithm remains efficient.
Algorithm 1 Generalized FitFlorea-Matula DLG Digit-serial Algorithm

Input: Odd integer $|A|_{2^k} = a_{k-1}a_{k-2} \ldots a_2a_1$

Semi-primitive root $|h|_{2^k} = h_{k-1}h_{k-2} \ldots h_2h_1$

Output: The factorization $(s, e)$ of $A$ as $A \equiv \langle -1 \rangle^s h^e$

1: $B := 1$; \hspace{1cm} \triangleright \text{Binary representation of 1 with } k \text{ bits}
2: $b := 0$; \hspace{1cm} \triangleright \text{Binary representation of 0 with } k - 2 \text{ bits}
3: if $(h_1 = 0$ and $a_1 = 1$) or $(h_1 = 1$ and $a_2 = 1)$ then \hspace{1cm} \triangleright \text{Identify the cases that } A \text{ is a}
4: \hspace{2cm} \triangleright \text{negative power of } h
5: \hspace{4cm} P := \langle -A \rangle_{2^k}$; \hspace{1cm} \triangleright \text{The two complement of } A
6: \hspace{3cm} s := 1;
7: else
8: \hspace{4cm} P := \langle A \rangle_{2^k}$;
9: \hspace{4cm} s := 0;
10: end if
11: if $|P|_{2^3} = |h|_{2^3}$ then \hspace{1cm} \triangleright \langle P \rangle_{2^3} = 001 \text{ or } \langle P \rangle_{2^3} = \langle h \rangle_{2^3} \text{ are the only possibilities}
12: \hspace{4cm} B := \langle A \rangle_{2^k}$;
13: \hspace{4cm} b := 1;
14: end if
15: $P := \langle P \times B \rangle_{2^k}$; \hspace{1cm} \triangleright \text{Binary multiplication of } k\text{-digit numbers}
16: \textbf{for } i \text{ from 3 to } k - 1 \text{ do}
17: \hspace{1cm} if $p_i = 1$ then
18: \hspace{2cm} $b := b + 2^{i-2}$; \hspace{1cm} \triangleright \text{This flips the } (i - 1)\text{-th bit of } b
19: \hspace{2cm} B := \langle B \times h^{2^{i-2}} \rangle_{2^k}$;
20: \hspace{2cm} $P := \langle P \times h^{2^{i-2}} \rangle_{2^k}$;
21: \hspace{1cm} end if
22: \textbf{end for}
23: $e = \langle 2^{k-2} - b \rangle_{2^k-2}$
24: return $(s, e)$
Chapter 7

Products of powers of semi-primitive roots equal to 1 in the ring of integers modulo $2^{127}$

In this chapter we consider the following problem involving semi-primitives modulo $2^{127}$.

Let $o_5 = \text{ord}_{\mathbb{Z}/2^{127}\mathbb{Z}}(5)$ and $o_3 = \text{ord}_{\mathbb{Z}/2^{127}\mathbb{Z}}(3)$ be the orders of 5 and 3 in the multiplicative ring $\mathbb{Z}/2^{127}\mathbb{Z}$, respectively. Then 5 and 3 are semi-primitive roots modulo $2^{127}$ because $o_5 = o_3 = 2^{125}$.

**Problem 7.1.** Consider the set $E = \{(k, l) \in (\mathbb{Z}/2^{126}\mathbb{Z})^2 \mid 5^k \cdot 3^l \equiv 1 \mod 2^{127}\}$. Find a lower bound for the set $M = \{k + l \mid (k, l) \in E \text{ and } k, l > 0\}$.

Problem 7.1 can be generalized as finding the minimal sum of exponents $k$ and $l$ such that $h_1^k \cdot h_2^l \equiv 1 \mod 2^k$ where $h_1$ and $h_2$ are semi-primitives for $k \geq 3$.

