Markus Schofnegger, BSc

# Implementing and Optimizing Lightweight Block Ciphers in the Context of a Signature Scheme

**MASTER'S THESIS**

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

**Graz University of Technology**

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Christian Rechberger

Institute of Applied Information Processing and Communications

Advisor

Dipl.-Ing. Sebastian Ramacher, BSc BSc MSc

Graz, May 2018

## AFFIDAVIT

# Acknowledgement

First and foremost, I want to express my gratitude to Christian Rechberger and Sebastian Ramacher for their continuous support, for their patience during my thesis, and for providing me the opportunity to join the work on a research paper, which turned out to be a new and very exciting experience.

I also wish to thank my parents for their emotional support and for continuously providing encouragement whenever it was needed.

Finally, I want to thank all members of Pink Floyd and especially David Gilmour. They will probably never read this, but their music was a big motivation during my studies and kept me going despite all the challenges.

# Abstract

Cryptography helps us to provide data security in digital communications while using a potentially insecure channel. One of the main principles made possible by cryptography is authentication, and in digital communications today this principle is implemented by applying digital signature protocols. These signature schemes play a crucial role in our everyday life and are often used without being noticed, such as in sensitive web applications like online banking. Hence, designing and implementing secure and efficient digital signature schemes is critically important.

Most of these schemes used in practice today, such as RSA, rely on the hardness of a mathematical problem, which can be solved efficiently by quantum computers. The construction of such a quantum computer is not possible yet, nevertheless it is important to search for new methods of providing data security in a post-quantum world. These methods consist of post-quantum secure algorithms, which are believed to withstand attacks by quantum computers.

This thesis discusses various post-quantum signature schemes and in particular the proof system ZKB++ [CDG+17] and a signature scheme based on this proof system. This signature scheme is built around the execution of a symmetric block cipher encryption, hence various block ciphers will also be discussed, where special emphasis is laid on their efficiency. Two of these block ciphers, MiMC [AGR+16] and GMiMC [AGP+18], were implemented during the work for this thesis, and various optimizations were used in order to reduce the total runtime of the signature protocol.

These changes include small adjustments to the protocol itself and the implementation of a custom math library, both of which were specifically added to make the protocol faster when using MiMC or GMiMC. The results of these optimizations are shown in the last chapter of this thesis.

# Kurzfassung

Kryptografie erlaubt es uns Datensicherheit zu gewährleisten, selbst, wenn die Kommunikation über einen potenziell unsicheren Kanal stattfindet. Eines der wichtigsten Prinzipien von Kryptografie ist die Authentifizierung, welche in der digitalen Kommunikation heute mithilfe diverser Signaturprotokolle implementiert wird. Diese Signaturalgorithmen spielen eine äußerst wichtige Rolle in unserem alltäglichen Leben und werden oft verwendet, ohne vom Benutzer bemerkt zu werden. Dies betrifft beispielweise Webanwendungen wie Onlinebanking, die mit Informationen vertrauenswürdig umgehen müssen.

Die meisten Signaturschemen, die heute in der Praxis verwendet werden, zum Beispiel RSA, verlassen sich bei ihrer Sicherheit auf schwer zu lösende mathematische Probleme, für die allerdings mit der Hilfe von Quantencomputern vergleichsweise effizient Lösungen gefunden werden können. Die Konstruktion solcher Geräte ist noch nicht möglich, aber dennoch ist es wichtig, bereits jetzt mit der Suche nach Verfahren zu beginnen, welche auch gegen Attacken von Quantencomputern ein gewisses Maß an Sicherheit gewährleisten. Diese Verfahren bestehen aus quantensicheren Algorithmen, von denen angenommen wird, dass sie auch in solchen Situationen sicher sind.

Diese Masterarbeit beschäftigt sich mit einigen quantensicheren Signaturschemen, insbesondere mit einem Beweisverfahren namens ZKB++ [CDG+17] und einem Signaturprotokoll basierend auf diesem Beweisschema. Dieses Signaturprotokoll ergibt sich aus der Verschlüsselungsfunktion einer symmetrischen Blockchiffre, weshalb diverse Blockchiffren ebenfalls diskutiert werden und besonders auf deren Effizienz und Eignung für ZKB++ eingegangen wird. Zwei dieser Blockchiffren, MiMC [AGR+16] und GMiMC [AGP+18], wurden im Zuge dieser Masterarbeit implementiert und mit diversen Methoden optimiert, um die Laufzeit des resultierenden Signaturschemas zu reduzieren.

Diese Änderungen beinhalten minimale Anpassungen des Protokolls selbst und die Implementierung einer eigenen Mathematik-Programmbibliothek. Sowohl diese Änderungen als auch die Bibliothek wurden hinzugefügt, um die Laufzeit des Protokolls speziell bei der Verwendung von MiMC oder GMiMC zu verbessern. Die Resultate dieser Optimierungen sind im letzten Kapitel der Arbeit zusammengefasst.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# 1 Introduction

Digital signatures play a crucial role in our everyday life. Whether it is browsing the internet or sending emails, without digital signatures we would not have the security most of us are requiring. The application areas of digital signatures are thus very manifold. For example, many of the websites we regularly visit, especially those containing sensitive data, use digital signatures together with TLS[1] in order to prove their identity to the user. This also includes online banking applications and other use cases, where the protection of data is crucially important. Furthermore, recent developments regarding the identification of citizens, one example being the Austrian mobile phone signature[2], are also only possible thanks to digital signatures. These mechanisms are used in the area of *e-government* and allow citizens to file tax returns or to view their electronic health records.

One of the more popular algorithms widely deployed in practice today is RSA. It is well-known and thought to be comparatively secure, since it relies on the hardness of the integer factorization problem, and there exists no publicly known polynomial-time algorithm to solve this problem. Another class of algorithms widely used today contains methods based on computations in elliptic curves. This includes the Elliptic Curve Digital Signature Algorithm (ECDSA), yielding smaller signature sizes than RSA for the same security level while at the same time being slower in the signature verification step, and the Edwards-curve Digital Signature Algorithm (EdDSA). Similar to RSA, both of them rely on the hardness of a specific problem, namely the problem of finding discrete logarithms, and are thus thought to be secure.

However, this is only the case if we do not consider quantum computers. As has been shown in 1997 by Shor [Sho97], RSA can in fact be broken relatively fast by making

[1]https://tools.ietf.org/rfc/rfc5246.txt
[2]https://www.digital.austria.gv.at/mobile-phone-signature

use of quantum computers[3]. Granted, quantum computers are not yet common and it will probably still take some time until such attacks can be reasonably used in practice. The transition to new cryptographic protocols, however, is not a trivial task and takes many years due to the adaptation of existing solutions and due to extensive analysis being necessary for new algorithms. Therefore, the NIST is already actively searching for new alternatives to traditional signature schemes, encryption schemes, and key encapsulation mechanisms [Nat17], in order to provide sufficient security when that time has come.

This thesis will focus on a class of post-quantum secure digital signatures based on symmetric encryption schemes. These schemes are already widely used today in the form of symmetric block ciphers, which are thought to be much more resistant to quantum computers than algorithms relying on the hardness of factoring integers or finding discrete logarithms. Moreover, it is possible to design block ciphers such that they have a low multiplicative complexity, i.e., a comparatively low number of multiplications. Encryption schemes with this property are not only useful for proof systems like ZKB++ but also in other areas such as multi-party computations [ARS+15], SNARKs [AGR+16], and ring signature schemes [DRS18]. One such block cipher, called GMiMC, is a crucial part of the new implementation of ZKB++, the main focus of this thesis.

## 1.1 Contributions and Structure of the Thesis

Contributions of this thesis include many different areas. Firstly, the original implementation of ZKB++ was specifically designed to work together with LowMC, a block cipher operating over the finite field $\mathbb{F}_2$. Thus, modular arithmetic over other finite fields was not supported, which in turn limits the flexibility of the current solution.

The new implementation supports modular arithmetic over a selection of predefined prime fields and binary fields, and compatibility with new fields can easily be added. In addition to this adjustment, the replacement of the inner circuit, a core component of ZKB++, with a different one was made significantly simpler.

[3]Note that even RSA can be made secure in a post-quantum world, as has been shown in [BHL+17]. However, this comes at a cost of impractically large key sizes and high execution times, and is thus only relevant in theory.

The second part of the practical work done during this thesis was about implementing the two block ciphers MiMC and GMiMC. These encryption schemes have not been used with ZKB++ before. Therefore, some small adjustments were made in order to make the computation of the final proof faster when using MiMC or GMiMC.

After implementing these block ciphers, which are based on arithmetic operations in prime fields or binary fields, these operations themselves were made faster by using specialized reduction methods. The resulting implementation of the proof system was then evaluated, and the total runtime with different parameters (e.g. different field sizes) was measured.

In the first chapter of this thesis, we give a short overview of post-quantum cryptography in general, and some of the methods in this field of cryptography, which are related to ZKB++ and the resulting signature scheme, are briefly described.

The second chapter gives an overview of the concepts used within this thesis, which includes the definition of zero-knowledge proofs, lightweight block ciphers, and finite fields.

After that, the proof system ZKB++ itself is described in detail, and some explanations regarding its security are also given. Then, some lightweight block ciphers used with ZKB++, in particular MiMC and GMiMC, which were implemented during this thesis, are explained.

Chapter 5 includes a selection of algorithms used for modular arithmetic in finite fields, in particular the methods used for the current implementation. Details of the implementation itself are described in the next chapter, where measurements and execution times are also given.

Finally, the last chapter gives an overview of some methods, which can probably be used to further optimize the computations and thus make the protocol faster.

## 1.2 Post-Quantum Cryptography and Related Methods

Post-quantum cryptography, in general, refers to a class of algorithms, which are believed to be secure against attacks from quantum computers. There are different methods in this field of cryptography, and four of these methods will be briefly described here.

### 1.2.1 Hash-Based Signatures

A hash function is a function $H : \{0,1\}^* \mapsto \{0,1\}^n$, where $H(x) = y$ is easy to compute, but getting $x$ by knowing $y$ should be computationally hard. With a variable input length and a fixed output length, a hash function obviously cannot be a permutation. However, given a random set of distinct inputs, a good hash function should map approximately the same number of input values to each of their output values. Furthermore, a cryptographically secure hash function must fulfill following properties:

> *Collision Resistance.* It should be hard to find any $x$ and $x'$, where $x \neq x'$, such that $H(x) = H(x')$.
> *Preimage Resistance.* Given only $y$, it should be hard to find $x$ such that $H(x) = y$.
> *2nd Preimage Resistance.* Given $x$ and $H(x) = y$, it should be hard to find $x' \neq x$ such that $H(x') = y$.

Hash-based cryptography refers to a class of algorithms, whose security relies on the security of hash functions. These are conjectured to be secure against attacks from quantum computers. As an example, this section will shortly describe one hash-based signature scheme and a classical way to improve it.

**Lamport-Diffie One-Time Signature Scheme.** The *Lamport-Diffie one-time signature scheme (Lamport OTS)* from 1979 [Lam79] uses a one-way function $f : \{0,1\}^n \mapsto \{0,1\}^n$ and a cryptographic hash function $H : \{0,1\}^* \mapsto \{0,1\}^n$. The private key $K_{private}$ is generated from $2n$ $n$-bit strings chosen at random:

$$K_{private} = (x_{n-1,0}, x_{n-1,1}, \ldots, x_{1,0}, x_{1,1}, x_{0,0}, x_{0,1}),$$

where $x_{i,j} \in \{0,1\}^n$ and $j \in \{0,1\}$. The public key (or verification key) $K_{public}$ is then

$$K_{public} = (y_{n-1,0}, y_{n-1,1}, \ldots, y_{1,0}, y_{1,1}, y_{0,0}, y_{0,1}),$$

where $y_{i,j} = f(x_{i,j}) \in \{0,1\}^n$ and $j \in \{0,1\}$. Now, given a bit string $M$ of arbitrary length, let $h = H(M)$. Then the signature $S$ of $h$ is

$$S = (x_{n-1,h_{n-1}}, \ldots, x_{1,h_1}, x_{0,h_0}),$$

where $x_{i,j} \in \{0,1\}^n$ and $j \in \{0,1\}$. Verification is done by computing the hash value and checking for equality, i.e., checking whether

$$(y_{n-1,h_{n-1}}, \ldots, y_{1,h_1}, y_{0,h_0}) = (f(S_{n-1}), \ldots, f(S_1), f(S_0)).$$

Note that the Lamport OTS needs $3n$ evaluations of $f$ in total. Improvements to this scheme include the *Winternitz one-time signature scheme (WOTS)* [DSS05], which produces shorter signatures, and *WOTS+* [Hül13], which can either reduce signature sizes further by using hash functions with shorter outputs or increase the security by maintaining the same signature size.

One-time signature schemes like the Lamport OTS must only be used to sign once, otherwise they would be easy to break, which means that new verification keys need to be sent for each signing process. This disadvantage can be mitigated to some extent by combining an OTS with Merkle trees [Mer80], resulting in a classical approach briefly explained now.

**Merkle Signature Scheme (MSS).** The idea of the Merkle signature scheme is to use a method based on full binary trees. It works with any one-time scheme, such as the Lamport OTS just described. The advantage of the MSS is that instead of relying on a fixed number of multiple verification keys, only one single public key is used to represent the validity of a signature. This public key is the root of the tree.

First, the user selects the parameter $d \geq 2 \in \mathbb{N}$. The resulting instantiation will then be able to sign exactly $2^d$ messages, which corresponds to the number of leaves of a full binary tree of depth $d$. Each leaf represents $H(m_i)$, where $H(\cdot)$ is again a cryptographic hash function equal to the one defined for the Lamport OTS, and $m_i$ is a message to be signed.

Before signing a document, the user has to create $2^d$ key pairs, here denoted as $((x_0, y_0), (x_1, y_1), \ldots, (x_{2^d-1}, y_{2^d-1}))$, where each $x_i$ is a private key and each $y_i$ is a (public) verification key. The whole private key is then the set containing every $x_i$.

The tree itself contains $2^{d+1} - 1 - 2^d$ internal nodes, and each of them stores the hash value of the concatenation of the hash values of its child nodes. This means that together with the hash values of each message to be signed, the key pair generation needs $2^{d+1} - 1 - 2^d + 2^d = 2^{d+1} - 1$ evaluations of the hash function.

In order to sign a message, the signer first chooses an index $i$ to indicate that message $m_i$ is to be signed. Then they create the one-time signature $s_i$ using the associated $x_i$. The Merkle signature $S_{m_i}$ for the message $m_i$ is then

$$S_{m_i} = (i, s_i, y_i, A_i),$$

where $A_i$ denotes the unique authentication path for $i$, containing all intermediate hash values stored in the internal nodes and needed by the verifier to compute all hash values up to the root of the tree.

Verification is done by first checking the given signature $s_i$ and then validating the public key $y_i$ by computing all intermediate hash values and comparing the public root hash value with the computed hash value.

An example for a Merkle tree is illustrated in Figure 1.1. Note that even though $2^d$ messages can be signed, this number is still very limited when compared to RSA, ECDSA or EdDSA, where one key pair is enough to sign an arbitrary number of messages. However, there have been various recent approaches to design viable hash-based signature schemes, one of them being SPHINCS-256 [BHH+15] and its improved version SPHINCS$^+$ [BDE+17], using SHA-256, SHAKE256 [Div14] or Haraka [KLM+16].

### 1.2.2 Lattices

A lattice is the set of all possible vectors generated from

$$\mathcal{L}(\mathbb{B}) = \left\{ \sum_{i=1}^{n} x_i \cdot v_i \ \mid x_i \in \mathbb{Z} \right\},$$

where $\mathbb{B} = (v_1, v_2, \ldots, v_n)$ is a set of column vectors and called the *basis* of the lattice. That is, a lattice is a collection of an infinite number of points, and each point can be generated by a certain combination of the vectors in $\mathbb{B}$.

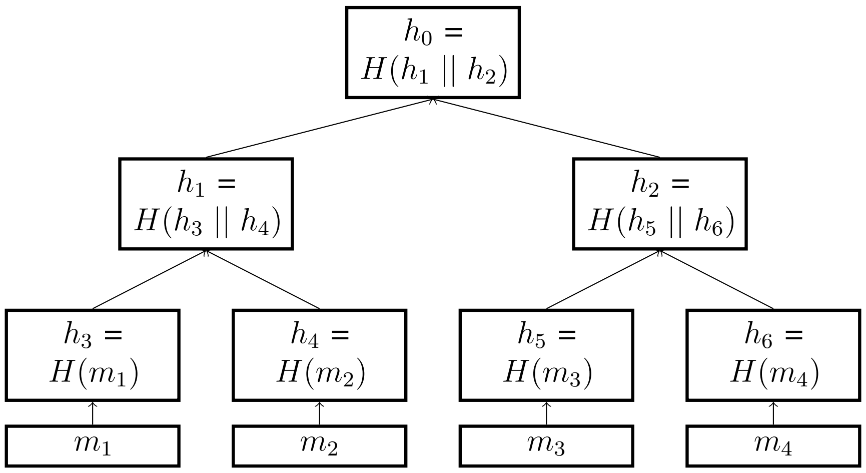

$h_0 = H(h_1 \mid\mid h_2)$
$h_1 = H(h_3 \mid\mid h_4)$
$h_2 = H(h_5 \mid\mid h_6)$
$h_3 = H(m_1)$
$h_4 = H(m_2)$
$h_5 = H(m_3)$
$h_6 = H(m_4)$
$m_1$
$m_2$
$m_3$
$m_4$

Figure 1.1: A Merkle tree with 4 leaves.

Lattices are not a new concept in mathematics. However, the usage for lattices in cryptography was first discovered in 1996 [Ajt96], where a new cryptographic hash function using lattices is also given. Two years later, NTRU was introduced [HPS98], a public-key signature scheme based on lattices. In general, cryptographic constructions based on lattices are believed to be secure based on the hardness of certain problems within lattices, such as the *closest vector problem*, which is the problem to find the closest point to a given vector in a known lattice [BBD08]. The conjectured PQ security of lattice-based cryptography follows from the fact that currently no quantum algorithm is known to solve lattice-based problems faster than classical algorithms [BBD08].

A modern lattice-based digital signature scheme believed to be post-quantum secure is TESLA [ABB+15], presented in 2015. Compared to the signature schemes *Picnic-FS* and *Picnic-UR* described later, it offers significantly better sign and verification times, and also smaller signatures sizes, but at the cost of much larger keys. A provably secure variation of TESLA, called qTESLA, also makes use of lattices, more precisely of the *learning with errors* (LWE) problem which is believed to be hard to solve, and was submitted to the NIST post-quantum cryptography competition in 2017.

### 1.2.3 Code-Based Cryptography

Code-based cryptography refers to a class of algorithms whose functionality and security rely on error correcting codes. Examples for error correcting codes include the binary Goppa code which is used in the original proposal for the McEliece public-key cryptosystem presented in 1978 [McE78]. This cryptosystem is still unbroken to this day, although some parameters needed to be changed in order to adapt to new security requirements.

Currently, no quantum algorithm is known in order to efficiently break the McEliece scheme, and similar to symmetric-key cryptography, a sufficient countermeasure in this scheme would be to increase the length of the codes. However, the main disadvantage is that the McEliece cryptosystem uses comparatively large keys, namely several hundred kilobytes in scenarios with realistic security levels. Various modifications, such as a variant proposed by Niederreiter in 1986 [Nie86], are able to reduce the key size, however at a cost of computation time in the general setting.

### 1.2.4 Block Ciphers and Symmetric-Key Cryptography

In contrast to stream ciphers, block ciphers operate on a fixed-length input of $n$ bits. They can be transformed into a stream cipher by different operation modes in order to use them for encryption and decryption in practice. Block ciphers are a comparatively old concept in cryptography and their origin goes back to the late 1940s [Sha49], when Claude Shannon first introduced the *product cipher*.

Most block ciphers are designed such that they use either a Feistel network, briefly illustrated in Figure 1.2, or a substitution-permutation network. Both concepts try to provide the two principal properties of encryption schemes: *diffusion* and *confusion*, where the former means that flipping a single bit of the plaintext should also change a large amount of ciphertext bits and vice versa, and the latter means hiding any connections between the ciphertext and the key bits.

Another characteristic of modern block ciphers is that they consist of a round function, which is then applied during multiple rounds within the encryption and the decryption. This round function often makes use of different round keys, which are typically derived from the master key. With an appropriate round function, increasing the number of rounds also increases the security of most block ciphers, at least up to a

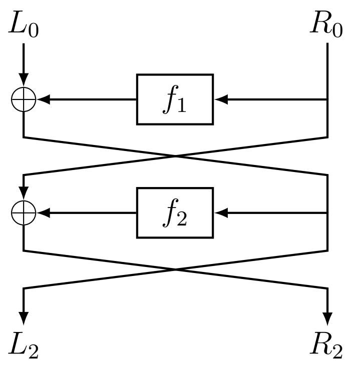


Figure 1.2: A 2-round Feistel-1 network. $L_i$ and $R_i$ denote the $n/2$ most significant bits and the $n/2$ least significant bits of the current block after round $i$, respectively.

certain degree. However, this has an impact on the encryption and decryption speed, and therefore the number of rounds is kept as low as possible while still providing the desired level of security.

**Security.** Very similar to $n$-bit hashes, which would resist preimage attacks with strength $2^{n/2}$, symmetric-key cryptography is believed to be relatively secure in that a block cipher with a $\kappa$-bit key offers $\frac{\kappa}{2}$ bits of security against attacks from quantum computers. This follows from Grover's search algorithm [Gro96], which is able to find an inverse $f^{-1}$ to a block cipher encryption function $f$ such that $f(x) = y$ and $f^{-1}(y) = x$ in $\mathcal{O}(\sqrt{2^\kappa}) = \mathcal{O}(2^{\kappa/2})$ operations [YI00], where $\kappa$ again denotes the key size of the block cipher in bits. Although this is significantly faster than the brute-force method requiring $\mathcal{O}(2^\kappa)$ operations, the attack can be weakened relatively easy by just doubling the key size.

The methods described in this thesis aim to provide a post-quantum security of $128$ bits, which means that the focus lies on block ciphers with a key size of $256$ bits.

# 2 Preliminaries

This chapter will now discuss the most important concepts used for this thesis and the new implementation. Notation given in this chapter will be used throughout the whole document. It will also give a short introduction to two different versions of zero-knowledge (ZK) proofs and secure multi-party computation (MPC), both of which are core components of ZKB++.

## 2.1 Digital Signatures

Encryption algorithms alone are not able to provide signatures, that is, an encrypted message is not proven to have a particular origin. In order to provide such a proof, which is called *authentication*, digital signatures have to be used. They are widely found throughout the internet, for example together with digital certificates to ensure a trustworthy way of communicating with websites. A digital signature involves a public and a private key, where the latter one is used to sign a message and the first one is used to verify it.

