MASTER IN COMPUTER SCIENCE

Implementation of a Threshold Post-Quantum Signature Scheme

Master Thesis

Dominique Roux

Faculty of Science
at the University of Bern

August 2022

Prof. Dr. Christian Cachin
Ignacio Amores Sesar

Cryptology and Data Security Research Group
Institute of Computer Science
University of Bern, Switzerland

UNIVERSITÄT BERN

UNIVERSITÉ DE NEUCHÂTEL

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Abstract

To answer the theoretical threat of quantum computers against current cryptographic systems, the National Institute of Standards and Technology (NIST) ran a post-quantum competition from January 2017 to July 2022. One of the candidates to be standardised as a post-quantum signature scheme is called *Dilithium*, representing this thesis' first cornerstone.

The second cornerstone is formed by *secure multi-party computation (MPC)*. One application of MPC is called threshold signatures. Threshold signatures are generated by multiple parties collaborating. They ensure privacy, availability, and correctness even in the presence of up to a threshold of corrupted parties.

This thesis results in a simplified version of Dilithium's threshold implementation with decreased security parameters. As a bridge between the high-level interface and the multi-party primitives, MP-SPDZ was used. The developed algorithm generates a valid signature of a one-byte message in the honest majority with passive adversaries security model and a 3-out-of-3 threshold setting in around 87 seconds. With a protocol ensuring privacy and completeness against a corrupted majority and active adversaries, the signing takes around 41 minutes in a 2-out-of-2 threshold setting.

Acknowledgments

A big thank you goes to Ignacio Amores Sesar for all the help, critical thinking, interesting discussions, and for always lifting my mood when needed. I would also like to thank Prof. Dr. Christian Cachin for all his support and for giving me the chance to work on this project. Another big thank you goes to the whole Cryptology and Data Security Research Group in Bern for welcoming me from the first second on. Furthermore, I would like to thank my girlfriend and my family for all the support during the master thesis and for reminding me to take a break once in a while. Although he might never see this, I would like to thank Marcel Keller a lot for his time helping me debug and providing fast bug fixes for MP-SPDZ.

Contents

1 Introduction

Quantum Computing. Classical computers are based on the theoretical foundation of Turing [32] and his introduction to Turing Machines. Furthermore, the used hardware is based on classical physics. This means every state is assumed to be fixed and is constantly evaluated. For example, if a value is written to the RAM, this state is fixed and, upon read-out, should result in the same value. Quantum mechanics, however, proclaimed that the evaluation influences the state. Schrödinger's [27] famous gedankenexperiment '*Schrödinger's Cat*' gives some intuition about this subject.

The science of physics is based on creating models of nature. The birth of computers led to a significant improvement in such physical modelling. Nowadays, models range from simple two-body experiments up to complex climate models. In 1980, Benioff [1] proposed a quantum mechanical model of the Turing Machine. The concrete idea of the quantum computer was born. Feynman furthermore indicated there might be difficulties in simulating quantum mechanics on classical computers. The idea was quite simple: why not use a quantum computer for modelling quantum mechanics? Another motivational question to develop a working quantum computer was asked by Deutsch [6]: can a quantum computer solve computational problems which do not have an efficient solution on a classical computer? Shor [29] demonstrated that two fundamental problems for cryptography could be solved efficiently on a quantum computer a decade later. The two problems, finding the prime factors of an integer, and the famous 'discrete logarithm' problem, are significant for current cryptographic systems. Therefore, it is theoretically known that today's cryptosystems would be broken if a quantum computer was large and stable enough. Although large companies, such as IBM[1] and Google[2], are funding huge quantum computing research labs, luckily for the world, current realisations of quantum computers are not able to break cryptography. The goal is clear, developing a stable quantum computer on a large scale. Of course, their main objective is not to build a quantum computer to break the current cryptosystem; since the beginning of quantum computers, more and more benefits of having a quantum computer crystallised. For example, quantum cryptography [2] and quantum machine learning [3]. However, as soon as quantum computers exist at the required scale, the current cryptosystems are not only theoretically but practically broken.

Due to the threat quantum computers pose against modern cryptosystems, the National Institute of Standards and Technology (NIST) started a post-quantum cryptography competition in 2017. The goal

[1] `https://www.ibm.com/quantum/roadmap`
[2] `https://quantumai.google/`

is to create new cryptographic standards presumed to be quantum-resistant. Candidates in the following sections were searched:

- Public-key Encryption and Key-establishment Algorithms,
- Digital Signature Algorithms.

At the beginning of this master thesis, March 2022, the competition's third round was still going on. One of the finalists was CRYSTALS Dilithium [7]. Dilithium is a lattice-based signature scheme, meaning the security assumptions are based on problems from lattices. On July 5, 2022, NIST announced the candidates to be standardised; Dilithium is one of them. In conclusion, in the five-year analysis of the signature scheme, no vulnerability was found that broke Dilithium's security in a way that poses a threat to the signature scheme. Of course, this does not mean that Dilithium is quantum-resistant; it just makes it more probable. Over the following years, Dilithium will be developed further until finally standardised.

Secure Multi-Party Computation. Nowadays, more and more user data is being processed. Concerns arise that big tech companies gather as much data about individuals as possible to increase their profits. The discussion focuses on the balance between enhanced usability and a threat to the privacy of applications and their features. For example, it might be beneficial for a user if she can see if one of her contacts is using the same application. On the other hand, the same user might not want to share all her contacts with the company to enable them to compare it with their active user list. One of the biggest questions which arise when discussing this corporative issue is thus: '*Whom do you trust?*'.

Yao [34] introduced a similar problem in his paper in 1982: '*Two millionaires wish to know who is richer; however, they do not want to find out inadvertently any additional information about each other's wealth. How can they carry out such a conversation?*'. One naive solution would be using a (trusted) third party. Both millionaires would tell the third party their respective wealth, and the third party would announce who of the two is richer. The big question arises: do both millionaires trust this third party? Yao introduced this problem as a motivation to found the field of research called secure multi-party computation (MPC). The main goal is that participants could compute a pre-defined and publicly known function without gaining more knowledge than the result of the function and their secret.

Since then, MPC was further developed, and another application arose: threshold signatures. In contrast to traditional signatures, not only one person but a group of persons are needed to compute a valid signature. Suppose there is a committee of 10 people, and in order for them to accept a proposal, six persons need to accept it. Eventually, the committee will sign an accepted proposal. Therefore, everyone could verify the signature of the proposal and thus ensuring its validity. This is an example of an application of threshold signatures. In this example, it is not essential which persons give their approval but that they are at least six. This is called a 6-out-of-10 threshold system. A more extensive application is if we take into account the responsibilities of the people in the committee. Suppose the committee consists still of 10 persons, but now in difference, there is a head of the committee. The rules are that a proposal only gets approved if any 7 or 4 members plus the head of the committee accept. One could think of implementing any of those two problems without threshold signatures. There, each member would individually sign the proposal. However, it directly follows that the corresponding logic needs to be implemented elsewhere. For example, another document or meta information would be needed to declare what a valid signed proposal must look like. Contrarily, threshold signatures include this logic directly inside the signing procedure. The verification algorithm stays the same while the signing procedure differs from the traditional signature scheme. In conclusion, if the committee approves and thus, computes a valid signature for the proposal, the signature can be verified as if a single person generated it. There are several threshold implementations of known signature schemes, such as RSA [30] and Schnorr [16].

This thesis combines both research topics: post-quantum signatures and secure multi-party computation. More precisely, the goal is to have a threshold implementation of Dilithium. Since there is no threshold

implementation of Dilithium available at the moment, the feasibility of this thesis is based on the theoretical analysis of Cozzo and Smart [4]. They discuss the required methods for the implementation and estimate a running time of 12 seconds to run a multi-party computation version of Dilithium. Furthermore, the security of the threshold model and the exact threshold setting is not fixed for the resulting implementation of this thesis.

Structure of the Thesis. The first two chapters concern the theoretical fundament of this thesis. Chapter 2 focuses on the details of lattice-based cryptography, especially on its presumedly hard problems. In the same chapter, Dilithium's construction is discussed. The algorithm is explained first in an overview and later in more detail. In Chapter 3, some fundamental theories on secure multi-party computation, including secret sharing schemes and garbled circuits, are given. Furthermore, the used framework in this thesis is presented. The implementation and the faced problems are discussed in Chapter 4. Chapter 5 provides the evaluation of the implementation. Finally, the conclusion and discussion about future work can be found in Chapter 6.

2 Lattice-Based Cryptography and Dilithium

This chapter focuses on the theoretical groundwork of Dilithium [7] and its functionalities. Dilithium is a lattice-based signature scheme developed by CRYSTALS[1].

In Section 2.1 the mathematical foundations of lattices and their corresponding hard problems, such as *Short Integer Solution (SIS)* and *Learning with Errors (LWE)* are discussed. Additionally, in Section 2.3 the approach called *Fiat-Shamir with Aborts* is discussed, on which Dilithium is based. Both sections form the foundation on which Dilithium bases its security and efficiency. The template algorithm of Dilithium is further explained in Section 2.4. The chapter will conclude in Section 2.5 by discussing the changes, mainly concerning efficiency, for the actual submission of Dilithium in the NIST competition.

Notation. Throughout this thesis, vectors are denoted as bold lower-case letters: $\mathbf{v}$. Matrices are denoted as upper-case letters: A. One-dimensional elements, such as integers or polynomials, are denoted as lower-case letters: c.

2.1 Lattices

The following section was taken from [31] and [7].

Definition 1. *A m-dimensional lattice L is a subgroup of $\mathbb{R}^n$ generated by $\mathbf{b_i} \in \mathbb{R}^n$:*

$$L = \left\{ \sum_{i=1}^{m} a_i \mathbf{b_i} : a_i \in \mathbb{Z} \right\} = \{B * \mathbf{a} : \mathbf{a} \in \mathbb{Z}^m\},$$

where $\mathbf{b_i}$ are independent vectors, called basis vectors. The matrix $B \in \mathbb{R}^{n \times m}$ is called the basis matrix and consists of the basis vectors.

Recall the definition of a m-dimensional vector subspace V:

[1] https://pq-crystals.org/

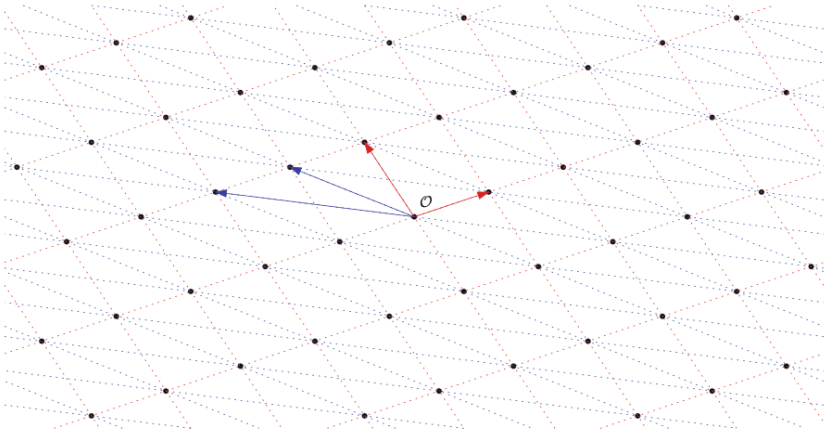

Figure 2.1: Example of a 2-dimensional lattice with two different bases. The red and the blue vectors form each a basis of the same lattice [31].

Definition 2. *A m-dimensional vector subspace V of $\mathbb{R}^n$ is generated by $\mathbf{b_i} \in \mathbb{R}^n$ as*

$$V = \left\{ \sum_{i=1}^{m} a_i \mathbf{b_i} : a_i \in \mathbb{R} \right\},$$

where $\mathbf{b_i}$ are independent vectors, called basis vectors.

A lattice L is very similar to a vector subspace V, but instead of taking any real linear combination of the basis vectors $\{\mathbf{b_1}, ..., \mathbf{b_m}\}$, only linear combinations with integers are allowed. Namely, $a_i \in \mathbb{Z}$ for a lattice and $a_i \in \mathbb{R}$ for a vector subspace. Like a vector subspace, a lattice does not have a distinct set of basis vectors. An example of a 2-dimensional lattice with two sets of basis vectors is shown in Figure 2.1.

Since a lattice is a discrete version of a vector subspace, there always exists a smallest vector besides the trivially zero vector. Out of this, we can define the non-zero minimum of any lattice L the following way [31]:

$$\lambda_1(L) := min\{||\mathbf{x}||_2 : \mathbf{x} \in L, \mathbf{x} \neq 0\}.$$

The successive minima $\lambda_i(L)$ can be defined as the smallest radius r such that the n-dimensional ball of radius r and centred on the origin contains i lattice points [31]. Nowadays, many problems, especially in cryptography, can be reduced to finding the smallest non-zero vector in a lattice.

q-ary Lattices. In cryptography, one does not calculate in infinite rings, for example, $\mathbb{Z}$ but rather in finite ones, such as $\mathbb{Z}_q$, for some prime q. Recall that $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z} = \{0, ..., q-1\}$. So-called q-ary lattices use the same approach but with lattices. A q-ary lattice L is one such that $q\mathbb{Z}^n \subset L \subset \mathbb{Z}^n$ for some integer q. Thus, we focus only on finite lattices.The following two q-ary lattices will be used later to reduce a hard lattice problem to find the smallest non-zero vector:

Definition 3. *Given a matrix $A \in \mathbb{Z}_q^{n \times m}$, with $m \geq n$; we define two m-dimensional q-ary lattices the following way [31]:*

$$\Lambda_q(A) = \{\mathbf{y} \in \mathbb{Z}^m : \mathbf{y} = A^T \mathbf{z} \text{ (mod } q) \text{ for some } \mathbf{z} \in \mathbb{Z}^n\},$$
$$\Lambda_q^{\perp}(A) = \{\mathbf{y} \in \mathbb{Z}^m : A\mathbf{y} = 0 \text{ (mod } q)\}.$$

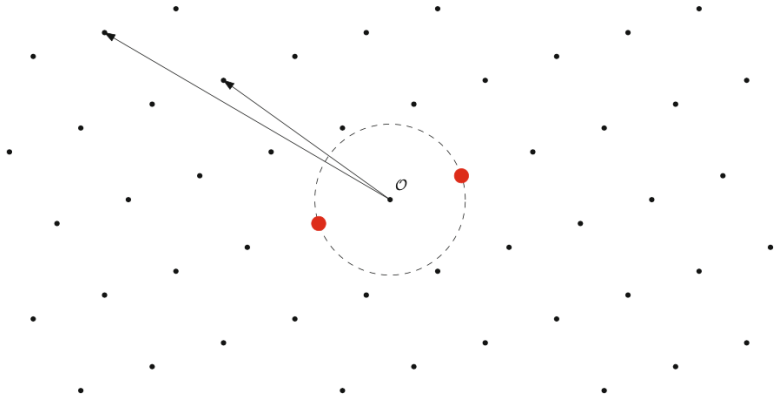

Figure 2.2: An example of the SV problem: given a lattice L, find the closest lattice points to the origin. As the two red points show, there might not be a unique solution [31].

2.2 Hard Lattice Problems

From the point of view of complexity theory, current cryptographic systems are not considered to be hard in their general case. Such cryptographic constructions are based on *average-case* hardness. A specific case must be taken to construct a hard problem. Factoring numbers, for example, is only considered hard if the number is large [24]. In the following section, we will discuss some problems presumed to be hard and used in the lattice-based signature scheme Dilithium. Additionally, they are proven to have the capability that solving a random case means solving any hard instance of the problem. This characteristic is called *worst-case hardness*. The idea behind a worst-case to average-case reduction is that we have a really hard instance of a problem. Then, we take a random instance of the problem and use reduction to show that this random instance has the same hardness as the worst-case instance. The random instance represents the average case of the problem. In conclusion, this means that the problem with this characteristic is always hard. On the other hand, this means that if there is an efficient algorithm breaking such a problem, the algorithm will be able to solve all instances of this problem efficiently [24].

