TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Jelizaveta Vakarjuk 192568IVCM

Converting a post-quantum signature scheme to a two-party signature scheme

Master's thesis

Supervisor:	Ahto Buldas
	PhD
Supervisor:	Jan Willemson
	PhD

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Jelizaveta Vakarjuk 192568IVCM

Kahe osapoolega signeerimisprotokoll postkvant signatuuriskeemi baasil

Magistritöö

Juhendaja:	Ahto Buldas
	PhD
Kaasjuhendaja:	Jan Willemson
	PhD

Tallinn 2021

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Jelizaveta Vakarjuk

19.04.2021

Abstract

Over the last years, there was a significant amount of research conducted in the area of post-quantum cryptography. The main motivation for this research is that the construction of the practical-scale quantum computer will break most of the currently used public-key cryptosystems, such as RSA and ECDSA. A two-party version of RSA digital signature is currently used in Smart-ID. Additionally, there is ongoing research that aims to construct an efficient two-party version of ECDSA. However, this is a non-trivial task to distribute the signing process between two parties and keep the protocol efficient at the same time. Once a practical-scale quantum computer is built, it will no longer be secure to use neither RSA nor ECDSA. Therefore, there is a need to find a quantum-resistant alternative to the two-party RSA that can be integrated into Smart-ID and used when practical-scale quantum computers will be built.

The goal of this research is to construct a two-party version of one of the post-quantum lattice-based signature schemes. During this research, there were several schemes designed and analysed before the final two-party signature scheme was constructed. This thesis describes three versions of the two-party signature scheme that were constructed and their associated problems and limitations. Additionally, this thesis presents the security proof for the final version of the signature scheme. This security proof considers a classical adversary that does not have access to a quantum computer. The security of the proposed two-party protocol relies on mathematical problems that are considered to be hard for both classical and quantum computers. Although the security proof presented in the thesis is in the classical model, it can be considered as an initial step for proving the quantum security of the proposed signature scheme.

This thesis is written in English and is 89 pages long, including 9 chapters, 9 figures and 19 tables.

Annotatsioon

Kahe osapoolega signeerimisprotokoll postkvant signatuuriskeemi baasil

Viimaste aastate jooksul on postkvant krüptograafia valdkonnas läbi viidud märkimisväärne hulk uuringuid. Nende uuringute motivatsioon on see, et kui piisavalt võimas kvantarvuti on ehitatud, muutuvad paljud avaliku võtme krüptograafilised algoritmid, näiteks RSA ja ECDSA, ebaturvalisteks. Kahe osapoolega RSA signeerimisprotokolli kasutatakse praegu Smart-ID süsteemis. Lisaks on käimas teadusuuringud, mille eesmärk on luua tõhus kahe osapoolega ECDSA. Allkirjastamisprotsessi jagamine kahe osapoole vahel samal ajal protokolli turvalisust ja tõhusust säilitades on osutunud äärmiselt mittetriviaalseks ülesandeks. Kui aga võimas kvantarvuti ehitatakse, ei ole enam turvaline ei RSA ega ECDSA kasutamine. Seetõttu on vaja leida kahe osapoolega RSA signeerimisprotokolli kvantkindel alternatiiv, mida saab integreerida Smart-ID süsteemi ja kasutada ka siis, kui piisavalt võimas kvantarvuti on ehitatud.

Selle töö eesmärk on konstrueerida ühe postkvant võrepõhise signatuuriskeemi baasil kahe osapoolega signeerimisprotokoll. Töö käigus, enne lõpliku signeerimisprotokolli koostamist, töötati välja ja analüüsiti mitmeid versioone. Töös kirjeldatakse võrepõhise kahe osapoolega signeerimisprotokolli kolme versiooni ning nendega seotuid probleeme ja piiranguid. Lisaks esitatakse selles töös välja pakutud kahe osapoolega signeerimisprotokolli lõpliku versioonile turvatõestus klassikalise vastase vastu, kellel pole juurdepääsu kvantarvutile. Lõpliku kahe osapoolega signeerimisprotokolli turvalisus tugineb matemaatilistel probleemidel, mida peetakse raskeks nii klassikaliste kui ka kvantarvutite jaoks. Kuigi lõputöös esitatud turvatõestus käsitleb klassikalist vastast, võib seda pidada esialgseks sammuks pakutava signeerimisprotokolli kvantturvalisuse tõestamisel.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 89 leheküljel, 9 peatükki, 9 joonist, 19 tabelit.