Formally, a digital signature consists of three distinct algorithms [Kat10]:

> *Generate.* The key generation $Generate(n) = (sk, vk)$, where $n$ is the security parameter in bits, is a randomized algorithm responsible for generating a secret (signing) key $sk$ used to sign a message and a public (verification) key $vk$ used to verify a signature. This step is usually performed only once.
> *Sign.* The algorithm $Sign(m, sk) = \sigma$ creates a signature $\sigma$ based on the private key $sk$ and on the message $m$ to be signed.
> *Verify.* A verification algorithm $Verify(m, \sigma, vk) = r$, where $r \in \{0, 1\}$, uses the message $m$, the provided signature $\sigma$, and the public key $vk$, and decides whether the signature is valid for the given message ($r = 1$).

All of these algorithms are *probabilistic polynomial-time (PPT)* algorithms, meaning that they run in polynomial time and have access to some random oracle providing random bits. Nevertheless, the result given by the *Verify(.)* algorithm is deterministic.

A digital signature has to fulfill certain properties in order to be regarded as secure:

> *Correctness.* For all key pairs $(sk, vk)$ output by $Generate(n)$ and for all possible messages $m$, we require that
>
> $$Verify(m, Sign(m, sk), vk) = 1.$$
>
> *Unforgeability.* It must be impossible for an adversary to forge a valid digital signature for a message in feasible time. There are different types of adversaries in this context, and strong signature schemes provide security even against attacks where the adversary does not have any control over the message for which a valid signature is forged, but can request signatures for arbitrary messages from the signature generation algorithm. These signature schemes are called *existentially unforgeable under chosen-message attacks (EUF-CMA)*.

One of the most popular digital signature schemes used in practice today is RSA, which is known to be vulnerable to attacks from quantum computers.

## 2.2 Interactive Zero-Knowledge Proofs

Similar to digital signatures, a zero-knowledge (ZK) proof enables an honest prover to prove a statement to an honest verifier with very high probability, where an honest party is a party that follows the protocol and does not try to gain more information than appropriate. However, in contrast to digital signatures, the definition of a zero-knowledge proof is that nothing should be known about both of them before the protocol starts and, more importantly, no additional knowledge should be gained throughout the protocol. Thus, the verifier, denoted by $V$, should only learn whether a statement made by the proving party, denoted by $P$, is true or not.

Given a language $L$ and a statement $x$, $x \in L$ denotes a true statement $x$. ZK proofs must then satisfy three properties:

*Soundness.* A dishonest prover $P$ can only convince a verifier $V$ of a false statement with small probability (more precisely, this probability has to be lower than $1/2$). That is, for all $x \notin L$, an honest verifier rejects with probability $> 1/2$.
*Completeness.* An honest verifier will be convinced of a true statement. That is, for all $x \in L$, an honest verifier accepts.
*Zero-Knowledge.* Independent from the fact whether the involved parties are cheating or not, the verifier will only learn if a statement is true or not and no new information is gained during the protocol.

The first two properties imply that a ZK protocol has a non-zero chance to yield an undesired result. This probability, however, can be made arbitrarily small by just repeating the protocol for a certain number of times. This means that ZK protocols are easily parallelizable, because distinct iterations of the protocol are independent from each other.

Moreover, the underlying scheme in ZKB$++$ is a sigma protocol ($\Sigma$-protocol). In such a protocol, the prover $P$ and the verifier $V$ have both access to a public $x \in L$, but only $P$ has access to a private witness $w$ such that $(x, w) \in R$, where $R = \{(x, w)\}$ is a binary relation and $L = \{x \mid \exists w : (x, w) \in R\}$. Then the protocol can be described by three consecutive steps:

1. *Commitment.* $P$ sends a commitment $c$ to $V$.
2. *Challenge.* $V$ creates a random challenge $e$ with $s$ bits of information and sends it to $P$.
3. *Verification.* $P$ replies to the challenge with a message $r$ and $V$ decides whether to accept based on $x$ and $(c, e, r)$.

Note that step $2$ is an interaction needed between the prover and the verifier. The version used in ZKB$++$ was made non-interactive using two possible transforms described in the next section.

Similar to ZK proofs, $\Sigma$-protocols have to fulfill three properties:

*Completeness.* If $P$ and $V$ follow the protocol correctly and $(x, w) \in R$, then $V$ will accept.
*n-Special Soundness.* Given $y$ and the accepted communications $(c, e_1, r_1)$, $(c, e_2, r_2)$, $\ldots$, $(c, e_n, r_n)$ with $e_i \neq e_j$ for $i \neq j$, it is possible to efficiently compute $w'$ such that $(x, w') \in R$.

> *Special Honest Verifier Zero-Knowledge.* Given $x$ and $e$, it is possible to compute $(c, e, r)$ with the same probability distribution as a “real” evaluation of the protocol between $P$ and $V$.

The soundness error of every $\Sigma$-protocol is $2^{-s}$, which corresponds to the probability of guessing the correct challenge beforehand. Zero-knowledge can be gained from $\Sigma$-protocols, and while proving that a combined ZK proof with independent and parallel repetitions is still zero-knowledge is non-trivial [GK90], any combined protocol of $\Sigma$-protocols is still a $\Sigma$-protocol [Dam10].

## 2.3 Non-Interactive Zero-Knowledge Proofs

The type of zero-knowledge proof just presented is an interactive zero-knowledge proof, meaning that the random challenge is created by the verifier and sent to the prover. In ZKB$++$, the underlying $\Sigma$-protocol was transformed into a non-interactive zero-knowledge (NIZK) proof, where the challenge is generated by the prover and thus one interaction step can be omitted. This was made possible by using two approaches shortly described below.

**Fiat-Shamir Transform.** The Fiat-Shamir transform [FS86] is a method to transform a given $\Sigma$-protocol into a non-interactive zero-knowledge proof by removing the interaction step between the prover $P$ and the verifier $V$ during the *Challenge* phase defined above. The idea is that the challenge is instead created by $P$ using a random oracle, which in practice is implemented by a hash function, and taking as input the commitment $c$. Hence, the challenge is also generated after computing $c$. Based on the commitment, the challenge can then also be created locally by $V$, who then decides whether to accept or reject.

This method is proven to be secure against attacks from adversaries without a quantum computer [FS86], and in a recent paper it has also been shown that it is secure against attacks from quantum computers under additional conditions [Unr17].

**Unruh Transform.** The Unruh transform [Unr15] is another method to transform a $\Sigma$-protocol into a non-interactive zero-knowledge proof. However, it can be proven to be secure in the post-quantum setting. At a high level, its idea is to not only use a single challenge for each repetition of the $\Sigma$-protocol, but to instead compute the response for a larger number of possible challenges. It then generates the final hash value to be verified by using the initial statement, the outputs of all the repetitions, and permutations of the responses.

Although the Unruh transform would result in a significantly larger proof size compared to the Fiat-Shamir transform, some properties of ZKB$++$ could be used in order to reduce the additional cost [CDG+17].

Note that in order to build a signature scheme from a zero-knowledge proof, the message $m$ to be signed has to be part of the computation. This is done by generating the challenge with a hash function, and by using both the commitment and $m$ as inputs to this function.

## 2.4 Secure Multi-Party Computation

Secure multi-party computation, abbreviated as MPC, means a joint computation of a common result by multiple parties, such that the final result is public (or only known to the computing parties), but every participant's input remains private to themself. As further explained in Chapter 3, this definition is only true to some extent for the MPC version used in ZKB$++$, but still manages to convey the general idea of the concept.

MPC was firstly mentioned in 1982 as *secure two-party computation* [Yao82] and later generalized to multi-party computation by Yao [Yao86]. The concept is thus comparatively old and finds use in many fields, such as electronic voting.

The version of MPC used in ZKB$++$ is similar to the GMW protocol introduced in 1987 [GMW87], because in this case MPC is also used to compute an arbitrary function. In particular, ZKB$++$ uses MPC to compute a number of gates, which together form a predefined circuit. This concept is known as "MPC in the Head" and was presented in 2007 [IKO+07].

## 2.5 Lightweight Block Ciphers

Block ciphers, more specifically lightweight block ciphers, will play a crucial part of this thesis and the new implementation of ZKB++. This section briefly explains the importance of lightweight block ciphers.

### 2.5.1 Design Rationale

Surprisingly and even though processing power became more and more powerful in the past, the need for lightweight block ciphers only emerged recently. The main reason is likely the increasing popularity of small devices (e.g. RFID tags or smart cards) in need of secure encryption schemes, and these small devices are more limited when it comes to properties like chip area, power consumption, and program size. This is also a reason for which AES, a well-known block cipher still resistant to many attacks, cannot easily be used on small devices due to its relatively large footprint.

Examples of lightweight block ciphers include PRESENT [BKL+07], PRINCE [BCG+12], LowMC [ARS+15], MiMC [AGR+16], and GMiMC [AGP+18].

### 2.5.2 Characteristics

The performance of lightweight block ciphers is measured using different characteristics, such as:

*Gate Equivalent (GE).* This is a unit of measure which helps to estimate the complexity of a resulting digital circuit of a specific cipher. Usually, one GE refers to the unit area of one NAND gate.
*Power Consumption.* The consumed power of a block cipher is naturally given in watts.
*Throughput.* More informally, this is the speed of the cipher, often measured in Gbps.

When designing a lightweight block cipher, a trade-off has to be made between these characteristics and the security of the resulting encryption scheme. For example, the NIST recommends a minimum key size of $112$ bits even for lightweight block ciphers [BR11], which is not reached by the $80$-bit instance of the above-mentioned PRESENT cipher.

## 2.6 Finite Fields

Finite fields are used by some of the encryption schemes discussed in Chapter 4. The definitions and the notations given in this chapter will be used throughout the whole document.

### 2.6.1 Definition

A field $F$ in mathematics is a set on which the operations *addition*, *subtraction*, *multiplication*, and *division* are defined. Common examples of fields include the set of real numbers $\mathbb{R}$ and the set of complex numbers $\mathbb{C}$.

A finite field $\mathbb{F}_q$ is a field with a finite number of elements, that is, $|\mathbb{F}_q| \in \mathbb{N}$. The number of elements in the field, also called the *order* of the field, is $q = p^k$, where $p$ is a prime number and $k \in \mathbb{N}$. In the following chapters we will focus on two types of finite fields:

> *Prime Fields.* For an arbitrary prime number $p$, the finite field $\mathbb{F}_p$ denotes the *prime field* defined by $p$, which is the set of residues $\bmod p$ and thus has $p$ elements, namely $0, 1, \ldots, p-1$.
> *Binary Fields.* For an arbitrary integer $n$, the finite field $\mathbb{F}_{2^n}$ denotes the *binary field* defined by an irreducible polynomial of degree $n$. The binary field $\mathbb{F}_{2^n}$ contains $2^n$ elements, namely all polynomials of degree at most $n-1$, including the zero polynomial.

For the block ciphers MiMC and GMiMC, further described in Chapter 4, we require that modular additive inverses and modular multiplicative inverses exist for every element of the field. This condition is fulfilled in every finite field $\mathbb{F}_q$. Note that division in a finite field was not implemented during this thesis, as its implementation is not necessary for the *Picnic-FS* and *Picnic-UR* signature schemes.

### 2.6.2 Permutation Polynomials

The core component of some of the block ciphers described later is the function $f(x) = x^3$, so we require this function to be a permutation.

**Theorem 1.** *For a finite field $\mathbb{F}_q$, the monomial $x^c$ with $c > 1 \in \mathbb{N}$ is a permutation polynomial of $\mathbb{F}_q$ if and only if $\gcd(c, q-1) = 1$, where $\gcd(a, b)$ denotes the greatest common divisor of $a$ and $b$.*

*Proof.* Let $\langle a \rangle$ be a finite cyclic group of order $m$. Then the order of $\langle a^c \rangle$ is the smallest positive integer $n$ such that $a^{cn} = 1$. Now let $g = \gcd(c, m)$. Then $a^{cn} = 1$ if and only if $m \mid cn$ or, equivalently, $\frac{m}{g} \mid n$. The smallest natural number $n$ fulfilling this property is $\frac{m}{g}$. Thus, the element $a^c$ generates a subgroup of order $\frac{m}{g}$.

The multiplicative group of $\mathbb{F}_q$ is cyclic of order $q - 1$. Therefore, let $m = q - 1$ and $g = \gcd(c, m) = 1$. Then the function $f \in \mathbb{F}_q \mapsto f^c$ is a bijection, which completes the proof. $\square$

**Theorem 2.** *For a natural number $n > 0$, $\gcd(3, 2^n - 1) = 1$ if and only if $n$ is odd.*

*Proof.* Obviously, $2^n \pmod 3 \neq 0$, since 3 is prime and 2 is the only prime factor of $2^n$. If $n$ is even, $2^n = 4^{n/2} \equiv 1^{n/2} \equiv 1 \pmod 3 \implies \gcd(2^n - 1, 3) = 3$. If $n$ is odd, $2^n = 2 \cdot 4^{\lfloor n/2 \rfloor} \equiv 2 \cdot 1^{\lfloor n/2 \rfloor} \equiv 2 \pmod 3 \implies \gcd(2^n - 1, 3) = 1$. $\square$

For $c = 3$, these two theorems conclude that in order for $f(x) = x^3$ to be a permutation, $\gcd(p - 1, 3) \overset{!}{=} 1$ for a prime field of order $p$ and $\gcd(2^n - 1, 3) \overset{!}{=} 1$ for a binary field of order $2^n$.

# 3 ZKBoo and ZKB$++$

This chapter will now describe ZKBoo and ZKB$++$, both of which are proof systems based on a $\Sigma$-protocol and on the evaluation of a predefined function. Implementing the resulting signature scheme and testing it with various configurations and circuits was the main focus of this thesis, which is why the concept of this scheme and its protocol will be described in detail.

## 3.1 Overview

The original version of the protocol, called ZKBoo, was first presented in 2016 [GMO16] as a protocol to prove the computation of an arbitrary boolean circuit while maintaining the zero-knowledge property. For our use case and the signature schemes *Picnic-FS* and *Picnic-UR*, this boolean circuit represents a one-way function $f(x) = y$, where $x$ and $y$ stand for the private and public key, respectively. That is, the signer wants to prove knowledge of $x$ such that $f(x) = y$ without disclosing information about $x$. While the function $f$ can be any function for which MPC gate definitions exist, it is of course necessary to choose a circuit which is hard to invert in order to maintain the security of the protocol (e.g. a hash function or a block cipher encryption using a secret key).

In 2017, ZKB$++$, an improved version of ZKBoo, was introduced [CDG+17]. ZK-Boo and ZKB$++$ are very similar, however, the latter includes many optimizations to the original scheme. Most of them result in a significantly reduced proof size and in a shorter computation time. For the new implementation, only the newer ZKB$++$ was implemented, therefore only ZKB$++$ will be described in this chapter.

## 3.2 Description

The core of ZKB++ is “MPC in the Head”, a concept presented in 2007 [IKO+07]. In its original version, the prover computes a function $g(y, s_0, s_1, \dots, s_{n-1})$ for all $n$ parties, where $y$ is known to all parties, $s_0 \oplus s_1 \oplus \cdots \oplus s_{n-1} = x$ is the private input only known to the prover, and $s_0, s_1, \dots, s_{n-1}$ are picked at random[1]. After this computation, the prover commits to the views. The verifier now chooses two different parties and asks the prover to open the committed views of the chosen parties. The verifier checks the values in these views and accepts if they are consistent.

The security follows from the fact that a dishonest prover cannot cheat by manipulating only one view, since the public value would also change. Thus, an adversary needs to create at least two inconsistent views. This pair of inconsistent views is detected by the verifier with probability $\binom{n}{2}^{-1}$. This probability can be increased by repeating the protocol multiple times.

### 3.2.1 MPC in ZKB++

In ZKB++, the number of parties is fixed ($n = 3$) and the protocol itself does not need the additional interaction step from the verifier to the prover, i.e., ZKB++ is a non-interactive ZK proof system. The shared computation of a function $f$ is illustrated in Figure 3.1. In this construction, ${f_j}^{(i)}$ and ${x_j}^{(i)}$ denote the function and the share for party $j$ at gate $i$, respectively. The functions for each gate are then defined as follows:

*Addition with a Constant ($c = a + k$).*

$$c_i = \begin{cases} a_i + k & \text{if } i = 0, \\ a_i & \text{otherwise.} \end{cases}$$

*Binary Addition ($c = a + b$).*

$$c_i = a_i + b_i.$$

[1]More precisely, $n - 1$ shares are picked at random and the last one is chosen such that $s_0 \oplus s_1 \oplus \cdots \oplus s_{n-1} = x$.

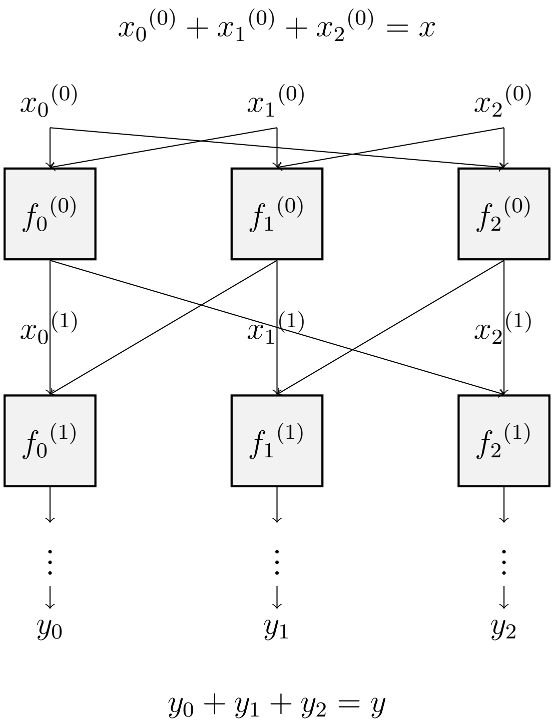


Figure 3.1: Circuit computation in ZKB++.

*Multiplication with a Constant ($c = a \cdot k$).*

$$c_i = a_i \cdot k.$$

*Binary Multiplication ($c = a \cdot b$).*

$$c_i = (a_i \cdot b_i) + (a_{i'} \cdot b_i) + (a_i \cdot b_{i'}) + R_i(m) - R_{i'}(m).$$

In these definitions, $a$, $b$ and $c$ denote shared values, and the subscripts $i$ and $i' = i+1 \pmod 3$ refer to the shares of parties $i$ and $i'$, respectively. A public constant is denoted by $k$ and $R_i(m)$ is a precomputed random number associated with party $i$ and multiplication gate $m$. These random numbers are necessary, otherwise the multiplication result would not be uniformly distributed over all possible values anymore.

In the implementation itself, each $R_i(m)$ is the output of a pseudorandom number generator (PRNG) seeded with the initial random tape $r_i$ of party $i$, because truly random $R_i(m)$ values would require the final proof to include each distinct $R_i(m)$. Thus, defining these numbers as outputs of a PRNG makes it possible to only store the relevant seed values in the proof.

Note that the addition with a constant is the only asymmetric gate, i.e., the function is not the same for each party. Moreover, the binary multiplication gate is the only gate needing the input of the “next” party, which heavily contributes to the final proof size.

Furthermore, all computations take place in a predefined field, i.e., in $\mathbb{F}_2$ the additions are bitwise XOR operations, and the multiplications are bitwise AND operations.

**Circuit Decomposition.** Before describing the protocol itself, a few functions need to be defined. These are necessary to create the decomposition of an arbitrary circuit. In this case, the circuit will be denoted by $f(x) = y$, where $x$ is the private key and $y$ is the public key.

To initiate the protocol, the private value $x$ needs to be shared first, creating three shares $x_0$, $x_1$, and $x_2$, where $x_0 + x_1 + x_2 = x$. This is done during the $\mathit{Share}$ function defined as follows:

$$({View_0}^{(0)}, {View_1}^{(0)}, {View_2}^{(0)}) \leftarrow \mathit{Share}(x, r_0, r_1, r_2),$$

where $r_0$, $r_1$, and $r_2$ are random binary strings of a predefined length and ${View_j}^{(0)}$ denotes the first value of the view of party $j$.

The circuit gates are computed using the definitions described above. This computation is denoted by the $\mathit{Update}$ function:

$${View_j}^{(i+1)} \leftarrow \mathit{Update}({View_j}^{(i)}, {View_{j'}}^{(i)}, r_j, r_{j'}),$$

where $j' = j + 1 \pmod 3$. This means that the $\mathit{Update}$ function takes the previous gate values of parties $j$ and $j'$, together with their random tapes, and computes the next gate value.

Finally, the function $\mathit{Output}$ extracts the last value of a view and assigns it to an output share:

$$y_j \leftarrow \mathit{Output}(View_j).$$

In the algorithms found in this section, the consecutive computation of all circuit gates (that is, the computation of the entire circuit) is denoted by

$$Update(\cdots(\cdots Update({x_j}^{(i)}, {x_{j'}}^{(i)}, {r_j}^{(i)}, {r_{j'}}^{(i)})\cdots)\cdots),$$

where again $j' = j + 1 \pmod 3$, and ${x_j}^{(i)}$ and ${x_{j'}}^{(i)}$ are intermediate shares. Note that view values are only generated for multiplication gates.

### 3.2.2 Protocol

**Proof Generation.** The proof generation algorithm assumes two public entities: the circuit $f$, and the value $f(x) = y$ used as the public key. Furthermore, three distinct hash functions $H_1$, $H_2$ and $H_3$ are needed. These can be created from the same core hash function, and in the current implementation SHA-256 is used for this purpose.

The first step of the protocol is to create a random tape $r_j \in \{0,1\}^s$ for each party $j$, where $s$ is the length of the random tape in bits. In the implementation of ZKB++, each $r_j$ is chosen by a pseudorandom number generator and is used as a seed to initialize a dedicated PRNG for later usage. The random tapes are also needed to create the initial shares for $x$, which is done during the $Share$ function defined above.

The circuit $f$ is then evaluated using the initial shares for $x$ and the gate definitions described earlier. For each binary multiplication, the output value of each party $j$ is added to $View_j$. That is, at the end of the protocol, $View_j$ is a collection of all binary multiplication results for party $j$.

After the circuit computation, the output shares $y_0$, $y_1$ and $y_2$, where again $y_0 + y_1 + y_2 = y$, are collected. For the commitment of each view, the hash function $H_3$, the random tape of the corresponding party, and all the view values are used. The challenge is computed by the prover, which makes the proof non-interactive.

The main part of the proof is then a deterministically chosen view based on the challenge, and the values needed for the verifier to start the verification process. The detailed process of generating a proof with ZKB++ using the Fiat-Shamir transform is given in Algorithm 3.1, where the circuit $f$, the public key $f(x) = y$, the hash functions $H_1$, $H_2$ and $H_3$, and an arbitrary pseudorandom number generator are known to both the prover and the verifier.