Using a “birthday paradox” argument, we can heuristically obtain an upper limit for this problem. Since $\gcd(5, 2^{127}) = 1$ and $\gcd(3, 2^{127}) = 1$, we can consider our modular equation in the form $5^k \equiv 3^{-l} \mod 2^{127}$. Among the first $m_1$ values of $5^k$ and the first $m_2$ values of
Chapter 7. Products of powers of semi-primitive roots equal to 1 in the ring of integers modulo \(2^{127}\)

3\(^{-l}\), we have \(m_1m_2\) different potential collisions. Treating these as independent events we can expect each of them to yield an actual collision with probability \(\frac{1}{2^{127}}\), that is, within approximately \(2^{127}\) potential collisions we should expect an actual collision. Since \(m_1 + m_2\) is minimized for a given value of \(m_1m_2\) when \(m_1 = m_2\), then we should expect the minimum value of \(m_1 + m_2\) to occur where \(m_1 \approx m_2\). Since \(m_1 \approx m_2\) yields \(m_1 \approx m_2 \approx 2^{64}\) and \(m_1 + m_2 \approx 2^{65}\). As long as \(m_1 \gg \log_2 2^{127} \approx 55\) and \(m_2 \gg \log_3 2^{127} \approx 80\), then we can expect the sets \(\{5^k \mid 0 \leq k < m_1\}\) and \(\{3^{-l} \mid 0 \leq l < m_2\}\) to be roughly equidistributed \(\mod 2^{127}\).

Nevertheless, a lower value can be obtained on using other imposed relations between the exponents \(k\) and \(l\).

The set \(E = \{(k,l) \in \mathbb{Z}/2^{125}\mathbb{Z})^2 \mid 5^k \cdot 3^l \equiv 1 \mod 2^{127}\}\) can be interpreted as a subspace of \((\mathbb{Z}/2^{125}\mathbb{Z}) \times (\mathbb{Z}/2^{125}\mathbb{Z})\).

Lemma 7.2. Given \(k,l \in \mathbb{Z}/2^{n-1}\mathbb{Z}\) such that \(5^k \cdot 3^l \equiv 1 \mod 2^n\), then \(5^k \cdot 3^l \equiv 1 \mod 2^{n-1}\).

Proof. Suppose that for \(k,l \in \mathbb{Z}/2^{n-1}\mathbb{Z}\) we have \(5^k \cdot 3^l \equiv 1 \mod 2^n\). Then:

\[
\begin{align*}
5^k \cdot 3^l &= 1 + t \cdot 2^n, \quad \text{for some } t \in \mathbb{Z} \\
5^k \cdot 3^l &= 1 + t \cdot 2 \cdot 2^{n-1} \\
5^k \cdot 3^l &= 1 + t' \cdot 2^{n-1} \\
5^k \cdot 3^l &\equiv 1 \mod 2^{n-1}
\end{align*}
\]

In general, the converse is not true. In the case of the modulus \(2^n\), given a solution to \(5^k \cdot 3^l \equiv 1 \mod 2^{n-1}\), we can try to lift \((k,l)\) from \(\mathbb{Z}/2^{n-1}\mathbb{Z}\) to \(\mathbb{Z}/2^n\mathbb{Z}\) by applying Lemma 6.8.
Chapter 7. Products of powers of semi-primitive roots equal to 1 in the ring of integers modulo $2^{127}$

**Fact 7.3.** Let $c = 4064729092413185736448652556727923386$, then the set $E' = \{(cm, 2m) \mid m \in \mathbb{Z}\}$ is a subset of $E$.

With this, we can write $k + l = (cm \mod 2^{125}) + (2m \mod 2^{125})$ with $0 < m < 2^{125}$.

Some basic results related to the exponents in problem 7.1 are shown below. The notation $\text{ord}(a)$ denotes the order of the element $a$ in the ring $\mathbb{Z}/2^{127}\mathbb{Z}$.

**Lemma 7.4.** Let $k$ and $\ell$ be integers such that $5^k \cdot 3^\ell \equiv 1 \mod 2^{127}$. Then $k$ and $\ell$ must be even.

*Proof.* First consider the case when $k = \ell$. Then $5^k \cdot 3^\ell \equiv 1 \mod 2^{127} \implies (5 \cdot 3)^k \equiv 1 \mod 2^{127} \implies 15^k \equiv 1 \mod 2^{127} \implies \text{ord}(15) \mid k$. Since $\text{ord}(15) = 2^{123}$, we have that $2^{123} \mid k$. Thus $k$ is even and so is $\ell$.