Acknowledgements

Thank you to my supervisors Ahto Buldas and Jan Willemson for proposing this research topic and supporting me all the way through it. Thank you to Alisa Pankova from Cybernetica for giving me a number of good ideas that helped me move forward in my thesis.

Acronyms

BDD Bounded Distance Decoding. 25

CRH Collision-Resistant Hash function. 40

CVP Closest Vector Problem. 24

DS-UF-CMA Distributed Signature Unforgeability Against Chosen Message Attacks. 58

DSA Digital Signature Algorithm. 22

ECDSA Elliptic Curve Digital Signature Algorithm. 11

FSwA Fiat-Shamir with Aborts. 19

GPV Gentry–Peikert–Vaikuntanathan. 20

LWE Learning with Errors. 25

MPC Multiparty Computation. 18

naHVZK No-Abort Honest-Verifier Zero Knowledge. 19, 33

NIST U.S Department of Commerce National Institute of Standards and Technology. 11

PQC Post-Quantum Cryptography. 11

QSCD Qualified Signature Creation Device. 11

RSA Rivest–Shamir–Adleman. 11

SHA Secure Hash Algorithm. 49

SHAKE Secure Hash Algorithm and KECCAK. 31

SIS Short Integer Solution. 25

SVP Shortest Vector Problem. 24

UF-CMA Existentially Unforgeable under Chosen Message Attack. 16

ZKP Zero-Knowledge Proof. 22

Contents

List of Figures

List of Tables

1 Introduction

1.1 Research motivation

Modern public key cryptography mostly relies on the hardness of solving two mathematical problems: integer factorisation and discrete logarithm. There exist efficient algorithms for quantum computers that can solve these problems [1]. Therefore, when a large-scale quantum computer will be built, it will no longer be secure to use cryptographic schemes that rely on these problems. Post-quantum cryptography is a new branch of cryptography that is looking for quantum-resistant cryptographic schemes. In 2016, U.S Department of Commerce National Institute of Standards and Technology (NIST) initiated the Post-Quantum Cryptography (PQC) competition. The project aims to standardize post-quantum public key cryptography algorithms in three major categories: public key encryption algorithms, key-establishment algorithms, and signature schemes [2]. For each category, several proposed schemes are analysed and evaluated to find the best alternatives to the currently used public key cryptosystems, such as Rivest–Shamir–Adleman (RSA), Elliptic Curve Digital Signature Algorithm (ECDSA), ElGamal, etc.

Post-Quantum cryptography is also concerned with the problem of the storage of secret keys. Once the secret key is extracted from an end-point device, it can be used by an attacker to impersonate a legitimate user. Threshold cryptography is used to solve this problem. In a (t, n)-threshold scheme, the secret key is shared between n users/devices. To create a valid signature or decrypt the message, a subset of t users/devices should collaborate and use their secret key shares [3]. In Baltic countries, this technique is used in Smart-ID, a mobile application that works as an authentication solution, and is recognised as a Qualified Signature Creation Device (QSCD) since November 2018 [4]. Smart-ID uses (2,2)-threshold version of RSA signing that was introduced by A. Buldas et. al. [5]. The RSA cryptosystem used in Smart-ID is vulnerable to quantum computer attacks as aforementioned. Considering the significant threat from quantum computers, there is a need to find a quantum-resistant alternative that can be used instead of RSA [6].

1.2 Research scope and goal

The main goal of this research is to propose a (2,2)-threshold signature scheme that is based on lattices. It was decided to concentrate on the lattice-based signatures as these are considered to be the most promising general-purpose digital signature schemes among the schemes submitted to the NIST PQC competition [7]. Furthermore, this work defines a threat model for the proposed scheme, gives the definition of security, and proves that the proposed scheme is secure with respect to this definition. The performance of the scheme is estimated according to the following metrics:

- number of communication rounds in key generation and signing protocols
- keys and signature sizes
- number of rejection sampling rounds.