**Algorithm 3.1:** Proof generation with ZKB++ using the Fiat-Shamir transform.

```
    Input: Private key $x$, number of iterations $t$.
    Output: The proof $p$.
 1  foreach iteration $i \in \{0, 1, \dots, t-1\}$ do
 2      Sample random tapes ${r_0}^{(i)}, {r_1}^{(i)}, {r_2}^{(i)}$.
 3      foreach party $P_j$, where $j \in \{0, 1, 2\}$ and $j' = j + 1 \pmod 3$ do
 4          $({x_0}^{(i)}, {x_1}^{(i)}, {x_2}^{(i)}) \leftarrow Share(x, {r_0}^{(i)}, {r_1}^{(i)}, {r_2}^{(i)}) =$
             $(H_1({r_0}^{(i)}), H_1({r_1}^{(i)}), x - H_1({r_0}^{(i)}) - H_1({r_1}^{(i)}))$.
 5          ${View_j}^{(i)} \leftarrow$
             $Update(\cdots(\cdots Update({x_j}^{(i)}, {x_{j'}}^{(i)}, {r_j}^{(i)}, {r_{j'}}^{(i)})\cdots)\cdots)$.
 6          ${y_j}^{(i)} \leftarrow Output({View_j}^{(i)})$.
 7          Commit $({C_j}^{(i)}, {D_j}^{(i)}) = (H_3({r_j}^{(i)}, {View_j}^{(i)}), {r_j}^{(i)} \mid\mid {View_j}^{(i)})$.
 8          $a^{(i)} \leftarrow ({y_0}^{(i)}, {y_1}^{(i)}, {y_2}^{(i)}, {C_0}^{(i)}, {C_1}^{(i)}, {C_2}^{(i)})$.
 9  Compute the challenge $e = H_2(a^{(0)}, a^{(1)}, \dots, a^{(t-1)})$.
10  Interpret $e$ deterministically such that $e^{(i)} \in \{0, 1, 2\}$.
11  foreach iteration $i \in \{0, 1, \dots, t-1\}$, where $e''^{(i)} = e^{(i)} + 2 \pmod 3$ do
12      $b^{(i)} \leftarrow ({y_{e''^{(i)}}}^{(i)}, {C_{e''^{(i)}}}^{(i)})$.
13      $z^{(i)}) \leftarrow \begin{cases} ({View_1}^{(i)}, {r_0}^{(i)}, {r_1}^{(i)}) \text{ if } e^{(i)} = 0, \\ ({View_2}^{(i)}, {r_1}^{(i)}, {r_2}^{(i)}, {x_2}^{(i)}) \text{ if } e^{(i)} = 1, \\ ({View_0}^{(i)}, {r_2}^{(i)}, {r_0}^{(i)}, {x_2}^{(i)}) \text{ if } e^{(i)} = 2. \end{cases}$
14  $p \leftarrow (e, (b^{(0)}, z^{(0)}), (b^{(1)}, z^{(1)}), \dots, (b^{(t-1)}, z^{(t-1)}))$.
15  return $p$.
```

**Proof Verification.** The verification process is similar to the proof generation. Instead of using the share function, the verifier uses the provided values and starts the computation of the circuit for two branches only. Whenever the evaluation of a binary multiplication gate is needed, the verifier picks the second input to this gate from the second branch. This can only be done for one of the branches, the computation for the second branch continues after the multiplication gate by using the respective value from the provided view.

At the end of the circuit computation, the verifier has calculated two output shares for the public key $y$. The third share can be calculated by using these two shares and $y$.

The verifier can now compute the commitments for two views themself and use them to recalculate the challenge. They accept if and only if the challenge provided by the prover is the same as the newly calculated one. A detailed description of the proof verification with ZKB++ using the Fiat-Shamir transform is given in Algorithm 3.2, where again the circuit $f$, the public key $f(x) = y$, the hash functions $H_1$, $H_2$ and $H_3$, and an arbitrary pseudorandom number generator are known to both the prover and the verifier.

**Algorithm 3.2:** Proof verification with ZKB++ using the Fiat-Shamir transform.

```
   Input: Public key $f(x) = y$, proof $p$.
   Output: Verification result.
1  foreach iteration $i \in \{0, 1, \ldots, t-1\}$, where ${e'}^{(i)} = e^{(i)} + 1 \pmod 3$ and ${e''}^{(i)} = e^{(i)} + 2 \pmod 3$ do
2      $({x_{e^{(i)}}}^{(i)}, {x_{e'^{(i)}}}^{(i)}) \leftarrow \begin{cases} (H_1({r_0}^{(i)}), H_1({r_1}^{(i)})) \text{ if } e^{(i)} = 0, \\ (H_1({r_1}^{(i)}), {x_2}^{(i)}) \text{ if } e^{(i)} = 1, \\ ({x_2}^{(i)}, H_1({r_0}^{(i)})) \text{ if } e^{(i)} = 2. \end{cases}$
3      ${View_{e^{(i)}}}^{(i)} \leftarrow$
         $Update(\cdots(\cdots Update({x_{e^{(i)}}}^{(i)}, {x_{e'^{(i)}}}^{(i)}, {r_{e^{(i)}}}^{(i)}, {r_{e'^{(i)}}}^{(i)})\cdots)\cdots)$.
4      ${y_{e^{(i)}}}^{(i)} \leftarrow Output({View_{e^{(i)}}}^{(i)})$.
5      ${y_{e'^{(i)}}}^{(i)} \leftarrow Output({View_{e'^{(i)}}}^{(i)})$.
6      ${y_{e''^{(i)}}}^{(i)} \leftarrow y - {y_{e^{(i)}}}^{(i)} - {y_{e'^{(i)}}}^{(i)}$.
7      foreach $j \in \{e^{(i)}, {e'}^{(i)}\}$ do
8          $({C_j}^{(i)}, {D_j}^{(i)}) \leftarrow (H_3({r_j}^{(i)}, {View_j}^{(i)}), {r_j}^{(i)} \mid\mid {View_j}^{(i)})$.
9      ${a'}^{(i)} \leftarrow ({y_0}^{(i)}, {y_1}^{(i)}, {y_2}^{(i)}, {C_0}^{(i)}, {C_1}^{(i)}, {C_2}^{(i)})$, where ${y_{e''^{(i)}}}^{(i)}$ and ${C_{e''^{(i)}}}^{(i)}$
         are part of $z^{(i)}$.
10 Compute the challenge $e' = H_2({a'}^{(0)}, {a'}^{(1)}, \ldots, {a'}^{(t-1)})$.
11 if $e' = e$ then
12     return "ACCEPT".
13 else
14     return "REJECT".
```

These explanations were for one iteration only. To reduce the soundness error, multiple iterations of ZKB++ have to be performed, as shown in the respective algorithms.

**Proof Size.** The expected proof size of ZKB++ using the Fiat-Shamir transform, resulting in *Picnic-FS*, is

$$S_1 = \gamma \cdot \left( c + 2t + \log_2(3) + \frac{2}{3} \cdot s + v \right)$$

bits, where $\gamma$ is the number of repetitions, $c$ is the size of the commitment (e.g. $c = 256$ bits for SHA-256), $t$ is the size of a random tape, $s$ is the size of a share, and $v$ is the view size (all in bits). The value $\log_2(3)$ is due to the challenge, where $\log_2(3)$ bits are needed to specify an element $e \in \{0, 1, 2\}$. The size of the share has to be multiplied by $\frac{2}{3}$, because only in two of three cases (namely, when the challenge is $1$ or $2$) one of the input shares is actually needed.

For the sake of completeness, the proof size of ZKB++ using a modified version of the Unruh transform, resulting in *Picnic-UR*, is

$$S_2 = \gamma \cdot (c + 3t + \log_2(3) + s + 2v)$$

bits, where the variables are the same as before.

## 3.3 Security

The ZKB++ proof system relies on a couple of properties, which aim to give Picnic a security of $128$ bits in the post-quantum setting. A more detailed security analysis of the protocol can be found in [GMO16].

**Zero-Knowledge Property.** First note that the underlying $\Sigma$-protocol of ZKB++ has 3-special soundness. This can be seen from the fact that three communications with the same commitments can be used to rewind the protocol beginning with the output shares and computing the circuit in reversed direction. By doing so, three input shares $x'_i$, where $i \in \{0, 1, 2\}$, can be found such that $x'_0 + x'_1 + x'_2 = x'$ and $f(x') = y$. Although all views are also included in only two distinct challenges, the protocol does not have 2-special soundness, because with two challenges only, the remaining branch cannot be verified. That is, the prover could simply calculate the protocol for an $x''$ with $f(x'') \neq y$ and then edit the view values accordingly for the needed output shares.

The zero-knowledge property follows from the construction of the MPC scheme within ZKB++. A verifier is only ever given two out of three shares, which means that the corresponding intermediate state cannot be extracted and, more importantly, it is uniformly distributed over all possible values. This property is called 2-privacy and affects all MPC gates defined in Section 3.2.1, and thus also the initial input $x$. Only the last state can be extracted, which is the public value $y$.

**Number of Iterations.** Given the definition of the circuit decomposition from above, ZKB++ has a soundness error of $2/3$. In order to achieve a soundness error of at most $2^{-128}$, which corresponds to a security of $128$ bits in the classical (non-quantum) setting, at least $219$ iterations are necessary, because $(2/3)^s < 2^{-128}$ for $s \geq 219$. In the post-quantum setting, this number has to be doubled due to the impact of Grover's search algorithm [Gro96], which leads to $t = 2 \cdot 219 = 438$ iterations. The attack using the search algorithm and further detail can be found in [CDG+17].

**Fiat-Shamir Transform and Unruh Transform.** The $\Sigma$-protocol in ZKB++ was transformed into a non-interactive zero-knowledge (NIZK) proof by applying the Fiat-Shamir transform [FS86] in the non-quantum setting, and by using the Unruh transform [Unr15] to gain security in the post-quantum setting, as further explained in 2.3.

# 4 Choice of Circuit for ZKB$++$

As explained in Chapter 3, the number of multiplications (or the number of AND gates) within a circuit plays an important role for the resulting proof size. Indeed, this number multiplied by the size of the used field is by far the most dominant part of the proof size.

However, the size of the proof is not the only problem. Due to the definition of the MPC binary multiplication gate in ZKB$++$, one single binary multiplication in the circuit actually results in $3$ multiplications and $4$ additions[1] for each party, which are $9$ multiplications and $12$ additions in the ZKB$++$ implementation. This is significantly higher than the final number of additional operations induced by the addition gates. Thus, even if additions and multiplications are similarly expensive on their own, multiplication gates are still much more expensive in the ZKB$++$ setting. Hence, reasonable choices for the resulting signature scheme include circuits with a comparatively small amount of multiplication gates.

In the original ZKBoo [GMO16], the hash function SHA-$256$ was used as a proof of concept for the circuit over which the public key was calculated. With its comparatively large amount of $25000$ AND gates, SHA-$256$ is not the best choice for a protocol like ZKBoo or ZKB$++$, and SHA-3 ($38400$ AND gates) would be even worse in this context. AES-128 needs $5440$ AND gates, while AES-192 and AES-256 use $6528$ and $7616$ AND gates, respectively. PRINCE needs $1920$ AND gates for an encryption and approximately $1400$ AND gates are required for LowMC and $256$ bits of security [CDG+17].

This chapter will describe some block ciphers which are more suited for ZKB$++$ due to a low multiplicative complexity. Note that MiMC is not exactly competitive, but still mentioned, because it serves as the foundation of GMiMC.

[1] One of these additions is a subtraction, but the computational cost is the same for additions and subtractions.

## 4.1 LowMC

LowMC was published in 2015 [ARS+15] and was actually the first circuit used in ZKB++ as a comparison to the former SHA-256. It is a highly parameterizable construction, i.e., it allows the user to change a selection of parameters. The number of AND gates of LowMC is still smaller than 800 for a 128-bit security level, which is significantly less than SHA-256 or AES-128.

**Description.** LowMC is a block cipher using a substitution-permutation network. It allows the user to freely choose a selection of parameters, such as the block size $n$ and the key size $\kappa$, the number of S-boxes, and the allowed data complexity $d$ of attacks, where $d = \log_2(n_{pairs})$ and $n_{pairs}$ refers to the number of (plaintext, ciphertext) pairs needed for an attack. The S-boxes of LowMC have a size of 3 bits and the number of S-boxes can be reduced in order to reduce the number of multiplications. LowMC also uses a partial S-box layer, which is not common for block ciphers with substitution-permutation networks.

One round of LowMC is shown in Figure 4.1. The output of each S-box $S(\cdot)$ is defined as

$$S(a, b, c) = (a + (b \cdot c), a + b + (a \cdot c), a + b + c + (a \cdot b)),$$

where $a$, $b$ and $c$ are bits, and additions and multiplications are executed in $\mathbb{F}_2$. The $m$ S-boxes are used for the first $3m$ bits of the $n$-bit state, while the remaining $n - 3m$ bits (for $n > 3m$) are passed to the affine layer unchanged.

Key addition and constant addition take place after the affine layer in $\mathbb{F}_2$ (XOR), where the round keys and the $n$-bit round constants are chosen during the instantiation of the cipher and then fixed. More precisely, each round key is generated by multiplying the initial key with a randomly chosen $n \times \kappa$ binary matrix of rank $\min(n, \kappa)$.

The affine layer itself consists of a multiplication in $\mathbb{F}_2$ of the current state with an $n \times n$ binary matrix, which is different for each round and chosen randomly out of all invertible $n \times n$ matrices. All of these matrices are randomly chosen and fixed during the instantiation of LowMC, thus generation takes place in constant time. Decryption is the same as encryption, but with a reversed order of steps and by inverting the matrices used in the affine layer. The pseudorandom bits needed for the matrices are generated by the Grain LFSR.

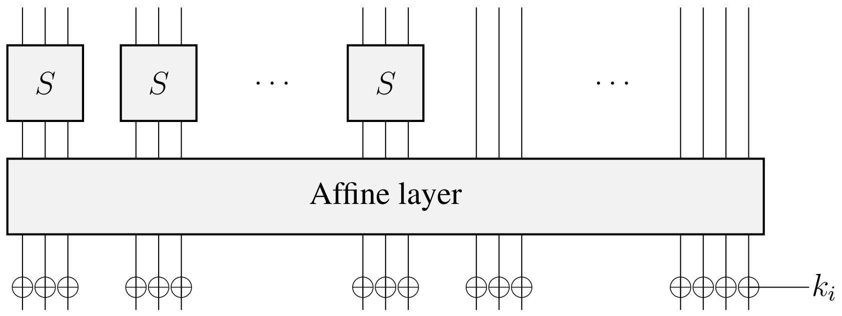


Figure 4.1: One round of LowMC.

**Security.** LowMC uses keys with a size $\kappa$ of e.g. $\kappa = 80$, $\kappa = 128$, or $\kappa = 256$ bits, where the last choice aims to provide a post-quantum security of 128 bits. With the affine layer already providing a rather large amount of diffusion, LowMC is designed to use a low amount of rounds, comparable to AES. Due to the flexibility of LowMC, the number of rounds depends on $n$, $\kappa$, $m$, and $d$. Table 4.1 gives a short overview of possible parameters and resulting round numbers for LowMC [ARS+15], and it also shows some of the configurations used for ZKB++, where the data complexity $d$ was set to 1.

| Block size $n$ | Key size $\kappa$ | # S-boxes $m$ | Data complexity $d$ | # rounds $r$ |
|---|---|---|---|---|
| 256 | 256 | 1 | 1 | 316 |
| 256 | 256 | 10 | 1 | 38 |
| 256 | 256 | 42 | 1 | 14 |
| 128 | 80 | 21 | 64 | 13 |
| 1024 | 80 | 21 | 64 | 28 |
| 256 | 128 | 42 | 128 | 13 |
| 1024 | 128 | 42 | 128 | 19 |
| 256 | 256 | 42 | 128 | 15 |
| 1024 | 256 | 42 | 128 | 21 |

Table 4.1: Some parameter sets of LowMC together with their respective round number.

As mentioned above, the advantage of LowMC in the ZKB++ proof setting is that the number of multiplications strongly contributing to the proof size can be reduced by using fewer S-boxes. However, by using the identity mapping for part of the state bits, a new attack vector is introduced. This has been shown in [DEM15], where the number of effective rounds of an 11-round LowMC instantiation could be reduced to 9.

## 4.2 MiMC

Compared to LowMC described in the previous section, MiMC [AGR+16] is a very different approach. The main distinction is probably that it does not use a substitution-permutation network. Instead, it uses a straightforward chain of functions for the rounds or, alternatively, a balanced Feistel network. This chapter will explain MiMC in detail and, most importantly, discuss its main disadvantages in the ZKB++ setting.

**Description.** MiMC is a block cipher whose core component is the function $f(x) = x^3$. The computation of this function takes place in $\mathbb{F}_q$, where $q = p$ or $q = 2^n$ for a prime number $p$ and a natural number $n$. That is, computations take place either in a prime field or a binary field. Two possible variants of MiMC are shortly described here, namely MiMC-$n/n$ (or MiMC-$p/p$ for prime fields) and MiMC-$2n/n$ (or MiMC-$2p/p$ for prime fields). For simplicity and because the functions are the same for prime fields, only MiMC-$n/n$ and MiMC-$2n/n$ will be discussed.

For the block cipher to be a permutation, $f$ has to be a permutation. The necessary criteria to fulfill this requirement are discussed in Section 2.6.2. Along with the permuting function $f$, MiMC also uses key additions and round constants, similar to LowMC. In detail, the encryption function of MiMC-$n/n$ is

$$E_k(x) = (F_{r-1} \circ F_{r-1} \circ \cdots \circ F_0)(x) \oplus k,$$

where $x$ is the plaintext, $r$ is the number of rounds, $F_i$ is the round function in round $i$, and $k$ is the key. Each $F_i$ is defined as

$$F_i(x) = (x \oplus k \oplus c_i)^3,$$

where $c_i$ is the round constant in round $i$ and $c_0 = 0$. The round constants are chosen as random elements of $\mathbb{F}_q$ at the instantiation of MiMC and then fixed. Note that there are no round keys, instead the same key is used in each round and once at the end. The encryption path for $r$ rounds of the non-Feistel version of MiMC is shown in Figure 4.2.

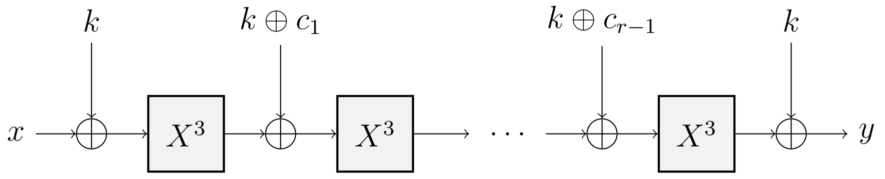


Figure 4.2: The MiMC encryption function with $r$ rounds.

For the formulas shown above, the block size is the same as the key size, and it also defines the size of the field in which the computations take place. Thus, the most expensive component of the encryption is the exponentiation in each round, and this is also one of the reasons for which MiMC is not a competitive choice for ZKB++. Nevertheless, the speed of this operation could be significantly increased as shown in Chapter 5, and it even turned out that in some cases using larger field sizes is beneficial for the runtime, although resulting in a larger proof size for ZKB++.

MiMC can also use a Feistel network. The round function of MiMC-$2n/n$ is then

$$F_i(x_{i_L} \mid\mid x_{i_R}) = x_{i_R} \oplus (x_{i_L} \oplus k \oplus c_i)^3 \mid\mid x_{i_L},$$

where $x_{i_L}$ and $x_{i_R}$ are the most significant and least significant $n$ bits of $x$ in round $i$, respectively, the block size is $2n$, and the key size is $n$.

Decryption was not needed for ZKB++ and thus not implemented. However, for the sake of completeness, it should be mentioned that decryption in MiMC is significantly more expensive than encryption. The reason is that for decryption the inverse of $f(x) = x^3 \pmod{m}$, for $m$ being a prime number or an irreducible polynomial, has to be used. For the prime case, the inverse would be $f_1(x) = x^{s_1} \pmod{m}$, where $s_1 \cdot 3 \equiv 1 \pmod{m-1}$ (this follows from Fermat's little theorem), and for the binary case it would be $f_2(x) = x^{s_2} \pmod{m}$, where $s_2 = \frac{2^{n+1}-1}{3}$ (a proof for this statement is given in [AGR+16]). Both $s_1$ and $s_2$ tend to be much higher than $3$.

Various methods like exponentiation by repeated squaring can speed up decryption. However, when used as a block cipher, it is nevertheless advisable to use MiMC with operation modes that do not require decryption at all (e.g. CTR mode [DE79]).

**Security.** The number of rounds for MiMC-$n/n$ to be deemed secure is

$$r_1 = \log_3(2^n) = \frac{\log_2(2^n)}{\log_2(3)} = \frac{n}{\log_2(3)}$$

for computations in binary fields and

$$r_2 = \log_3(p) = \frac{\log_2(p)}{\log_2(3)}$$

for computations in prime fields. The round number for MiMC-$2n/n$ is similar with the numerator of the fraction being doubled in each case.

As the core component of MiMC is the function $f(x) = x^3$, main focus during the security analysis of MiMC was laid on algebraic attacks such as a new "GCD" attack further described in [AGR+16]. More specifically, the number of rounds for MiMC-$n/n$ was derived from a possible interpolation attack[2], which was found to be the most impactful one against this construction.

## 4.3 GMiMC

This section will now describe the GMiMC block cipher, which was one of the main focuses of this thesis. This is also the second block cipher being used with the new implementation and thus served as an alternative to MiMC described in the previous section. As the name suggests, MiMC and GMiMC, which stands for Generalized MiMC, are closely related.

Most of the methods described in Chapter 5 were specifically needed for GMiMC and also optimized for its various field sizes and instantiations. Hence, this block cipher will now be described in a little more detail.

[2]During an interpolation attack [JK97], the adversary tries to find a polynomial which corresponds to the encryption function without knowledge of the key. The coefficients of this polynomial can be determined by Lagrange interpolation, where known (plaintext, ciphertext) pairs are used as data points. This method can also be extended to mount a key-recovery attack.

### 4.3.1 Description

Similar to the original version of MiMC, GMiMC also uses the function $f(x) = x^3$ as the main component of its construction. However, where MiMC used a function chain or, alternatively, a balanced Feistel network, GMiMC uses a different approach, namely unbalanced Feistel networks. These are Feistel networks where the two (or more) branches of the network are not of the same size. In the case of GMiMC, for most field sizes more than two branches are used.

In this section, two modes of GMiMC will be described, namely GMiMC$_{crf}$ and GMiMC$_{erf}$. These have proved to be the most competitive choices for the ZKB++ proof setting, where the latter was ultimately chosen as the best choice. Justifications for this statement will be given after describing both modes.

For the following discussions, let $X_i$ denote the value of a Feistel branch, where $X_0$ denotes the value of the leftmost branch and $X_{t-1}$ denotes the value of the rightmost branch, with the number of branches being denoted by $t$. For example, the intermediate value of a 4-branch Feistel network is thus denoted by $(X_0 \mid\mid X_1 \mid\mid X_2 \mid\mid X_3)$, where $X_0$ contains the $\frac{n}{4}$ most significant bits of the whole $n$-bit value. As before, $n$ is the block size and $\kappa$ is the key size.