Now we consider when $k \neq \ell$. Without loss of generality, we take $k > \ell$, say $k = \ell + m$ for some positive integer $m$. Hence,

\[
\begin{align*}
5^k \cdot 3^\ell &\equiv 1 \\
5^{\ell+m} \cdot 3^\ell &\equiv 1 \\
5^\ell \cdot 3^\ell \cdot 5^m &\equiv 1 \\
15^\ell &\equiv 5^{-m} \\
(15^\ell)^{\text{ord}(15)} &\equiv (5^{-m})^{\text{ord}(15)} \\
1 &\equiv (5^{-1})^{m \cdot \text{ord}(15)}  
\end{align*}
\]

Since $\text{ord}(5^{-1}) = \text{ord}(5) = 2^{125}$ and from congruence (7.0.1) we see that $\text{ord}(5^{-1}) \mid m\text{ord}(15)$. Then $2^{125} \mid m \cdot 2^{123} \implies m \cdot 2^{123} = 2^{125} \cdot u$, for some $u \in \mathbb{Z} \implies m = 2^2 \cdot u \implies m$ is even.

Because $m$ is even and $m = k - \ell$, both $k$ and $\ell$ are even or both are odd.

Suppose that both $k$ and $\ell$ are odd.
Since \( \text{ord}(5) = \text{ord}(3) = 2^{125} \), let \( o \) denote the orders of the elements 5 and 3. The congruences \( 5^k \cdot 3^\ell \equiv 1 \mod 2^{127} \) and \( 5^o \cdot 3^o \equiv 1 \mod 2^{127} \) imply that \( 5^k \cdot 3^\ell \equiv 5^o \cdot 3^o \). Thus, 
\[ 5^{k-o} \equiv 3^{o-\ell}. \]
So the order of \( 5^{k-o} \) and the order of \( 3^{o-\ell} \) must be the same in \( \mathbb{Z}/2^{127}\mathbb{Z} \).

Thus,
\[
5^{k-o} \equiv 3^{o-\ell} \implies \frac{o}{\gcd(o,k-o)} = \frac{o}{\gcd(o,o-\ell)} \implies \gcd(o,k-o) = \gcd(o,o-\ell)
\]

Let \( d = \gcd(o,k-o) \). Thus, \( d \) divides \( o, k-o \) and \( o-\ell \). If \( d \mid o \) then \( d \mid 2^{125} \). Hence \( d \) is even and consequently \( k-o \) and \( o-\ell \) are also even. By assumption, \( k \) and \( \ell \) are odd, then
\[ k-o = (2q+1) - 2q' = 2(q-q') + 1 \]
for some \( q, q' \in \mathbb{Z} \), which is a contradiction. Similar contraction is obtained for \( o-\ell \).

**Corollary 7.5.** \( k+\ell \) is divisible by 4.

**Proof.** If \( k = \ell \) and using the same argument in the proof above, then \( \text{ord}(15) = 2^{123} \mid k \).

Hence, \( 4 \mid k \) and \( 4 \mid \ell \). Consequently, \( 4 \mid (k+\ell) \). Suppose that \( k \neq \ell \). Without loss of generality, we assume that \( k > \ell \), that is, \( k = \ell + m \) for some positive integer \( m \).

According to Lemma 7.4, \( k, \ell \) and \( m \) are even. Thus,
\[
k + \ell = \ell + \ell + m = 2q + 2q + m, \quad q \in \mathbb{Z}
\]
\[
= 4q + m \tag{7.0.2}
\]

Also in the proof of Lemma 7.4, we showed that \( m \) is of the form \( m = 2^2u \) for some \( u \in \mathbb{Z} \).

Therefore, \( 4q + m = 4q + 4u \implies 4 \mid (k+\ell) \).