Shortest Vector Problem (SVP). The most famous hard problem in a lattice is to determine the shortest vector within the lattice. One can think of this problem as given a basis, compute the shortest non-zero vector.

Definition 4. *There are three variants of the shortest vector problem, given a lattice basis B [31]:*

- *The shortest vector problem (SVP) is to find a non-zero vector $\mathbf{x}$ in the lattice L generated by B for which $||\mathbf{x}||_2 \leq ||\mathbf{y}||_2$ for all non-zero $\mathbf{y} \in L$, i.e. $||\mathbf{x}||_2 = \lambda_1(L)$.*
- *The approximate-SVP (SVP_γ) is to find a $\mathbf{x}$ such that $||\mathbf{x}||_2 \leq \gamma\lambda_1(L)$ for some 'small' constant γ.*
- *The γ-unique SVP ($uSVP_\gamma$) is given a lattice and a constant $\gamma > 1$ such that $\lambda_2(L) > \gamma\lambda_1(L)$, find a non-zero $\mathbf{x} \in L$ of length $\lambda_1(L)$.*

This problem can be visualised as in Figure 2.2. The solution to the SV problem is not unique; if we have $\mathbf{x} \in L$, then we also have $-\mathbf{x} \in L$.

This problem can be thought of in the following way: two bases, B_g and B_b, are given and depending on which basis one gets, the problem is hard or easy to solve. In this example B_g describes a 'good' basis

and B_b describes a 'bad' basis. 'Good' and 'bad' mean that with the 'good' basis, the SV problem is easy to solve; however, the problem is hard to solve with the 'bad' basis. An example of two different bases can be seen in Figure 2.1, where the red vecotrs form B_g and the blue vectors form B_b. Here, one can intuitively see that with the blue basis, the construction of the lattice and, thus, the solution to the SV problem is more challenging to do than with the red basis. We can see that a 'good' basis is 'more orthogonal' than the 'bad' one. In general, there is no guarantee of the existence of an orthogonal basis in a lattice. However, there exists an algorithm called the LLL-algorithm[2] which will create a reduced basis. We will not discuss this algorithm in this thesis, but further information can be found in [31].

Short Integer Solution (SIS) Problem. The next hard problem is called *Short Integer Solution* problem and is very similar to the SV problem.

Definition 5. *Given an integer q and vectors $\mathbf{a_1}, ..., \mathbf{a_m} \in \mathbb{Z}_q^n$ the SIS problem is to find a short vector $\mathbf{z} \in \mathbb{Z}^m$ such that $z_1\mathbf{a}_1 + ... + z_m\mathbf{a}_m = 0$ (mod q)*

'Short' refers to $||\mathbf{z}||_\infty = 1$.

The relation between the SIS problem and q-ary lattices is the following: if we set $A = (\mathbf{a_1}, ..., \mathbf{a_m}) \in \mathbb{Z}_q^{n\times m}$ and set

$$\Lambda_q^\perp(A) = \{\mathbf{z} \in \mathbb{Z}^m : A\mathbf{z} = 0 \text{ (mod } q)\},$$

the SIS problem becomes the SVP for the lattice $\Lambda^\perp(A)$. It is known that the SIS problem has worst-case hardness. The proof for this statement can be found in [22].

Ring-LWE. The next hard lattice problem is called learning with errors (LWE). A more specific formulation of LWE is called Ring-LWE. Given a ring R_q and an error distribution $D_{R,\sigma}$, the Ring-LWE problems are defined the following way [31]:

Definition 6. *Ring-LWE Search Problem: Pick $a, s \in R_q$ and $e \leftarrow D_{R,\sigma}$ and set $b := as + e$ (mod q). The search problem is given the pair (a, b) to output the value s.*

Definition 7. *Ring-LWE Decision Problem: Given (a, b) where $a, b \in R_q$ determine which of the following two cases holds:*

1. *b is chosen uniformly at random from R_q,*
2. *$b := as + e$ (mod q) where $e \leftarrow D_{R,\sigma}$ and $s \in R_q$.*

The ring R_q is defined as the ring $R = \mathbb{Z}_q[X]/F(X)$ reduced modulo q, where $F(X)$ denotes a integer polynomial of degree n. The distribution $D_{R,\sigma}$ produces polynomials of degree less than n and whose coefficients are distributed like a rounded normal distribution with a mean zero and a standard deviation of σ. The Ring-LWE problems are lattice-based problems with a worst-case to average-case reduction and thus have a worst-case hardness [23, 31].

2.3 Fiat-Shamir with Aborts

The Fiat-Shamir transform [12] can be used to transform an identification scheme into a signature scheme. An identification scheme, which we assume to be a Σ-Protocol, is a protocol that allows the holder of a secret key to prove its identity to anyone with access to the public key without disclosing the secret key [20]. The identification scheme consists of a key-generation algorithm and an interactive protocol between a prover, who owns a secret key, and a verifier with the public key [21].

The general identification scheme is a three-way protocol:

[2] named after its inventors: Lenstra, Lenstra, and Lovàsz [17].

1. Commitment phase: the prover commits to a certain value.
2. Challenge phase: the verifier responds with a challenge.
3. Verification phase: the prover has to provide a final response, called proof, which is connected to the challenge and the commitment. The verifier must be able to verify this proof.

This thesis focuses only on lattice-based identification schemes. More information about identification schemes and how the Fiat-Shamir transformation works can be found in [12, 15, 20, 21, 31].

One such lattice-based identification scheme is shown in Figure 2.3. In the first step, the prover picks a m-dimensional vector $\hat{\mathbf{y}}$ from some distribution D_y^m, and commits to it by sending $\mathbf{Y} = h(\hat{\mathbf{y}})$, where h is a cryptographic hash function, to the verifier. Conversely, the verifier selects a random challenge $\mathbf{c}$ from another distribution D_c and sends this challenge to the prover. The prover then computes $\hat{\mathbf{z}} = \hat{\mathbf{s}}\mathbf{c} + \hat{\mathbf{y}}$. If this $\hat{\mathbf{z}}$ falls into G^m, where G^m defines the space with elements that all suffice all security requirements, then the prover sends this result to the verifier. Otherwise, the protocol is aborted. The intuition behind aborting is that nothing about the secret key should be disclosed. Thus, as we see in Figure 2.3 in step 3, the prover will compute $\hat{\mathbf{z}} = \hat{\mathbf{s}}\mathbf{c} + \hat{\mathbf{y}}$. One can easily see that if the prover would only compute $\hat{\mathbf{z}} = \hat{\mathbf{s}}\mathbf{c}$, the verifier could simply conclude what $\hat{\mathbf{s}}$ is. Therefore, the masking vector $\hat{\mathbf{y}}$ is needed. Hence, the name *Fiat-Shamir with Aborts*. The main reason for aborting is to remain witness-indistinguishable. An identification scheme is said to be perfectly witness-indistinguishable if for any public key S, and any two valid secret keys s, s' (i.e. $s, s' \in D_s$ and $g^s \bmod N = g^{s'} \bmod N = S$), the view of any (possibly malicious) verifier has the exact same distribution in the interaction where the prover uses s as in the view where the prover uses s' [21].

Intuitively, it would make sense to pick $\hat{\mathbf{y}}$ uniformly random from a much larger space than $\hat{\mathbf{s}}$. However, in the lattice-based scheme, this would be infeasible because in doing so, much stronger complexity assumptions would be required, which would decrease the efficiency of the protocol [21]. Therefore, $\hat{\mathbf{y}}$ needs to be large enough to successfully mask the secret key $\hat{\mathbf{s}}$ and thus make the protocol witness-indistinguishable. On the other hand, it must be small enough to make the protocol efficient. If $\hat{\mathbf{y}}$ is not in the correct range, the prover aborts the protocol if the randomly picked $\hat{\mathbf{y}}$ is not in the correct range. Finally, the prover will accept the interaction[3] if $\hat{\mathbf{z}} \in G^m$ and $h(\hat{\mathbf{z}}) = \mathbf{S}\mathbf{c} + \mathbf{Y}$. That this identification scheme is sound and complete is proven in [21].

Private key: $\hat{\mathbf{s}} \xleftarrow{\$} D_s^m$
Public key: $h \xleftarrow{\$} \mathcal{H}(R, D, m), \mathbf{S} \leftarrow h(\hat{\mathbf{s}})$

Prover		Verifier
$\hat{\mathbf{y}} \xleftarrow{\$} D_y^m,\ \mathbf{Y} \leftarrow h(\hat{\mathbf{y}})$	$\xrightarrow{\mathbf{Y}}$	
	$\xleftarrow{\mathbf{c}}$	$\mathbf{c} \xleftarrow{\$} D_c$
$\hat{\mathbf{z}} \leftarrow \hat{\mathbf{s}}\mathbf{c} + \hat{\mathbf{y}}$		
if $\hat{\mathbf{z}} \notin \mathrm{G}^m$ then $\hat{\mathbf{z}} \leftarrow \perp$	$\xrightarrow{\hat{\mathbf{z}}}$	
		Accept iff $\hat{\mathbf{z}} \in \mathrm{G}^m$ and $h(\hat{\mathbf{z}}) = \mathbf{S}\mathbf{c} + \mathbf{Y}$

Figure 2.3: Lattice-Based Identification Scheme [21]. The prover wants to convince the verifier that she knows the private key $\hat{\mathbf{s}}$ without disclosing it. This is done by first committing to a value, in this case, the hash of the masking vector $\hat{\mathbf{y}}$. Upon receiving such a commitment, the verifier responds with a random challenge $\mathbf{c}$. Now, the prover can generate a proof ($\hat{\mathbf{z}}$) and send it back to the verifier. Finally, the verifier accepts if the proof matches the computation with the public key and the challenge.

[3]Note: for this specific scheme to work, h needs to have homomorphic properties.

2.4 Dilithium: Template Algorithm

First, we analyse a simplified version of the algorithm that captures all the functionalities optimised in the full version. Additionally, one can directly see the previously stated theoretical foundations from the sections above. The security of Dilithium is based on the short integer solution problem and learning with errors. Both problems depend on the kind of coefficients their parameters ($\mathbf{z}$ for SIS, and $\mathbf{e}$ for LWE) use. Whereas LWE mainly gets harder the larger the coefficient is, SIS gets easier. This intuitively makes sense since if the added error distribution is large, the reconstruction of the original secret vector $\mathbf{s}$ gets harder. On the other hand, the SIS problem is harder for short vectors per definition. Thus, a scheme like Dilithium aims to find the parameters in a way that both problems become equally hard. The simple Dilithium pseudo-code is shown in Algorithm 2.

Algorithm 1 Dilithium Template: State

$A \in R_q^{k \times l} := \{0\}^{k \times l}$	▷ base matrix of lattice
$\mathbf{s_1} \in S_\eta^l : 0\{0\}^l$	▷ secret vector
$\mathbf{s_2} \in S_\eta^k := \{0\}^k$	▷ secret vector
$\mathbf{t} \in R_q^k := \{0\}^k$	▷ public vector
$\mathbf{z} \in R_q^l := \{0\}^l$	▷ signature candidate
$\mathbf{y} \in R_q^l := \{0\}^l$	▷ masking vector
$\mathbf{w_1} \in R_q^k := \{0\}^k$	▷ high bits of $A\mathbf{y}$
$c \in R_q := 0$	▷ challenge

2.4.1 Basic Ring Operations

Before we go through Algorithm 2, we need to define certain operations. All definitions are taken from [7].

Modular Reductions (*centred reduction modulo* α.) $r' = r \bmod^{\pm} \alpha$: for an even α, r' is the unique element in range $-\frac{\alpha}{2} < r' \leq \frac{\alpha}{2}$ for an odd α, r' is the unique element in range $-\frac{\alpha-1}{2} \leq r' \leq \frac{\alpha-1}{2}$ such that $r' \equiv r \pmod{\alpha}$.

Size of elements. For an element $\hat{w} \in \mathbb{Z}_q$, we write $||\hat{w}||_\infty$ to mean $|\hat{w} \bmod^{\pm} q|$. We define the l_∞ and l_2 norms for $w = w_0 + w_1 X + ... + w_{n-1} X^{n-1} \in R$:

$$||w||_\infty = \max_i ||w_i||_\infty, \; ||w|| = \sqrt{||w_0||_\infty^2 + ... + ||w_{n-1}]||_\infty^2}.$$

Similarly, for $\mathbf{w} = (w_1, ..., w_k) \in R^k$, we define

$$||\mathbf{w}||_\infty = \max_i ||w_i||_\infty, \; ||\mathbf{w}|| = \sqrt{||w_1||^2 + ... + ||w_k||^2}. \tag{2.1}$$

We will write S_η to denote all elements $w \in R$ such that $||w||_\infty \leq \eta$.

Hashing to a ball. In this concrete application, hashing to a ball means that an element of a fixed length with all besides τ coefficients set to zero is created starting from a random source. This is shown in Algorithm 3. The τ coefficients different from zero take values from the set $\{-1, 1\}$. This exact form is needed for two purposes in Dilithium. First, it is a challenge; thus, some source of randomness is required. In Dilithium, this is done via the extendable output function of the SHA3-Suite [9], called SHAKE. The underlying *sponge*-function absorbs a set of values derived from the message, and

Algorithm 2 Dilithium Template

function GEN
  $A \leftarrow R_q^{k \times l}$
  $(\mathbf{s_1}, \mathbf{s_2}) \leftarrow S_\eta^l \times S_\eta^k$
  $\mathbf{t} := A\mathbf{s_1} + \mathbf{s_2}$
  return $(pk = (A, \mathbf{t}), sk = (A, \mathbf{t}, \mathbf{s_1}, \mathbf{s_2}))$

function SIGN(sk,M)
  $\mathbf{z} := \perp$
  while $\mathbf{z} = \perp$ **do**
    $\mathbf{y} \leftarrow S^l_{\gamma_1 - 1}$
    $\mathbf{w_1} :=$ HIGHBITS$(A\mathbf{y}, 2\gamma_2)$ ▷ only part of the information of $A\mathbf{y}$ is needed
    $c :=$ H$(M||\mathbf{w_1})$ ▷ the output of the hash function is sampled to a ball of radius τ
    $\mathbf{z} := \mathbf{y} + c\mathbf{s_1}$
    ▷ checks to ensure completeness and no secret information gets disclosed
    if $||\mathbf{z}||_\infty \geq \gamma_1 - \beta \vee ||$LOWBITS$(A\mathbf{y} - c\mathbf{s_2}, 2\gamma_2)||_\infty \geq \gamma_2 - \beta$ **then**
      $\mathbf{z} := \perp$ ▷ signature candidate gets rejected and computation starts again
  return $\sigma = (\mathbf{z}, c)$

function VERIFY(pk,M,$\sigma = (\mathbf{z}, c)$)
  $\mathbf{w_1'} :=$ HIGHBITS$(A\mathbf{z} - c\mathbf{t}, 2\gamma_2)$
  if $||\mathbf{z}||_\infty < \gamma_1 - \beta \wedge c =$ H$(M||\mathbf{w_1'})$ **then**
    return true
  else
    return false

with the according *squeeze*-function, one can generate as many random bits as needed. Second, the limitation of τ elements to be either plus or minus one is needed for the completeness of the protocol.

Algorithm 3 SampleInBall

1: Initialize $\mathbf{c} = c_0 c_1 ... c_{255} := 00...0$
2: **for** $i := 256 - \tau$ to 255 **do**
3: $j \leftarrow \{0, 1, ..., i\}$ ▷ get a random index between 0 and i
4: $s \leftarrow \{0, 1\}$
5: $c_i := c_j$ ▷ switch integer and random index with current loop-index i
6: $c_j := (-1)^s$ ▷ set the value at random index j to either -1 or plus 1.
7: **return c**

2.4.2 Algorithm

We will discuss now the three different parts of Algorithm 2, namely the `Gen()`, the `Sign(sk,M)`, and the `Verify(pk,M,`σ`)` subroutines.