**GMiMC$_{crf}$.** GMiMC$_{crf}$ uses an unbalanced Feistel network with a contracting round function, which can be described as

$$(X_0, X_1, \ldots, X_{t-1}) \leftarrow (X_1, X_2, \ldots, X_{t-1}, X_0 + F(X_1, X_2, \ldots, X_{t-1})),$$

where $F(\cdot)$ is the round function in round $j$ defined as

$$F(X_1, X_2, \ldots, X_{t-1}) = \left( \left( \sum_{i=1}^{t-1} X_i \right) + k_j + c_j \right)^3$$

for $c_j$ and $k_j$ denoting the round constant and round key in round $j$, respectively. Note that in contrast to MiMC, distinct round keys are used here. They have to be of the same size as the field in which the computations take place and are derived from the $\kappa$-bit master key $k$. In detail, if the key size is the same as the field size, the same key is used for each round. If the key size is larger than the field size (which is the case for most instantiations), then $k_j = k'_{j \pmod{l}}$, where $k = (k'_0 \mid\mid k'_1 \mid\mid \ldots \mid\mid k'_{l-1})$ and $\kappa = l \cdot n$.

One round of GMiMC$_{crf}$ is shown in Figure 4.3.

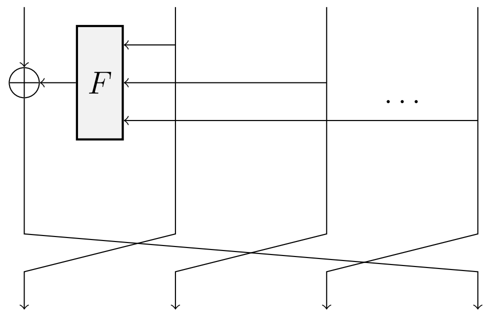


Figure 4.3: One round of GMiMC$_{crf}$ in an unbalanced Feistel network.

**GMiMC$_{erf}$.** GMiMC$_{erf}$ also uses an unbalanced Feistel network, but in contrast to GMiMC$_{crf}$, it has an expanding round function. It can be described as

$$(X_0, X_1, \dots, X_{t-1}) \leftarrow (X_1 + F(X_0), X_2 + F(X_0), \dots, X_{t-1} + F(X_0), X_0),$$

where $F(\cdot)$ is again the round function in round $j$ defined as

$$F(X_i) = (X_i + k_j + c_j)^3$$

and both $k_j$ and $c_j$ are the same as in GMiMC$_{crf}$.

One round of GMiMC$_{erf}$ is shown in Figure 4.4.

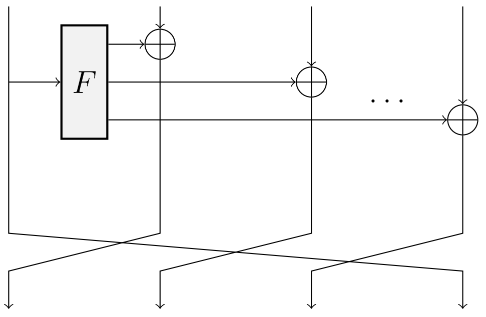


Figure 4.4: One round of GMiMC$_{erf}$ in an unbalanced Feistel network.

The number of additions and multiplications in each round is the same for GMiMC$_{crf}$ and GMiMC$_{erf}$. However, GMiMC$_{erf}$ still manages to be more competitive, because the round number is lower while maintaining the same level of security.

GMiMC$_{crf}$ and GMiMC$_{erf}$ are not the only constructions of GMiMC. Other possible constructions include GMiMC$_{nyb}$ using a Feistel network construction proposed in [Nyb96], and GMiMC$_{mrf}$ using a Multi-Rotating structure introduced in [AGP+18]. Both of them work only for an even number of branches and were not added to the new implementation of ZKB++, because they would result in larger proof sizes compared to the other two constructions. The main reason is that the round function, which includes the exponentiation, is called $\frac{t}{2}$ times in each round, whereas it is only called once in each round in GMiMC$_{crf}$ and GMiMC$_{erf}$.

### 4.3.2 Security

For ZKB++, the GMiMC family of block ciphers is aiming for a security of $n = 256$ bits, which are believed to correspond to $128$ bits of security in a post-quantum world.

The number of rounds for GMiMC$_{crf}$ is

$$r_{crf} = \left\lceil 1.262 \cdot \frac{n}{t} \right\rceil + 6t - 4$$

and the number of rounds for GMiMC$_{erf}$ is

$$r_{erf} = \left\lceil 1.262 \cdot \frac{n}{t} \right\rceil + 4t - 3,$$

where $n$ is the block size and $t$ is the number of branches. Thus, $\frac{n}{t}$ is the size of a single branch value and also the size of the field in which the computations take place. Both of these round numbers correspond to the number of rounds necessary to provide security against an interpolation attack, which, similar to MiMC, proved to be the most powerful attack against these constructions of GMiMC.

However, these round numbers are for the general use case, where no restrictions on data complexity are given. In the ZKB++ setting, a possible adversary has only access to a very limited number of (plaintext, ciphertext) pairs, which is why the round numbers from above can be reduced. The new round number for GMiMC$_{erf}$ is then

$$r_{erf} = \left\lceil 1.262 \cdot \frac{n}{t} - 4 \cdot \log_3\left(\frac{n}{t}\right) \right\rceil + 4t + 3,$$

where again $n$ is the block size and $t$ denotes the number of branches. In this low-data scenario, GMiMC$_{erf}$ turned out to be the most efficient GMiMC configuration for ZKB++.

**Field Sizes and the Consequences for ZKB++.** GMiMC supports many different configurations, mainly by choosing different field sizes. ZKB++ aims for a post-quantum security of $128$ bits, hence the encryption scheme used should have a key size of at least $256$ bits. For a block size of $256$ bits, every chosen field size results in a corresponding number of branches. For example, $t = 4$ branches are chosen for a $64$-bit field.

In the *Picnic-FS* and *Picnic-UR* signature schemes, the goal is to achieve a small signature while also providing a short computation time. If we only want to optimize the former, then $r_{erf} \cdot n'$ has to be minimized (the number of multiplications is $2$ and thus the same in each round). It turns out that in this case, the field size $n' = \frac{n}{t} = \frac{258}{86} = 3$ is the best choice ($r_{erf} = 347$). However, this also results in a relatively slow computation.

The lowest number of rounds, on the other hand, is achieved by choosing a field size of $n' = 32$ bits ($r_{erf} = 63$). This is also beneficial for the final implementation, where the block size is exactly $256$ bits and operations with $32$-bit values can be implemented efficiently. Indeed, when using prime fields, a field size of $n' = 32$ bits turned out to yield the fastest computation in our setting.

The relation between field sizes and the resulting computational cost for various operations is shown in Section 6.4.

To further emphasize the difference between MiMC and GMiMC with regard to the number of multiplication gates in configurations providing at least $256$ bits of non-quantum security, Table 4.2 shows the multiplicative complexities of various instantiations and the resulting computational cost in bits.

| Scheme | $(n, t, r)$ | $m = 2 \cdot r$ | $c = n \cdot m$ |
|---|---|---|---|
| **MiMC** | $(256, 1, 162)$ | 324 | 82944 bits |
| | $(272, 1, 172)$ | 344 | 93568 bits |
| **GMiMC**$_{erf}$ ($\mathbb{F}_p$) | $(3, 86, 347)$ | 694 | 2082 bits |
| | $(4, 64, 260)$ | 520 | 2080 bits |
| | $(16, 16, 78)$ | 156 | 2496 bits |
| | $(32, 8, 63)$ | 126 | 4032 bits |
| | $(64, 4, 85)$ | 170 | 10880 bits |
| | $(136, 2, 165)$ | 330 | 44880 bits |
| **GMiMC**$_{erf}$ ($\mathbb{F}_{2^n}$) | $(3, 86, 347)$ | 694 | 2082 bits |
| | $(33, 8, 64)$ | 128 | 4224 bits |
| | $(65, 4, 86)$ | 172 | 11180 bits |
| **GMiMC**$_{crf}$ ($\mathbb{F}_p$) | $(3, 86, 516)$ | 1032 | 3096 bits |
| | $(4, 64, 386)$ | 772 | 3088 bits |
| | $(16, 16, 113)$ | 226 | 3616 bits |
| | $(32, 8, 85)$ | 170 | 5440 bits |
| | $(64, 4, 101)$ | 202 | 12928 bits |
| | $(136, 2, 180)$ | 360 | 48960 bits |
| **GMiMC**$_{crf}$ ($\mathbb{F}_{2^n}$) | $(3, 86, 516)$ | 1032 | 3096 bits |
| | $(33, 8, 86)$ | 172 | 5676 bits |
| | $(65, 4, 103)$ | 206 | 13390 bits |

Table 4.2: Number of multiplication gates $m$ and final computational cost $c$ for various instantiations of MiMC and GMiMC, where $n$ denotes the field size, $t$ denotes the number of branches, and $r$ denotes the number of rounds.

# 5 Modular Arithmetic in MiMC and GMiMC

This chapter will now discuss modular arithmetic in general and various algorithms in this area. The implementation of some of these methods and their optimization was the second part of this thesis' practical work.

The necessity of these algorithms comes from the fact that both MiMC and GMiMC heavily rely on modular arithmetic. While additions and subtractions in a finite field are relatively cheap in terms of computation time, multiplications are much more expensive. The reason is that the second part of a modular multiplication is the reduction, which for most field sizes (especially smaller ones) is even more expensive than the multiplication itself.

Additionally, a shared multiplication operation used by ZKB$++$ in the MPC setting needs $9$ multiplications ($3$ for each party), hence $18$ multiplications for $f(x) = x^3$ in each round, which makes fast modular multiplications even more important.

This chapter will first focus on classical approaches of modular arithmetic, and then discuss faster methods for special moduli. In all algorithms and explanations used in this chapter, $W$ denotes the word size in bits and $N$ denotes the number of words. For example, if $W = 64$, then in a 136-bit finite field $N = 3$ words have to be used to represent a value $x$ in this field. In general, $N = \lceil n/W \rceil$, where $n$ denotes the field size in bits. Moreover, $x[0]$ denotes the least significant word of a value $x$, and $x[N-1]$ denotes the most significant word of the same value.

## 5.1 Additions and Subtractions

Both additions and subtractions are faster than multiplications. The main reason is that an addition of two $n$-bit values has a result which is at most one bit larger. A similar argument holds for subtractions.

### 5.1.1 Prime Fields

In prime fields, the result $r$ of an addition of two values $a$ and $b$, with $a, b \in \mathbb{F}_p$, where $p$ is the prime modulus, needs at most one subtraction in order to get a new value $r' \in \mathbb{F}_p$. This is obvious, because $a < p, b < p \implies r' = r - p = a + b - p < p$.

For subtractions, at most one addition is necessary in order to compute a new value $r' \in \mathbb{F}_p$. This addition is needed if the second operand is larger than the first one, in which case the result is a negative number $r$ with $|r| < p$, where $r = a - b, a < p, b < p$, and $b > a$.

Classical addition and subtraction are shown in Algorithm 5.1 and Algorithm 5.2, respectively. Note that these algorithms are linear-time algorithms and need $\mathcal{O}(N)$ operations.

**Algorithm 5.1:** Word-wise addition.

```
   **Input:** Integers $a$ and $b$ using $N$ words.
   **Output:** $(\beta \mid\mid c)$, where $\beta$ is the carry bit and $c = (a + b) \pmod{2^{WN}}$.
1  $(\beta \mid\mid c[0]) \leftarrow a[0] + b[0]$.
2  for $i \leftarrow 1$ to $N - 1$ do
3      $(\beta \mid\mid c[i]) \leftarrow a[i] + b[i] + \beta$.
4  return $(\beta \mid\mid c)$.
```

Both of these algorithms can easily be transformed into their modular versions, given in Algorithm 5.3 and Algorithm 5.4.

**Algorithm 5.2:** Word-wise subtraction.

```
   Input: Integers $a$ and $b$ using $N$ words.
   Output: $(\beta \mid\mid c)$, where $\beta$ is the borrow bit and $c = (a - b) \pmod{2^{WN}}$.
1  $(\beta \mid\mid c[0]) \leftarrow a[0] - b[0]$.
2  for $i \leftarrow 1$ to $N - 1$ do
3      $(\beta \mid\mid c[i]) \leftarrow a[i] - b[i] - \beta$.
4  return $(\beta \mid\mid c)$.
```

**Algorithm 5.3:** Modular word-wise addition in $\mathbb{F}_p$.

```
   Input: Prime number $p$ and integers $a, b$ such that $a, b \in \mathbb{F}_p$.
   Output: $c = (a + b) \pmod{p}$.
1  $(\beta \mid\mid c) \leftarrow Add(a, b)$.
2  if $\beta = 1$ or $c \geq p$ then
3      $c \leftarrow c - p$.
4  return $c$.
```

### 5.1.2 Binary Fields

In a binary field $\mathbb{F}_{2^n}$, modular addition is the same as modular subtraction, which can both be done by a bitwise XOR ($\oplus$) operation between two operands. The reason is that in an $n$-bit binary field, the irreducible polynomial has order $n$ and thus occupies $n + 1$ bits, which means that every $n$-bit value is already reduced. Furthermore, by using XOR operations, every sum of two $n$-bit values is again an $n$-bit value, which is not always the case in prime fields. This means that also every result of an addition in $\mathbb{F}_{2^n}$ is already reduced, and thus *if* clauses are not needed. Therefore, modular additions and subtractions in binary fields tend to be faster than their counterparts in prime fields.

Both modular addition and modular subtraction can be implemented by using Algorithm 5.5. This is also a linear-time algorithm and uses $\mathcal{O}(N)$ operations.

**Algorithm 5.4:** Modular word-wise subtraction in $\mathbb{F}_p$.

```
   Input: Prime number $p$ and integers $a, b$ such that $a, b \in \mathbb{F}_p$.
   Output: $c = (a - b) \pmod{p}$.
1  $(\beta \mid\mid c) \leftarrow Sub(a, b)$.
2  if $\beta = 1$ then
3      $c \leftarrow c + p$.
4  return $c$.
```

**Algorithm 5.5:** Modular word-wise addition/subtraction in $\mathbb{F}_{2^n}$.

```
   Input: Polynomials $a, b$ such that $a, b \in \mathbb{F}_{2^n}$ and $n < W \cdot N$.
   Output: $c = (a \oplus b)$.
1  for $i \leftarrow 0$ to $N - 1$ do
2      $c[i] \leftarrow a[i] \oplus b[i]$.
3  return $c$.
```

## 5.2 Modular Multiplications in Prime Fields

While modular additions and modular subtractions can be implemented with a rather straightforward approach, this is not at all the case for modular multiplications. The main reason is that the result of an addition or subtraction has at most $n + 1$ bits, where $n$ is the number of bits of the larger of the two operands, and that, following the explanations from above, at most one additional operation is necessary.

However, a multiplication result $r = a \cdot b$, where $a, b \in \mathbb{F}_p$, has at most $2 \cdot \lceil \log_2(p) \rceil$ bits, which is why reductions for multiplications are in general not as fast.

### 5.2.1 Basic Multiplication

Before reducing a multiplication result, the multiplication itself has to be performed. The new implementation uses a word-wise version, which needs $\mathcal{O}(N^2)$ operations (quadratic time). There are various other multiplication techniques, such as the Karatsuba algorithm [KO62], which will be discussed in Chapter 7.

Given two integers $a, b \in \mathbb{F}_p$, Algorithm 5.6 computes the result $r = a \cdot b$.

**Algorithm 5.6:** Word-wise multiplication.

```
Input: Integers $a$ and $b$ using $N$ words.
Output: $c = (a \cdot b)$.
1 for $i \leftarrow 0$ to $N - 1$ do
2     $c[i] \leftarrow 0$.
3 for $i \leftarrow 0$ to $N - 1$ do
4     $u \leftarrow 0$.
5     for $j \leftarrow 0$ to $N - 1$ do
6         $(u \mid\mid v) \leftarrow c[i + j] + (a[i] \cdot b[j]) + u$.
7         $c[i + j] \leftarrow v$.
8     $c[N + i] \leftarrow u$.
9 return $c$.
```

This algorithm is an operand-scanning approach, where $(u \mid\mid v)$ denotes a $(2W)$-bit double word, which has to be supported by the underlying programming language and architecture. In general, it is wise to choose $W$ such that a $(2W)$-bit integer is the maximum that can be handled.

The main multiplication of the current word is performed in line $6$, which is also called the *inner product operation* [BBD08]. Both $c[i + j]$ and $u$ are at most $2^W - 1$, which is the maximum value that can be represented with $W$ bits. The word-wise multiplication result $a[i] \cdot b[j]$ is at most $(2^W - 1)^2$. Hence,

$$\begin{aligned} c[i + j] + (a[i] \cdot b[j]) + u &\leq 2 \cdot (2^W - 1) + (2^W - 1)^2 \\ &= 2^{W+1} - 2 + 2^{2W} - 2^{W+1} + 1 = 2^{2W} - 1 \end{aligned}$$

is a value that can be stored in $(u \mid\mid v)$.

### 5.2.2 Classical Reduction and the Native Modulo Operator

After a multiplication, a $(2n)$-bit result $r$ needs to be reduced to an $n$-bit field element. By applying the same method as for additions and subtractions discussed in Section 5.1.1, it is possible to reduce such a result by just subtracting the modulus $p$ from $r$ until an integer $r' < p$ is reached. However, with $r$ and $p$ being a $(2n)$-bit and $n$-bit quantity, respectively, roughly $2^n$ subtractions would be necessary. While this

| Bits \ Operation | ADD | SUB | MUL | DIV | MOD | XOR | AND | CLMUL |
|---|---|---|---|---|---|---|---|---|
| 8 bits | 3 ns | 3 ns | 4 ns | 44 ns | 42 ns | 3 ns | 3 ns | 2 ns |
| 16 bits | 3 ns | 2 ns | 4 ns | 43 ns | 42 ns | 3 ns | 2 ns | 2 ns |
| 32 bits | 2 ns | 3 ns | 5 ns | 42 ns | 41 ns | 3 ns | 3 ns | 2 ns |
| 64 bits | 2 ns | 3 ns | 4 ns | 42 ns | 118 ns | 3 ns | 3 ns | 2 ns |

Table 5.1: Execution times of various CPU instructions in nanoseconds (ns).

method is indeed fast for very small field sizes (e.g. 3 bits), it is obviously too slow in general.

The most straightforward approach in most programming languages is to use the native modulo operator. For example, in C and C++ this is the % operator and it works up to a certain field size (up to $64$ bits on the tested machine). However, this method is not well-suited for larger field sizes, and is sometimes slower than different methods even in small fields.

Actual benchmark results of various operators, including the modulo operator, are given in Table 5.1. For comparison, the bitwise operators XOR and AND were also tested, together with the CLMUL instruction discussed in Section 5.3. This benchmark was run on an Intel i7-6700 CPU @ 3.40 GHz with GCC and compiler optimizations set to *O2*. All operations were tested in blocks of 5, these blocks were executed $10^6$ times, and the numbers indicate the average execution time for each block.

As one would assume, additions and subtractions are equally fast. Multiplications are slightly slower, but the difference is not noticeable in most applications. However, we can see that divisions and modulo operations, which use the division instruction internally, are significantly slower. The reason is that both of these operations are heavily microcoded, whereas additions and multiplications map to a single CPU instruction. This difference is also illustrated in [Fog17], where the number of microcode operations is given for older CPU generations. The instruction table also shows a significantly higher average number of clock cycles for DIV instructions on the Intel Skylake architecture, which is the architecture of the CPU used for the benchmark. The execution time for the last MOD value is higher than that for the division, because for each field size $n$, a $(2n)$-bit number is reduced modulo an $n$-bit number, which means that for $64$ bits the number to be reduced exceeds the word size.

### 5.2.3 Barrett Reduction

The following algorithms will now reduce a number $x \pmod m$ without needing the expensive division operation from the previous section. The first one of these algorithms is the Barrett reduction presented in 1986 [Bar86] and illustrated in Algorithm 5.7. This method works for any positive integer numbers $x$ and $m$. It needs the precomputation of $\mu = \lfloor B^{2k}/m \rfloor$, where $B$ is typically a quantity close to the word size of the CPU (e.g. $B = 2^l$ for some $l$) for multi-precision integers. Computing $\mu$ requires the division instruction in most cases, which makes it an expensive operation. Therefore, this value is often hard-coded and can then be used for every modular multiplication where the modulus is the same.

**Algorithm 5.7:** Barrett reduction.

```
Input: Integers $m, B \geq 3, k = \lfloor \log_B(m) \rfloor + 1, 0 \leq x < B^{2k}, \mu = \left\lfloor \frac{B^{2k}}{m} \right\rfloor$.
Output: $r = x \pmod m$.
1 $q \leftarrow \lfloor x/B^{k-1} \rfloor$.
2 $q \leftarrow q \cdot \mu$.
3 $q \leftarrow \lfloor q/B^{k+1} \rfloor$.
4 $r \leftarrow (x \pmod{B^{k+1}}) - ((q \cdot m) \pmod{B^{k+1}})$.
5 if $r < 0$ then
6     $r \leftarrow r + B^{k+1}$.
7 while $r \geq m$ do
8     $r \leftarrow r - m$.
9 return $r$.
```

Note that there are also two divisions in this algorithm, namely in lines 1 and 3. However, the divisors of these divisions are powers of 2, because $B^k = (2^l)^k = 2^{lk}$, where $k$ and $l$ are positive integers. Hence, these operations can be implemented efficiently with shift instructions. A similar argument holds for the modulo operations in line 4, which can be replaced by bitwise AND instructions. That is,

$$x/B^{k-1} = x \gg (\log_2(B) \cdot (k-1)), q/B^{k+1} = q \gg (\log_2(B) \cdot (k+1))$$

and

$$x \pmod{B^{k+1}} = x \,\&\, (B^{k+1} - 1), (q \cdot m) \pmod{B^{k+1}} = (q \cdot m) \,\&\, (B^{k+1} - 1),$$

where $\gg$ and $\&$ denote a bitwise right shift and a bitwise AND operation, respectively.

The *while* loop at the end is responsible for correcting subtractions, if the final value is still greater than or equal to $m$. However, in most cases no subtractions are required and only very rarely more than one subtraction is needed [Bar86]. A correctness proof of the algorithm is given in [HMV03].

### 5.2.4 Montgomery Conversion and Reduction

The Montgomery reduction algorithm is due to Montgomery and was introduced in 1985 [Mon85]. Its main idea is to perform the modular multiplication in a different domain, the *Montgomery domain*, where certain modular operations are cheaper. The resulting product is then transformed back into the original domain to obtain the final result.