Using the results about the discrete log with base semi-primitive roots from previous chapter, we can lift a solution of \( 5^k3^2 \mod 2^3 \) to find that \( 5^o3^2 \equiv 1 \mod 2^{127} \) using the following “depth-first search” algorithm.
Algorithm 2 Recursive algorithm that finds exponent modular with two generators mod $2^{127}$

**Input:** $5^k \cdot 3^2 \equiv 1 \mod 2^3$, for some $k \in \mathbb{Z}_{>0}$

**Output:** $5^c \cdot 3^2 \equiv 1 \mod 2^{127}$, $c \in \mathbb{Z}_{>0}$

1: Let $k$ be such that $5^k 3^2 \equiv 1 \mod 2^3$ ($k = 2$);
2: $n = 4$;
3: while $n \leq 127$ do
4: if $5^k 3^2 \equiv 1 \mod 2^n$ then
5: $n \rightarrow n + 1$;
6: else
7: $k \rightarrow 2^{n-3} + k$; \hspace{1cm} $\triangleright$ Applying lemma 6.8
8: end if
9: end while
10: return $k$

The algorithm returns $c = 40647290924413185736448652556727923386$. Now we can use continued fractions of $c/2^n$ to obtain small values of $(cm \mod 2^n)$ and consequently of $l + k$.

We want to obtain the smallest value of $r$ such that $c = i \cdot r$ for some $i \in \mathbb{Z}$ and also $r \mid 2^n$, that is, $2^n = j \cdot r$ for some $j \in \mathbb{Z}$. Therefore, $\frac{c}{2^n} = \frac{i}{j}$.

Using a simple and fast SAGE algorithm, we have that the continued fraction of $c/2^{127}$ is $[0, 2, 10, 1, 3, 4, 37, 1, 1, 6, 1, 58, 1, 4, 14, 5, 7, 1, 1, 1, 3, 1, 1, 8, 1, 9, 12, 1, 8, 1, 1, 1, 2, 1, 5, 2, 36, 1, 1, 1, 3, 1, 5, 1, 1, 6, 3, 1, 4, 1, 8, 1, 2, 4, 4, 1, 1, 1, 2, 1, 1, 6, 1, 16, 1, 18, 1, 3, 1, 4, 4, 1, 61, 1, 10]$.

The lowest values of $k$ and $l$ found so far are

\[ l_o = 11726533429350798020 \]
\[ k_0 = 391079140617450804 \]
\[ l_0 + k_0 = 12117612569968248824 \]

For the values above we have $2^{63} < l_0 + k_0 < 2^{64}$, which so far is in accordance with the “birthday paradox” argument.

An exhaustive search on small values of $k$ and $l$ was also attempted. Applying lemma
7.4 and corollary 7.5, we searched even exponents whose sum is a multiple of 4 in the range $0 < k, l \leq 2^{20}$ but none of such exponents resulted in the congruence $5^k \cdot 3^l \equiv 1 \mod 2^{127}$. Therefore, it remains an open problem to find a lower bound on $k+l$ such that the congruence above is satisfied.
Chapter 8

Conclusion

Cayley hash functions are elegant cryptographic hash functions constructed from Cayley graphs. We have described a new Cayley hash where “0” and “1” bits are hashed by linear functions over $\mathbb{F}_p$. We suggested the pair of linear functions $f_0(x) = 2x + 1$ and $f_1(x) = 3x + 1$.

Our proposal is very efficient and it is possible to compute the hash values without even using multiplications, hence performing between $3n$ and $4n$ additions in $\mathbb{F}_p$ to hash a bit string of length $n$. In addition, the computation of any Cayley hash function can be easily parallelized, which can improve the efficiency of our proposal. This new proposed hash function outputs values of length $2\log p$ while other proposals also using matrices over $\mathbb{F}_p$ to hash output values of length $4\log p$.

We also evaluated how random the outputs of our hash function are. It has successfully passed all the pseudorandomness tests in the NIST Statistical Test Suit and the Pearson’s goodness of fit test performed.

A lower bound of $\log_3 p$ for the girth of the Cayley graph of the semigroup generated by $f_0(x)$ and $f_1(x)$ is given, that is, a lower bound on the length of bit strings that may result in collisions. If $p$ is on the order of $2^{256}$ (or larger), then with the suggested pair of linear
functions, our hash function does not have collisions unless the length of at least one of the colliding bit strings is at least 162.