Gen(): The first line of the `Gen()` subroutine of Algorithm 2 describes a random sampling of the matrix A. The coefficients of this matrix are in the polynomial ring $a_{ij} \in R_q$, where $q = 2^{23} - 2^{13} + 1 = 8380417$. Thus, each coefficient is a polynomial of degree n, where $n = 256$. Then again, each polynomial coefficient is in $\mathbb{Z}_q$.

Next, the function will sample the secret key vectors $\mathbf{s_1}$ and $\mathbf{s_2}$. Again, both vectors will have their coefficients as elements of R_q. In contrast to sampling the matrix A, the coefficients of the secret vectors will be 'small'. 'Small' in this case means of size at most η and thus be elements of S_η.

On the third line, the rest of the public key is generated by $\mathbf{t} = A\mathbf{s_1} + \mathbf{s_2}$. Here, one can see the connection to the LWE problem as discussed in Section 2.2. According to the LWE problem, it is hard to calculate the two secret vectors $\mathbf{s_1}$ and $\mathbf{s_2}$, given only the matrix A and the vector $\mathbf{t}$. On the other hand, it is straightforward, only linear operations, to calculate $\mathbf{t}$ if one knows $A, \mathbf{s_1}$, and $\mathbf{s_2}$.

Finally, the public key consists of the matrix A and the vector $\mathbf{t}$, while the secret key consists of the parts from the public key and the secret vectors $\mathbf{s_1}, \mathbf{s_2}$.

Sign(sk,M): The `Sign(sk,M)` function takes two arguments, the secret key `sk`=$(A, \mathbf{t}, \mathbf{s_1}, \mathbf{s_2})$ and the message `M`.

One of the significant differences to traditional digital signature schemes, such as RSA [25], is that Dilithium uses a rejection sampling for the signature and thus, does not directly compute a single signature but rather a potential signature. This scheme, called *Fiat-Shamir with Aborts*, is discussed in more detail in Section 2.3. If this potential signature does not fulfil specific criteria, it will be discarded, and the procedure will start again.

The first step is to sample a random vector, called the masking vector, $\mathbf{y}$ of size l. Once more, the elements of the vector $\mathbf{y}$ are elements of the polynomial ring R_q. Similar to the secret vectors, the coefficients of $\mathbf{y}$ have some limitations on their sizes and must not be larger than $\gamma_1 - 1$. The parameter γ_1 is defined so that it is large enough not to reveal the secret key, yet it is chosen to be small enough that the signature cannot be easily forged.

In the next step, the function will calculate $A\mathbf{y}$ and compute $\mathbf{w_1}$ as the 'high-bits' of the resulting vector. The concrete function which will calculate the 'high-bits' will be defined in Section 2.5. For

now, we focus on the intuition of what 'high-bits' means: each coefficient w_i of a vector $\mathbf{w}$ can be written in a canonical way: $w_i = w_{i,1} * 2\gamma_2 + w_{i,0}$, where $|w_{i,0}| \leq \gamma_2$. The vector $\mathbf{w_1}$ consists of the coefficients $w_{i,1}$.

As described in Section 2.3, schemes based on the Fiat-Shamir transformation need a challenge. In Dilithium, the challenge c is computed from the hash of the vector $\mathbf{w_1}$ and the message `M`. Furthermore, the challenge c is an element of R_q too. However, in difference to the other elements in R_q, c is designed to have precisely 60 coefficients that are either +1 or -1. This scheme is called *Hashing to a Ball* and is discussed in Subsection 2.4.2. Finally, the potential signature is calculated the following way: $\mathbf{z} = \mathbf{y} + c\mathbf{s_1}$.

To prevent disclosure of the secret key and to not be easily forgeable, the potential signature $\mathbf{z}$ needs to fulfil specific requirements. First, the parameter β is defined in that it corresponds to the maximum possible coefficients of $c\mathbf{s_i}$. Since the maximum coefficient of $\mathbf{s_i}$ is η and c has at most 60 ± 1's, β must be less or equal to 60η. Now, the potential signature must fulfil that its coefficients are smaller than $\gamma_1 - \beta$ and that the 'low-bits'[4] of $A\mathbf{z} - c\mathbf{t}$ is less or equal than $\gamma_2 - \beta$. The first requirement concerns security, whereas the second is needed for the security and correctness of the signature.

If the requirements are met, the signature $\sigma = (\mathbf{z}, c)$ is returned.

For an adversary to find a $\mathbf{z}'$ without owning $\mathbf{s_1}$ and $\mathbf{s_2}$ is the SIS problem described in Section 2.2.

Verify(pk, M, σ): Similar to the `Sign(sk,M)`, the `Verify(pk,M,`σ`)` function takes a set of parameters: the public key `pk`=$(A, \mathbf{t})$, the message `M` and the signature $\sigma = (\mathbf{z}, c)$. Similar to the $\mathbf{w_1}$, thus the naming, the elements of the vector $\mathbf{w_1'}$ are the 'high-bits' of the calculation $A\mathbf{z} - c\mathbf{t}$. The signature is accepted, if all coefficients of $\mathbf{z}$ are less than $\gamma_1 - \beta$ and c is equal to $H(M||w_1')$. The correctness, also known as the completeness property, will be shown immediately.

Completeness. There are different requirements for a signature scheme; one requirement is completeness. Completeness means that the verification algorithm must accept each signature generated according to the protocol, given the corresponding public key.

Theorem 1. *The simple Dilithium pseudo-code as defined in Algorithm 2 is complete.*

Proof. To show that a signature scheme is *correct*, one has to show that if the creation procedure for a signature is followed correctly, the `Verify` function will accept the resulting signature. To check the first condition of the `Verify` function, namely that $||\mathbf{z}||_\infty < \gamma_1 - \beta$, is trivial, because it follows directly the construction of the signature. The second part is more extensive. It is simple to see that we need to have $\mathbf{w}_1' = \mathbf{w}_1$ to have a correct signature. Therefore, let us start by showing that $A\mathbf{z} - c\mathbf{t} = A\mathbf{y} - c\mathbf{s_2}$:

$$\begin{aligned} A\mathbf{z} - c\mathbf{t} &= A(\mathbf{y} + c\mathbf{s_1}) - c(A\mathbf{s_1} + \mathbf{s_2}), \\ &= A\mathbf{y} + cA\mathbf{s_1} - cA\mathbf{s_1} - c\mathbf{s_2}, \\ &= A\mathbf{y} - c\mathbf{s_2}. \end{aligned}$$

Then, the only thing that is left to show is

$$\text{HighBits}_q(A\mathbf{y}, 2\gamma_2) = \text{HighBits}_q(A\mathbf{y} - c\mathbf{s_2}, 2\gamma_2).$$

For this, we show the following property [7]: If $||s||_\infty \leq \beta$ and $||\text{LowBits}_q(\mathbf{r}, \alpha)||_\infty < \alpha/2 - \beta$, then

$$\text{HighBits}_q(\mathbf{r} + \mathbf{s}, \alpha) = \text{HighBits}_q(\mathbf{r}, \alpha).$$

[4]The 'low-bits' use the same logic as the 'high-bits' but in difference, the w_0 coefficients are used instead of the the w_1's.

The next part of the proof is done for one component r. The extension to a vector $\mathbf{r}$ follows trivially. From $||\text{LowBits}_q(r,\alpha)||_\infty < \frac{\alpha}{2} - \beta$ it follows that $r = r_1 * \alpha + r_0$, with[5] $-\frac{\alpha}{2} + \beta < r_0 \leq \frac{\alpha}{2} - \beta$. Therefore, we know that $r + s = r_1\alpha + (r_0 + s)$, with $-\frac{\alpha}{2} < r_0 + s \leq \frac{\alpha}{2}$. This is because we know that $|s| \leq \beta$; thus, we 'enlarge' the range by β. Next, we deduce that $r + s \bmod^{\pm} \alpha = r_0 + s$. This is basically the $\text{LowerBits}_q(r + s)$ and $r_1\alpha$ is simply 'removed' by mod α. Out of this, we can show:

$$(r + s) - ((r + s) \bmod^{\pm} \alpha) = r_1\alpha = r - (r \bmod^{\pm} \alpha).$$

Here are the short explanations:

$$(r + s) - ((r + s) \bmod^{\pm} \alpha) = \text{HighBits}_q(r + s, \alpha),$$

this property holds because we have the following statements: firstly,

$$(r + s) = r_1\alpha + (r_0 + s),$$
$$(r + s) \bmod^{\pm} \alpha = r_0 + s.$$

and secondly,

$$r_1\alpha = r - (r \bmod^{\pm} \alpha) = \text{HighBits}_q(r, \alpha),$$

this is due to the following fact:

$$r = r_1\alpha + r_0,$$
$$r \bmod^{\pm} \alpha = r_0.$$

In conclusion, we have:

$$\begin{aligned}\text{HighBits}_q(r + s, \alpha) &= (r + s) - ((r + s) \bmod^{\pm} \alpha),\\ &= r_1\alpha,\\ &= r - (r \bmod^{\pm} \alpha),\\ &= \text{HighBits}_q(r, \alpha).\end{aligned}$$

□

2.5 Dilithium: Optimized Implementation

The algorithm above in Section 2.4 was provided by [7] to explain the functionality of Dilithium in a more simplified manner. The actual implementation, the one CRYSTAL submitted for the NIST competition, is more advanced. Most of the changes introduced in the submitted implementation are reasoned by efficiency. The idea of the changes was to introduce a better trade-off, for example, in terms of key sizes against signature sizes. In addition, the changes also help to introduce other techniques to decrease the signature sizes in general.

First, we discuss some helper functions, shown in Algorithm 4.

Power2Round. Breaks up an element $r = r_1 2^d + r_0$, so we have $r_0 = r \bmod^{\pm} 2^d$ and $r_1 = \frac{r - r_0}{2^d}$.

Decompose. The $\texttt{Decompose}_q(r, \alpha)$ function will split up the value r into a unique form of $r = r_1 * \alpha + r_0$, where $0 \leq r_0 < \alpha$. Figuratively, one can explain the $\texttt{Decompose}_q(r, \alpha)$ by using an example: you have the numbers $r = 10$ and $\alpha = 6$, then the function would start calculating how many times 6 fits into 10 ($= r_1$) and then the rest is r_0. In this case: $r_1 = 1, r_0 = 4$. This is only an example and a substantial simplification and should only help for better understanding and not be taken as a fact. The actual definition can be found in [7].

[5] In the paper [7] it is stated the following way: $-\frac{\alpha}{2} + \beta < r_0 \leq \frac{\alpha}{2} + \beta$, while this is technically not wrong, it might be a bit misleading.

Algorithm 4 Dilithium: Helper Functions

function $\text{POWER2ROUND}_q(r, d)$ ▷ breaks up an element $r = r_1 \cdot 2^d + r_0$
  $r := r \bmod^+ q$
  $r_0 := r \bmod^\pm 2^d$
  return $((r - r_0)/2^d, r_0)$

function $\text{MAKEHINT}_q(z, r, \alpha)$ ▷ creates a hint depending on the carry of the addition
  $r_1 := \text{DECOMPOSE}_q(r, \alpha)$
  $v_1 := \text{HIGHBITS}_q(r + z, \alpha)$
  if $r_1 \neq v_1$ **then**
    return true
  else
    return false

function $\text{USEHINT}_q(h, r, \alpha)$ ▷ restore the actual addition with the help of a hint
  $m := (q - 1)/\alpha$
  $(r_1, r_0) := \text{DECOMPOSE}_q(r, \alpha)$
  if $h = 1 \wedge r_0 > 0$ **then**
    return $(r_1 + 1) \bmod^+ m$
  if $h = 1 \wedge r_0 \leq 0$ **then**
    return $(r_1 - 1) \bmod^+ m$

function $\text{DECOMPOSE}_q(r, \alpha)$ ▷ $r = r_1 \cdot \alpha + r_0$
  $r := r \bmod^+ q$
  $r_0 := r \bmod^\pm \alpha$
  if $r - r_0 = q - 1$ **then**
    $r_1 := 0$
    $r_0 := r_0 - 1$
  else
    $r_1 := (r - r_0)/\alpha$
  return (r_1, r_0)

function $\text{HIGHBITS}_q(r, \alpha)$ ▷ r_1 from the break-up of an element ($r = r_1 \cdot \alpha + r_0$)
  $(r_1, r_0) := \text{DECOMPOSE}_q(r, \alpha)$
  return r_1

function $\text{LOWBITS}_q(r, \alpha)$ ▷ r_0 from the break-up of an element ($r = r_1 \cdot \alpha + r_0$)
  $(r_1, r_0) := \text{DECOMPOSE}_q(r, \alpha)$
  return r_0

HighBits. The 'high-bits' are thus simply the r_1 coefficients after decomposing the value r.

LowBits. Similar to the 'high-bits', the 'low-bits' are calculated with the $\texttt{Decompose}_q(r, \alpha)$ function. In difference, the 'low-bits' are taking the r_0 coefficients.

MakeHint. The $\texttt{MakeHint}(h, r, \alpha)$ function decides if a hint is needed to verify a signature correctly. This is done by comparing if the $\texttt{HighBits}_q(r, \alpha)$ function changes if we have the sum or not. If it changes, we need a hint; if it does not, we can omit the hint.

UseHint. In simplification, this function is used for restoring the 'high-bits' if a hint is present. If the hint is present, we must decide if we have to add or subtract 1 from the 'high-bits'. The idea behind the *Hints* is to reduce the public key size.

Expand*. The $\texttt{Expand*}(\rho)$ functions are used to create whatever is needed from a seed deterministically. For example, given a seed ρ, a matrix A could be reconstructed. The idea is that less space is needed to store ρ compared to the whole matrix A. On the other hand, more computations are needed to restore the matrix A every time from the seed.

Algorithm 5 Dilithium: State

$\rho := \{0\}^{256}$	▷ seed to restore the matrix A
$K := \{0\}^{256}$	▷ Seed to create $\mathbf{y}$
$A \in R_q^{k \times l} := \{0\}^{k \times l}$	▷ base matrix of lattice
$\mathbf{s_1} \in S_\eta^l := \{0\}^l$	▷ secret vector
$\mathbf{s_2} \in S_\eta^k := \{0\}^k$	▷ secret vector
$\mathbf{t} \in R_q^k := \{0\}^k$	▷ public vector
$\mathbf{z} \in R_q^l := \{0\}^l$	▷ signature candidate
$\mathbf{y} \in R_q^l := \{0\}^l$	▷ masking vector, deterministically created from K
$\mathbf{w_1} \in R_q^k := \{0\}^k$	▷ high bits of $A\mathbf{y}$
$c \in R_q := 0$	▷ challenge
$\mathbf{t_1} \in R_q^k := \{0\}^k$	▷ higher part of $\mathbf{t}$
$\mathbf{t_0} \in R_q^k := \{0\}^k$	▷ lower part of $\mathbf{t}$
$tr := \{0\}^{384}$	▷ more efficient way to store information about the public key
$\kappa \in \mathbb{Z} := 0$	▷ index in order to have a different $\mathbf{y}$ after each rejection
$\mathbf{h} \in R_q^k := \{0\}^k$	▷ vector of 1-bit hints

The first round submission from CRYSTAL is shown in Algorithm 6[6]. It can be directly seen that the optimised Dilithium implementation is more complex than the template algorithm discussed in Subsection 2.4.2. We will go through the functions and mention the similarities but, more important, the differences between the template algorithm.

Gen(): In difference to the template algorithm from Subsection 2.4.2, A is not randomly sampled but rather deterministically generated from a randomly sampled seed $\rho \leftarrow \{0,1\}^{256}$. Since A is part of the public key, ρ will use less space than A. In addition, A is stored in NTT Domain Representation[7]. The NTT representation will help to make multiplications with polynomials more efficient [19].

[6]At the beginning (March 2022) of working on the thesis, the third round in the NIST competition was running. At the end of the thesis, Dilithium is one among three to be standardised. There are some slight differences between Dilithium's first and third-round submission, which will not be discussed in this thesis. The changelog can be found in the latest specifications at `https://pq-crystals.org/dilithium/index.shtml`.