Let $m$ be the modulus as before and $x$ an arbitrary field element. Furthermore, let $R = 2^k$, where $k = \lfloor \log_2(m) \rfloor + 1$ (hence, $k$ is the number of bits of $m$ and $R$ is the smallest power of 2 greater than $m$). Then the Montgomery representation of $x$ is

$$x' = x \cdot R \pmod{m}$$

and the conversion back to the original $x$ is

$$x = x' \cdot R^{-1} \pmod{m},$$

where $R^{-1} \pmod{m}$ is the modular multiplicative inverse of $R \pmod{m}$ such that $R \cdot R^{-1} \equiv 1 \pmod{m}$. This means that an algorithm is needed in order to efficiently divide a number $x'$ by $R$. The main idea here is to use the fact that $R$ is a power of 2. If the $k$ least significant bits of $x'$ are all zeros, then a division by $R$ can be performed in an efficient way by simply shifting $x'$ to the right for $k$ bits. However, in general this is not the case and the idea is then to add a certain value $t$ to $x'$, which makes the $k$ least significant bits of $x'$ become zero. The quantity $t$ is chosen in such a way that

$$\begin{aligned} t &= c \cdot m, \\ x' + t &= x' \pmod{m}, \\ x' + t &= 0 \pmod{R}, \end{aligned}$$

where $c \geq 1 \in \mathbb{N}$. Note that

$$x' + c \cdot m \equiv 0 \pmod{R} \implies c \equiv -x' \cdot m^{-1} \pmod{R}.$$

In order to calculate this, $m_{-inv} = -m^{-1} \pmod{R}$ has to be precomputed first, such that $c = x' \cdot m_{-inv} \pmod{R}$. This value is the same for every reduction modulo $m$.

The entire procedure is summarized in Algorithm 5.8.

**Algorithm 5.8:** Montgomery reduction.

```
  Input: Integers $x', R, m, m_{-inv}$.
  Output: $x = x' \cdot R^{-1} \pmod{m}$.
1 $c \leftarrow x' \cdot m_{-inv} \pmod{R}$.
2 $x \leftarrow x' + (c \cdot m)$.
3 $x \leftarrow x/R$.
4 if $x \geq m$ then
5     $x \leftarrow x - m$.
6 return $x$.
```

Note that the modulo operation in line 1 can be implemented with an AND operation and the division in line 3 is a right shift operation, since $R$ is a power of 2. Now, to compute $x \cdot y \pmod{m}$, one multiplication and two reductions are needed, as shown in Algorithm 5.9. In this algorithm, line 4 yields the Montgomery representation of $x \cdot y \pmod{m}$, and another conversion is then used to obtain the final result in line 5. This works because

$$\begin{aligned} x' &= x \cdot R \pmod{m}, \\ y' &= y \cdot R \pmod{m}, \end{aligned}$$

and therefore

$$x' \cdot y' \equiv (x \cdot R) \cdot (y \cdot R) \equiv z \cdot R^2 \equiv z' \cdot R \pmod{m}.$$

The computational cost for the Montgomery multiplication is comparatively high for one single multiplication, mainly because the input values need to be converted to the Montgomery domain first, and converted back to their original form after the

**Algorithm 5.9:** Montgomery multiplication.

```
  Input: Integers $x, y \in \{0, 1, \dots, m-1\}, R, m$.
  Output: $z = x \cdot y \pmod{m}$.
1 $x' \leftarrow x \cdot R \pmod{m}$.
2 $y' \leftarrow y \cdot R \pmod{m}$.
3 $t \leftarrow x' \cdot y'$.
4 $z' \leftarrow MontgomeryReduction(t)$.
5 $z \leftarrow MontgomeryReduction(z')$.
6 return $z$.
```

procedure. These transformations are expensive. However, if an exponentiation with a sufficiently large exponent is needed (such as for RSA), a bitwise Montgomery exponentiation only needs to transform these values once for all subsequent multiplications. In this setting, the Montgomery method tends to be faster than Barrett's reduction, as further explained in [HMV03]. The exponentiation, however, was not implemented for ZKB++, hence it will not be described here.

A short comparison of the Montgomery multiplication and the normal multiplication together with the Barrett reduction in prime fields of sizes $16$ bits and $32$ bits is given in Table 5.2. The number of runs, the hardware, and the parameters are the same as for the previous benchmark. Normal multiplications with subsequent Barrett reductions prove to be faster in this setting, mainly due to the computationally expensive conversions into the Montgomery domain, which require modular operations themselves. Each of these methods is almost equally fast for both $16$-bit and $32$-bit fields, because all operations are word size operations.

Note that in order to reduce the computational cost of Montgomery multiplications in ZKB$++$, it is also possible to perform the expensive Montgomery conversions only at the beginning and at the end of the protocol. This optimization was not implemented during the practical part of this thesis, but is further described in Section 7.2.

| Field size \ Method | Normal MUL + Barrett reduction | Montgomery MUL |
|---|---|---|
| 16 bits | 17 ns | 41 ns |
| 32 bits | 17 ns | 38 ns |

Table 5.2: Comparison of Barrett reduction and Montgomery reduction.

### 5.2.5 Crandall Prime Number Reduction

The previous two reduction methods, explained in Section 5.2.3 and Section 5.2.4, work for arbitrary prime numbers and do not exploit any properties of a specific prime. The following two reduction methods, on the other hand, make use of prime numbers of special form. In further consequence, these methods make reductions modulo special prime numbers significantly faster.

The first one of these two algorithms is the reduction in a prime field $\mathbb{F}_p$, where $p = 2^n - c$ and $c$ is a small integer number that can be stored in one word. Prime numbers with this property are called Crandall prime numbers or Pseudo-Mersenne prime numbers.

Let $p = 2^n - c$ be a Crandall prime number and let $r = a \cdot b$, where $a, b \in \mathbb{F}_p$, be the result of a non-modular multiplication. Then $r < p^2$ and

$$r = (r' \cdot 2^n) + t \equiv (r' \cdot c) + t \pmod{p},$$

because $2^n \equiv c \pmod{p}$. Applying this congruence recursively on $r'$ yields Algorithm 5.10.

This algorithm was presented in 1992 [Cra92] and a correctness proof is given in [MOV96]. Note that all modulo operations and divisions can be replaced by bitwise AND operations and right shift operations, respectively, as explained in Section 5.2.3. Aside from the conditioned *while* loops, the most expensive operation is the multiplication in line $4$, especially in larger fields.

### 5.2.6 Solinas Prime Number Reduction

This section will now describe a reduction algorithm presented in 1999 [Sol99], which is based on Solinas prime numbers (also called Generalized Mersenne primes).

**Algorithm 5.10:** Crandall prime number reduction.

```
  Input: Integers $x, n$ and $c$ such that $p = 2^n - c$ is prime and $c < 2^W$.
  Output: $r = x \pmod{p}$.
1 $q \leftarrow x/2^n$.
2 $r \leftarrow x \pmod{2^n}$.
3 while $q > 0$ do
4     $t \leftarrow c \cdot q$.
5     $r \leftarrow r + (t \pmod{2^n})$.
6     $q \leftarrow t/2^n$.
7 while $r \geq p$ do
8     $r \leftarrow r - p$.
9 return $x$.
```

Similar to the reduction using Crandall numbers, this algorithm makes use of special properties of a prime number. However, compared to the Crandall reduction, it needs a substantial amount of precomputation, which will now be explained in detail.

A Solinas prime number (or Generalized Mersenne prime number) is a prime number of the form

$$p = t^n - \left(\sum_{i=0}^{n-1} c_i \cdot t^i\right) = t^n - c_{n-1} \cdot t^{n-1} - \cdots - c_0,$$

where $c_i \in \{-1, 0, 1\}$ for the use cases presented in this thesis. The idea of the fast reduction method is based on the fact that the degree of $p$ can be kept very low for certain prime numbers in a specific base representation. For example, consider the 64-bit prime number $p_{64} = 2^{64} - 2^8 - 1$. Then, by converting $p_{64}$ from base 2 to base $2^8$, we have

$$p_{64} = 2^{64} - 2^8 - 1 = (2^8)^{(64/8)} - (2^8)^{(8/8)} - (2^8)^{(0/8)} = t^8 - t - 1.$$

Consider now a prime number $p_t = t^d - c_{d-1} \cdot t^{d-1} - \cdots - c_0$ of degree $d$ converted to base $t$. The first step of the reduction algorithm is to generate the *modular reduction matrix* $X$ of $p_t$, which is done by computing all residues of $t^i \pmod{p_t}$, where

$d \leq i < 2d$. That is,

$$X = \begin{pmatrix} X_{0,0} & \cdots & X_{0,d-1} \\ \vdots & \ddots & \vdots \\ X_{d-1,0} & \cdots & X_{d-1,d-1} \end{pmatrix},$$

where the matrix entries are defined as follows:

$$X_{0,j} = c_j \text{ for } 0 \leq j < d,$$
$$X_{i,j} = \begin{cases} X_{i-1,d-1} \cdot c_0 \text{ if } j = 0, \\ X_{i-1,j-1} + (X_{i-1,d-1} \cdot c_j) \text{ otherwise.} \end{cases}$$

The matrix now contains the residues mentioned above, that is,

$$t^{d+i} \equiv \sum_{j=0}^{d-1} X_{i,j} \cdot t^j \pmod{p_t},$$

where $0 \leq i < d$, or, equivalently,

$$\begin{pmatrix} t^d \\ \vdots \\ t^{2d-1} \end{pmatrix} \equiv X \begin{pmatrix} 1 \\ \vdots \\ t^{d-1} \end{pmatrix} \pmod{p_t}.$$

Now, let $r = (r_{2d-1} \mid\mid r_{2d-2} \mid\mid \cdots \mid\mid r_0) < {p_t}^2$ be an integer which has to be reduced. Note that $r < {p_t}^2$ if $r$ is the result of a multiplication of two integers smaller than $p_t$. Then

$$\sum_{i=0}^{2d-1} r_i \cdot t^i = \begin{pmatrix} r_0 & r_1 & \cdots & r_{d-1} \end{pmatrix} \begin{pmatrix} 1 \\ \vdots \\ t^{d-1} \end{pmatrix} + \begin{pmatrix} r_d & r_{d+1} & \cdots & r_{2d-1} \end{pmatrix} \begin{pmatrix} t^d \\ \vdots \\ t^{2d-1} \end{pmatrix}$$
$$\equiv \left( \begin{pmatrix} r_0 & r_1 & \cdots & r_{d-1} \end{pmatrix} + \begin{pmatrix} r_d & r_{d+1} & \cdots & r_{2d-1} \end{pmatrix} \cdot X \right) \begin{pmatrix} 1 \\ \vdots \\ t^{d-1} \end{pmatrix} \pmod{p_t}.$$

Let

$$r' = \begin{pmatrix} r'_0 & r'_1 & \cdots & r'_{d-1} \end{pmatrix} = \begin{pmatrix} r_0 & r_1 & \cdots & r_{d-1} \end{pmatrix} + \begin{pmatrix} r_d & r_{d+1} & \cdots & r_{2d-1} \end{pmatrix} \cdot X.$$

Then

$$\sum_{i=0}^{2d-1} r_i \cdot t^i \equiv \sum_{i=0}^{d-1} r'_i \cdot t^i \pmod{p_t}.$$

Thus, $r' \equiv r \pmod{p_t}$, and converting $r'$ back to base 2 gives the final result.

Most of the values, including the matrix, can be precomputed. However, in order to find $r'$, a matrix multiplication with base-$t$ values of $r$ and the reduction matrix $X$ is necessary. This can be implemented in a more efficient way, which will now be explained.

The matrix multiplication can be accelerated by making use of two properties. Firstly, the number of entries of $X$ is limited and, more importantly, relatively small, since we aim for Solinas primes with a small degree $d$. This means that multiplications can be replaced by a sequence of modular additions and subtractions. Secondly, the matrix product is a row vector and can be written as a sum of row vectors representing $d$-digit integer numbers. This method is illustrated in Algorithm 5.11 for modular additions, and works analogously for the subtractions. Using this method and adding $S^+$ and $S^-$ to $(r_{d-1} \,||\, r_{d-2} \,||\, \cdots \,||\, r_0)$ yields the final result $r' = r \pmod{p_t}$.

**Algorithm 5.11:** Modular additions for Solinas prime reduction.

```
   Input: Integer $r = (r_{2d-1} \,||\, r_{2d-2} \,||\, \cdots \,||\, r_0) < {p_t}^2$, $X$.
   Output: Sum $S^+$ of modular additions.
1  $S^+ \leftarrow 0$.
2  $temp = (temp_{d-1} \,||\, temp_{d-2} \,||\, \cdots \,||\, temp_0) \leftarrow 0$.
3  while $\exists X_{i,j}$ such that $X_{i,j} > 0$ do
4      for each column $j$ of $X$ do
5          if $\exists X_{i,j}$ such that $X_{i,j} > 0$ then
6              $i \leftarrow \min\{k : X_{k,j} > 0\}$.
7              $temp_j \leftarrow r_{d+i}$.
8              $X_{i,j} \leftarrow X_{i,j} - 1$.
9          else
10             $temp_j \leftarrow 0$.
11     $S^+ \leftarrow S^+ + temp \pmod{p_t}$.
12 return $S^+$.
```

However, this calculation still includes two loops and needs a comparatively large amount of operations. This can be improved by exploiting the fact that the positions of positive and negative entries of $X$ are known beforehand and also fixed during the precomputation. Moreover, note that the number of modular additions (and subtractions) depends on the number of evaluations of the *while* loop in line $3$. This loop is executed exactly $w_+$ times, where $w_+$ is the largest column sum of positive entries in $X$. For the subtractions, the decrement is replaced by an increment and thus the loop is executed $w_-$ times, where $-w_-$ is the smallest column sum of negative entries in $X$. For the following descriptions, $w_+$ denotes the *modular addition weight* and $w_-$ denotes the *modular subtraction weight*.

By considering these properties, the matrix $X$ can be separated into a positive matrix $X^+$ and a negative matrix $X^-$, containing all strictly positive entries and strictly negative entries of $X$, respectively. Thus,

$$X^+{}_{i,j} = \begin{cases} X_{i,j} & \text{if } X_{i,j} > 0, \\ 0 & \text{otherwise.} \end{cases}$$

and

$$X^-{}_{i,j} = \begin{cases} X_{i,j} & \text{if } X_{i,j} < 0, \\ 0 & \text{otherwise.} \end{cases}$$

Using the matrix $X^+$ and the weight $w_+$, the indices for the modular additions can be precomputed by applying Algorithm 5.12. Again, the algorithm to compute the indices for the modular subtractions is very similar and just replaces the decrement in line $8$ by an increment. The assignment in line $10$ has no effect other than to distinguish this entry from legit indices. If the index “$-1$” is found during the reduction, the corresponding quantity of the number is $0$, because no strictly positive entry was found in $X^+$.

Having computed the modular addition and subtraction matrices $M_+$ and $M_-$, the implementation of the final reduction algorithm is straightforward and given in Algorithm 5.13. By choosing the prime numbers carefully, all loops can be avoided, which will be explained in the next chapter. Moreover, this algorithm is the only one presented here which works exclusively with modular additions and subtractions.

**Example for $p_{64}$.** Here, a short example for the $64$-bit prime number $p_{64} = 2^{64} - 2^8 - 1$ will be given. Using the transformation to base $t$ yields $p_t = t^8 - t - 1$, where

**Algorithm 5.12:** Modular addition matrix for Solinas prime reduction.

```
   Input: $X^+$, $w_+$.
   Output: Matrix $M_+$ containing indices for modular additions.
1  for $k \leftarrow 0$ to $w_+ - 1$ do
2      for $j \leftarrow 0$ to $d - 1$ do
3          $i \leftarrow 0$.
4          while $i < d$ and ${X^+}_{i,j} = 0$ do
5              $i \leftarrow i + 1$.
6          if $i < d$ then
7              ${M_+}_{k,j} \leftarrow d + i$.
8              ${X^+}_{i,j} \leftarrow {X^+}_{i,j} - 1$.
9          else
10             ${M_+}_{k,j} \leftarrow -1$.
11 return $M_+$.
```

$t = 2^8$. Calculating the residues of $t^{15}, t^{14}, \dots, t^8 \pmod{p_t}$ results in the $d \times d$ matrix $X$, where $d = 8$. Then, following holds for $i = 0, 1, \dots, 7$:

$$t^{8+i} \equiv \sum_{j=0}^{7} (X_{i,j} \cdot t^j) \pmod{p_t}.$$

Now, let $r = (r_{15} \mid\mid r_{14} \mid\mid \cdots \mid\mid r_0)$ be the result of a multiplication of two 64-bit integers, where each $r_i$ is a 8-bit quantity. Then

$$r = \sum_{i=0}^{15} (r_i \cdot t^i) \equiv \sum_{i=0}^{7} (r'_i \cdot t^i) \pmod{p_t},$$

where

$$\begin{pmatrix} r'_0 & r'_1 & \cdots & r'_7 \end{pmatrix} = \begin{pmatrix} r_0 & r_1 & \cdots & r_7 \end{pmatrix} + \begin{pmatrix} r_8 & r_9 & \cdots & r_{15} \end{pmatrix} \cdot X.$$

In order to avoid the matrix multiplications, the modular addition matrix $M_+$ and the modular subtraction matrix $M_-$ are precomputed using the algorithms described above.

**Algorithm 5.13:** Solinas reduction algorithm.

```
Input: Integer $r = (r_{2d-1} \,||\, r_{2d-2} \,||\, \cdots \,||\, r_0) < {p_t}^2$, weights $w_+$ and $w_-$,
       matrices $M_+$ and $M_-$.
Output: $r' = (r'_{d-1} \,||\, r'_{d-2} \,||\, \cdots \,||\, r'_0) = r \pmod{p_t}$.
1  $temp = (temp_{d-1} \,||\, temp_{d-2} \,||\, \cdots \,||\, temp_0) \leftarrow 0$.
2  $r' \leftarrow (r_{d-1} \,||\, r_{d-2} \,||\, \cdots \,||\, r_0)$.
3  for $i \leftarrow 0$ to $w_+ - 1$ do
4      for $j \leftarrow 0$ to $d - 1$ do
5          if $M_{+i,j} = -1$ then
6              $temp_j \leftarrow 0$.
7          else
8              $temp_j \leftarrow r_{M_{+i,j}}$.
9      $r' \leftarrow r' + temp \pmod{p_t}$.
10 for $i \leftarrow 0$ to $w_- - 1$ do
11     for $j \leftarrow 0$ to $d - 1$ do
12         if $M_{-i,j} = -1$ then
13             $temp_j \leftarrow 0$.
14         else
15             $temp_j \leftarrow r_{M_{-i,j}}$.
16     $r' \leftarrow r' - temp \pmod{p_t}$.
17 return $r'$.
```

Solinas primes for fast reductions are used in many different areas of cryptography. For example, the NIST recommends to use prime numbers which fulfill the properties described above to implement modular arithmetic for elliptic curves [KR13]. Moreover, an elliptic curve presented in 2015 [Ham15] makes use of a Solinas prime. The main difference between this prime and the primes suggested by the NIST is that the prime number $p = 2^{448} - 2^{224} - 1$ used in this publication is of the special form $2^n - 2^{n/2} - 1$, and the largest exponent, $448$, is divided without remainder by 28, 32, and $56$. This supports fast computations with a comparatively low amount of indexing operations on 32-bit machines and on $64$-bit machines. Moreover, the modulus of a very recent proposal for a post-quantum secure key encapsulation mechanism by the same author [Ham17] is also a Solinas prime number of the form $2^n - 2^{n/2} - 1$.

Choosing prime numbers of this special form is, unfortunately, not possible for most of the field sizes used in the new implementation of MiMC and GMiMC.

### 5.2.7 Modular Addition and Modular Subtraction Matrices for Chosen Solinas Primes

Table 5.3 gives an overview of all the prime numbers used for MiMC and GMiMC in the new implementation of ZKB++. Every prime number $p$ listed here fulfills the requirement $\gcd(p-1,3) \overset{!}{=} 1$.

| Field size | $p$ | $t$ | $p_t$ | $w_+$ | $w_-$ |
|---|---|---|---|---|---|
| 3 bits | $2^3-2-1$ | $2$ | $t^3-t-1$ | 3 | 0 |
| 4 bits | $2^4-2^2-1$ | $2^2$ | $t^2-t-1$ | 3 | 0 |
| 16 bits | $2^{16}-2^4-1$ | $2^4$ | $t^4-t-1$ | 3 | 0 |
| 32 bits | $2^{32}-2^4-1$ | $2^4$ | $t^8-t-1$ | 3 | 0 |
| 32 bits | $2^{32}-2^{24}-1$ | $2^8$ | $t^4-t^3-1$ | 5 | 0 |
| 64 bits | $2^{64}-2^8-1$ | $2^8$ | $t^8-t-1$ | 3 | 0 |
| 136 bits | $2^{136}-2^8-1$ | $2^8$ | $t^{17}-t-1$ | 3 | 0 |
| 256 bits | $2^{256}-2^{184}+2^{32}+1$ | $2^8$ | $t^{32}-t^{23}+t^4+1$ | 4 | 8 |
| 272 bits | $2^{272}-2^{40}-1$ | $2^8$ | $t^{34}-t^5-1$ | 3 | 0 |

Table 5.3: Prime numbers used for the implementation of MiMC and GMiMC.

The corresponding modular addition and modular subtraction matrices $M_+$ and $M_-$ are given below. In these matrices, the entry "$-1$" indicates that the value to be added or subtracted has only zero bits at this position.