This work focused on developing a two-party lattice-based signature scheme and proving the security of the scheme against a classical adversary that does not have access to a quantum computer. The future research will focus on the implementation of the proposed scheme and the proofs against a quantum adversary. Moreover, for the security proofs, it is assumed that the underlying mathematical problems are computationally hard both for classical and quantum computers.

The research focused on the NIST PQC competition of which the material is open-source and it is encouraged to develop novel and original research to further the community knowledge of PQC. All the materials (documentation, source code, benchmarking results) regarding the signature schemes that are analysed are publicly available on the webpage of the NIST PQC competition or from the Cryptology ePrint Archive.

1.3 Research questions

The research questions that are answered in the current work are defined as follows:

1. How is it possible to convert one of the schemes proposed in the NIST PQC competition – Crystals-Dilithium [8] to a threshold signature scheme?
2. Is it possible to propose a new lattice-based threshold signature scheme?
3. What threat model is suitable for the signature scheme proposed in the second research question?
4. Which security properties are satisfied in the proposed scheme? Prove that the scheme is secure with respect to the given security definition.

1.4 Contribution

There exist several studies within the topic of this research, however, there is no threshold signature scheme proposed that is purely based on one of the NIST PQC competition submissions. This work analyses whether it is possible to construct a threshold signature scheme based on the Crystals-Dilithium signature scheme that was submitted to the NIST PQC competition. During the research, it became apparent that constructing a threshold scheme based on the NIST PQC submission was not possible, so the current work explains the details and problems that occurred. Moreover, a suitable lattice-based signature scheme is found and a threshold version of it is proposed. Analysis of existing studies in this area suggests that no scheme suits perfectly to Smart-ID framework, therefore the scheme that is proposed in this research takes into account the requirements of the framework.

1.5 Research methods

The research method that is used to construct a lattice-based two-party signature scheme and prove its security is analytical. Firstly, the literature review was conducted to find the existing post-quantum threshold signature schemes. Then, these schemes were analysed to identify their limitations and disadvantages. The first version of a two-party signature protocol was designed and analysed. The analysis of the first version identified several problems in the protocol. These problems were fixed in the second version, but the analysis of the second version identified some other mistakes. The final version of the protocol was constructed, analysed, and proven to be secure against a classical adversary. In this work, it is proven that the essential cryptographic properties for a signature scheme hold. The security proof of the final version is based on the proof from [9], and [10].

The next step is the empirical evidence collection that consists of estimating the performance of the scheme according to the metrics that were defined above.

1.6 Thesis organisation

The following work contains nine chapters. Chapter 1 gives an introduction to the topic. Chapter 2 proceeds with describing the notation that is used in this work and introduces the cryptographic background. Chapter 3 analyses related work. Chapter 4 gives an introduction to post-quantum cryptography and focuses mainly on the concepts related to lattice-based cryptography. Chapter 5 describes two lattice-based signature schemes that are used in this work to create a two-party signature scheme. In chapter 6, the process of designing the two-party signature scheme is described. This chapter contains three versions of the two-party protocol that were created during this process, with the last version being the final two-party signature scheme. Chapter 7 gives the security definition for the proposed two-party protocol and contains proof that the signature scheme is secure according to that definition. Chapter 8 gives performance estimations of the scheme. Chapter 9 contains the conclusion and the future work.

2 Cryptographic Background

The following section introduces the notation and the main definitions that are used in this work.