[7]Check `https://www.nayuki.io/page/number-theoretic-transform-integer-dft` or `https://cgyurgyik.github.io/posts/2021/04/brief-introduction-to-ntt/` for more information on this topic

Algorithm 6 Dilithium

function GEN
- $\rho \leftarrow \{0,1\}^{256}$
- $K \leftarrow \{0,1\}^{256}$
- $(\mathbf{s_1}, \mathbf{s_2}) \leftarrow S_\eta^l \times S_\eta^k$
- $A \in R_q^{k \times l} :=$ EXPANDA(ρ)
- $\mathbf{t} := A\mathbf{s_1} + \mathbf{s_2}$
- $(\mathbf{t_1}, \mathbf{t_0}) :=$ POWER2ROUND$_q(\mathbf{t}, d)$ ▷ distribute the information of $\mathbf{t}$ to two elements
- $tr \in \{0,1\}^{384} :=$ CRH($\rho || \mathbf{t_1}$)
- **return** $(pk = (\rho, \mathbf{t_1}), sk = (\rho, K, tr, \mathbf{s_1}, \mathbf{s_2}, \mathbf{t_0}))$ ▷ smaller public key than in the simpler version

function SIGN(sk, M)
- $A \in R_q^{k \times l} :=$ EXPANDA(ρ)
- $\mu \in \{0,1\}^{384} :=$ CRH($tr || M$)
- $\kappa := 0, (\mathbf{z}, \mathbf{h}) := \perp$
- **while** $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
  - $\mathbf{y} \in S_{\gamma_1 - 1}^l :=$ EXPANDMASK($K || \mu || \kappa$)
  - $\mathbf{w} := A\mathbf{y}$
  - $\mathbf{w_1} :=$ HIGHBITS$_q(\mathbf{w}, 2\gamma_2)$
  - $c \in B_{60} :=$ H($\mu || \mathbf{w_1}$)
  - $\mathbf{z} := \mathbf{y} + c\mathbf{s_1}$
  - $(\mathbf{r_1}, \mathbf{r_0}) :=$ DECOMPOSE$_q(\mathbf{w} - c\mathbf{s_2}, 2\gamma_2)$
  - **if** $||\mathbf{z}||_\infty \geq \gamma_1 - \beta \vee ||\mathbf{r_0}||_\infty \geq \gamma_2 - \beta \vee \mathbf{r_1} \neq \mathbf{w_1}$ **then**
    - $(\mathbf{z}, \mathbf{h}) := \perp$
  - **else**
    - $\mathbf{h} :=$ MAKEHINT$_q(-c\mathbf{t_0}, \mathbf{w} - c\mathbf{s_2} + c\mathbf{t_0}, 2\gamma_2)$ ▷ use hints in order to shrink signature size
    - **if** $||c\mathbf{t_0}||_\infty \geq \gamma_2 \vee$ # of 1's in $\mathbf{h}$ is greater than ω **then**
      - $(\mathbf{z}, \mathbf{h}) := \perp$
  - $\kappa := \kappa + 1$
- **return** $\sigma = (\mathbf{z}, \mathbf{h}, c)$

function VERIFY($pk, M, \sigma = (\mathbf{z}, \mathbf{h}, c)$)
- $A \in R_q^{k \times l} :=$ EXPANDA(ρ)
- $\mu \in \{0,1\}^{384} :=$ CRH(CRH($\rho || \mathbf{t_1}) || M$)
- $\mathbf{w_1'} :=$ USEHINT$_q(\mathbf{h}, A\mathbf{z} - c\mathbf{t_1} \cdot 2^d, 2\gamma_2)$
- **if** $||\mathbf{z}||_\infty < \gamma_1 - \beta \wedge c =$ H($\mu || \mathbf{w_1'}$) $\wedge$ # of 1's in $\mathbf{h}$ is $\leq \omega$ **then**
  - **return** true
- **else**
  - **return** false

Similarly to the template algorithm, we calculate $\mathbf{t} := A\mathbf{s_1} + \mathbf{s_2}$, but in difference to before, $\mathbf{t}$ is not used in the public key but only parts of it, namely $\mathbf{t_1}$. $\mathbf{t_1}$ is computed the following way: $(\mathbf{t_1}, \mathbf{t_0}) := \texttt{Power2Round}_q(\mathbf{t}, d)$. The public key consists of the tuple $\texttt{pk} = (\rho, \mathbf{t_1})$ and the secret key $\texttt{sk} = (\rho, K, tr, \mathbf{s_1}, \mathbf{s_2}, \mathbf{t_0})$.

Sign(sk,M): This function receives the secret key sk and the message M as parameters. The first step is to create the matrix A with the help of the $\texttt{ExpandA}(\rho)$ function.

μ saves the hash of tr concatenated with the message $\mu = \text{CRH}(tr||M)$[8]. Again, a signature candidate is computed, which means that if the candidate does not withstand the rejection sampling, it will be discarded, and a new candidate will be computed. Here, κ is used in the form of a counter. Each time the signature candidate is discarded, κ will be incremented, and the sampling will be restarted. This is needed because otherwise, the ExpandMask function would always return the same thing, and therefore, the procedure would be stuck in an infinite loop.

In difference to the template algorithm, $\mathbf{y}$ is not sampled randomly but deterministically. For this, one uses K, μ and κ the following way $\mathbf{y} := \texttt{ExpandMask}(K||\mu||\kappa)$.

The calculation of $\mathbf{w_1}$ is again equal to the template algorithm which is $\mathbf{w_1} := \texttt{HighBits}_q(A\mathbf{y}, 2\gamma_2)$.

The challenge c is not calculated as the hash of the message M and $\mathbf{w_1}$ but instead as the hash of μ and $\mathbf{w_1}$.

The computation for the signature candidate $\mathbf{z}$ is once more equal to the template scheme.

Before one jumps to the rejection-sampling part, in the Dilithium scheme, one computes the decomposition of $\mathbf{w} - c\mathbf{s_2}$ subject to $2\gamma_2$: $(\mathbf{r_1}, \mathbf{r_0}) := \texttt{Decompose}_q(\mathbf{w} - c\mathbf{s_2}, 2\gamma_2)$.

The rejection sampling checks if $||\mathbf{z}||_\infty \geq \gamma_1 - \beta \vee ||\mathbf{r_0}||_\infty \geq \gamma_2 - \beta \vee \mathbf{r_1} \neq \mathbf{w_1}$. If one statement returns true, then the candidate is rejected.

If the candidate is good, we then create a hint: $\mathbf{h} := \texttt{MakeHint}_q(-c\mathbf{t_0}, \mathbf{w} - c\mathbf{s_2} + c\mathbf{t_0}, 2\gamma_2)$. This creates a vector $\mathbf{h}$ in which each 'coordinate' express one bit. The idea behind hints is to reduce the size of the signature. If the bit equals 1, the summation causes a 'carry'.

There are two conditions which must not be met:

1. if $||c\mathbf{t_0}||_\infty \geq \gamma_2$ (the largest coefficient of $c\mathbf{t_0}$ is larger than γ_2),
2. if the number of 1's in $\mathbf{h}$ is greater than w.

The signature is then composed by $\sigma = (\mathbf{z}, \mathbf{h}, c)$.

Verify(pk,M,σ): The verification algorithm takes the public key pk, the message M, and the signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$ as parameters.

First, the matrix A must be reconstructed from the seed ρ. Then, μ gets created equally as in the Gen and Sign algorithm.

To reconstruct $\mathbf{w_1'}$ one needs to use the hint: $\mathbf{w_1'} = \texttt{UseHint}_q(\mathbf{v}, A\mathbf{z} - c\mathbf{t_1}2^d, 2\gamma_2)$. Recall, the $\texttt{UseHint}_q\,(h, r, \alpha)$ is like the inverse function to the $\texttt{MakeHint}\,(h, r, \alpha)$ function.

Finally, the signature is accepted if:

1. $||\mathbf{z}||_\infty < \gamma_1 - \beta$,
2. $c = \text{H}(\mu||\mathbf{w_1'})$,
3. Number of 1's in $\mathbf{h} \leq w$.

[8]CRH means Collision Resistant Hash-Function.

Completeness. Similar to the simplified version, the completeness needs to be proven. The following lemma is needed for this proof [7]:

Lemma 1. *Suppose that q and α are positive integers satisfying $q > 2\alpha, q \equiv 1 (mod\ \alpha)$ and α is even. Let $\mathbf{r}$ and $\mathbf{z}$ be vectors of elements in R_q where $||z||_\infty \leq \alpha/2$, and let $\mathbf{h}, \mathbf{h}'$ be vectors of bits. Then, the $\texttt{HighBits}_q$, $\texttt{MakeHint}_q$, and $\texttt{UseHint}_q$ algorithms satisfy the following properties:*

1. $\texttt{UseHint}_q(\texttt{MakeHint}_q(\mathbf{z}, \mathbf{r}, \alpha), \mathbf{r}, \alpha) = \texttt{HighBits}_q(\mathbf{r} + \mathbf{z}, \alpha)$.
2. *Let $\mathbf{v_1} = \texttt{UseHing}_q(\mathbf{h}, \mathbf{r}, \alpha)$. Then, $||\mathbf{r} - \mathbf{v_1}\alpha||_\infty \leq \alpha + 1$. Furthermore, if the number of 1's in $\mathbf{h}$ is w, then all except at most w coefficients of $\mathbf{r} - \mathbf{v_1}\alpha$ will have magnitude at most $\alpha/2$ after centred reduction modulo q.*
3. *For any $\mathbf{h}, \mathbf{h}'$, if $\texttt{UseHint}_q(\mathbf{h}, \mathbf{r}, \alpha) = \texttt{UseHint}_q(\mathbf{h}', \mathbf{r}, \alpha)$, then $\mathbf{h} = \mathbf{h}'$.*

Sketch. The first and the third part are proven by applying the corresponding helper functions to the given variables. The second part of the lemma is proven by using three cases ($h = 0$, $h = 1, r_0 > 0$, and $h = 1, r_0 \leq 0$) and show that for each case, the condition still holds. □

A more technical and rigorous proof can be found in appendix A of [7].

Theorem 2. *The Dilithium pseudo-code as defined in Algorithm 6 is complete.*

Proof. Requirements one and three from the verification function to accept a signature are trivially fulfilled since they follow directly from the creation of the signature. What is left to show is that:

$$\mathbf{w_1'} = \mathbf{w_1}.$$

If $||c\mathbf{t_0}||_\infty < \gamma_2$ and with the help of Lemma 1, we know that:

$$\begin{aligned}
\texttt{UseHint}_q(\mathbf{h}, \mathbf{w} - c\mathbf{s_2} + c\mathbf{t_0}, 2\gamma_2) &= \texttt{UseHint}_q(\texttt{MakeHint}_q(-c\mathbf{t_0}, \mathbf{w} - c\mathbf{s_2} + c\mathbf{t_0}, 2\gamma_2), \mathbf{w} - c\mathbf{s_2} + c\mathbf{t_0}, 2\gamma_2), \\
&= \texttt{HighBits}_q(\mathbf{w} - c\mathbf{s_2} + c\mathbf{t_0} + (-c\mathbf{t_0}), 2\gamma_2), \\
&= \texttt{HighBits}(\mathbf{w} - c\mathbf{s_2}, 2\gamma_2).
\end{aligned}$$

Since, $\mathbf{w} = A\mathbf{y}$ and $\mathbf{t} = A\mathbf{s_1} + \mathbf{s_2}$, we can conclude that:

$$\begin{aligned}
\mathbf{w} - c\mathbf{s_2} &= A\mathbf{y} - c\mathbf{s_2}, \\
&= A(\mathbf{z} - c\mathbf{s_1}), \\
&= A\mathbf{z} - c\mathbf{t}.
\end{aligned}$$

Now, we show that $\mathbf{w} - c\mathbf{s_2} + c\mathbf{t_0} = A\mathbf{z} - c\mathbf{t_1}2^d$:

$$\begin{aligned}
A\mathbf{z} - c\mathbf{t_1}2^d &= A(\mathbf{y} + c\mathbf{s_1}) - c(\frac{\mathbf{t} - \mathbf{t_0}}{2^d})2^d, \\
&= A\mathbf{y} + cA\mathbf{s_1} - c\mathbf{t} + c\mathbf{t_0}, \\
&= A\mathbf{y} + cA\mathbf{s_1} - c(A\mathbf{s_1} + \mathbf{s_2}) + c\mathbf{t_0}, \\
&= A\mathbf{y} + cA\mathbf{s_1} - cA\mathbf{s_1} - c\mathbf{s_2} + c\mathbf{t_0}, \\
&= \mathbf{w} - c\mathbf{s_2} + c\mathbf{t_0}.
\end{aligned}$$

Here are some explanations, all received by using the helper functions:

$$\begin{aligned}
\mathbf{t_1} &= \frac{\mathbf{t} - \mathbf{t_0}}{2^d}, \\
\mathbf{t} &= A\mathbf{s_1} + \mathbf{s_2}, \\
\mathbf{w} &= A\mathbf{y}, \\
\mathbf{z} &= \mathbf{y} + c\mathbf{s_1}.
\end{aligned}$$

Thus, the verifier computes:

$$\texttt{UseHint}_q(\mathbf{h}, \mathbf{Az} - c\mathbf{t_1}2^d, 2\gamma) = \texttt{HighBits}_q(\mathbf{w} - c\mathbf{s_2}, 2\gamma_2).$$

Since $\mathbf{r_1} = \mathbf{w_1}$ this is equivalent to:

$$\texttt{HighBits}_q(\mathbf{w} - c\mathbf{s_2}, 2\gamma_2) = \texttt{HighBits}_q(\mathbf{w}, 2\gamma_2).$$

Because of the facts from the algorithm:

$$\mathbf{r_1} = \texttt{HighBits}_q(\mathbf{w} - c\mathbf{s_2}, 2\gamma_2),$$
$$\mathbf{w_1} = \texttt{HighBits}_q(\mathbf{w}, 2\gamma_2).$$

□

3
Secure Multi-Party Computation

This chapter will discuss the general ideas and approaches of secure multi-party computation (MPC), which form this thesis' second main theoretical topic. First, we will discuss some preliminaries in Section 3.1 and progress to the two main approaches in MPC. Section 3.2 will focus on the principles of garbled circuits. The second approach, secret sharing, will be discussed in Section 3.3. These sections are mainly following [31]. In Section 3.4 the used framework for this thesis, MP-SPDZ, is presented.

3.1 Preliminaries

Representations of Computations. Computations can be represented in two equivalent forms [5, 31]. The first representation is called *boolean circuits*. As the name suggests, a computation is represented using a set of wires $W = \{w_1, ..., w_n\}$ and boolean gates $G = \{g_1, ..., g_m\}$. In addition, the wires transmit boolean values. Universal boolean gates are `OR`, `AND`, and `NOT`. The MPC approach concerning boolean circuits is called *Garbled Circuits* and was Yao's direct answer to his millionaire's problem.

The second representation is called *arithmetic circuit*. Here, the boolean values are substituted with elements from a finite field. The computation is still represented as a circuit. The difference in boolean circuits, the wires transmit elements of the finite field, and the gates are represented as multiplications and additions over the corresponding field. Secret sharing schemes use the secure multi-party computation approach based on arithmetic circuits.

There is no better representation since it depends on the computation. For example, linear arithmetic computations, such as simple additions, are more efficient in arithmetic than in boolean circuits. On the other hand, comparisons are performed more efficiently in boolean circuits. A bridge form exists for the wires to have the flexibility to switch between the two representations in the implementation. Hence, the values can be used in both secret sharing schemes as well as in garbled circuits. The technique used for this is called doubly-authenticated bits, in short *daBits* [26] and the improved version: *edaBits* [10]. This thesis will not explain this in further detail.

Security Models. Security models describe the kind of adversaries a protocol can defend. First, we have *honest-but-curious* adversaries. These adversaries will follow the protocol but try to extract as much

information as possible from other participants. The corresponding security model, which will ensure safety against such adversaries, is called *semi-honest*, and protocols are said to have *passive security*.

Second, the adversaries can deviate from the protocol and may pass incorrect data to the other participants. Such adversaries are called *malicious adversaries*. The security model ensuring safety against such malicious adversaries is called *active*, and such protocols are called robust. In addition, in every protocol, there will be a point where the parties need to publish their result and thus share them with the other participants.