**Matrices for $p_3 = 2^3 - 2 - 1$.**

$$M_+ = \begin{pmatrix} 3 & 3 & 4 \\ 5 & 4 & 5 \\ -1 & 5 & -1 \end{pmatrix}$$

and

$$M_- = N/A.$$

**Matrices for $p_4 = 2^4 - 2^2 - 1$.**

$$M_+ = \begin{pmatrix} 2 & 2 \\ 3 & 3 \\ -1 & 3 \end{pmatrix}$$

and

$$M_- = N/A.$$

**Matrices for $p_{16} = 2^{16} - 2^4 - 1$.**

$$M_+ = \begin{pmatrix} 4 & 4 & 5 & 6 \\ 7 & 5 & 6 & 7 \\ -1 & 7 & -1 & -1 \end{pmatrix}$$

and

$$M_- = N/A.$$

**Matrices for $p_{32} = 2^{32} - 2^4 - 1$.**

$$M_+ = \begin{pmatrix} 8 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ 15 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ -1 & 15 & -1 & -1 & -1 & -1 & -1 & -1 \end{pmatrix}$$

and

$$M_- = N/A.$$

**Matrices for $p'_{32} = 2^{32} - 2^{24} - 1$.**

$$M_+ = \begin{pmatrix} 4 & 5 & 6 & 4 \\ 5 & 6 & 7 & 5 \\ 6 & 7 & -1 & 6 \\ 7 & -1 & -1 & 7 \\ -1 & -1 & -1 & 7 \end{pmatrix}$$

and

$$M_- = N/A.$$

**Matrices for $p_{64} = 2^{64} - 2^8 - 1$.**

$$M_+ = \begin{pmatrix} 8 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ 15 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ -1 & 15 & -1 & -1 & -1 & -1 & -1 & -1 \end{pmatrix}$$

and

$$M_- = N/A.$$

**Matrices for $p_{136} = 2^{136} - 2^8 - 1$.**

$$M_+ = \begin{pmatrix} {M_+}^{(1)} & {M_+}^{(2)} \end{pmatrix},$$

where

$${M_+}^{(1)} = \begin{pmatrix} 17 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \\ 33 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 \\ -1 & 33 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{pmatrix},$$

$${M_+}^{(2)} = \begin{pmatrix} 25 & 26 & 27 & 28 & 29 & 30 & 31 & 32 \\ 26 & 27 & 28 & 29 & 30 & 31 & 32 & 33 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{pmatrix},$$

and

$$M_- = N/A.$$

**Matrices for $p_{256} = 2^{256} - 2^{184} + 2^{32} + 1$.**

$$M_+ = \begin{pmatrix} {M_+}^{(1)} & {M_+}^{(2)} & {M_+}^{(3)} \end{pmatrix},$$

where

$${M_+}^{(1)} = \begin{pmatrix} 60 & 61 & 62 & 63 & 60 & 61 & 62 & 63 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{pmatrix},$$

$$M_+{}^{(2)} = \begin{pmatrix} -1 & \cdots & -1 \\ \vdots & \ddots & \vdots \\ -1 & \cdots & -1 \end{pmatrix}_{4\times 15},$$

$$M_+{}^{(3)} = \begin{pmatrix} 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 \\ 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 & 49 \\ 50 & 51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 \\ 59 & 60 & 61 & 62 & 63 & -1 & -1 & -1 & -1 \end{pmatrix},$$

and

$$M_- = \begin{pmatrix} M_-{}^{(1)} & M_-{}^{(2)} & M_-{}^{(3)} \end{pmatrix},$$

where

$$M_-{}^{(1)} = \begin{pmatrix} 32 & 33 & 34 & 35 & 32 & 33 & 34 & 35 & 36 & 37 & 38 \\ 41 & 42 & 43 & 44 & 36 & 37 & 38 & 39 & 40 & 41 & 42 \\ 50 & 51 & 52 & 53 & 41 & 42 & 43 & 44 & 45 & 46 & 47 \\ 59 & 60 & 61 & 62 & 45 & 46 & 47 & 48 & 49 & 50 & 51 \\ -1 & -1 & -1 & -1 & 50 & 51 & 52 & 53 & 54 & 55 & 56 \\ -1 & -1 & -1 & -1 & 54 & 55 & 56 & 57 & 58 & 59 & 60 \\ -1 & -1 & -1 & -1 & 59 & 60 & 61 & 62 & 63 & -1 & -1 \\ -1 & -1 & -1 & -1 & 63 & -1 & -1 & -1 & -1 & -1 & -1 \end{pmatrix},$$

$$M_-{}^{(2)} = \begin{pmatrix} 39 & 40 & 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 & 49 \\ 43 & 44 & 45 & 46 & 47 & 48 & 49 & 50 & 51 & 52 & 53 \\ 48 & 49 & 50 & 51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 \\ 52 & 53 & 54 & 55 & 56 & 57 & 58 & 59 & 60 & 61 & 62 \\ 57 & 58 & 59 & 60 & 61 & 62 & 63 & -1 & -1 & -1 & -1 \\ 61 & 62 & 63 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{pmatrix},$$

$$M_-{}^{(3)} = \begin{pmatrix} 50 & 51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 & 59 \\ 54 & 55 & 56 & 57 & 58 & 59 & 60 & 61 & 62 & 63 \\ 59 & 60 & 61 & 62 & 63 & -1 & -1 & -1 & -1 & -1 \\ 63 & 60 & 61 & 62 & 63 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{pmatrix}.$$

**Matrices for $p_{272} = 2^{272} - 2^{40} - 1$.**

$$M_+ = \begin{pmatrix} M_+{}^{(1)} & M_+{}^{(2)} & M_+{}^{(3)} \end{pmatrix},$$

where

$$M_+{}^{(1)} = \begin{pmatrix} 34 & 35 & 36 & 37 & 38 & 34 & 35 & 36 & 37 & 38 & 39 & 40 \\ 63 & 64 & 65 & 66 & 67 & 39 & 40 & 41 & 42 & 43 & 44 & 45 \\ -1 & -1 & -1 & -1 & -1 & 63 & 64 & 65 & 66 & 67 & -1 & -1 \end{pmatrix},$$

$$M_+{}^{(2)} = \begin{pmatrix} 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 & 49 & 50 & 51 \\ 46 & 47 & 48 & 49 & 50 & 51 & 52 & 53 & 54 & 55 & 56 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{pmatrix},$$

$$M_+{}^{(3)} = \begin{pmatrix} 52 & 53 & 54 & 55 & 56 & 57 & 58 & 59 & 60 & 61 & 62 \\ 57 & 58 & 59 & 60 & 61 & 62 & 63 & 64 & 65 & 66 & 67 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{pmatrix},$$

and

$$M_- = N/A.$$

Intuitively, maximizing the new base $t$ should yield the best performance, because it means that fewer shift and indexing operations have to be performed in order to change the base-$t$ value of a number at a certain position. Yet, this is not always true. For example, $p'_{32}$ uses $8$-bit quantities, whereas $p_{32}$ uses $4$-bit quantities. However, the former also needs $5$ modular additions, while the latter needs only $3$. This makes computations in $\mathbb{F}_{p_{32}}$ noticeably faster than computations in $\mathbb{F}_{p'_{32}}$.

A similar argument holds for field sizes, where operations in smaller fields are, in general, faster. Nonetheless, modular multiplications in $\mathbb{F}_{p_{256}}$ are slower then those in $\mathbb{F}_{p_{272}}$. The reason is that, even though non-modular multiplications are faster for $256$-bit values, $\mathbb{F}_{p_{256}}$ needs more modular additions and modular subtractions during the reduction steps. So, when the number of additions and subtractions in a circuit is not significantly higher than the number of multiplications (and if storage is not an issue), it is better to choose $\mathbb{F}_{p_{272}}$ over $\mathbb{F}_{p_{256}}$.

Another convenient property of most of these Solinas primes is that when representing them as a Crandall prime number $2^n - c$, then $c$ is very close to a power of $2$. This means that the multiplications used in the Crandall reduction algorithm can actually be implemented by using a shift operation together with one addition. The impact of this optimization is, however, dependent on the underlying architecture and on the compiler. On the tested machine (Intel i7-6700 CPU @ 3.40 GHz, GCC compiler) there was no noticeable difference.

To conclude, prime numbers were chosen by trying to maximize $t$ and, at the same time, keep the number of modular additions, modular subtractions, and indexing operations as low as possible.

## 5.3 Modular Multiplications in Binary Fields

Binary field multiplications make use of carry-less multiplications, hence they have to be implemented differently. This section will describe multiplications and reductions used for binary fields in GMiMC.

### 5.3.1 Basic Multiplication

In every reduction method described here, the multiplication result $r$ of two polynomials $a, b \in \mathbb{F}_{2^n}$ is first computed separately, making $r$ a number of at most $2n - 1$ bits.

**Polynomial Multiplication Using the CLMUL Instruction.** The CLMUL (carry-less multiplication) instruction is a CPU instruction for performing multiplications of two polynomials of degree at most $63$ and is supported by most modern architectures. It is slightly faster than the standard multiplication instruction, and the fastest method of multiplying two polynomials on the tested machine. However, for binary fields with more than $64$ bits, the instruction has to be applied within a word-wise method. The only binary field with more than $64$ bits used in this implementation of GMiMC is $\mathbb{F}_{2^{65}}$ (note that $\mathbb{F}_{2^{64}}$ cannot be used, due to reasons described in Section 2.6.2). The non-modular multiplication with operands in this field is shown in Algorithm 5.14, where $(r_1||r_0)$ is a $128$-bit quantity. Most of the necessary operations for larger field sizes up to $128$ bits can be omitted in the $65$-bit case, because both $a[1]$ and $b[1]$ only store one bit for $W = 64$ and $N = 2$, and thus any polynomial multiplication of two numbers using these values is the same as a normal multiplication with either $0$ or $1$. This means that the CLMUL instruction, which would have to be called $4$ times in the general 2-word case, is only called once here.

**Algorithm 5.14:** Polynomial multiplication of two $65$-bit values using the CLMUL instruction, where $W = 64$ and $N = 2$.

```
   Input: Binary polynomials $a, b \in \mathbb{F}_{2^{65}}$.
   Output: $c = a \cdot b$.
1  $(r_1 \mid\mid r_0) \leftarrow CLMUL(a[0], b[0])$.
2  $c[0] \leftarrow r_0$.
3  $c[1] \leftarrow r_1$.
4  $t \leftarrow a[0] \cdot b[1]$.
5  $c[1] \leftarrow c[1] \oplus t$.
6  $t \leftarrow a[1] \cdot b[0]$.
7  $c[1] \leftarrow c[1] \oplus t$.
8  $t \leftarrow a[1] \cdot b[1]$.
9  $c[2] \leftarrow t$.
10 return $c$.
```

**Polynomial Multiplication Using Windows of Width $w$.** For architectures not supporting the CLMUL instruction, a different method was implemented as an alternative. This method multiplies the two polynomials in windows of width $w$, where increasing $w$ yields a faster computation at the cost of a larger storage overhead.

The detailed approach is shown in Algorithm 5.15. The computation of $T$ in the first loop can be accelerated by adding $b$ to previously computed results of $T_t$ instead of using a new multiplication for each value. A window width of $w = 4$ was chosen for the binary fields in GMiMC.

**Algorithm 5.15:** Polynomial multiplication using windows of width $w$.

```
   Input: Binary polynomials $a$ and $b$ using $N$ words, window width $w$.
   Output: $c = (c_{N'-1} \mid\mid c_{N'-2} \mid\mid \cdots \mid\mid c_0) = a \cdot b$.
1  foreach polynomial $t$ of degree at most $(w-1)$ do
2      $T_t \leftarrow b \cdot t$.
3  for $i \leftarrow ((W/w) - 1)$ to $0$ do
4      for $j \leftarrow 0$ to $N - 1$ do
5          $t \leftarrow a[j]_{wi}$, where $a[j]_{wi}$ is the $w$-bit quantity starting at bit $wi$ of $a[j]$.
6          $(c_{N'-1} \mid\mid c_{N'-2} \mid\mid \cdots \mid\mid c_j) \leftarrow (c_{N'-1} \mid\mid c_{N'-2} \mid\mid \cdots \mid\mid c_j) \oplus T_t$.
7      if $i > 0$ then
8          $c \leftarrow c \cdot 2^w$.
9  return $c$.
```

## 5.3.2 Classical Reduction

After computing the $(2n-1)$-bit product $r$ of two $n$-bit polynomials in $\mathbb{F}_{2^n}$, the next step is to reduce $r$ in order to obtain $r' = r \pmod{f_n}$, where $f_n$ is the irreducible polynomial of degree $n$ (stored in $n+1$ bits) used for reduction. A classical method for arbitrary irreducible polynomials is given in Algorithm 5.16 [HMV03], where $f_n = 2^n + m$ is the irreducible polynomial, and the set $T$, where $T_k = 2^k \cdot m$ for $0 \leq k \leq (W-1)$, is precomputed. This is a bitwise reduction and is thus rather expensive, especially when compared to the prime field reduction algorithms discussed in Section 5.2.

## 5.3.3 Faster Word-Wise Reduction

If the irreducible polynomial can be freely chosen, the reduction algorithm can be implemented using a significantly faster approach. In particular, trinomials and

**Algorithm 5.16:** Bitwise reduction for arbitrary irreducible polynomials.

```
Input: Binary polynomial $r$ of degree at most $2n-2$ and set of precomputed
       values $T$, where $T_k = 2^k \cdot m$ for $0 \leq k \leq (W-1)$.
Output: $r' = (r'_{N-1} \mid\mid r'_{N-2} \mid\mid \cdots \mid\mid r'_0) = r \pmod{f_n}$.
1 for $i \leftarrow (2n-2)$ to $n$ do
2     if $r_i = 1$ then
3         $j \leftarrow \lfloor (i-n)/W \rfloor$.
4         $k \leftarrow (i-n) - (W \cdot j)$.
5         $(r'_{N-1} \mid\mid r'_{N-2} \mid\mid \cdots \mid\mid r'_j) \leftarrow (r'_{N-1} \mid\mid r'_{N-2} \mid\mid \cdots \mid\mid r'_j) \oplus T_k$.
6 return $r'$.
```

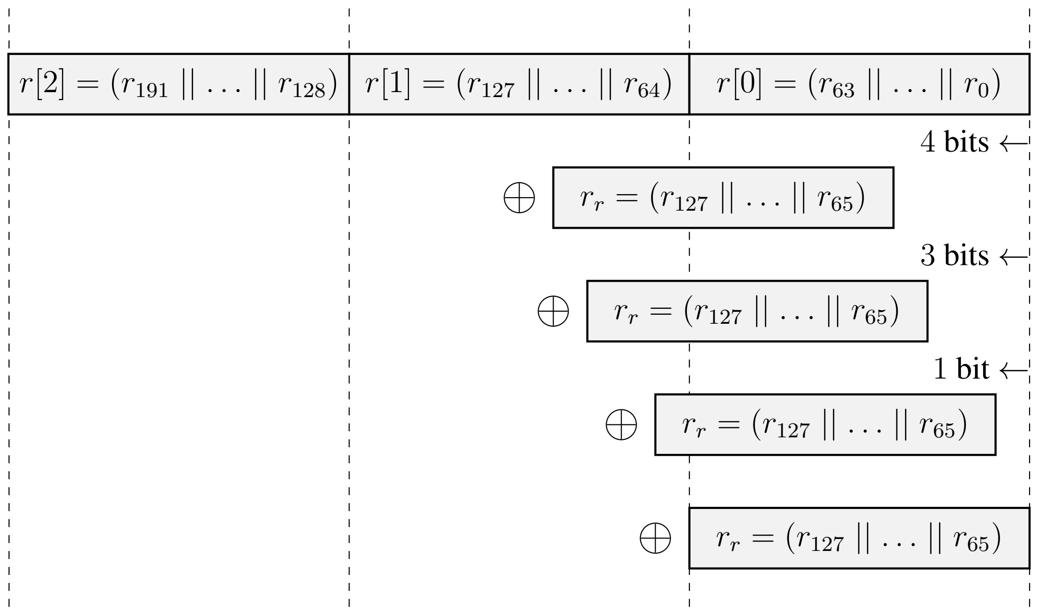
pentanomials with middle terms close to each other allow for a comparatively fast method [AM05].

For example, let $f_{65} = 2^{65} + 2^4 + 2^3 + 2 + 1$ be the irreducible polynomial used for computations in a 65-bit binary field. Given a $(2n-1)$-bit product $r$ of two $n$-bit polynomials, following equations hold:

$$\begin{aligned}
2^{65} &\equiv 2^4 + 2^3 + 2 + 1 \pmod{f_{65}}, \\
2^{66} &\equiv 2^5 + 2^4 + 2^2 + 2 \pmod{f_{65}}, \\
&\vdots \\
2^{128} &\equiv 2^{67} + 2^{66} + 2^{64} + 2^{63} \pmod{f_{65}}.
\end{aligned}$$

These congruences allow to reduce all coefficients of $2^{128}, 2^{127}, \ldots, 2^{65}$ by adding them consecutively to bits $4, 3, 1$, and $0$ of $r$. This method is illustrated in Figure 5.1, where the $63$ most significant bits of $r[1]$ are reduced by adding them to $r$ at the corresponding degrees. The illustrated step is the second step of the algorithm, and the first step is adding the least significant bit of $r[2]$ to the correct positions of $r$. Hence, reduction is performed one word at a time, starting from the most significant word. The final result $r'$ can then be obtained by simply taking the $65$ least significant bits of $r$ after adding the last word.

**Irreducible Polynomials Chosen for GMiMC.** Due to the reasons mentioned in Section 2.6.2, suitable irreducible polynomials are easier to find than suitable Solinas



$r[2] = (r_{191} \mid\mid \ldots \mid\mid r_{128})$
$r[1] = (r_{127} \mid\mid \ldots \mid\mid r_{64})$
$r[0] = (r_{63} \mid\mid \ldots \mid\mid r_0)$
4 bits $\leftarrow$
$\bigoplus$ $r_r = (r_{127} \mid\mid \ldots \mid\mid r_{65})$
3 bits $\leftarrow$
$\bigoplus$ $r_r = (r_{127} \mid\mid \ldots \mid\mid r_{65})$
1 bit $\leftarrow$
$\bigoplus$ $r_r = (r_{127} \mid\mid \ldots \mid\mid r_{65})$
$\bigoplus$ $r_r = (r_{127} \mid\mid \ldots \mid\mid r_{65})$

Figure 5.1: Reduction of a single word in $\mathbb{F}_{2^{65}}$.

prime numbers. Indeed, every irreducible trinomial and pentanomial can be used for the reduction algorithm explained above. Table 5.4 gives a short overview of the polynomials used in this implementation, and also shows the number of XOR and shift operations needed for the reduction, where all operations are done word-wise.

| Field size | Irreducible polynomial | # XORs | # shifts |
|---|---|---|---|
| 3 bits | $f(z) = z^3 + z + 1$ | 2 | 1 |
| 33 bits | $f(z) = z^{33} + z^6 + z^3 + z + 1$ | 6 | 5 |
| 65 bits | $f(z) = z^{65} + z^4 + z^3 + z + 1$ | 11 | 11 |

Table 5.4: Irreducible polynomials used for the implementation of binary field arithmetic in GMiMC.

# 6 Implementation

This chapter will now give an overview of the new implementation of ZKB++, MiMC, and GMiMC.

## 6.1 Implementation Details

During the new implementation, focus was laid on multiple advantageous properties. Some of them are shortly described here.

**Minimizing Loop Iterations and Indexing Operations.** Some of the algorithms described in the previous chapter, in particular the algorithms used for classical multi-precision multiplication and for the Solinas reduction, make use of various *for* loops. These slow down the computation of the corresponding operation significantly due to additional instructions necessary for the loop tests and the loop variable. Therefore, loops were unrolled where possible.

Moreover, all indexing operations for the Solinas reduction algorithm are hard-coded, as the reduction matrices are known beforehand. The number of these operations was reduced by using built-in C++ functions such as *memset(·)* and *memcpy(·)*, and by making use of special properties of the prime numbers chosen for reduction. This means that Algorithm 5.13 is actually implemented differently for every prime number.

**Easily Interchangeable Circuits.** The previous implementations of *Picnic-FS* and *Picnic-UR* are closely related to LowMC. In the new implementation, the initial main focus was laid on clearly separating the signature protocol from any circuits being used within the protocol. Hence, the proof system itself is stored in one file, whereas the circuit and all functionalities regarding the circuit are stored in an entirely different file. This not only makes it easier to change independent sections of the program, but more importantly it allows to replace the used circuit with a different one by simply changing one single function. This has also been made possible by merging both the prover and the verifier circuit into one single circuit, which calls the appropriate functions (for the prover or for the verifier) depending on the current state of the protocol. Moreover, all parts of the protocol which are not directly related to the internal block cipher, e.g. preparing the shares for each Feistel branch, are not part of the circuit itself, which makes the circuit function only contain the block cipher encryption and nothing else.

**Lightweight Implementation.** The first version of the new implementation used NTL (Number Theory Library) [Sho17] for all modular computations within MiMC and GMiMC. Unfortunately, NTL does not support some of the reduction algorithms described in Chapter 5, in particular not the methods used for special prime moduli and irreducible polynomials of special form. Understandably, the library itself is also relatively large and heavily contributes to the final size of the executable file.

For these reasons, a custom math library for calculations with big integer numbers was written, and the main focus was laid on the performance of the reduction algorithms. This divides the new implementation into three main parts: the protocol itself, the circuit implementation, and the custom math library. All of them are clearly separated into different files and classes.

Apart from built-in C++ functions, the only external library used is OpenSSL [Ope18], which is needed for SHA-256, AES-128, and the generation of random bytes. This results in a comparatively small executable file and makes it easier to install ZKB++ on different environments.

**Faster Squaring in ZKB++.** When using ZKB++ with MiMC or GMiMC, all multiplications are due to the permuting function $f(x) = x^3$, which means that half of the multiplications in the protocol are actually squarings. By using the gate definitions from Section 3.2.1, $a_i = b_i$, where $i$ refers to the share of party $i$ and

$a = (a_0 + a_1 + a_2) = (b_0 + b_1 + b_2) = b$. Hence,

$$\begin{aligned} c_i &= (a_i \cdot b_i) + (a_{i'} \cdot b_i) + (a_i \cdot b_{i'}) + R_i(m) - R_{i'}(m) \\ &= (a_i \cdot b_i) + t + t + R_i(m) - R_{i'}(m), \end{aligned}$$

where $i'$ and $R_i(\cdot)$ are defined as in Section 3.2.1, and $t = (a_{i'} \cdot b_i) = (a_i \cdot b_{i'})$. This method reduces the number of multiplications for the squaring from $9$ to $6$, and thus reduces the total number of multiplications necessary for two multiplication gates calculating $f(x) = x^3$ from $18$ to $15$.

## 6.2 NTL vs. Custom Library

Using NTL, there were two main disadvantages. Firstly, NTL does not natively support fast reductions modulo special prime moduli or irreducible polynomials of special form. Secondly, the library uses its own data types to represent integer numbers. This is not necessarily a disadvantage, but in the ZKB$++$ implementation primitive data types like character pointers were needed for the final proof representation. Therefore, conversions between the NTL data types and ordinary buffers were required, which resulted in a significantly slower computation. Moreover, the compiled executable file is rather large when using NTL, which may be an additional drawback in restricted environments.

Due to these reasons, a custom library for handling modular computations had to be written. Most methods described in Chapter 5 were implemented, in particular the Solinas reductions for prime fields and the faster word-wise reductions for binary fields. All algorithms in the new version operate on primitive data types, hence no conversions are necessary.

Table 6.1 shows a comparison of modular additions and modular multiplications using NTL and the custom library in prime fields of various sizes. The given times represent the average execution time of $10^6$ runs, and are purely for the specified operation, thus conversion times are not included.

Modular multiplications in the custom library are especially slow in the $256$-bit prime field. The reason is the prime number chosen for this field. As shown in Section 5.2.7, this number does not have the same advantageous properties of the other prime numbers given in this section, in particular it results in comparatively many

| Field size \ Operation | ModAdd (NTL) | ModAdd (Custom) | ModMul (NTL) | ModMul (Custom) |
|---|---|---|---|---|
| 16 bits | 48 ns | 8 ns | 266 ns | 14 ns |
| 32 bits | 47 ns | 9 ns | 267 ns | 13 ns |
| 64 bits | 48 ns | 9 ns | 268 ns | 19 ns |
| 136 bits | 50 ns | 20 ns | 403 ns | 201 ns |
| 256 bits | 79 ns | 33 ns | 414 ns | 1495 ns |
| 272 bits | 80 ns | 49 ns | 570 ns | 319 ns |

Table 6.1: Runtime comparison of modular additions and modular multiplications for various prime fields, using NTL and the custom library.

modular additions and modular subtractions. Therefore, modular multiplications in the specified 272-bit prime field are faster, even if the field itself is larger and field elements need more words.