2.1 Notation

- Let $\mathbb{Z}$ be a ring of all integers. $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ denotes a ring of residue classes modulo q. $\mathbb{Z}[x]$ denotes a ring of polynomials in the variable x with integer coefficients.
- R denotes a quotient ring $\mathbb{Z}[x]/(x^n + 1)$, where $n \in \mathbb{N}$ and R_q denotes a quotient ring $\mathbb{Z}_q[x]/(x^n + 1)$, where $n \in \mathbb{N}$.
- Polynomials are denoted in italic lowercase p. $p \in R_q$ is a polynomial of degree bound by n: $p = p_0 + p_1 x + ... + p_{n-1}x^{n-1}$. It can be also expressed in a vector notation through its coefficients $(p_0, p_1, ..., p_{n-1})$.
- Vectors are denoted in bold lowercase $\mathbf{v}$. $\mathbf{v} \in R_q^n$ is a vector of dimension n: $\mathbf{v} = (v_0, ..., v_{n-1})$, where each element v_i is a polynomial in R_q.
- Matrices are denoted in bold uppercase $\mathbf{A}$. $\mathbf{A} \in R_q^{n\times m}$ is a $n \times m$ matrix with elements in R_q.
- For an even positive integer α and for every $x \in \mathbb{Z}$, define $x' = x \mod {}^{\pm}\alpha$, as x' in the range $-\frac{\alpha}{2} < x' \leq \frac{\alpha}{2}$ such that $x' \equiv x \pmod{\alpha}$. For an odd positive integer α and for every $x \in \mathbb{Z}$, define $x' = x \mod {}^{\pm}\alpha$, as x' in the range $-\frac{\alpha-1}{2} \leq x' \leq \frac{\alpha-1}{2}$ such that $x' \equiv x \pmod{\alpha}$. For any positive integer α, define $x' = x \mod \alpha$, as x' in the range $0 \leq x' < \alpha$ such that $x' \equiv x \pmod{\alpha}$.
- For an element $x \in \mathbb{Z}_q$, its infinity norm is defined as $||x||_\infty = |x \mod {}^{\pm}q|$, where $|x|$ denotes the absolute value of the element. For an element $p = p_0 + p_1 x + ... + p_{n-1}x^{n-1} \in R_q$, $||p||_\infty = \max_i ||p_i||_\infty$. Similarly for an element $\mathbf{v} = (p_0, ..., p_n) \in R_q^n$, $||\mathbf{v}||_\infty = \max_i ||p_i||_\infty$.
- S_η denotes a set of all elements $p \in R$ such that $||p||_\infty \leq \eta$.
- $a \leftarrow A$ denotes sampling an element uniformly at random from the set A.
- $a \leftarrow \chi(A)$ denotes sampling an element from the distribution χ defined over the set A.
- $\lceil x \rceil$ denotes mapping x to the least integer greater than or equal to x (e.g $\lceil 5.2 \rceil = 6$).

2.2 Definitions

The following definition is adopted from [11].

Definition 1. **Digital signature** is a triple of probabilistic algorithms (*KeyGen, Sign, Verify*) that work as follows:

1. Generate keypair consisting of secret key and public key using security parameter λ as input: $(sk, pk) \leftarrow \text{KeyGen}(1^{\lambda})$.
2. To sign a message m use the secret key: $\sigma \leftarrow \text{Sign}(sk, m)$. Signature is published together with the corresponding message.
3. To verify signature, verifier needs to check if $\text{Verify}(pk, m, \sigma) = 1$. If signature was generated correctly, verification should always succeed.

The following definition is adopted from [12].

Definition 2. A **hash function** is a function $H : X \rightarrow R$ which takes arbitrary length input $x \in X$ and produces a fixed-length output $r \in R$.

Definition 3. A hash function family $\mathcal{F}$ is called **collision resistant** if for a randomly chosen function $f \leftarrow \mathcal{F}$, where $f : X \rightarrow R$, the probability that adversary, given access to f finds $x, x' \in X, x \neq x'$ such that $f(x) = f(x')$ is negligible.

Advantage of adversary $\mathcal{A}$ in breaking collision resistance of a hash function $f : X \rightarrow R$ can be defined as follows:

$$\text{Adv}^{\text{CR}}(\mathcal{A}) := \Pr[f(x) = f(x') \wedge x \neq x' : f \leftarrow \mathcal{F}, (x, x') \leftarrow \mathcal{A}(f)].$$

Definition 4. Let $+$ be an operation defined over X (the set of inputs of a hash function) and let $\oplus$ be an operation defined over R (the set of outputs of the hash function). Let $x_1, x_2 \in X$ be any two inputs to the hash function. A hash function $f : X \rightarrow R$ is homomorphic if it holds that:

$$f(x_1 + x_2) = f(x_1) \oplus f(x_2).$$

The following definitions are adopted from [3].