After analyzing known attacks on Cayley hash functions succeeding against our proposed hash, we suggested also that the prime $p$ be such that $p - 1 = 2q$ where $q$ is prime. With $p$ of this form, attacks using subgroup structure and elements of small order can be prevented. In our analysis, we also determined that there are obstacles that make the lifting attack difficult against our hash function since it seems that there is no clear way of finding a “good” lifting in this scenario. Therefore, we conclude there is no visible threat to the security of our proposed hash function.

We concluded this work with the presentation of the discrete logarithm algorithm described by Fit-Florea and Matula in [9] and therein results extended to any semi-primitive root modulo $2^k$ as logarithmic base. The adjustments do not change the efficiency of the algorithm, which requires $O(k)$ number of binary additions and multiplications. With the general results presented here, the discrete logarithm modulo $2^k$ can be computed directly for bases other than 3.
Appendices
Appendix A

Cayley Graph for $p = 5$
Figure A.1: Cayley graph of matrix semigroup generated by $f(x) = 2x + 1$ and $g(x) = 3x + 1$ modulo $p = 5$. 
Appendix B

Python algorithm for the runtime of evaluating compositions of modular linear functions

```python
# This function computes the runtime to evaluate the composition of linear functions f=ax+b and g=cx+d
# Python

o = open('summary_primes.txt', 'w');
text = "Summary_primes_python.txt";
o.write(text); o.write('
'); o.write('
'); o.write('
'); #
text = "Evaluation of the hash H(x) at x=0";
o.write(text); o.write('
'); o.write('
'); o.write('
'); #
text0="comptime_prime_";
#
import time
from random import *
#
# Function that generates a random bit string of length m
```
Appendix B. Python algorithm for the runtime of evaluating compositions of modular linear functions

```python
def randBinList(m):
    B=[randint(0,1) for t in range(1,m+1)];
    return B

# Moduli
primes=[2**127-1, 2**137-555, 2**147-387, 2**157-213, 2**167-771,
    2**177-919, 2**187-477, 2**197-775, 2**207-429, 2**217-675,
    2**227-721, 2**237-949, 2**247-309, 2**256-1053];

# str_primes=['2ˆ127−1', '2ˆ137−555', '2ˆ147−387', '2ˆ157−213',
# '2ˆ167−771', '2ˆ177−919', '2ˆ187−477', '2ˆ197−775', '2ˆ207−429',
# '2ˆ217−675', '2ˆ227−721', '2ˆ237−949', '2ˆ247−309', '2ˆ256−1053'];
# len_primes=len(primes);
#
# for ind in range(len_primes):
#    #modulo
#    Mod=primes[ind];
#    str_mod=str_primes[ind];
#    print("Modulo "+str_mod);
#    # f=ax+b
#    a = 2; b = 1;
#    text1=", "+str(a)+"x+"+str(b)+"mod "+str_mod+"-> bit 0 ";
#    o.write(text1);o.write('\n');
#    a = a%Mod; b = b%Mod;
#    # g=cx+d
#    c = 3; d = 1;
#    text2="g ="+str(c)+"x +"+str(d)+"mod "+str_mod+"-> bit 1 ";
#    o.write(text2);o.write('\n');
#    c = c%Mod; d = d%Mod;
#    text3="***************************";
#    o.write(text3);o.write('\n');o.write('\n');
#    #number of random bit strings generated.
#    n=100;
#    #lengths of the bit strings
#    L=[100, 200, 400, 800, 1000, 2000];
#    #for i in range(len(L)):
#        l=L[i]; #length of the strings
#        print(l); textL="_length_";
#        text4=text0+str_mod;
#        text4=text4+textL+str(l)+".txt"
#        ol = open(text4 , 'w');ol.write(text4);
#        ol.write('\n');ol.write('\n');
#        ol.write(text1);ol.write('\n');
#        ol.write(text2);
#        ol.write('\n');ol.write(text3);
```
Appendix B. Python algorithm for the runtime of evaluating compositions of modular linear functions

```
o1.write(\n'); o1.write(\n');
text5= "Length of the composition is "+str(l);
o1.write(text5); o1.write(\n'); o1.write(\n');
o.write(text5); o.write(\n');
#
meanTime=0;
for j in range(n):
    B[randBinList(l)];
o1.write(str(B)); o1.write(\n');
time_=0;
X0=0; X1=1;
#time performing the composition
t0 = time.time();
for k in range(l):
    if B[k]==0:
        X0=(a*X0+b)%Mod;
        X1=(a*X1+b)%Mod;
    else:
        X0=(c*X0+d)%Mod;
        X1=(c*X1+d)%Mod;
    t1 = time.time();
    time_=t1-t0;
    #o1.write(str(X0)); o1.write(\n');
    #o1.write(str(X1)); o1.write(\n');
    #o1.write('\{:20f\}'.format(time_));
    o1.write(" s"); o1.write(str(time_));
    o1.write(" s"); o1.write(\n');
    #o1.write(str(t0)); o1.write(" s");
    #o1.write(\n');
    #o1.write(str(t1)); o1.write(" s");
    #o1.write(\n');
    meanTime=meanTime+time_;
    # Average time for n random strings of length l
    meanTime=meanTime/n;
text7="Average time for compositions = ",
o1.write(text7); o1.write(str(meanTime));
o1.write(\n');
    o.close();
o.write(text7); o.write(str(meanTime));
o.write(" sec"); o.write(\n');
print("THE END!!!")
o.close();
```
Appendix C