Both of the mentioned security models can either be applied in a *honest majority* or a *corrupt majority* setting. An honest majority means that although there are adversaries present, they will make up less than half of the participants. On the other hand, in a corrupt majority, there might be more adversaries than honest participants. Noteworthy, a protocol cannot guarantee output in a corrupt majority with an active adversaries security model.

More information about security models can be found in [5, 18, 31].

3.2 Garbled Circuits

Yao [35] presented a solution to the secure two-party computation problem called garbled circuits. The main idea of garbled circuits is to use the publicly known function in the boolean circuit representation and encrypt the different gates so that not a single party can learn more than the result of the function and their respective input. In the first phase, a party called A constructs the garbled circuit. In a second phase, the other party, B, evaluates the garbled circuit.

Garbled Circuit Construction. Recall: a circuit consists of a set of wires $W = \{w_1, ..., w_n\}$ and a set of gates $G = \{g_1, ..., g_m\}$. The garbled circuit is constructed by party A as follows:

- For each wire w_i a random value $\rho_i \in \{0, 1\}$ is chosen. This value is needed to encrypt the actual wire value. In more detail, this means that if the real value of the wire i is v_i, then the encrypted value is given by $e_i = v_i \oplus \rho_i$.
- For each wire w_i two random cryptographic keys, k_i^0 and k_i^1, are selected. The first represents the encryption of the zero value of the wire i, whereas the second represents the encryption of the wire i with value one. The encrypted value e_i will be encrypted again with the corresponding cryptographic key.
- The 'garbled table' is computed for each gate, representing the gate's function on these encrypted values. Here is an example for gate g_i that has the input wires w_{i_0} and w_{i_1} and the output wire w_{i_2}. Then, the table is the following four values, each for a combination of a and b, for some encryption function E: $c_{a,b}^{w_{i_2}} = E_{k_{w_{i_0}}^{a \oplus \rho_{i_0}}, k_{w_{i_1}}^{b \oplus \rho_{i_1}}}(k_{w_{i_2}}^{o_{a,b}} || (o_{a,b} \oplus \rho_{i_2}))$ for $a, b \in \{0, 1\}$, where $o_{a,b} = g_i(a \oplus \rho_{i_0}, b \oplus \rho_{i_1})$.

Since the formula for the garbled table looks complicated, it is easier to explain using an example. Considering the `AND` gate: $f(\{w_1\}, \{w_2\}) = w_1 \wedge w_2$, where w_1 is the input wire from party A and w_2 is the input wire of party B. The respective binary circuit is shown in Figure 3.1. Further, suppose the following values $\rho_1 = 1, \rho_2 = \rho_3 = 0$ are given. The two garbled values for the first wire w_1 are k_1^0 and k_1^1, representing the 0 and 1 values, respectively. Since the corresponding ρ_1 is 1, the external value e_1 of the internal value 1 is thus 0. Therefore, the representation of the wire looks as follows: $(k_1^0||1, k_1^1||0)$. Similar for the second wire: $(k_2^0||0, k_2^1||1)$. The next step is to look at the used `AND` gate. The first entry in the final garbled table corresponds to $a = b = 0$. The ρ values for the first and second wires are 1 and 0, respectively. Thus, the first entry in the table corresponds to what would happen if the keys k_1^1 and k_2^0 are

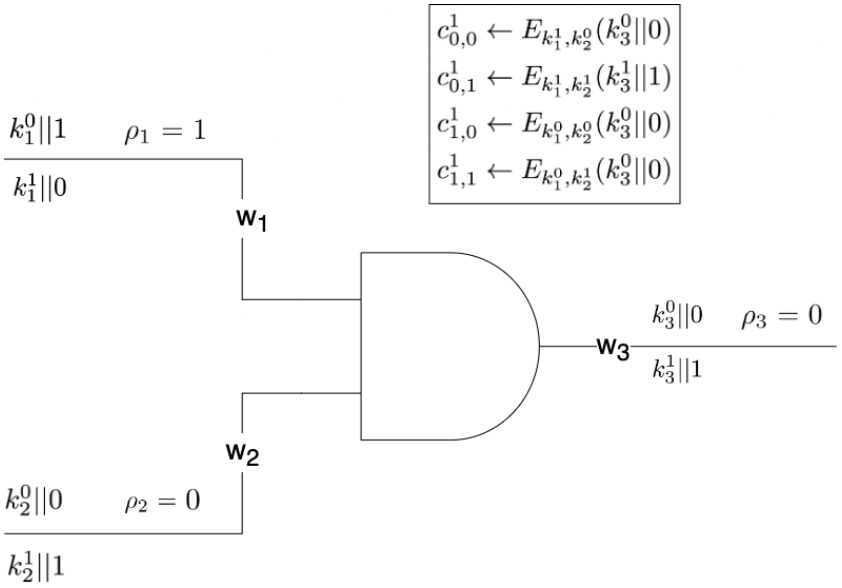

Figure 3.1: Binary circuit of an AND function with the corresponding wire values and the garbled table.

seen. This is due to the fact that $1 = 1 \oplus 0 = \rho_1 \oplus a$ and $0 = 0 \oplus 0 = \rho_2 \oplus b$. Similarly, the rest of the garbled table can be constructed. This leads to the following:

$$
\begin{aligned}
c^1_{0,0} &\leftarrow E_{k^1_1,k^0_2}(k^0_3||0),\\
c^1_{0,1} &\leftarrow E_{k^1_1,k^1_2}(k^1_3||1),\\
c^1_{1,0} &\leftarrow E_{k^0_1,k^0_2}(k^0_3||0),\\
c^1_{1,1} &\leftarrow E_{k^0_1,k^1_2}(k^0_3||0).
\end{aligned}
$$

Garbled Circuit Evaluation. We take the example from Figure 3.1 to explain party B's evaluation of the garbled circuit. Suppose B received, in some way[1], the values $(k^0_1||1)$ and $(k^1_2||1)$ for the input wires to the `AND` gate. In addition, A will share all the ρ_i values of the final output gates of the circuit but no intermediate ρ_j. In this case, it is only ρ_3, the value from the output of the `AND` gate. After the sharing, the evaluation starts. B knows the external value of wire 1 is 1, and the external value of wire 2 is 1. Thus, B takes the value $c^1_{1,1}$ from the garbled table of gate 1. Since B is in possesion of k^0_1 and k^1_2, B is able to decrypt $c^1_{1,1}$ and receive the value $k^0_3||0$. Note, if this wire went into some next gate, B would not know the internal value of this wire since A would not have shared ρ_3 with him. In this example, this is already the final output. Thus, B can, with the help of ρ_3, determine the actual output value which, in this example, is 0, since $0 = 0 \oplus 0 = 0 \oplus \rho_3$. In the end, B will share the final value with A.

A general remark is that no protocol does provide security against a simple deduction of the input variables. In this example, if party B has the input 1 and the result is 0, B directly knows that the input of A was 0. On the other hand, each participant cannot extract more information than given by the function and the own input. In the same example, if the final output is 0, party B would not know the input of party A.

[1] In Yao's protocol, this would be done by oblivious transfer [11, 31].

3.3 Secret Sharing Schemes

Secret sharing schemes focus, as the name suggests, on sharing information within a group of participants secretly. Therefore, no other participant can reconstruct the original information without the group's collaboration.

3.3.1 Shamir's Secret Sharing

Shamir proposed a scheme [28] to secretly share information among n participants so that no t or fewer participants can collude to restore the secret. Thus, presenting a scheme with a $(t+1)$-out-of-n-threshold.

Given n participants, a $(t+1)$-out-of-n-threshold setting, we suppose that there is a trusted dealer who wants to share its secret $s \in \mathbb{F}_q$ among the n participants. Here, $\mathbb{F}_q$ is a finite field over q, where q is a prime larger than n. First, the dealer creates a random secret polynomial $f(X)$ of degree t with $f(0) = s$:

$$f(X) = s + f_1 * X + ... + f_t * X^t.$$

Further, the dealer creates shares s_i for each participant i in the following way:

$$s_i = f(i), \text{ for } i = 1, ..., n.$$

To reconstruct the secret s, at least $t+1$ participants must share their shares. The mathematical reasoning behind this is that with $t+1$ shares, one could reconstruct the polynomial with the help of Lagrange interpolation [33]. To be more concrete, one can reconstruct the polynomial, or more directly the secret s the following way:

$$s = f(0) = \sum_{i=1}^{t+1} s_i \prod_{j=1, j\neq i}^{t+1} \frac{-x_j}{x_i - x_j}.$$

We can define the components $(r_1, ..., r_{t+1})$ of the recombination vector $\mathbf{r}$ the following way

$$r_i = \prod_{j=1, j\neq i}^{t+1} \frac{-x_j}{x_i - x_j}.$$

Important to note is that given only t shares, no information is leaked. However, the Lagrange interpolation discussion is beyond this thesis' scope.

3.3.2 Multi-Party computation

Given Shamir's secret sharing scheme, we can now define a protocol to compute a polynomial function over a finite field $\mathbb{F}_q$, where q is a prime. Since we work with a polynomial function, it can be computed using addition and multiplication. Thus, the protocol must support addition and multiplication gates for multi-party computation. To compute the result of the actual function, all parties need to reveal their shares of the last gate in the circuit. The restoring is then equal to the one described by Shamir's secret sharing scheme.

Addition Gates. Given that two parties have each a secret a and b which are shared using the polynomials $f(X) = a + f_1 * X + ... + f_t * X^t$ and $g(X) = b + f_1 * X + ... + f_t * X^t$, by computing the shares $a^{(i)} = f(i)$ and $b^{(i)} = g(i)$. The polynomial $h(X) = f(X) + g(X)$ provides a sharing of the sum $c = a + b$. Thus, we have $c^{(i)} = h(i) = f(i) + g(i) = a^{(i)} + b^{(i)}$. In conclusion, the parties can compute the share result of the addition solely locally.

Multiplication Gates. In contrast to the addition gates, the multiplication gates are more expensive. Again, we consider two secrets a and b shared with the help of the polynomials $f(X)$ and $g(X)$. Each party posses a share $a^{(i)} = f(i)$ and $b^{(i)} = g(i)$. We now wish to compute a share $c^{(i)} = h(i)$ such that $h(0) = c = a \cdot b$. The protocol for the multiplication gate is the following:

- Each party i locally computes $d^{(i)} = a^{(i)} \cdot b^{(i)}$.
- Each party i produces a polynomial $\delta_i(X)$ of degree at most t such that $\delta_i(0) = d^{(i)}$.
- Each party i distributes to party j the value $d_{i,j} = \delta_i(j)$.
- Each party j computes $c^{(j)} = \sum_{i=1}^{n} r_i \cdot d_{i,j}$.

Further examples and information about other multi-party computation protocols based on secret sharing schemes and arithmetic circuits can be found in [5, 31].

3.4 MP-SPDZ

Keller [13] created with Multi-Protocol SPDZ (MP-SPDZ) an implementation of the MPC protocol called SPDZ. The idea is to provide a high-level programming interface based on Python and allow the user to choose which MPC protocol to use. Therefore, the framework can ideally be used for benchmarking different protocols in different security models with the same high-level programming interface. For example, the protocol MASCOT [14] ensures active security, thus, ensuring safety against malicious adversaries. Additionally, MASCOT even ensures safety against a malicious majority, but of course, cannot guarantee an output of the protocol. On the other hand, protocols based on Shamir's secret sharing ensure safety against a malicious but honest majority. In the case of this thesis, the framework should enable easy access to the MPC primitives without the need to implement them.

The general workflow in MP-SPDZ is the following:

1. Create the program in the high-level programming interface based on Python.
2. Compile the program with the help of the provided compiler. The compilation will generate the corresponding byte-code to be run by the selected protocol.
3. Run the program with a selected protocol. Next to the output of the computation, one receives some benchmarking results and even remarks and hints about which configuration might be even faster.

The high-level programming interface provides a set of MPC primitives, such as data types and sub-routines, to work. For example, there are different data types that one knows from other programming languages. Some are listed here:

- `sint`: secret integer. This integer is shared in a secret way among the other parties.
- `cint`: clear integer. This integer is shared among the participants.
- `Array`: arrays exist for different kinds of data types. For example, there might be a sint array. Of course, multi-dimensional arrays are supported as well.
- `sbitvector`: a secret vector of bits. This vector is secretly shared among all.

Next to the primitive data types, there are other handy functions provided as well:

- `@for_range(n)`: creates a runtime loop.
- `@if_e`: provides an if/else on secret data.

There is a extensive documentation about all the different types and functions online[2]. In addition, MP-SPDZ is continually under development and questions or problems are discussed frequently in the GitHub repository[3].

[2]Here: `https://mp-spdz.readthedocs.io/en/latest/Compiler.html`.
[3]Here: `https://github.com/data61/MP-SPDZ`.

4
Implementation

At the beginning of this thesis, in March 2022, only one paper [4] concerning Dilithium's threshold implementation was published. Concretely, the paper examined the round-2 NIST Post-Quantum Competition signature schemes. Since it covered all candidates, it did not go into much detail but only briefly discussed each candidate and what are their concerns. However, the key focus was on how fast a signature could be computed in a multi-party setting. For Dilithium, the estimation was 12 seconds. The estimation was founded by analysing what multi-party computations are needed. Therefore, whereas the article does not provide an implementation, it analyses the different lines from Dilithium and discusses the needed computations. The mentioned computation used the standard primitives in MPC. Those include garbled circuits and secret sharing schemes, as discussed in Chapter 3. Thus, MP-SPDZ was chosen for the implementation.

The considered pseudo-code to implement is shown in Algorithm 7.

Algorithm 7 Dilithium Template: Sign(sk,M) function

1: **function** SIGN(sk,M)
2: $\mathbf{z} := \perp$
3: **while** $\mathbf{z} = \perp$ **do**
4: $\mathbf{y} \leftarrow S^{l}_{\gamma_1 - 1}$
5: $\mathbf{w_1} := \text{HIGHBITS}(A\mathbf{y}, 2\gamma_2)$
6: $c \in B_\tau := \text{H}(M||\mathbf{w_1})$ ▷ only part of the information of $A\mathbf{y}$ is needed
7: $\mathbf{z} := \mathbf{y} + c\mathbf{s_1}$
▷ checks to ensure completeness and no secret information gets disclosed
8: **if** $||\mathbf{z}||_\infty \geq \gamma_1 - \beta \vee ||\text{LOWBITS}(A\mathbf{y} - c\mathbf{s_2}, 2\gamma_2)||_\infty \geq \gamma_2 - \beta$ **then**
9: $\mathbf{z} := \perp$ ▷ signature candidate gets rejected and computation starts again
10: **return** $\sigma = (\mathbf{z}, c)$

The goal of the implementation was to translate the lines of pseudo-code into the high-level programming interface of MP-SPDZ. In addition to the relatively easy handling, MP-SPDZ supports many different protocols and, thus, different security models. In the *honest majority* setting, computing is done by only using secret sharing schemes. In contrary, the *corrupt majority* uses computations based on

public-key cryptography [13]. Whereas secret sharing schemes are information-theoretically secure and thus, quantum-resistant [31], public-key cryptography is generally not quantum resistant. In conclusion, to have a quantum-resistant implementation of Dilithium, only the *honest majority* setting could be used.

Section 4.1 focuses on the set-up of MP-SPDZ as well as showing a first example of how MP-SPDZ can be used. The preliminaries to develop Dilithium's threshold implementation are discussed in Section 4.2. Section 4.3 discusses the implementation of the signature generation, as well as the related problems. Lastly, the implementation of the verification algorithm is discussed in Section 4.4.

4.1 Set-Up and First Steps with MP-SPDZ

Set-up. The set-up of MP-SPDZ was straightforward:

1. Clone the Git repository: `git clone https://github.com/data61/MP-SPDZ.git`.
2. Download all submodules for the compilation: `git submodules update --init`.
3. Install all the needed dependencies: `make tldr`.
4. Compile all needed libraries and executables to run MP-SPDZ: `make`.