Note that in small fields, the custom library is much faster. However, the difference for modular multiplications gets smaller with increasing field sizes. While the reductions keep being faster than the reductions in NTL, the multiplications themselves are slower in the custom library. The reason is that multiplications in the custom library use $\mathcal{O}(n^2)$ operations, whereas NTL uses the Karatsuba algorithm for comparatively small values, which only needs $\mathcal{O}(n^{\log_2(3)})$ operations (in both cases, $n$ denotes the number of digits).

The Karatsuba algorithm is shortly described in Section 7.1. However, it was not implemented during the work for this thesis, because using smaller finite fields in GMiMC proved to result in faster computations and smaller signatures.

Both the runtime of various instantiations of GMiMC and the corresponding proof sizes are given in Table 6.3.

## 6.3 Prime Fields vs. Binary Fields

In contrast to GMiMC over prime fields, the implementation shows that for the tested field sizes, using binary fields leads to slightly shorter execution times. Although most of the operations are the same for both fields, the main differences lie in the modular multiplication and in the random number generation.

**Modular Operations.** For field sizes of up to $64$ bits, the non-modular result of a multiplication in a prime field can be calculated by using the native multiplication instruction. In binary fields, the CLMUL instruction has to be used instead, which is slightly faster than the native multiplication operator, as shown in Table 5.1. The difference is also noticeable when comparing the execution time in a $64$-bit prime field with the execution time in a $65$-bit binary field (a $64$-bit binary field would not result in a permutation for $f(x) = x^3$), even though a word-wise multiplication has to be performed for the $65$-bit field, whereas the multiplication instruction is still usable for $64$ bits.

The reduction itself is also slightly faster in binary fields. This is due to the fact that the reduction algorithm works with XOR and shift operations exclusively, and all operations are word size operations, whereas for prime fields the reduction algorithm may require to use lower quantities (e.g. $8$-bit quantities) for the additions and also uses *if* clauses. This is, however, heavily dependent on the ordering of the indices used for the Solinas reduction algorithm.

Finally, additions and subtractions are faster in binary fields, where they can be executed by simple XOR operations and no reductions are needed. Together with the increased performance of multiplications, this makes computations in binary fields more efficient.

**Random Number Generation.** During the computation of multiplication gates in ZKB$++$, random field elements have to be used to maintain the zero-knowledge property. In binary fields, every $n$-bit polynomial is an element of $\mathbb{F}_{2^n}$, thus every random $n$-bit quantity is guaranteed to be an element of $\mathbb{F}_{2^n}$, which makes the generation of additional random numbers unnecessary.

This is not the case for prime fields, where a random $n$-bit quantity might not be an element of the respective $n$-bit prime field $\mathbb{F}_p$. This event occurs with probability $\frac{2^n-p}{2^n} = 1 - \frac{p}{2^n} \leq \frac{1}{2^s}$, where $s$ denotes the number of leading ones in the binary representation of the corresponding prime number $p$.

Let $S$ be a stream of uniformly distributed bits. We want to make $S$ as short as possible, while at the same time containing enough bits to create all random numbers as elements of the field. Let a *sampling* $s(n)$ denote the process of selecting $n$ bits from $S$ and let the random variable $X$ represent the number of samplings necessary in order to find a random number $x \in \mathbb{F}_p$. Then the expected number of samplings required for one single random number is $\mathbb{E}(X)$, and the expected number of samplings required for all random numbers of one party in one iteration is $m \cdot \mathbb{E}(X)$, where $m$ denotes the number of multiplications in a single encryption using MiMC or GMiMC. Furthermore,

$$\begin{aligned}
\mathbb{E}(X) &= \sum_{k=1}^{\infty} k \cdot \Big(1 - \frac{p}{2^n}\Big)^{k-1} \cdot \Big(\frac{p}{2^n}\Big) \\
&= \sum_{k=1}^{\infty} k \cdot \Big(\frac{2^n - p}{2^n}\Big)^{k-1} \cdot \Big(\frac{p}{2^n}\Big) \\
&= \sum_{k=1}^{\infty} k \cdot \Big(\frac{2^n}{2^n - p}\Big) \cdot \Big(\frac{2^n - p}{2^n}\Big)^{k} \cdot \Big(\frac{p}{2^n}\Big) \\
&= \sum_{k=1}^{\infty} k \cdot \Big(\frac{2^n - p}{2^n}\Big)^{k} \cdot \Big(\frac{2^n p}{2^{2n} - 2^n p}\Big) \\
&= \sum_{k=1}^{\infty} k \cdot \Big(\frac{2^n - p}{2^n}\Big)^{k} \cdot \Big(\frac{p}{2^n - p}\Big),
\end{aligned}$$

which is a geometric series with

$$\sum_{k=1}^{n} a_0 \cdot q^k \cdot k = a_0 \cdot \frac{n \cdot q^{n+2} - (n+1) \cdot q^{n+1} + q}{(q-1)^2},$$

where $a_0 = \Big(\frac{p}{2^n - p}\Big)$ and $q = \Big(\frac{2^n - p}{2^n}\Big) < 1$. Thus,

$$\begin{aligned}
\lim_{n \to \infty} (n \cdot q^{n+2}) &= 0, \\
\lim_{n \to \infty} ((n+1) \cdot q^{n+1}) &= 0,
\end{aligned}$$

and

$$\sum_{k=1}^{\infty} a_0 \cdot q^k \cdot k = a_0 \cdot \frac{q}{(q-1)^2}.$$

This corresponds to the expected number of trials $e$ needed for a success event of a Bernoulli distributed random variable Y, where $\mathbb{P}[Y = 1] = \frac{p}{2^n}$ and $e = \frac{1}{p/2^n} = \frac{2^n}{p}$.

For most of the Solinas prime numbers given in Section 5.2.7 and for suitable Crandall prime numbers, this probability is relatively low, and thus most random numbers do not need to be replaced by new ones. However, in small fields, this probability is significantly larger and given the amount of random numbers needed, many of them have to be replaced, which means that the pseudorandom number generator is used more often.

Another approach would be to select more bits than needed for each random number and to reduce the resulting number afterwards. For example, $s = 32$ bits could be selected additionally to the bits needed for the field. However, this approach results in a comparatively large amount of random bits needed.

The expected number of bits needed for both approaches in each prime field used for MiMC and GMiMC is given in Table 6.2. In either case, though, the check whether a random number is an element of a given prime field always has to be performed.

| n | $p$ | $\mathbb{P}[s(n) \in \mathbb{F}_p] = \frac{p}{2^n}$ | $\mathbb{E}(X)$ | $m$ | $\lceil 438 \cdot 3 \cdot m \cdot n \cdot \mathbb{E}(X) \rceil$ | $438 \cdot 3 \cdot m \cdot (n + 32)$ |
|---|---|---|---|---|---|---|
| 3 | $2^3 - 2 - 1$ | 0.6250 | 1.6 | 694 | 4377197 bits $\approx$ 534 KB | 31917060 bits $\approx$ 3896 KB |
| 4 | $2^4 - 2^2 - 1$ | 0.6875 | $\approx 1.4545$ | 520 | 3975448 bits $\approx$ 485 KB | 24598080 bits $\approx$ 3002 KB |
| 16 | $2^{16} - 2^4 - 1$ | $\approx 0.9961$ | $\approx 1.004$ | 156 | 3280595 bits $\approx$ 400 KB | 9839232 bits $\approx$ 1201 KB |
| 32 | $2^{32} - 2^4 - 1$ | $\approx 1$ | $\approx 1$ | 126 | 5298049 bits $\approx$ 646 KB | 10596096 bits $\approx$ 1293 KB |
| 64 | $2^{64} - 2^8 - 1$ | $\approx 1$ | $\approx 1$ | 170 | 14296321 bits $\approx$ 1745 KB | 21444480 bits $\approx$ 2617 KB |
| 136 | $2^{136} - 2^8 - 1$ | $\approx 1$ | $\approx 1$ | 330 | 58972321 bits $\approx$ 7198 KB | 72848160 bits $\approx$ 8892 KB |
| 256 | $2^{256} - 2^{184} + 2^{32} + 1$ | $\approx 1$ | $\approx 1$ | 324 | 108988417 bits $\approx$ 13304 KB | 122611968 bits $\approx$ 14967 KB |
| 272 | $2^{272} - 2^{40} - 1$ | $\approx 1$ | $\approx 1$ | 344 | 122948353 bits $\approx$ 15008 KB | 137412864 bits $\approx$ 16774 KB |

Table 6.2: Comparison of random sampling methods.

## 6.4 Detailed Timings

All benchmarks listed in this section, just as the benchmarks in Chapter 5, were obtained using an Intel i7-6700 CPU @ 3.40 GHz. The implementation was compiled with GCC 4.8.2, and the optimization option *O2* was used. Moreover, the tested machine has $64$ GB of memory. For time measurements, various methods from the *std::chrono* namespace were used, in particular *high_resolution_clock::now()*.

The new implementation of ZKB++ was tested using MiMC and GMiMC, and by trying different field sizes for both of them. All field sizes (and number of branches in the case of GMiMC) were chosen such that the resulting block size is at least $256$ bits, and preferably as close to it as possible. Solinas prime numbers given in Section 5.2.7 and irreducible polynomials given in Section 5.3.3 were used for prime fields and binary fields, respectively. For small prime fields, where non-modular multiplication results are less than $2^{64}$, both the Barrett reduction and the native modulo operator came very close to the Solinas reduction algorithm, and in some cases there was not even a noticeable difference. However, when non-modular multiplication results cannot be stored in a single $64$-bit word anymore, the Solinas reduction algorithm becomes much faster than all other tested methods.

The execution times of the implemented instantiations of MiMC and GMiMC are given in Table 6.3. Each configuration was run $50$ times, and the average execution time was chosen. The number of iterations in ZKB$++$ was set to $438$. This table lists the times for distinct phases of the protocol:

> *GenSign.* This is the amount of time it takes to generate memory buffers for the view values, random tapes, and key shares. The generation of random tapes and random key shares takes also place in this step.
> *Sign.* This is the total execution time for the signing process, including the time given for *GenSign*, the multi-party computation of the circuit, and the generation of the final proof.
> *GenVerify.* This is very similar to *GenSign*, but smaller buffers have to be created here. For example, memory for only one view is needed. Moreover, the generation of random tapes and random key shares is not necessary during the verification process.
> *Verify.* Similar to *Sign*, this includes both the time given for *GenVerify* and the subsequent circuit computation, together with the evaluation of the final result. Note that execution times are shorter here, because the computation does not have to be performed for all three views.

| Scheme | $(n, t, r)$ | GenSign | Sign | GenVerify | Verify | $\upsilon$ | $\lvert\sigma\rvert$ |
|---|---|---|---|---|---|---|---|
| MiMC | $(256, 1, 162)$ | 1.29 ms | 441.03 ms | 0.32 ms | 168.87 ms | 83456 bits | $\approx$ 4512 KB |
| | $(272, 1, 172)$ | 1.43 ms | 185.14 ms | 0.38 ms | 89.52 ms | 94112 bits | $\approx$ 5084 KB |
| GMiMC$_{erf}$ ($\mathbb{F}_p$) | $(3, 86, 347)$ | 0.99 ms | 157.73 ms | 0.26 ms | 107.07 ms | 2082 bits | $\approx$ 161 KB |
| | $(4, 64, 260)$ | 0.89 ms | 94.54 ms | 0.23 ms | 62.77 ms | 2080 bits | $\approx$ 161 KB |
| | $(16, 16, 78)$ | 0.62 ms | 17.84 ms | 0.13 ms | 10.81 ms | 2496 bits | $\approx$ 183 KB |
| | $(32, 8, 63)$ | 0.59 ms | 12.97 ms | 0.13 ms | 7.65 ms | 4032 bits | $\approx$ 265 KB |
| | $(64, 4, 85)$ | 0.63 ms | 22.06 ms | 0.14 ms | 11.94 ms | 10880 bits | $\approx$ 631 KB |
| | $(136, 2, 165)$ | 0.98 ms | 115.17 ms | 0.29 ms | 55.64 ms | 44880 bits | $\approx$ 2452 KB |
| GMiMC$_{erf}$ ($\mathbb{F}_{2^n}$) | $(3, 86, 347)$ | 1.00 ms | 127.79 ms | 0.25 ms | 92.26 ms | 2082 bits | $\approx$ 161 KB |
| | $(33, 8, 64)$ | 0.60 ms | 12.19 ms | 0.13 ms | 7.51 ms | 4224 bits | $\approx$ 277 KB |
| | $(65, 4, 86)$ | 0.76 ms | 21.93 ms | 0.19 ms | 13.35 ms | 11180 bits | $\approx$ 648 KB |
| GMiMC$_{crf}$ ($\mathbb{F}_p$) | $(3, 86, 516)$ | 1.14 ms | 232.31 ms | 0.30 ms | 167.83 ms | 3096 bits | $\approx$ 216 KB |
| | $(4, 64, 386)$ | 1.03 ms | 141.99 ms | 0.26 ms | 102.23 ms | 3088 bits | $\approx$ 215 KB |
| | $(16, 16, 113)$ | 0.68 ms | 25.04 ms | 0.16 ms | 15.68 ms | 3616 bits | $\approx$ 243 KB |
| | $(32, 8, 85)$ | 0.63 ms | 17.04 ms | 0.14 ms | 10.11 ms | 5440 bits | $\approx$ 341 KB |
| | $(64, 4, 101)$ | 0.66 ms | 26.73 ms | 0.14 ms | 14.59 ms | 12928 bits | $\approx$ 741 KB |
| | $(136, 2, 180)$ | 1.02 ms | 125.25 ms | 0.30 ms | 60.47 ms | 48960 bits | $\approx$ 2670 KB |
| GMiMC$_{crf}$ ($\mathbb{F}_{2^n}$) | $(3, 86, 516)$ | 1.16 ms | 191.72 ms | 0.29 ms | 144.51 ms | 3096 bits | $\approx$ 216 KB |
| | $(33, 8, 86)$ | 0.63 ms | 15.77 ms | 0.14 ms | 9.81 ms | 5676 bits | $\approx$ 354 KB |
| | $(65, 4, 103)$ | 0.82 ms | 26.26 ms | 0.20 ms | 16.23 ms | 13390 bits | $\approx$ 766 KB |

Table 6.3: Execution times and proof sizes of MiMC and GMiMC$_{erf}$ in the ZKB$++$ proof system using the Fiat-Shamir transform, where $\upsilon$ denotes the view size and $|\sigma|$ denotes the proof size. As a comparison, the numbers for the same configurations using GMiMC$_{crf}$ are also included.

To better understand the effects of different field sizes on the final computation time, the following diagrams show the total amount of instructions spent in various functions for different field sizes.

**Random Number Generation Cost for Various Field Sizes.** As further discussed in Section 6.3, random number generation is faster in binary fields, especially in smaller fields. This is shown in Figure 6.1, and the difference is most noticeable in the 3-bit field. The reason is that in a 3-bit prime field, many randomly generated 3-bit values are greater than or equal to the respective prime number and thus need to be replaced by new random values. In larger prime fields, most generated values are already in the field. However, the random number generation is still slower than in binary fields due to the *if* clauses used to check whether a random value is an element of a field.

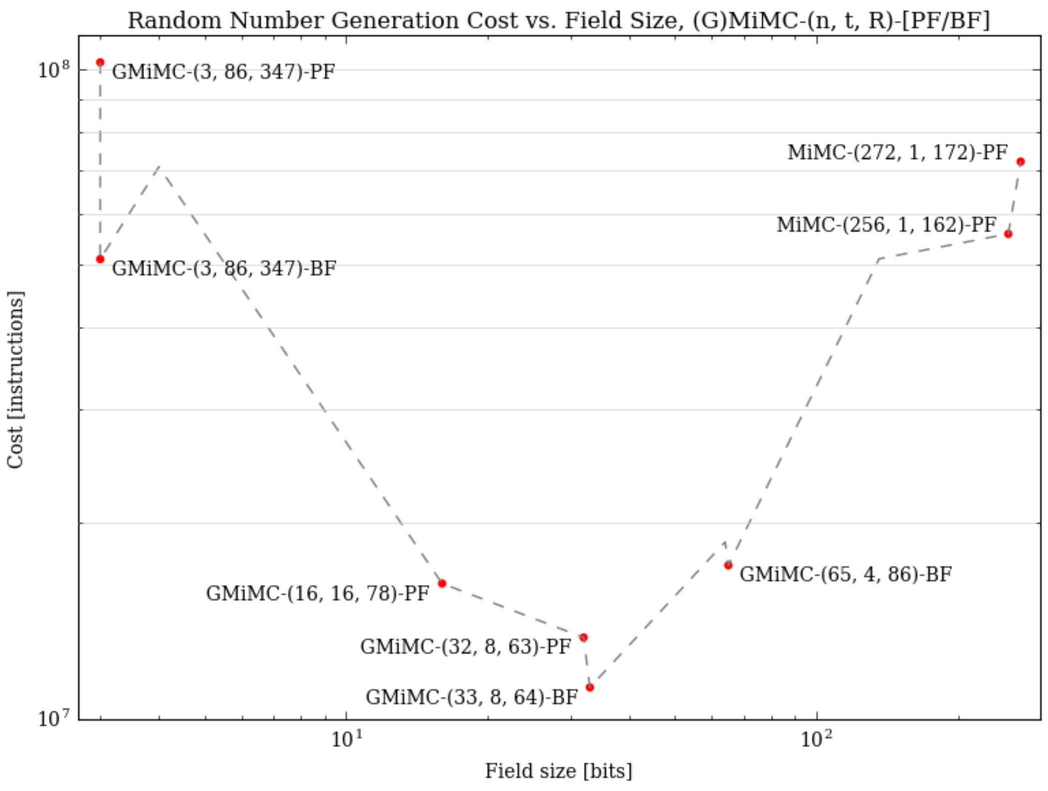


Figure 6.1: Total cost of random number generation for various field sizes.

**Addition Cost for Various Field Sizes.** As discussed in Chapter 5, additions are XOR operations in binary fields and thus cheaper than in prime fields, because no *if* clauses are needed. This is reflected in Figure 6.2, where additions in binary fields are clearly faster than additions in prime fields of similar size. It can also be seen that these operations are costly in small fields in general, because many operations are needed in these fields. The reason is that only a low number of bits can be processed during each CPU instruction, and thus more CPU instructions are needed to process the same amount of bits. This is directly related to the number of branches and rounds of the GMiMC Feistel network, both of which increase with smaller field sizes. However, additions are also costly in rather large fields, mainly because they cannot be replaced by single CPU instructions.

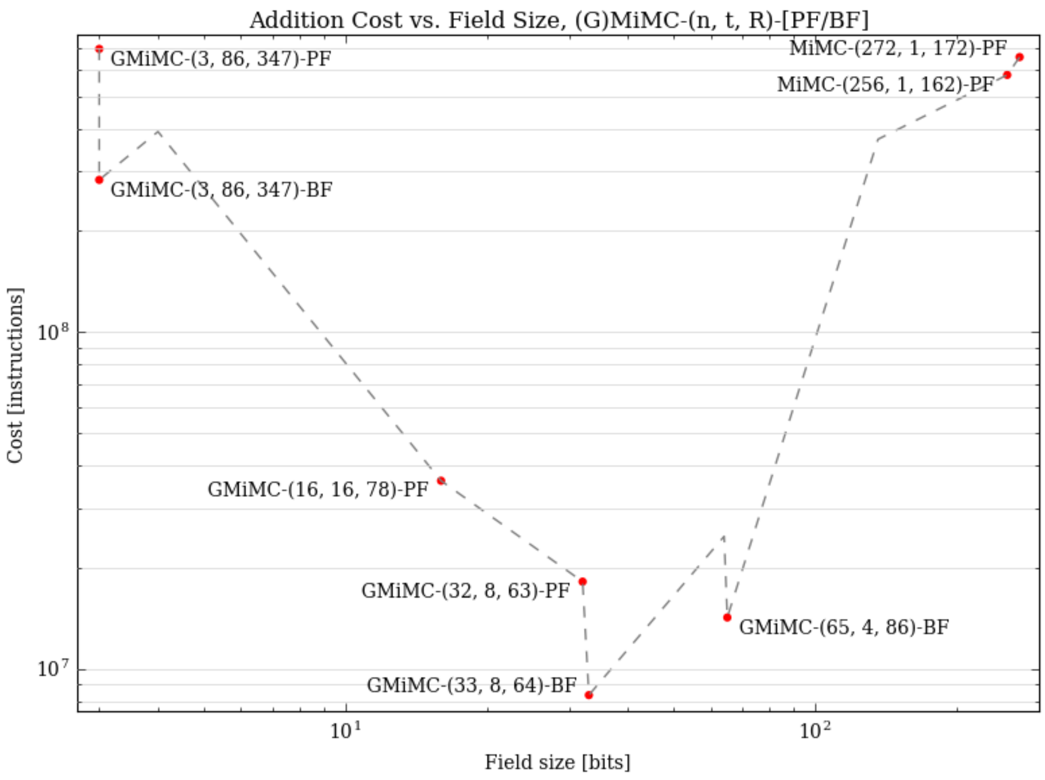


Figure 6.2: Total cost of additions for various field sizes.

**Multiplication Cost for Various Field Sizes.** Note that in each round, the round function $f(x) = x^3$ is executed exactly once, regardless of the field size. Thus, the total cost for multiplications is mainly dependent on the number of rounds used within the cipher. This means that, in contrast to the cost for additions, the cost for multiplications does not change too much for fields up to a certain size. However, the cost starts to become substantial with larger fields, where the non-modular multiplication cannot be replaced by a single CPU instruction anymore. This behavior is reflected in Figure 6.3.