SAGE Algorithm for generating sequences modular linear hash values

```python
### Setting
# Modulo
Mod = 2**1257787 - 1;
str_mod = "2^1257787 - 1";
# Format the number of bits in the modulo for the printing file
string_ = '{0:01257787b}'
# Indexation used to create the output files names
Range_X = range(1, 2);
# n=100#number of random bit strings to be generated
l = 1500000; # lengths of the bit strings
#
# f=ax+b—> bit 0
# a = 2; b = 1;
#
# g=cx+d—> bit 1
c = 3; d = 1;
```
Appendix C. SAGE Algorithm for generating sequences modular linear hash values

### Preamble

```python
import numpy as np
from random import *

# Function that generates a random bit string of length m
def randBinList(m):
    B=[randint(0,1) for t in range(1,m+1)];
    return B
```

### Main function

```python
for IndeX in Range_X:
    # txt files
    o1=open(str(IndeX)+"Const"+str_mod+"_inputLeng_
    +str(l)+".txt ", 'w ');
    o2=open(str(IndeX)+"Const"+str_mod+"_inputLeng_"
    +str(l)+".txt ", 'w ');
    o1.write('

n
n'); o2.write('

n');
    f=[a,b]; #f=a*x+b as a vector
    g=[c,d]; #g=c*x+d as a vector
    for i in range(n):
        B=randBinList(1);
        comp=[1,0]; #identity function comp=1*x+0
        for k in range(1):
            if B[k]==0:
                comp=[((comp[0]*f[0])%Mod,
                ((comp[0]*f[1])%Mod+comp[1])%Mod];
            else:
                comp=[((comp[0]*g[0])%Mod,
                ((comp[0]*g[1])%Mod+comp[1])%Mod];
        o1.write(string_.format(comp[0])); o1.write('t ');
        o2.write(string_.format(comp[1])); o2.write('t ');
        print(i); print(" ");
        o1.close();
        o2.close();
        print("THE END!!!")
```
Appendix D

Python code for the Chi-square test

```python
### Chi-Square Testing
###
### Python
###
#### Settings

# Modulo
Mod = 2**20 - 153;
str_mod = "2^20 - 153";

# Indexation used to create the output files names
Range_X = range(1, 101);

# number of hash values generated (number of samples observed)
n = 8000000

# lengths of the bit strings input for the composition function
l = 100;

# Critical values at 99.
x0 = [1051793.5894] # Mod number of bins => dof = Mod - 1

# main txt file
o = open("Const_term_uniformity_testing"+str_mod+".txt","w");
text="Const_term_uniformity_testing"+str_mod+".txt"
o.write(text); o.write("\n \n")
```

85
Appendix D. Python code for the Chi-square test

# f=ax+b→ bit 0
a = 2; b = 1;

# g=cx+d→ bit 1
c = 3; d = 1;

### Preamble
import numpy as np
import math
from random import *

# Function that generates a random bit string of length m
def randBinList(m):
    B=[randint(0,1) for t in range(1,m+1)];
    return B