Implementing the Millionaires Problem in MP-SPDZ. First, the famous *Millionaires Problem* was implemented to get to know more about MP-SPDZ. It turns out that, although the problem is complex, the implementation in MP-SPDZ is trivial; as shown by the following code:

```
# millionaires.mpc

num0 = sint.get_input_from(0) # read secret input from millionaire 0
num1 = sint.get_input_from(1) # read secret input from millionaire 1

richer = (num0 < num1).if_else(1, 0) # Decide which millionaire is richer

print_ln('Millionaire %s has more money', richer.reveal()) # Reveal who is reacher
```

Important note: MP-SPDZ follows a strict directory hierarchy.

- `./Scripts`: containing all needed scripts and all different protocol binaries.
- `./Programs/Source`: contains the source code of the high-level programs.
- `./Player-Data`: contains the player-related input and output data.

Given that the source code is stored at the correct location, the compilation of this high-level program is done by issuing the following command:

```
$ ./compile.py millionaires
```

This will generate a byte-code, which any protocol supported by MP-SPDZ can run. The program will first read the secret input from each player. In this case, the secret input will be the respective millionaire's wealth. Suppose millionaire zero's wealth is 1.000.000 and millionaire one's wealth is 1.000.100. Then, the final run with the MASCOT protocol and with two participants of the aforementioned `millionaires.mpc` looks the following:

```
$ ./Scripts/mascot.sh millionaires
Running /PATH/MP-SPDZ/Scripts/../mascot-party.x 0 \
   millionaires -pn 15641 -h localhost -N 2
Running /PATH/mp-spdz-0.3.0/Scripts/../mascot-party.x 1 \
   millionaires -pn 15641 -h localhost -N 2
Millionaire 1 has more money
Significant amount of unused triples of SPDZ gfp. For more accurate benchmarks,\
   consider reducing the batch size with -b.
Significant amount of unused bits of SPDZ gfp. For more accurate benchmarks,\
   consider reducing the batch size with -b.
The following timing is inclusive preprocessing.
Time = 1.23661 seconds
Data sent = 65.64 MB in ~92 rounds (party 0)
Global data sent = 131.28 MB (all parties)
This program might benefit from some protocol options.
Consider adding the following at the beginning of 'millionaires.mpc':
   program.use_edabit(True)
```

First, the script `mascot.sh` will, according to the configuration, start the correct amount of parties before the compiled byte-code gets run. The parties are indicated in the output above with `Running`, in this case there are only two participants. Together with the output of the actual program, MP-SPDZ provides extensive information for each run. This information includes how long a run took and how much data was sent among the parties. Additionally, hints and tips to increase efficiency are provided with the output.

One advantage of MP-SPDZ is that the same byte-code can be run with a different protocol without recompiling. For example, the output with a protocol based on Shamir's secret sharing in a 3-out-of-3 setting looks the following:

```
$ ./Scripts/shamir.sh millionaires
Running /PATH/MP-SPDZ/Scripts/../shamir-party.x 0 millionaires \
   -pn 11404 -h localhost -N 3
Running /PATH/MP-SPDZ/Scripts/../shamir-party.x 1 millionaires \
   -pn 11404 -h localhost -N 3
Running /PATH/MP-SPDZ/Scripts/../shamir-party.x 2 millionaires \
   -pn 11404 -h localhost -N 3
Using security parameter 40
Millionaire 1 has more money
Significant amount of unused bits of Shamir gfp. For more accurate benchmarks,\
   consider reducing the batch size with -b.
The following timing is inclusive preprocessing.
Time = 0.0268549 seconds
Data sent = 0.643936 MB in ~13 rounds (party 0)
Global data sent = 1.61168 MB (all parties)
This program might benefit from some protocol options.
Consider adding the following at the beginning of 'millionaires.mpc':
      program.use_edabit(True)
```

There are some noteworthy differences between the two runs. First, one can see that the Shamir protocol needs three participants, whereas MASCOT works in the two-party case. Secondly, since the security assumptions of the Shamir protocol are less strict, less interaction between the parties is needed; thus, fewer data must be sent. Additionally, the Shamir Protocol uses less time for computing the same result.

4.2 Threshold Dilithium: Preliminaries

MP-SPDZ configuration. At one point during the implementation phase the following error was produced at the start of threshold Dilithium:

```
Not compiled for 1088-bit primes
Compile with -DGFP_MOD_SZ=17
```

The error was due to the large bit-vectors which are either sent to the hash function or receive the hash output. Thus, the suggested config flag needed to be set. Since a recompilation of MP-SPDZ is needed for the adaptation of the given configuration flag for the compiler, the procedure was as straightforward as the whole setup of MP-SPDZ. With the changed MP-SPDZ environment, the same program was run again. After some seconds, the program failed with a segmentation fault. From the output, it was not clear where the bug was located. Since a complete understanding of how MP-SPDZ works were out of this thesis' scope, further debugging in this direction was aborted. Nevertheless, an issue on GitHub was opened[1]. The issue was resolved after some days thanks to the main developer behind MP-SPDZ, Marcel Keller.

Architecture. The architecture of Dilithium's threshold implementation is mainly taken from Dilithium's reference C implementation. There are some deviations due to simplicity and of course general differences between the programming languages. This resulted in many small functions, such as adding or multiplying two polynomials. A great benefit of this approach is that each function could be tested isolated from the rest. This is because running only a small subset of all functions takes less time than running the template Dilithium algorithm. In addition, all results could be tested against the reference implementation because a simple program, including the corresponding libraries, could be programmed without much work. Of course, toy values were chosen for these tests. For example, a polynomial would have the same coefficient regardless of the degree.

System requirements. At some point in the implementation, the compilation of Dilithium took close to one hour to complete. This enormous long compilation time was due to the available RAM on the VM. There was in total 15GiB RAM available on the server, but the compilation needs around 14GiB. Thus, many things were swapped out to the slow disk. In conclusion, to compile Dilithium's threshold implementation, around 14GiB of free RAM is required.

Dilithium parameters and NTT representation. First, Dilithium's security parameters[2] were fixed to $L = 5, K = 4, \gamma_1 = \text{G1} = 523776, \gamma_2 = \text{G2} = 261888,\ \text{Q} = 8380417, \tau = 49, \beta = 275, N = 256$. During the implementation phase, the parameters were changed to $L = 1, K = 1, \gamma_1 = \text{G1} = 524288, \gamma_2 = \text{G2} = 261888,\ \text{Q} = 8380417, \tau = 49, \beta = 196, N = 256$. The change was mainly due to the long compilation and running time. The long compilation and running time showed up after realising that Dilithium is not using coefficients from an integer ring but from a polynomial ring. Implementation wise, for exmaple the vector $\mathbf{y}$ must thus be a two-dimensional array of size [l][n] and not a one-dimensional array of size [l]. This erorr introduced some small and some severe changes. For example, the random sampling of the vector $\mathbf{y}$ could be adapted by simply adding a second loop. The participants need to randomly sample $L \cdot n = 1024$ integers in $\mathbb{Z}_q$, rather than only L. A more complicated problem is the matrix-vector multiplications such as $A\mathbf{y}$. Multiplication and addition in $\mathbb{Z}_q$ can easily be implemented by doing the multiplication or the addition in $\mathbb{Z}$ and afterwards, using the modulo operator. The addition in R_q works similarly: adding up the two polynomials, then reducing by q. For example: given the following ring $R_q = \mathbb{Z}_7/F[X]$, where $F[X] = x^2 - 1$, the addition of $4x^2 + x$ and $1x^2 + 6x$ results in

[1] Can be found here: https://github.com/data61/MP-SPDZ/issues/619.

[2] Throughout the thesis, the dimensions K and k respectively L and l are used interchangeable.

$4x^2 + 1x^2 + x + 6x = 5x^2 + 7x = 5x^2$. However, multiplication is more complicated. By multiplying a polynomial with a polynomial, there will be higher degrees. Thus, the reduction process consists of two phases: degree reduction and coefficient reduction. Consulting Dilithium's reference implementation and the specification [7], all corresponding vectors and matrices are transformed using the number theoretic transform (NTT) before multiplying them. NTT is a discrete Fourier transform (DFT) restricted to a finite field. In addition, the inverse NTT transformation uses Montgomery reductions instead of the usage of the modulo operator '%' in C. Further discussion about those two topics is out of the scope of this thesis. Important to know is that in the NTT representation, the multiplication is more efficient. For the implementation, the code concerning all operations regarding NTT from Dilithium's C reference implementation was ported to MP-SPDZ. Around 300 lines had to be added to support the NTT and Montgomery representation transformation. This included all multiplications needed by Dilithium. A first remark was that the transformation to the NTT representation was time-consuming. A test implementation with the following steps was programmed:

1. Create two polynomials of degree 256.
2. Transform both polynomials into the NTT representation.
3. Multiply both polynomials.
4. Transform back from the NTT representation.

The compilation time, with dimensions $K = 4, L = 5$, was around 30 minutes, and the actual running time was around 45 minutes with the MASCOT protocol[3]. Dilithium needs several such multiplications and additions as well. Since the multiplications and additions are mixed, several transformations to and from NTT are needed. Thus, resulting in a significant workload only for the number transformations. According to a mail from Gregor Seiler, a developer of Dilithium, the parameters L and K, corresponding to the dimensions of the matrices and vectors, can be decreased. On the other hand, changes in the degree N of the polynomials would require more time and effort because the required parameters for the NTT transformation are hard-coded in the reference implementation. A change in the degree N of the polynomials would thus require recalculating those parameters. Since understanding the NTT transformation in full was beyond the scope of the thesis, this was omitted. Therefore, the degree of the polynomials $N = 256$ stayed. In conclusion, for further development, the parameters L and K were set to 1, working mainly only with polynomials rather than matrices and vectors. All these changes reduced the compilation and running time and made debugging realistic again.

Key generation. This thesis focused mainly on the procedure described in [4]. There, they assumed that the public and private keys were already shared; thus, assuming a trusted dealer. The trusted dealer owns the secret key and secretly shares it at the start of a message signing. This was equally implemented: one participant will read the private and public keys from a file and secretly share the values among all the participants. The keys were generated using Dilithium's single-party C implementation. To further decrease the compilation and running time in the threshold implementation, the values of the keys are already stored in the NTT representation. Therefore, the parameters of the keys were not stored in the standard integer format but directly in the NTT representation. The whole `KeyGen()` subroutine could also be implemented in MP-SPDZ but would, as already mentioned, increase the workload.

4.3 Threshold Dilithium: Signature Generation

The focus of this section is going through line-by-line of Algorithm 7 and explaining the encountered barriers and limitations.

[3] Remark: the code was not yet optimised at this time.

Line 4: random sampling of vector y. Starting from line 4, this can easily be implemented by generating 20 bits at random, testing if the result is less than $2 \cdot \gamma_1 - 2$, and then subtracting γ_1 from the generated number [4]. Therefore, the implementation was quite easy:

```
# random sampling of y

# Defining the threshold
thresh = MemValue(2 * G1 - 2)

# Loop through dimension L
@for_range(L)
def _(i):
   # Loop through the degree of the polynomials
   @for_range(N)
   def r(j):
      @do_while
      def _():
         # Define a return value to stay or break the loop
         ret = MemValue(0)
         # Sample a random number of 20-bits size
         rand = sint.get_random_int(bits=20)

         # Check if the generated number is less than a certain threshold
         @if_e((rand < thresh).reveal())
         def _():
            # assign the final value
            y[i][j] = rand - G1
            # set the return value in order to break the @do_while loop
            ret.write(0)
         @else_
         def _():
            # Stay in the loop
            ret.write(1)

      return ret
```

Line 5: calculating the HighBits. The computation of $\mathbf{w_1} := \texttt{HIGHBITS}(A\mathbf{y}, 2\gamma_2)$ on line 5, is split into two parts. As already mentioned, multiplications restricted to a polynomial ring are not straightforward. Thus, the coefficients of A and $\mathbf{y}$ are in NTT representation. Whereas MP-SPDZ would directly support the matrix-vector multiplication, there was a lack of such if the result have to lay in a polynomial ring. Therefore, the function for a matrix-vector multiplication needed to be translated from the C reference implementation of Dilithium. In the C implementation of Dilithium, all the functions relying on NTT transformations and computations heavily use C's casting functionality. For example, the following lines are part of the inverse NTT to Montgomery transformation in C:

```
int32_t montgomery_reduce(int64_t a) {
   int32_t t;

   t = (int64_t)(int32_t)a*QINV;
   t = (a - (int64_t)t*Q) >> 32;
   return t;
}
```

Since casting is not natively supported in MP-SPDZ, it has to be programmed by hand. By default, the integer size in MP-SPDZ is 64-bit. Thus, casting to 64-bit is not necessary. Casting to 32-bit integers, on the other hand, must be implemented. The following code was used in MP-SPDZ to resolve this problem:

```
def sintto32bit(a):
```

```
    a += 2**31
    a = a.mod2m(32)
    a -= 2**31
    return a
```

For the second part of line 5, `HighBits(`$A\mathbf{y}, 2\gamma_2$`)`, the `HighBits` function needed to be implemented. This consisted of implementing the `Decompose`$_q$`(`r, α`)` function, as described in Algorithm 4. Here, the following problem arose: MP-SPDZ only supports modulo a power of two. Since q and α are not a power of two, this could not be implemented straight-forward. The reference implementation in C of Dilithium does not use the modulo operation but uses bit-shifts, allowing its implementation with MP-SPDZ:

```
def reduce32(a):
    a = sintto32bit(a)
    t = a + 2**22
    t = t.right_shift(23)
    t = a - t*Q
    return t
```

With this, the implementation of the `HighBits` could easily be translated from Dilithium's C implementation.

Line 6: hashing to the ball. Line 6 consists again of two different parts. First, a cryptographic hash function, H, must output a hash given the message M concatenated with the previously computed high-bits vector $\mathbf{w_1}$. Secondly, this hash output will be mapped to a vector of dimension 256 with τ coefficients either $\{-1, +1\}$. The main challenge is that the hashing must be done in secret. Therefore, the input must be secret variables; thus, the computation must produce the 'correct' output given the shared input. 'Correct' means that the hash created with secret values must be equal to the hash if the input was known. As discussed in Section 2.4, Dilithium uses SHAKE256 for the hashing. MP-SPDZ only supports SHA3-256 but not SHAKE256 out of the box. Since SHA3-256 and SHAKE256 are part of the SHA3-Suite, implementing SHAKE256 was thought to be straightforward. The functionality and the specification [9] had to be read and understood thoroughly to see a simple difference. In the end, the implementation of a simplified version was straightforward since one of the differences between SHA3-256 and SHAKE256 is the different padding scheme. Therefore, the SHA3-256 function was changed the following way to compute a valid SHAKE256 hash:

- Instead of using `0x06` as the domain separation byte, `0x1f` was used.
- Instead of producing a 256-bit output, a 1088-bit output is produced.

However, the implementation of SHAKE256 was not complete. In summary, the following functionalities were missing:

- There is no such thing as an `absorb` function. Thus, the function can only be called once with a string smaller than 1088 bits[4].
- There is no such thing as an `squeeze` function. Hence, given an input, one could not produce infinitely many random bits.

In conclusion, this implementation returns once a correct 1088-bit long SHAKE256 string. At first, this was thought to be enough to proceed further with the implementation.

As discussed in Section 4.2, the correct configuration flag `-DGFP_MOD_SZ=17` must be set in order to properly run. Furthermore, at first, this was not possible due to a bug in MP-SPDZ. While waiting for

[4]This is the maximum process rate before a permutation is needed [9].

the bug fix to be published, other solutions to this problem were tried. It was known that the problem was caused by the bit-vector used for the hashing. Recall that the hashing looks the following: `H(M||w1)` and responds with a bit-vector. Exactly this bit-vector, the result of the hashing, was 1088-bit long. Due to the need to have a long output from the hashing function, this could not be decreased.