Definition 5. A signature scheme (*KeyGen, Sign, Verify*) is called Existentially Unforgeable under Chosen Message Attack (UF-CMA) iff for any probabilistic polynomial time adversary $\mathcal{A}$, its advantage of creating successful signature forgery is negligible in λ. Advantage of adversary is defined as:

$$\text{Adv}^{\text{UF-CMA}}(\mathcal{A}) = \Pr[Verify(pk, m, \sigma) = 1 \text{ and } m \text{ is fresh } : (pk, sk) \leftarrow KeyGen(1^{\lambda}), (m, \sigma) \leftarrow \mathcal{A}^{Sign(\cdot)}(pk)] \leq \text{negl}(\lambda).$$

Definition 6. (t, n)-threshold signature is a digital signature scheme where n parties share a secret key and authorized subset of $t \leq n$ parties is sufficient for every signing, but any set of less than t parties can do nothing.

A special case of the general (t, n)-threshold signature is (n, n)-signature. In this case, all n parties are needed to produce a valid signature, for a subset of less than n parties it is not possible to generate a valid signature. Below, is a more formal definition for (n, n)-signature or distributed signature protocol. The following definitions are adopted from [9].

Definition 7. Distributed signature protocol is a protocol between $P_1, ..., P_n$ parties that consists of the following algorithms:

- Generate public parameters *par* using security parameter λ as input: $par \leftarrow \text{Setup}(1^\lambda)$.
- Each party P_j generates a keypair consisting of secret key share and public key using interactive algorithm and public parameters as input: $(sk_j, pk) \leftarrow \text{KeyGen}_j(par)$ for each $j \in \{1, ..., n\}$.
- To sign a message m, each party P_j runs interactive signing algorithm using secret key share: $(\sigma) \leftarrow \text{Sign}_j(sk_j, m)$ for each $j \in \{1, ..., n\}$.
- To verify signature, verifier needs to check if $\text{Verify}(pk, m, \sigma) = 1$. If signature was generated correctly, verification should always succeed.

Definition 8. Multi-signature protocol is a protocol consisting of the following algorithms:

- Generate public parameters *par* using security parameter λ as input: $par \leftarrow \text{Setup}(1^\lambda)$
- A **non-interactive** key generation algorithm produces a keypair consisting of secret key and public key using public parameters as input: $(sk, pk) \leftarrow \text{KeyGen}(par)$.
- To sign a message m, party P runs interactive signing algorithm using their own keypair (sk, pk) and a set of co-signers' public keys L: $(\sigma) \leftarrow \text{Sign}(sk, pk, m, L)$.
- To verify signature, verifier needs to check if $\text{Verify}(L, m, \sigma) = 1$. If signature was generated correctly, verification should always succeed.

The main difference between a distributed signature and a multi-signature protocol is that the key generation process is not interactive for multi-signature protocol. Each party generates its keypair that does not depend on inputs from the other parties.

The following definitions are adopted from [13].

Definition 9. A **commitment** protocol is a two-party protocol that consists of two phases: commit and open. In the commit phase, the first party creates a commitment to some value and sends it to the second party. In the opening phase, the first party opens the value that was inside the commitment, and the second party checks if the commitment was opened correctly.

Two basic properties should hold for the security of the commitment scheme:

Definition 10. Binding property: after giving away commitment, the first party can no longer change the value inside the commitment.

Definition 11. Hiding property: when the second party receives the commitment, it cannot see what is inside the commitment until the opening phase.

The following definition is adopted from [14]:

Definition 12. Multiparty Computation (MPC): Parties $P_1, ..., P_n$ participate in the protocol, each party P_i has its own secret input x_i. All the parties agree on some function f that takes n inputs, the goal of the parties is to compute $f(x_1, ..., x_n) = y$ such that the following conditions are satisfied:

- the value y is computed correctly (correctness),
- y is the only information revealed to the parties, no information about the private data of the parties is revealed (privacy).

The following definitions are adopted from [15].