# Expected frequency
Exp_freq = float(n/Mod);

# significance levels
.alpha = [.01]

# function that creates an array with the bin edges
#def Bins(k,w):
    #b=[0]*4+1;
    #for i in range(1,k+1):
        #b[i]=b[i-1]+w;
    #return b

###

####

H_0: The distributions of constant terms of
the hash function is uniform

o.write(text); o.write(’\n\n’)
text="X_0^2 = ”+str(x0);
    o.write(’\n\n’)
text= "Length of input bit strings = ”+str(l);
    o.write(’\n\n’)
text= "Number of hash values tested = ”+str(n);
    o.write(’\n\n’)
text= "Individual values used as bins”;
    o.write(’\n\n’)

    text1= ”f = ”+str(a)” x +”+str(b)” mod ”+str_mod+” ——> bit 0”;
    o.write(text1);
    text2= ”g = ”+str(c)” x +”+str(d)” mod ”+str_mod+” ——> bit 1”;
    o.write(text2);
    text3= "***************************************”;
    o.write(text3);
Appendix D. Python code for the Chi-square test

nRejections_1=0

### Main function

```python
for Index in range(X):
    # Initial value of chi square statistic
    chi_square = 0.0*1.0
    f = [a,b]; #f=a*x+b as a vector
    g = [c,d]; #g=c*x+d as a vector

    #text1= "f = \begin{pmatrix} \text{a} \\ \text{b} \end{pmatrix} \Rightarrow \text{bit 0};
    #o. write(text1); o. write('n');
    #text2= \begin{pmatrix} \text{c} \\ \text{d} \end{pmatrix} \Rightarrow \text{bit 1};
    #o. write(text2); o. write('n');
    #Vector that stores the hash values
    text= \text{Test } \text{Index} ; print(text)
    o. write(text); o. write('n')
    freq=np.zeros(Mod);
    #vector=[];
    for i in range(n):
        B=randBinList(1);
        comp=[1,0]; #identity function comp=1*x+0
        for j in range(l):
            if B[j]==0:
                comp=[(comp[0]*f[0])%Mod, ((comp[0]*f[1])%Mod+comp[1])%Mod];
            else:
                comp=[(comp[0]*g[0])%Mod, ((comp[0]*g[1])%Mod+comp[1])%Mod];
        freq[int(comp[1])]=freq[int(comp[1])]+1;

    for i in range(Mod):
        f=freq[i] - Exp_freq
        chi_square += f * f
        chi_square /= Exp_freq; print(chi_square)

    text=\text{Chi_square is } \begin{pmatrix} \text{Chi_square} \end{pmatrix}
    o. write(text); o. write('n n');

    for t in range(len(x0)):
        _alpha_t = '{0:.2f}'.format(float(_alpha[t]))
        if chi_square >= x0[t]:
            text="REJECT CLAIM";
```

```python
```
Appendix D. Python code for the Chi-square test

```python
# print(text)
oc.write(text); o.write('\n \n');
nRejections_1 +=1
else:
text="FAILED TO REJECT CLAIM";
# print(text)
oc.write(text); o.write('\n \n');
print("number of rejections = "+ str(nRejections_1))
o.write(text3); o.write('\n \n');
text="Number of rejections of claim at level 0.01="+str(nRejections_1)
o.write(text); o.write('\n \n');
o.close();

print ("Number of rejections of claim")
print(nRejections_1)
print("THE END!!! \n")
```
Appendix E

SAGE code for the FitFlorea-Matula
DLG Digit-serial Algorithm

```python
import numpy as np

# Number of bits of modulo
k=5

# Log Base - semiprimitive root mod 2^k
logbase=11

# Number x to find the dlg(x)
x=3

# conversion of x to a binary array with exactly k bits
A=np.zeros(k)
ta=x.bitsof();

# if number of bits of x is > k then drop the k-len(ta) bits because any t-th bit with t>=k+1 is irrelevant mod 2^k
if (len(ta)>k):
```