An idea to resolve this problem was to use the `digest()` method provided by MP-SPDZ which exists on `cint` data types. Of course, this solution would drift further away from the Dilithium specification since the input of the hashing would not be shared securely, and the cryptographic hash function would differ. According to the documentation, the digest would be based on libsodium[5]. However, experiments with the digest method showed that it is not revealing the same digest although the same input was given. Since the task for the hash function in Dilithium is to provide random output given a defined input, this approach was not usable.

Another approach would be to use Python's SHAKE256 algorithm. There, the input to the SHAKE256 function would be in clear-text, requiring stronger security assumptions than the original Dilithium [4]. Of course, this was very much not wished, since this would affect the threshold system. But in order to proceed, further time was invested to work on this solution. Since there is a SHAKE256 implementation in Python provided by PyCryptodome, the implementation would be straightforward. However, a problem arises: there was no method to transform an MP-SPDZ data type, such as `cint`, to a python data type, such as `int` or even binary. Therefore, this resulted in the following compilation error:

```
TypeError: Object type <class 'Compiler.types.cint'> cannot be passed to C code.
```

Thus, this approach was rejected as well.

The last idea to solve this problem was to implement SHAKE256 correctly. In more detail, this means that an `absorb(input)` and an `squeeze()` function must be implemented. Since the input bit-vector was, with the current parameters $L = K = 1$, also 1032-bit long, a correct implementation of the `absorb(input)` was also wished. This way, the input bit-vector, created from $M||w_1$, could be split into smaller values and absorbed by the SHAKE256 function. For this, Dilithium's reference implementation was consulted again. After some tests, it became clear that the current implementation in MP-SPDZ could not easily be adapted to absorb multiple times and give the correct output. For example, the output of the `squeeze()` function after the absorption of the message 'ab' is not equal to the output after absorbing 'a' first and then absorbing 'b'. According to the specification [9], the `absorb(input)` subroutine waits until either the maximum input, in this case, 1088 bits, is reached, or no more input is coming. If no more input is coming, the domain separation byte and the padding is added, and the construction is ready to produce the output with the `squeeze()` subroutine. On the other hand, if the maximum input is overshot, then the permutation function f is called to allow more input to come. See Figure 4.1 for a better overview of how the underlying sponge construction works. In the current MP-SPDZ implementation of SHAKE256, the separation byte and the padding are directly added after each absorption. In addition, the permutation function is only called in the `squeeze()` subroutine. Thus, the absorption of first 'a' and then 'b' would lead to the overwriting of the input 'a' by the input 'b'. In conclusion, the current absorption function is not constructed to handle several calls.

The SHAKE256 function would need to be separated into several functions, namely:

- `init()`: Initialises all the needed global variables.
- `absorb(input)`: Absorbs input until the maximum rate is reached; Upon reaching the maximum rate, run the permutation function and continue filling up.
- `finalize()`: Add the domain separation byte and fill up the rest with the padding information.
- `squeeze()`: Absorb a full block, 1088-bits, from the SHAKE256 function.

[5]More information can be found here: `https://doc.libsodium.org/`.

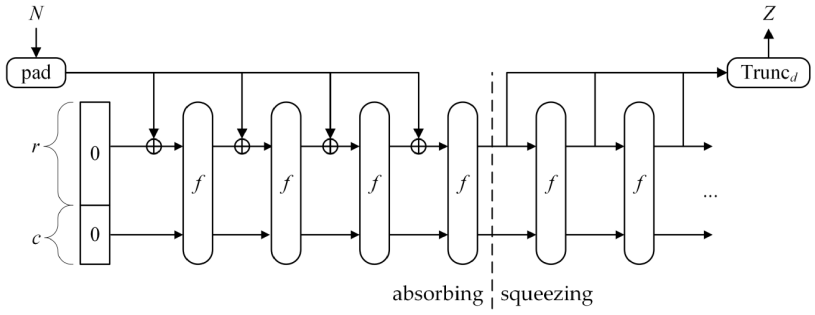

Figure 4.1: The sponge construction used in the SHAKE256 algorithm. N is the input, f is the permutation function, pad denotes the padding, and Z is the output of the hash function [9].

The `squeeze()` and the `init()` functions would be implemented easily since most of the old code could have been reused. The other two functions, namely the `absorb(input)` and the `finalize()` functions, would have been more exhaustive. The main reason was that the reference implementation of SHAKE256 is substantially different from the existing SHA3-256 implementation in MP-SPDZ. SHA3-256 and SHAKE256 are comparable because both use the same underlying permutation function as a foundation. The main difference between MP-SPDZ and the reference implementation was how the data type was built-up before the permutation function was called. In SHA-3's reference implementation, the input to the permutation function is a simple array of size 25 of 64-bit integers; in MP-SPDZ the input was a 3-dimensional bit-vector array of sizes 5, 5, and 64, respectively. In both implementations, the total size is 1600 bits long, but the relationship between the indices is not straightforward. Luckily, after all the mentioned considerations above, the bug fix for the segmentation fault was published; thus, the further implementation of SHAKE256 was not required at this point anymore.

Line 7: compute a signature candidate. On line 7, a signature candidate is computed. As mentioned in Subsection 2.4.2, Dilithium is using a rejection sampling. Thus, not any computed signature will be accepted by the `Sign(sk,M)` abortion rules. At first, the signature has always been rejected. Therefore, one of the two infinity norms must have been too large. After some days of debugging, including checking all functions in the code, the mistake was found. In Dilithium's C implementation, they start the rejection sampling with the following line:

```
z = y;
```

Thus, all the calculations from the pseudo-code corresponding to `y` are done with `z` in the C implementation. All besides the one calculating the signature candidate $\mathbf{z} = \mathbf{y} + c\mathbf{s_1}$. Because of this confusion, the calculations in the MP-SPDZ implementation were done with `y` all the time. However, in difference from the C implementation, the MP-SPDZ implementation transforms `y` into the NTT representation at the beginning. Since the computation of $\mathbf{z}$ requires the standard representation, $\mathbf{z}$ was consistently miscalculated. First, it was thought that the inverse NTT to Montgomery representation could be used, meaning that before computing $\mathbf{z}$ `y` would be transformed into the inverse NTT to Montgomery representation. The results were, however, still incorrect. As soon as $\mathbf{z}$ was computed with the correct $\mathbf{y}$, the infinity norm of the signature candidate was in the correct range. Therefore, the rejection sampling accepted the signature candidate, and thus, the algorithm stopped.

Line 8: rejection sampling. As mentioned just above, there was an effort in debugging to get the infinity norm of $\mathbf{z}$ and $\texttt{LowBits}(A\mathbf{y} - c\mathbf{s_2}, 2\gamma_2)$ in the correct range. At first, a naive approach, by trying to check if the actual or the negated value was above respective below the given threshold, was used. Consulting Dilithium's reference implementation, their more sophisticated approach was adopted to MP-SPDZ:

```
# Check the infinity norm of a polynomial
def poly_chknorm(a, bound):
   sb32 = sbits.get_type(32)
   ret_chk = MemValue(0)

   @if_(bound > (Q-1)/8)
   def _():
      ret_chk.write(1)

   @for_range(N)
   def _(i):
      t = a[i].right_shift(31)
      bit_t = sb32(t)
      bit_a = sb32(2*a[i])
      temp = sint.bit_compose(bit_t & bit_a)
      t = sintto32bit(a[i] - temp)

      @if_((t >= bound).reveal())
      def _():
         ret_chk.write(1)

return ret_chk.reveal()
```

Similar to the whole C implementation of Dilithium, this function also heavily uses bit shifts and logic operators. To implement them in MP-SPDZ, the secret integers need to be transformed into secret bit-vectors, before the logic operators can be used.

4.4 Threshold Dilithium: Verification Algorithm

Since the implemented threshold Dilithium is not compatible with the reference implementation, the generated signatures cannot be verified by the standardised implementation. Furthermore, at the time of writing this thesis, there was no peer-reviewed implementation of the simplified Dilithium algorithm. Thus, a verification algorithm was programmed in MP-SPDZ as well. Therefore, the generated signatures could at least be verified against one verification algorithm. The implementation was done in MP-SPDZ to ensure compatibility since many of the same functions were needed. Otherwise, all the needed functions would need to be translated into another programming language and thus, the probability of long debugging would have been very large. On the other hand, MP-SPDZ does not provide a protocol to run the verification algorithm in a single-party setting. Since the idea is to run the signature computation in multi-party and afterwards, everyone could verify this signature on its own, having a single-party verification algorithm would be better. Since all needed functions could be copied from the threshold implementation of Dilithium, the implementation of the verification algorithm was straightforward. The only bug concerned the representation of the public vector $\mathbf{t}$. Whereas the own implementation of the key generation algorithm created the $\mathbf{t}$-vector in the NTT representation, the vector gathered from the C implementation was in the Montgomery representation. Hence, in the first version of the verification algorithm, the expected format of $\mathbf{t}$ was NTT. Before further calculations, the $\mathbf{t}$ vector must be transformed to the NTT representation. After implementing this change, the verification algorithm verified the previously generated signature.

5 Evaluation

The compilation of the Dilithium MPC algorithm with simplified parameters, $k = l = 1$, was around 15 minutes, and the resulting byte code is 144MiB large. The compilation time of the verification algorithm was around 12 minutes and the resulting byte code is 112MiB large. A random 8-bit message was signed and verified for benchmarking the different methods. Note that all the parties were run on the same virtual machine and no party was actually misbehaving. Thus, the protocol only had to prepare for active or honest-but-curious adversaries. The following methods were tested:

- **Honest majority**:
  - **Honest-but-curious adversaries:**
    - **S(3,3)**: 3-out-of-3 Shamir-based,
    - **S(2,3)**: 2-out-of-3 Shamir-based,
    - **S(5,5)**: 5-out-of-5 Shamir-based,
    - **S(3,5)**: 3-out-of-5 Shamir-based.
  - **Active adversaries:**
    - **MS(3,3)**: 3-out-of-3 Shamir-based.
- **Corrupted majority with active adversaries**:
  - **MA(3,3)**: 3-out-of-3 MASCOT.

Recap from Section 3.1: in the *honest majority* security models, less than half of the parties are adversaries. The adversaries might be honest-but-curious; thus, following the protocol but trying to extract as much information as possible from the other participants. On the other hand, the adversaries might be active; thus, not following the protocol. In the *corrupted majority with active adversaries* security model, the protocol cannot guarantee output, *i.e.,* the protocol might abort.

With the honest-but-curious Shamir-based protocol, 300 signatures were computed. The MASCOT protocol was run 50 times and the Shamir-based protocol ensuring security against malicious adversaries in an honest majority security model (*malicious Shamir-based protocl*) was run only 35 times. This is because computing one signature with the MASCOT or the malicious Shamir-based protocol took significantly

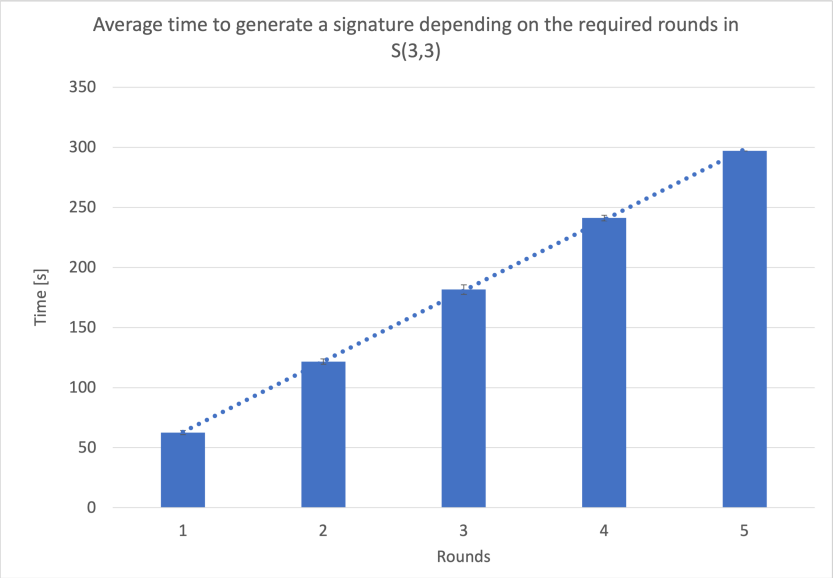

Figure 5.1: Average time needed to produce a valid signature in S(3,3), separated by the number of required rejection-sampling rounds, including a linear trendline.

longer than computing one signature with the honest-but-curious Shamir-based protocol. The benchmarks ran on a virtual machine with 45GiB of RAM and an AMD EPYC-Rome 2.2GHz CPU with 8 threads. The following measurements were taken for each computation of a signature of each method:

- Time, in seconds, required to produce a signature.
- Number of rounds needed to compute a signature.

Out of the given measurements, the following values were obtained for each method:

- Average time, in seconds, required to produce a signature.
- Average number of rounds needed to compute a signature.

In addition, the amount of data sent by one party as well as the number of MPC-rounds needed for the computation was measured for a one-round rejection sampling of Dilithium.

In Figure 5.1, the average amount of time required to produce a signature in S(3,3) is shown. The plotted linear trendline shows that each round takes, on average, the same amount of time. Since there are some computations outside of the rejection sampling, the trendline will not go through the origin.

Figure 5.2 shows how many rounds were required for each of the 300 computed signatures. Out of the plotted exponential trendline, it can be deduced that the probability of having more rounds decreases exponentially. Of course, this figure could be produced with a single-party implementation as well.

The computed average timings for each method are shown in Figure 5.3. Noteworthy, the figure uses a logarithmic time axis. Thus, the MASCOT and the malicious Shamir-based protocol require about an order of magnitude longer than the honest-but-curious Shamir-based protocols. Whereas the S(3,3) and the S(2,3) protocols are the fastest with around 87 seconds, the malicious Shamir-based protocol is the slowest using around nearly 1.5 hours. As a reference, Dilithium's C implementation[1] computes a signature in

[1] Run with the NIST Security Level 5, which is defined in [8].

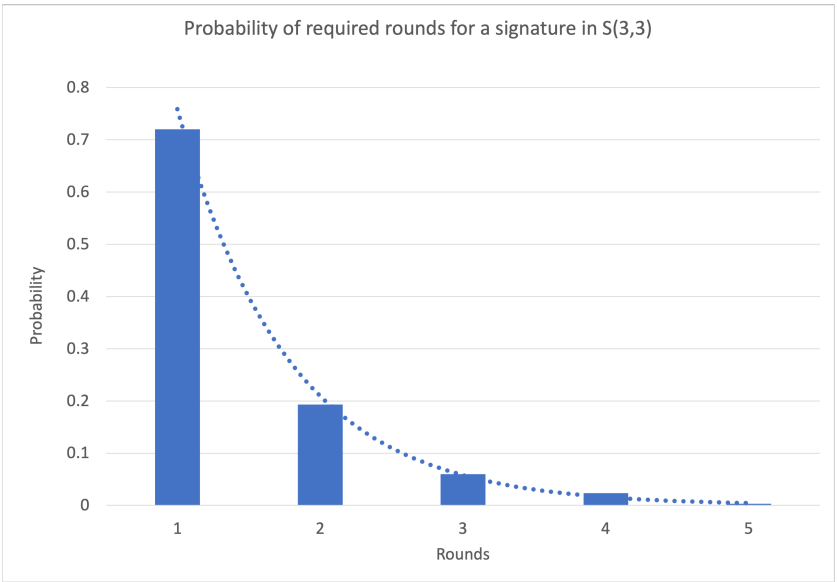

Figure 5.2: Probability of required number of rejection-sampling rounds to produce a valid signature with S(3,3), including an exponential trendline.

around 1ms and verifies it in less than 1ms on the same machine as before. The shown variance in the figure should indicate that due to the rejection sampling, the running time required to compute a signature is not fixed.

On average, a valid signature needed 1.4 rejection-sampling rounds. In difference to the required time, this value is not depending on the used method. This is because the amount of required rounds depends not on the method but solely on the code itself. Thus, this value would as well have been computed by a single-party implementation.