Finally, Figure 6.4 shows the different configurations of MiMC and GMiMC$_{erf}$, and the resulting timings and proof sizes when using them in the context of *Picnic-FS*. Small proof sizes can be achieved by using smaller fields, where a field size of $3$ bits is the optimum if the smallest proof size is desired. However, this also means

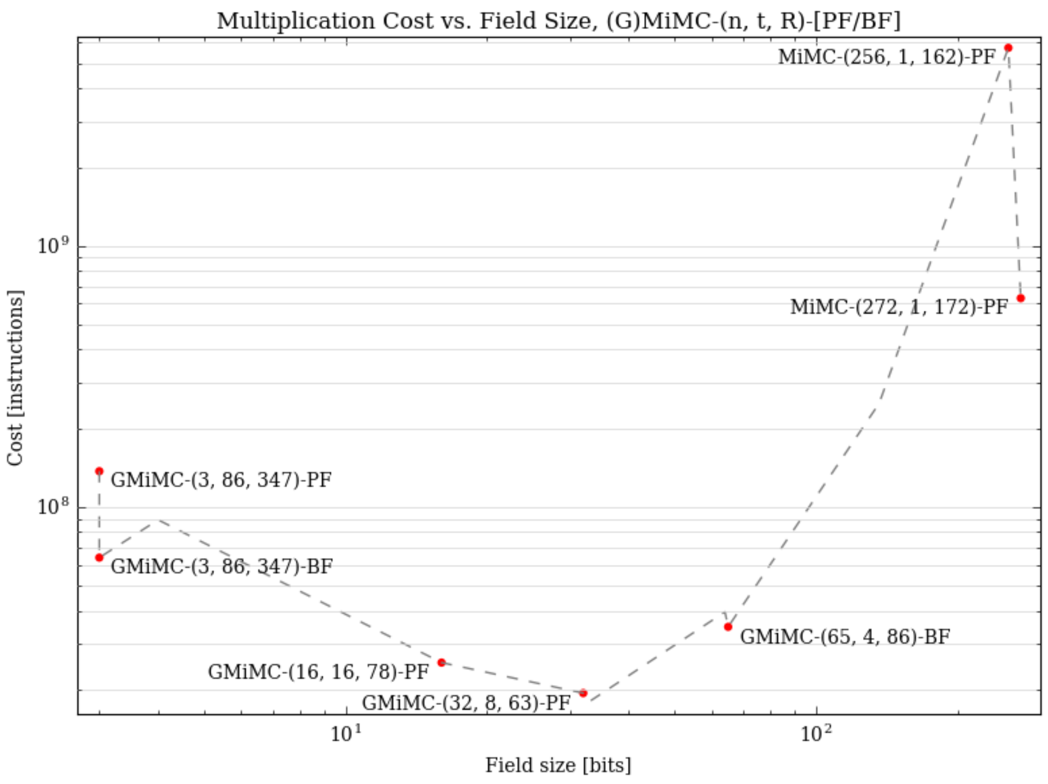


Figure 6.3: Total cost of multiplications for various field sizes.

that only a small number of bits can be processed during each computation, which increases the runtime of the procedure.

Using larger fields, it can be seen that the respective view sizes (and with them also the proof sizes) begin to increase rather quickly. This means that the buffers for the views need to be larger, thus requiring a noticeably longer generation time. Furthermore, for both very small and very large fields, the high number of block cipher rounds results in a higher computational cost, partly due to the larger amount of random values needed for the multiplications.

Comparing MiMC with GMiMC makes clear that GMiMC is the better choice for both a much faster computation and a significantly smaller proof size. It can also be seen that, unsurprisingly, GMiMC$_{erf}$ is a better choice than GMiMC$_{crf}$, which is slightly slower due to requiring more block cipher rounds.

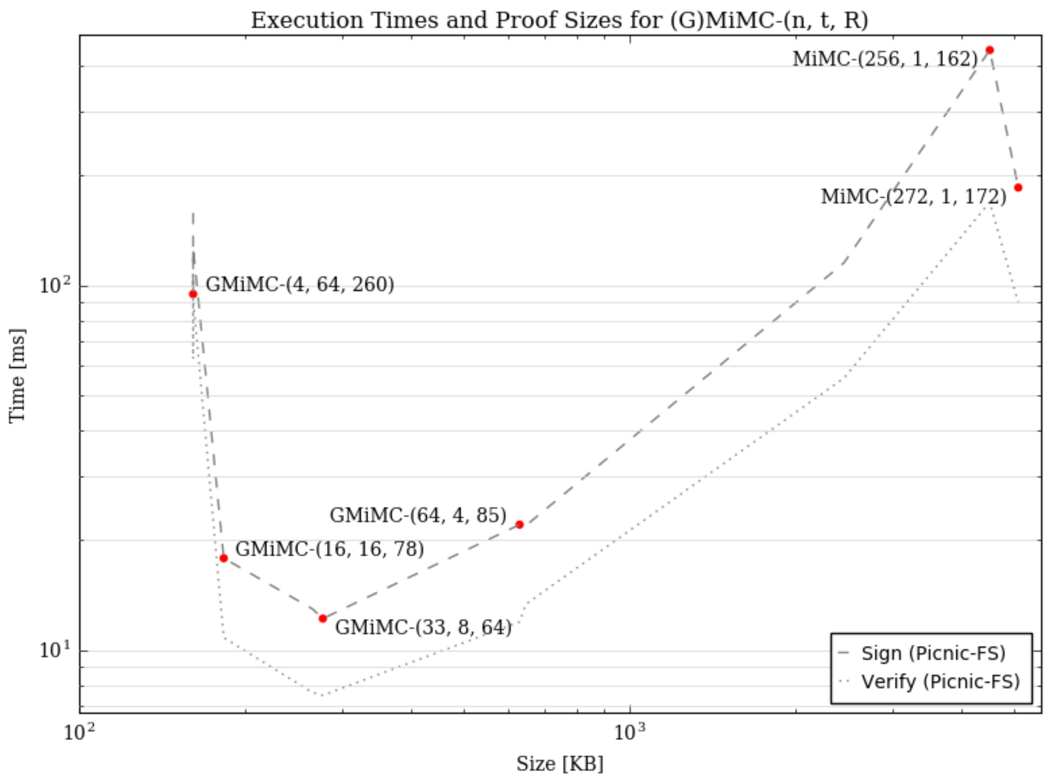


Figure 6.4: Execution times and proof sizes for configurations from Table 6.3. Numbers for GMiMC$_{crf}$ are not included in this plot.

Good trade-offs according to these measurements seem to be GMiMC$_{erf}$ using a 16-bit prime field and GMiMC$_{erf}$ using a 33-bit binary field, where both the execution time and the final proof size can be kept relatively low.

Using similarly sized binary fields tends to be faster than using prime fields of order $< 2^{64}$, as can be seen in the benchmark. The main reasons are that in binary fields no *if* clauses have to be used for modular additions and multiplications, and the cost for generating random values is also slightly smaller, as further explained in Section 6.4.

# 7 Future Work

This chapter discusses some topics, which may allow to increase the performance of the current implementation in specific scenarios.

## 7.1 Karatsuba Multiplication for Large Fields

The current implementation uses the same word-wise multiplication algorithm for every field size. This algorithm needs $\mathcal{O}(N^2)$ operations, where $N$ denotes the number of words needed for a field element. As seen in Section 6.2, using a different non-modular multiplication method, such as the Karatsuba algorithm published in 1962 [KO62], is advantageous for larger fields.

The Karatsuba algorithm is a divide and conquer approach which reduces the number of operations to $\mathcal{O}(N^{\log_2(3)})$. Let $a = (a_1 \cdot 2^l) + a_0$ and $b = (b_1 \cdot 2^l) + b_0$ be two $2l$-bit integer numbers. Then

$$\begin{aligned} a \cdot b &= (a_1 2^l + a_0) \cdot (b_1 2^l + b_0) \\ &= a_1 \cdot b_1 \cdot 2^{2l} + ((a_0 + a_1) \cdot (b_0 + b_1) - (a_1 \cdot b_1) - (a_0 \cdot b_0)) \cdot 2^l + (a_0 \cdot b_0). \end{aligned}$$

This method results in 3 multiplications of $l$-bit integers, 2 additions, and 2 subtractions. In larger fields, the cost of the additions and subtractions is outweighed by the cost of the multiplications. The Karatsuba algorithm is applied until the operands can be stored in CPU words, followed by native methods to finalize the multiplication.

## 7.2 Montgomery Multiplication without Conversions

As further explained in Section 5.2.4 and shown in Table 5.2, the Montgomery method is not suitable in the classic setting, that is, when for every multiplication both operands need to be transformed into the Montgomery domain and then back to the previous domain after multiplying both numbers. With this method, the cost for the large amount of conversions is too high.

However, it may be possible to perform all computations in the Montgomery domain without using any conversions. For example, the private key $x$ and the key of the underlying block cipher encryption in ZKB$++$ can be understood as values already in the Montgomery domain. This is possible, because both a value $a$ and its Montgomery representation $a'$ can be represented by the same number of bits. Of course, this means that the resulting public key $f(x) = y$ is also a value in the Montgomery domain.

The same method can be applied to the random values sampled during the ZKB$++$ protocol, where the conversion to the Montgomery domain can be omitted and each value can be used directly instead.

## 7.3 Computation of Multiple Field Values

The current implementation computes the field values for each addition and multiplication consecutively. This is also part of the reason for which ZKB$++$ in smaller fields is generally slower, because less bits can be processed in each step.

There may be a possibility of computing multiple field values at once in smaller fields. In particular, this method could be applied to binary fields, where additions and subtractions are XOR operations and do not need carry or borrow values. Especially in the beginning of a circuit based on a Feistel network, where the last share for each branch value needs to be calculated, this can be done all at once with a single XOR operation instead of handling each branch separately.

## 7.4 Lazy Reductions for Small Fields

Note that reductions are applied after every operation, regardless of the number of bits of the result. For example, in a $t$-branch Feistel network using GMiMC$_{crf}$, $t-2$ consecutive additions are needed for the branch sum in the round function. If $t = 16$ and $n = 256$, where $n$ denotes the block size, the field size is $16$ bits. This means that $t-2 = 14$ additions are applied, which add at most $14$ bits to the $16$-bit value of the initial branch when not using any reductions. Note that the Solinas reduction algorithm works for every positive value smaller than $p^2$, where $p$ is the $16$-bit prime number for this field. By adding $14$ bits, the result is a $30$-bit number, and the reduction algorithm can obviously be applied.

By using this method, the $14$ *if* clauses and at most $14$ subtractions for the calculation of the branch sum in GMiMC$_{crf}$ can be avoided, and they are replaced by a single Solinas reduction. This optimization is only advantageous when using prime fields, because in binary fields the additions are simple XOR operations and already comparatively cheap.

The situation is different in GMiMC$_{erf}$, where there is not a comparatively large amount of consecutive additions in a single Feistel branch. The round function needs two additions, but two distinct *if* clauses and their subtractions are probably faster than a single Solinas reduction. However, it may be possible to let the number of bits increase during multiple rounds. For example, in GMiMC$_{erf}$ with $16$ branches, the last branch $X_{15}$ becomes the branch $X_{14}$ in the next round after one addition, and then the branch $X_{13}$ after another addition. Hence, the values for most of the branches only need to be reduced after $\approx 16$ rounds, which again helps to avoid a large number of *if* clauses.

# 8 Conclusion

With fully functional quantum computers, most of the cryptographic signature protocols used in practice today would be severely broken. Although building such devices might still take some time, the search for post-quantum secure solutions has already begun.

One example of such a solution, ZKB++ and the resulting signature scheme, has been explained in detail in this thesis. Focus was laid on the protocol definition, on the security of the protocol, and on the choice of the circuit over which the protocol is executed. The latter resulted in an analysis comparing various block ciphers, most notably the GMiMC family of block ciphers. Unfortunately, the proof sizes could not be made smaller and LowMC is still the recommended choice for the protocol. However, various faster reduction methods, such as the Solinas prime reduction, and other performance improvements were implemented in order to optimize GMiMC itself significantly. These improvements are shown in the second main part of the thesis, and detailed comparisons describing the performance differences between various methods are given.

In the final part, new methods to improve the current implementation are suggested. Of all these methods, the *Lazy Reduction* approach would probably result in the most noticeable performance improvement and thus represents an attractive addition for future versions of the block cipher implementation.

In conclusion, we can say that finding new post-quantum secure cryptographic protocols and optimizing them is of major interest, mainly because they need to be ready and sufficiently studied when quantum computers finally become available.

# Bibliography

[ABB+15] Erdem Alkim, Nina Bindel, Johannes A. Buchmann, et al. "TESLA: Tightly-Secure Efficient Signatures from Standard Lattices." In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 755 (cit. on p. 7).

[AGP+18] Martin R. Albrecht, Lorenzo Grassi, Leo Perrin, et al. "On Feistel Structures with Low-Degree Round Functions." submitted to Crypto 2018. Feb. 2018 (cit. on pp. iv, v, 15, 35).

[AGR+16] Martin R. Albrecht, Lorenzo Grassi, Christian Rechberger, et al. "MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity." In: *ASIACRYPT (1)*. Vol. 10031. Lecture Notes in Computer Science. 2016, pp. 191–219 (cit. on pp. iv, v, 2, 15, 30–32).

[Ajt96] Miklós Ajtai. "Generating Hard Instances of Lattice Problems (Extended Abstract)." In: *STOC*. ACM, 1996, pp. 99–108 (cit. on p. 7).

[AM05] Omran Ahmadi and Alfred Menezes. "On the Number of Trace-One Elements in Polynomial Bases for $F_{2^n}$." In: *Des. Codes Cryptography* 37.3 (2005), pp. 493–507 (cit. on p. 63).

[ARS+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, et al. "Ciphers for MPC and FHE." In: *EUROCRYPT (1)*. Vol. 9056. Lecture Notes in Computer Science. Springer, 2015, pp. 430–454 (cit. on pp. 2, 15, 28, 29).

[Bar86] Paul Barrett. "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor." In: *CRYPTO*. Vol. 263. Lecture Notes in Computer Science. Springer, 1986, pp. 311–323 (cit. on pp. 44, 45).

[BBD08] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. *Post-Quantum Cryptography*. 1st. Springer Publishing Company, Incorporated, 2008. ISBN: 9783540887010 (cit. on pp. 7, 42).

[BCG+12] Julia Borghoff, Anne Canteaut, Tim Güneysu, et al. "PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract." In: *ASIACRYPT*. Vol. 7658. Lecture Notes in Computer Science. Springer, 2012, pp. 208–225 (cit. on p. 15).

[BDE+17] Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, et al. "SPHINCS+ - Submission to the NIST post-quantum project." submitted to the NIST post-quantum project. Nov. 2017 (cit. on p. 6).

[BHH+15] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, et al. "SPHINCS: Practical Stateless Hash-Based Signatures." In: *EUROCRYPT (1)*. Vol. 9056. Lecture Notes in Computer Science. Springer, 2015, pp. 368–397 (cit. on p. 6).

[BHL+17] Daniel J. Bernstein, Nadia Heninger, Paul Lou, et al. "Post-quantum RSA." In: *PQCrypto*. Vol. 10346. Lecture Notes in Computer Science. Springer, 2017, pp. 311–329 (cit. on p. 2).

[BKL+07] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, et al. "PRESENT: An Ultra-Lightweight Block Cipher." In: *CHES*. Vol. 4727. Lecture Notes in Computer Science. Springer, 2007, pp. 450–466 (cit. on p. 15).

[BR11] Elaine B. Barker and Allen L. Roginsky. *SP 800-131A. Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths*. Tech. rep. Gaithersburg, MD, United States, 2011 (cit. on p. 16).

[CDG+17] Melissa Chase, David Derler, Steven Goldfeder, et al. "Post-Quantum Zero-Knowledge and Signatures from Symmetric-Key Primitives." In: *CCS*. ACM, 2017, pp. 1825–1842 (cit. on pp. iv, v, 14, 18, 26, 27).

[Cra92] Richard E. Crandall. *Method and apparatus for public key exchange in a cryptographic system*. US Patent 5,159,632. 1992 (cit. on p. 48).

[Dam10] Ivan Damgård. *Lecture Notes in Cryptologic Protocol Theory*. Mar. 2010 (cit. on p. 13).

[DE79] Whitfield Diffie and Martin E. Hellman. "Privacy and authentication: An introduction to cryptography." In: 67 (Apr. 1979), pp. 397–427 (cit. on p. 31).

[DEM15] Christoph Dobraunig, Maria Eichlseder, and Florian Mendel. "Higher-Order Cryptanalysis of LowMC." In: *ICISC*. Vol. 9558. Lecture Notes in Computer Science. Springer, 2015, pp. 87–101 (cit. on p. 30).

[Div14] NIST Computer Security Division. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. FIPS Publication 202. National Institute of Standards and Technology, U.S. Department of Commerce, 2014. URL: `http://csrc.nist.gov/publications/drafts/fips-202/fips_202_draft.pdf` (cit. on p. 6).

[DRS18] David Derler, Sebastian Ramacher, and Daniel Slamanig. "Post-Quantum Zero-Knowledge Proofs for Accumulators with Applications to Ring Signatures from Symmetric-Key Primitives." In: *PQCrypto*. Vol. 10786. Lecture Notes in Computer Science. Springer, 2018, pp. 419–440 (cit. on p. 2).

[DSS05] C. Dods, Nigel P. Smart, and Martijn Stam. "Hash Based Digital Signature Schemes." In: *IMA Int. Conf.* Vol. 3796. Lecture Notes in Computer Science. Springer, 2005, pp. 96–115 (cit. on p. 5).

[Fog17] Agner Fog. *Instruction tables - Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. 2017. URL: `http://www.agner.org/optimize/instruction_tables.pdf` (visited on 03/01/2018) (cit. on p. 43).

[FS86] Amos Fiat and Adi Shamir. "How to Prove Yourself: Practical Solutions to Identification and Signature Problems." In: *CRYPTO*. Vol. 263. Lecture Notes in Computer Science. Springer, 1986, pp. 186–194 (cit. on pp. 13, 26).

[GK90] Oded Goldreich and Hugo Krawczyk. "On the Composition of Zero-Knowledge Proof Systems." In: *ICALP*. Vol. 443. Lecture Notes in Computer Science. Springer, 1990, pp. 268–282 (cit. on p. 13).

[GMO16] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. "ZKBoo: Faster Zero-Knowledge for Boolean Circuits." In: *USENIX Security Symposium*. USENIX Association, 2016, pp. 1069–1083 (cit. on pp. 18, 25, 27).

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority." In: *STOC*. ACM, 1987, pp. 218–229 (cit. on p. 14).

[Gro96] Lov K. Grover. "A Fast Quantum Mechanical Algorithm for Database Search." In: *STOC*. ACM, 1996, pp. 212–219 (cit. on pp. 9, 26).

[Ham15] Mike Hamburg. “Ed448-Goldilocks, a new elliptic curve.” In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 625 (cit. on p. 54).

[Ham17] Mike Hamburg. “Post-quantum cryptography proposal: ThreeBears.” submitted to the NIST post-quantum project. Dec. 2017 (cit. on p. 54).

[HMV03] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003. ISBN: 038795273X (cit. on pp. 45, 47, 62).

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. “NTRU: A Ring-Based Public Key Cryptosystem.” In: *ANTS*. Vol. 1423. Lecture Notes in Computer Science. Springer, 1998, pp. 267–288 (cit. on p. 7).

[Hül13] Andreas Hülsing. “W-OTS+ - Shorter Signatures for Hash-Based Signature Schemes.” In: *AFRICACRYPT*. Vol. 7918. Lecture Notes in Computer Science. Springer, 2013, pp. 173–188 (cit. on p. 5).

[IKO+07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, et al. “Zero-knowledge from secure multiparty computation.” In: *STOC*. ACM, 2007, pp. 21–30 (cit. on pp. 14, 19).

[JK97] Thomas Jakobsen and Lars R. Knudsen. “The Interpolation Attack on Block Ciphers.” In: *FSE*. Vol. 1267. Lecture Notes in Computer Science. Springer, 1997, pp. 28–40 (cit. on p. 32).

[Kat10] Jonathan Katz. *Digital Signatures*. Springer, 2010 (cit. on p. 10).

[KLM+16] Stefan Kölbl, Martin M. Lauridsen, Florian Mendel, et al. “Haraka v2 - Efficient Short-Input Hashing for Post-Quantum Applications.” In: *IACR Trans. Symmetric Cryptol.* 2016.2 (2016), pp. 1–29 (cit. on p. 6).

[KO62] Anatoly Karatsuba and Yuri Ofman. “Multiplication of Many-Digital Numbers by Automatic Computers.” In: *Doklady Akad. Nauk SSSR* 145 (1962). Translation in Physics-Doklady 7, 595–596, 1963, pp. 293–294 (cit. on pp. 41, 78).

[KR13] Cameron F. Kerry and Charles Romine. *FIPS PUB 186-4 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION Digital Signature Standard (DSS)*. 2013 (cit. on p. 54).

[Lam79] Leslie Lamport. *Constructing Digital Signatures from a One Way Function*. Tech. rep. Oct. 1979. URL: https://www.microsoft.com/en-us/research/publication/constructing-digital-signatures-one-way-function/ (cit. on p. 4).

[McE78] R. J. McEliece. "A Public-Key Cryptosystem Based on Algebraic Coding Theory." In: 44 (May 1978) (cit. on p. 8).

[Mer80] Ralph C. Merkle. "Protocols for Public Key Cryptosystems." In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1980, pp. 122–134 (cit. on p. 5).

[Mon85] Peter L. Montgomery. "Modular Multiplication without Trial Division." In: *Mathematics of Computation* 44 (1985) (cit. on p. 45).

[MOV96] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996 (cit. on p. 48).

[Nat17] National Institute of Standards and Technology. *Post-Quantum Cryptography*. 2017. URL: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography (visited on 02/27/2018) (cit. on p. 2).

[Nie86] H Niederreiter. "Knapsack Type Cryptosystems and Algebraic Coding Theory." In: 15 (Jan. 1986) (cit. on p. 8).

[Nyb96] Kaisa Nyberg. "Generalized Feistel Networks." In: *ASIACRYPT*. Vol. 1163. Lecture Notes in Computer Science. Springer, 1996, pp. 91–104 (cit. on p. 35).

[Ope18] OpenSSL Development Team. *OpenSSL - Cryptography and SSL/TLS Toolkit*. 2018. URL: https://www.openssl.org/ (visited on 03/11/2018) (cit. on p. 66).

[Sha49] C. E. Shannon. "Communication theory of secrecy systems." In: *The Bell System Technical Journal* 28.4 (Oct. 1949), pp. 656–715. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1949.tb00928.x (cit. on p. 8).

[Sho17] Victor Shoup. *NTL: A Library for doing Number Theory*. 2017. URL: http://www.shoup.net/ntl/ (visited on 03/11/2018) (cit. on p. 66).

[Sho97] Peter W. Shor. “Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer.” In: *SIAM J. Comput.* 26.5 (1997), pp. 1484–1509 (cit. on p. 1).

[Sol99] Jerome A. Solinas. *Generalized Mersenne Numbers*. Tech. rep. National Security Agency, 1999 (cit. on p. 48).

[Unr15] Dominique Unruh. “Non-Interactive Zero-Knowledge Proofs in the Quantum Random Oracle Model.” In: *EUROCRYPT (2)*. Vol. 9057. Lecture Notes in Computer Science. Springer, 2015, pp. 755–784 (cit. on pp. 14, 26).

[Unr17] Dominique Unruh. “Post-quantum Security of Fiat-Shamir.” In: *ASIACRYPT (1)*. Vol. 10624. Lecture Notes in Computer Science. Springer, 2017, pp. 65–95 (cit. on p. 13).

[Yao82] Andrew Chi-Chih Yao. “Protocols for Secure Computations (Extended Abstract).” In: *FOCS*. IEEE Computer Society, 1982, pp. 160–164 (cit. on p. 14).

[Yao86] Andrew Chi-Chih Yao. “How to Generate and Exchange Secrets (Extended Abstract).” In: *FOCS*. IEEE Computer Society, 1986, pp. 162–167 (cit. on p. 14).

[YI00] Akihiro Yamamura and Hirokazu Ishizuka. “Quantum cryptanalysis of block ciphers.” In: (Jan. 2000) (cit. on p. 9).