89
for v in range(k):
    A[v]=int(ta[v]);
else:
    for v in range(len(ta)):
        A[v]=int(ta[v]);
# Initialization of the multiplicative inverse of A
B=np.zeros_like(A)
B[0]=1;
# Initialization of the exponent (dlg)
b=np.zeros(k-2)
# conversion of the log base to a binary array
with exactly k bits
    tn=logbase.bits();
N=np.zeros(k)
# if number of bits of logbase is > k then drop the
# k-len(ta) bits because any t-th bit with t>=k+1
# is irrelevant mod 2^k
if (len(tn)>k):
    for u in range(k):
        N[u]=tn[u]
else:
    for u in range(len(tn)):
        N[u]=tn[u]

### Function to roll and pad with zeros an array
def rollzeropad(a, shift):
   """
   Roll elements in a 1-row-array.
   Elements off the end of the array are treated as zeros.
   """
   a = np.asanyarray(a); n = len(a)
   if shift == 0:
       return a
   if np.abs(shift) > n:
       res = np.zeros_like(a)
else:
    zeros = np.zeros_like(a.take(np.arange(n-shift,n)))
    res = np.hstack((a.take(np.arange(n-shift,n)), zeros))

return res

### Function to add two binary arrays mod $2^k$

def add(array1, array2):
    """Add two binary arrays of same length mod $2^k$ where ‘k’
    represents the number of bits in each array.
    The result is a binary array of same length as the
    original ones.
    Parameters
    ""
    array1 : array_like
        Input array.
    array2 : array_like
        Input array.
    Returns
    ""
    sum_ : ndarray
        Output array, with the same number of bits as ‘array1’
        and ‘array2’.
    ""
    l = len(array1);
    sum_ = [];
    carry = 0;
    for i in range(l):
        bit1 = array1[i];
        bit2 = array2[i];
        s = carry + int(bit1) + int(bit2);
        if s <= 1:
            carry = 0;
        else:
            carry = 1;
        if s % 2 == 0:
            sum_.append(0);
        else:
Appendix E. SAGE code for the FitFlorea-Matula DLG Digit-serial Algorithm

```python
sum_.append(1);
return sum_

### Function to multiply two binary arrays of same length mod 2^k

def prod(array1, array2):
    
    Multiply two binary arrays of same length reducing mod 2^k where 'k' represents the number of bits in each array.
    The result is a binary array of same length as the original ones.

    Parameters
    ----------
    array1 : array_like
        Input array.
    array2 : array_like
        Input array.

    Returns
    -------
    prod_ : ndarray
        Output array, with the same number of bits as 'array1' and 'array2'.

    l = len(array1);
    prod_ = np.zeros_like(array1);
    for i in range(l):
        if array2[i] == 1:
            temp_ = roll_zero_pad(array1, i);
            prod_ = add(prod_, temp_);
    return prod_

### Function to determine the two's complement of a binary array mod 2^k

def TwosCompl(array1):
    
    Compute the two's complement of a binary array mod 2^k where 'k' represents the number of bits in the array.
The result is a binary array of same length as the original one.
```
Parameter

array1 : array_like
Input array.

Returns

compl_1 : ndarray
Output array, with the same number of bits as ‘array1’.

```python
l = len(array1);
compl_1 = np.zeros_like(array1);
compl_2 = np.zeros_like(array1);
compl_2[0] = 1;
for i in range(l):
    if array1[i] == 0:
        compl_1[i] = 1;
    else:
        compl_1[i] = 0;
compl_1 = add(compl_1, compl_2);
return compl_1
```

```python
### Main function

P = TwosCompl(A)
s = 1;
else:
P = A;
s = 0;
if (P[:3] == N[:3]).all():
    B = N
    b[0] = 1
P = prod(P, B);

for j in range(3, k):
    N = prod(N, N);
    if P[j] == 1:
        M = np.zeros_like(b)
        M[j - 2] = 1
        b = add(b, M);
        B = prod(B, N);
        P = prod(P, N);
result1 = TwosCompl(b)
result2 = 0;
```
for j in range(k-2):
    result2=result2+(result1[j]*2^j)

print "decimal dlg e=": print result2
print "sign is s=": print s
Bibliography