The amount of data sent by one party in the process of creating a valid signature in a single rejection-sampling round is shown in Figure 5.4. Given that a signature, saved as a simple text file, is around 4KiB, the amount of data sent among the parties is immense independent of the used method. As a further comparison, the amount of data sent in the Millionaires Problem with the S(3,3) protocol, as described in Section 4.1, is shown as well. In addition, this figure uses a logarithmic scale for the amount of data. Thus, the MASCOT and the malicious Shamir-based protocol use again one to two orders of magnitudes more data than the honest-but-curious Shamir-based protocols. The number of MPC rounds required by a single party is shown in Figure 5.5. Again, as a comparison, the number of MPC rounds for the Millionaires Problem in S(3,3) is shown. Furthermore, the y-axis is again on a logarithmic scale. Thus, similar to the previous experiments, the MASCOT and the malicious Shamir-based protocol are using more than an order of magnitude more rounds than the honest-but-curious Shamir-based methods. In addition, with the comparison to the Millionaires Problem, one can estimate the complexity of the Dilithium implementation.

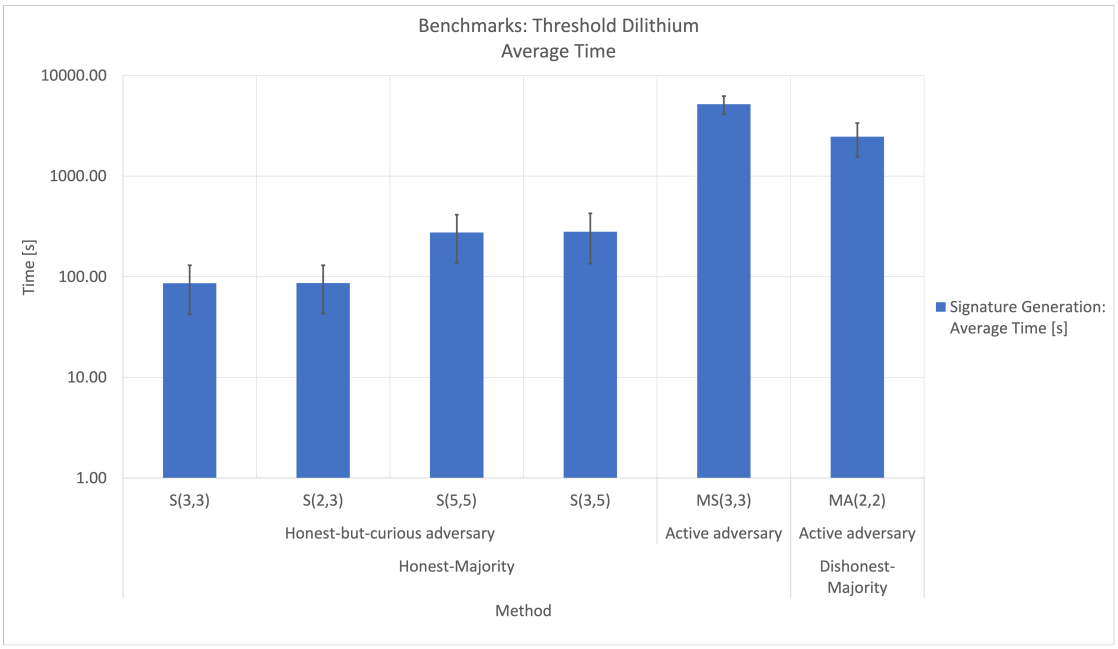

Figure 5.3: Required average time in seconds to compute a signature separated by the used method. The time axis is on a logarithmic scale.

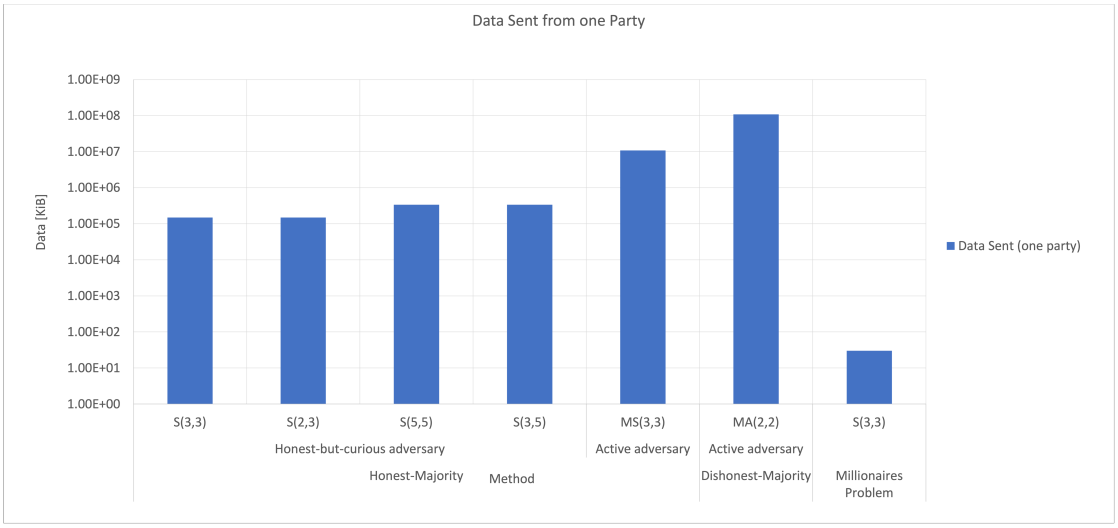

Figure 5.4: Data, in MiB, sent from one party to others in logarithmic scale separated by the method used. Only one round in the rejection sampling was needed to create a valid signature. As a reference, the Millionaires Problem is listed in the last column.

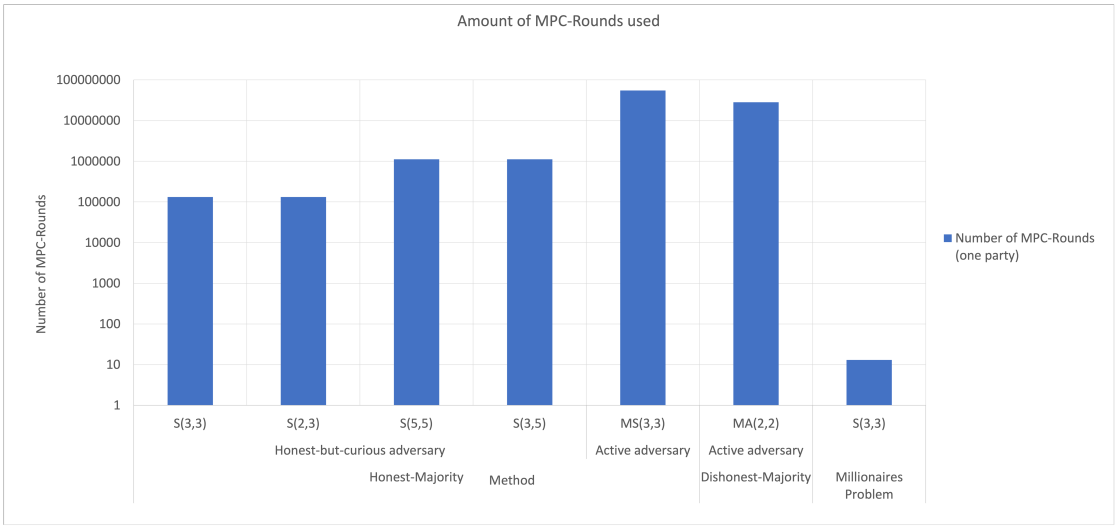

Figure 5.5: Used amount of MPC-rounds required by one party separated by the method used. Only one round in the rejection sampling was needed to create a valid signature. As a reference, the Millionaires Problem is listed in the last column.

6 Conclusion

MPC implementations already exist for other protocols, such as RSA and Schnorr, but there is a lack of such an implementation of a quantum-resistant signature scheme. Hence, this and the ongoing progress in building quantum computers motivated this thesis. The aim was to create an MPC implementation of Dilithium, a candidate to be standardised for the NIST post-quantum competition. This thesis not only discussed the theoretical foundations of lattice-based cryptography and secure multi-party computation needed to accomplish the implementation, but also provided the first version of an MPC implementation of Dilithium. The provided version allows signing an 8-bit message with Dilithium's template algorithm and toy security parameters (*i.e.,* the dimensions $k = l = 1$). In an *honest majority* setting with passive adversaries and three participants, the algorithm needed, on average, around 87 seconds to generate a signature. On average, the verification algorithm in the equal setting needed around 44 seconds. The implementation uses different MP-SPDZ primitives and computations. Thus, it serves as a model for using MP-SPDZ in other projects. Furthermore, this implementation might help to show the limitations of MP-SPDZ.

While this implementation is not yet usable in the real world, it provides a solid groundwork on which further research and development can be elaborated. There are several possible paths for future development in realising an MPC implementation of a quantum-resistant signature scheme. The first option would be to enhance the provided implementation by this thesis. Currently, there are three main shortcomings:

1. The keys are generated by one party and then shared. Therefore, the trust remains in one person.
2. Given the estimation of 12 seconds in [4] suggests that the average running time of 87 seconds is too long, even though the recommended security parameters of Dilithium specified in [7] are not met, and the loosest security model is used.
3. The generated signatures are verified by a self-made verification algorithm. Thus, the correctness is not verifiable by the official implementation.

Problem one is rooted in the missing `KeyGen()` subroutine in MP-SPDZ, although it could be implemented straightforward. Therefore, the secret key would be generated and thus shared by all participants

securely instead of belonging to one person. Hence, with this change, trust distribution would be fully implemented.

Verifying a generated signature with the reference implementation of Dilithium is impossible due to the incompatibility of the signatures. Thus, the generated signatures are verified by a self-made verification algorithm. Furthermore, this includes the risk of a faulty implementation resulting in security vulnerabilities. The implementation needs to be developed to the optimised version of Dilithium to overcome the incompatibility problem. This is because the optimised version uses different values for the public and secret keys and the signature. Nevertheless, developing the optimised version of Dilithium could benefit from reusing some implemented functionalities, such as the NTT representation. Furthermore, the SHAKE256 algorithm must be implemented in MP-SPDZ as defined in the specifications. This change would enable the equal output of the `squeeze` subroutine and allow the signing of a large message. In addition, the adaptation of the security-relevant parameters of Dilithium, such as $k, l, \tau, \gamma_1, \gamma_2$ would be straightforward. Thus, providing full compatibility with Dilithium's reference implementation.

However, shortcoming two, the long running time, would be harder to overcome. It is assumed that the provided code from the thesis can be optimised, but the resulting running time might still be far away from the expected 12 seconds. In addition, further development and optimisations regarding MP-SPDZ might further decrease the running time. The second option to go from here would be to use a more sophisticated threshold implementation of Dilithium. Therefore, not relying on the basic MPC primitives, such as secret sharing schemes and garbled circuits, but instead directly using the mathematical foundation of Dilithium to achieve a threshold signature. Of course, this will require a more thorough understanding of the mathematical foundations of Dilithium as well as a clear assessment of the security assumptions in the threshold setting. While this approach might introduce small changes in the signing procedure of the reference implementation, it is expected to have a relatively small running time.

Bibliography

[1] Paul Benioff. The computer as a physical system: A microscopic quantum mechanical hamiltonian model of computers as represented by turing machines. *Journal of Statistical Physics*, 22:563–591, 1980.

[2] Charles H. Bennett, François Bessette, Gilles Brassard, Louis Salvail, and John A. Smolin. Experimental quantum cryptography. *J. Cryptol.*, 5(1):3–28, 1992.

[3] Jacob D. Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. *Nat.*, 549(7671):195–202, 2017.

[4] Daniele Cozzo and Nigel P. Smart. Sharing the LUOV: threshold post-quantum signatures. In Martin Albrecht, editor, *Cryptography and Coding - 17th IMA International Conference, IMACC 2019, Oxford, UK, December 16-18, 2019, Proceedings*, volume 11929 of *Lecture Notes in Computer Science*, pages 128–153. Springer, 2019.

[5] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015.

[6] David Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London Series A*, 400(1818):97–117, 1985.

[7] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: A lattice-based digital signature scheme. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(1):238–268, 2018.

[8] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: Algorithm specifications and supporting documentation (version 3.1). Technical report, 2021. `https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf`.

[9] Morris Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions. Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, 2015.

[10] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 823–852. Springer, 2020.

[11] David Evans, Vladimir Kolesnikov, and Mike Rosulek. A pragmatic introduction to secure multi-party computation. *Found. Trends Priv. Secur.*, 2(2-3):70–246, 2018.

[12] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986.

[13] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1575–1590. ACM, 2020.

[14] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 830–842. ACM, 2016.

[15] Eike Kiltz, Vadim Lyubashevsky, and Christian Schaffner. A concrete treatment of fiat-shamir signatures in the quantum random-oracle model. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 552–586. Springer, 2018.

[16] Chelsea Komlo and Ian Goldberg. FROST: flexible round-optimized schnorr threshold signatures. In Orr Dunkelman, Michael J. Jacobson Jr., and Colin O'Flynn, editors, *Selected Areas in Cryptography - SAC 2020 - 27th International Conference, Halifax, NS, Canada (Virtual Event), October 21-23, 2020, Revised Selected Papers*, volume 12804 of *Lecture Notes in Computer Science*, pages 34–65. Springer, 2020.

[17] Arjen K. Lenstra, Hendrik W. Lenstra, and Laszlo Lovasz. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[18] Yehuda Lindell. Secure multiparty computation. *Commun. ACM*, 64(1):86–96, 2021.

[19] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In Sara Foresti and Giuseppe Persiano, editors, *Cryptology and Network Security - 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings*, volume 10052 of *Lecture Notes in Computer Science*, pages 124–139, 2016.

[20] Vadim Lyubashevsky. Lattice-based identification schemes secure under active attacks. In Ronald Cramer, editor, *Public Key Cryptography - PKC 2008, 11th International Workshop on Practice and Theory in Public-Key Cryptography, Barcelona, Spain, March 9-12, 2008. Proceedings*, volume 4939 of *Lecture Notes in Computer Science*, pages 162–179. Springer, 2008.

[21] Vadim Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 598–616. Springer, 2009.

[22] Daniele Micciancio and Oded Regev. Worst-case to average-case reductions based on gaussian measures. *SIAM J. Comput.*, 37(1):267–302, 2007.

[23] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 84–93. ACM, 2005.

[24] Oded Regev. Lattice-based cryptography. In Cynthia Dwork, editor, *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings*, volume 4117 of *Lecture Notes in Computer Science*, pages 131–141. Springer, 2006.

[25] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[26] Dragos Rotaru and Tim Wood. Marbled circuits: Mixing arithmetic and boolean circuits with active security. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *Progress in Cryptology - INDOCRYPT 2019 - 20th International Conference on Cryptology in India, Hyderabad, India, December 15-18, 2019, Proceedings*, volume 11898 of *Lecture Notes in Computer Science*, pages 227–249. Springer, 2019.

[27] Erwin Schrödinger. Die gegenwärtige situation in der quantenmechanik. *Naturwissenschaften*, 23(50):844–849, 1935.

[28] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[29] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 124–134. IEEE Computer Society, 1994.

[30] Victor Shoup. Practical threshold signatures. In Bart Preneel, editor, *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, volume 1807 of *Lecture Notes in Computer Science*, pages 207–220. Springer, 2000.

[31] Nigel P. Smart. *Cryptography Made Simple*. Springer, 2016.

[32] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, s2-42(1):230–265, 1937.

[33] E Waring. VII. problems concerning interpolations. *Philos. Trans. R. Soc. Lond.*, 69(0):59–67, 1779.

[34] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 160–164. IEEE Computer Society, 1982.

[35] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167. IEEE Computer Society, 1986.

Erklärung

gemäss Art. 30 RSL Phil.-nat. 18

Name/Vorname:	Roux Dominique Marcel
Matrikelnummer:	16-100-661
Studiengang:	Computer Science

Bachelor ☐ Master ☑ Dissertation ☐

Titel der Arbeit:	Implementation of a Threshold Post-Quantum Signature Scheme
LeiterIn der Arbeit:	Prof. Dr. Christian Cachin

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.
Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Bern, 22.08.2022

Ort/Datum

Unterschrift