Definition 13. Statistical distance. Let X and Y be probability distributions defined over some set A. The statistical distance $\mathrm{SD}(X, Y)$ between X and Y is defined as

$$\mathrm{SD}(X, Y) := \max_{T \subseteq A} |\Pr[X \in T] - \Pr[Y \in T]|.$$

3 Related Work

In 2017, A. Buldas, A. Kalu, P. Laud, and M. Oruaas proposed a server-supported RSA signature that is currently used in the Smart-ID [5]. The scheme has several properties that should also be considered in the process of developing a post-quantum alternative. Firstly, the secret key is divided into two parts, one part is stored on the user's device and the other part is stored on the server's side. This means that the client alone cannot create a valid signature using only their share of the secret key and the same is true for the server. Additionally, the client's key share looks uniformly random and it is encrypted with a password (chosen by the user). Therefore, an adversary who gets access to the password-encrypted key share cannot perform dictionary attacks to find out the key share. The resulting signature looks the same as the standard RSA signature. This means that the signature produced using Smart-ID can be easily verified by the third party using standard cryptographic libraries. Finally, there is a mechanism called clone detection: it detects if the client's secret key was cloned and used by an unauthorized user. If key cloning was detected, the device that sent the signing request is blocked. However, as was mentioned earlier, the scheme will be broken once a large-scale quantum computer will be built.

Several previous works described lattice-based threshold signatures or lattice-based multisignatures. The works [16, 17, 18, 19, 20, 21] focused on creating multisignatures that followed the Fiat-Shamir with Aborts (FSwA) paradigm. Rejection sampling is a special technique used in signatures that follow FSwA paradigm. Rejection sampling consists of several checks that are performed on the signature before it is output. Rejection sampling helps to remove dependency of signature on the secret key and prevent the leakage of information about the secret key. The signing process is repeated until all the conditions in the rejection sampling are satisfied. One of the main problems of the signature schemes constructed using the FSwA paradigm is that due to the use of rejection sampling, intermediate values produced during the signature generation process need to be kept secret until the rejection sampling has been completed. There are currently no known attacks that exploit aborted executions of the protocol and there are no techniques of proving the underlying identification scheme has No-Abort Honest-Verifier Zero Knowledge (naHVZK) property [9]. In some of the previously mentioned multisignatures ([16], [18], [19], [20], [17]), intermediate values were published by parties before the rejection sampling that results in incomplete security proofs in these works [9]. The work by M. Fukumitsu and S. Hasegawa [21] relies on a non-standard hardness assumption (rejected LWE) to solve the problem with the aborted executions and proof the security of the scheme in the quantum random oracle model.

In 2019, D. Cozzo and N. Smart in the work [22] analysed the second round NIST PQC competition signature schemes to determine whether it is possible to turn these schemes into threshold versions. The authors used only generic multiparty computation techniques, like linear secret sharing and garbled circuits to propose a way how to *thresholdize* each of the second round signature schemes. There were no details and concrete instantiations of the MPC protocols described that should be used. As a result, the authors proposed that the most suitable signature scheme is Rainbow [23] which belongs to the multivariate-based family. The authors described a threshold version of Crystals-Dilithium [8], that, by estimations, takes around 12 seconds to produce a single signature. The authors noted that the problems with performance come from the fact that the signature scheme consists of both linear and non-linear operations and it is inefficient to switch between these representations using generic MPC techniques. The work considered a broader case of multiple parties trying to jointly create a signature, but the goal of the current work is to focus on a specific scenario with two parties that has its technicalities.

R. Bendlin, S. Krehbiel, and C. Peikert in the work [24] proposed threshold protocols for generating a hard lattice with trapdoor and sampling from the discrete Gaussian distribution using the trapdoor. These two protocols are the main building blocks for the Gentry–Peikert–Vaikuntanathan (GPV) signature scheme (based on hash-and-sign paradigm), where generating a hard lattice is needed for the key generation and Gaussian sampling is needed for the signing process. The work describes a threshold version of the GPV signature but does not contain an exact selection of the parameters, implementation, or performance estimation.

M. Kansal and R. Dutta in the work [25] proposed a lattice-based multisignature scheme with a single round signature generation, that has key aggregation and signature compression. Key aggregation guarantees that the public key size is the same as the size of a public key of a single signer. Signature compression makes multisignature size the same as the size of a single signature. The underlying signature scheme, however, follows neither hash-and-sign nor FSwA paradigm that are the main techniques used to construct lattice-based signature schemes. The work does not contain an exact selection of parameters, implementation, or performance estimation.

In 2020 I. Damgård, C. Orlandi, A. Takahashi, and M. Tibouchidamgard proposed a lattice-based multisignature and distributed signing protocols that are based on the Dilithium-G signature scheme in the work [9]. Dilithium-G is a version of Crystals-Dilithium signature that requires sampling from a discrete Gaussian distribution [26]. The work contains the construction of a trapdoor commitment that is required to construct a two-round distributed signature scheme and the complete classical security proofs for the proposed schemes. The work solves the problem with the aborted executions by using commitments such that in the case of abort only commitment is published, the intermediate value itself stays secret. The proposed distributed signature scheme could potentially fit the Smart-ID framework, however, some questions need to be addressed. More precisely, the scheme is based on a modified version of Crystals-Dilithium from the NIST PQC competition project and uses Gaussian sampling. It is known that generating samples from the Gaussian distribution is non-trivial which means that the insecure implementation may lead to the side-channel attacks [27]. The open question is whether it is possible to use the version of the scheme more similar to the one being submitted to the NIST PQC competition.

4 Post-quantum cryptography

In 1994, Peter Shor described a quantum algorithm that can efficiently solve mathematical problems that are the basis of modern public key cryptography [1]. These problems are integer factorization and discrete logarithm. The security of almost all currently used public key cryptosystems rely on these two problems. When a large-scale quantum computer will be built these cryptographic schemes will become insecure including RSA (both encryption and signing), Digital Signature Algorithm (DSA), ECDSA, Diffie-Hellman key exchange, etc. By estimations of Michele Mosca, RSA-2048 will be broken by 2026 with a 1/6 chance and by 2031 with 1/2 chance [6]. It should be noted that quantum computers will have an impact on symmetric-key cryptography as well, but this impact will not be as pronounced compared to the impact to public key cryptography.

To address the issue of securing communication in the quantum computing era, the research community started looking for alternative cryptographic schemes that will be resistant to the known quantum computer attacks. The new branch of cryptography dealing with these schemes is called post-quantum cryptography. In 2016, NIST initiated a competition that aims to standardize post-quantum public key cryptography algorithms in three major categories: public key encryption, signature schemes, and key exchange algorithms [2]. For the first round of the competition 82 candidate algorithms were submitted and NIST accepted 69 of them based on the submission requirements and minimum acceptability criteria [28]. Among all the accepted schemes 20 were digital signature schemes. In 2019, NIST selected 26 second-round candidates from the 69 first-round candidates based on the security, cost and performance and algorithm and implementation characteristics [28]. Among all the second-round candidates 9 were digital signature schemes from the following families: lattice-based, multivariate, hash-based and based on the Zero-Knowledge Proof (ZKP).

In 2020, NIST announced 15 third-round candidates, seven were selected as finalists and eight as alternate candidates [7]. Among all the third-round schemes six are digital signature schemes. NIST PQC competition finalists in the digital signature category are [7]:

- Crystals-Dilithium – a lattice-based signature scheme that follows FSwA paradigm [8].
- Falcon – a lattice-based signature scheme that follows hash-and-sign paradigm [29].

- Rainbow – a multivariate digital signature scheme [23].

Crystals-Dilithium and Falcon use different lattice problems as the underlying security assumptions. Falcon has better performance and signature and public key sizes, but its implementation is more complex, as it requires sampling from the Gaussian distribution and using floating-point numbers to implement an optimised polynomial multiplication [29]. Crystals-Dilithium has slightly slower performance results and the sizes of signature and public key are bigger than ones in Falcon but it has a simpler structure in terms of implementation [8]. It was mentioned in the NIST report that only one of the previously named lattice-based signature schemes will be standardized [7].

Alternate candidates will have another round of evaluation and are considered as potential candidates for standardisation [7]. Some of the alternate candidates have high security estimations and potential for future improvement, but worse performance. During the third round additional analysis of these schemes will be done in order to select more suitable schemes for future standardisation. Alternate candidates in the digital signature category are [7]:

- GeMSS – multivariate signature scheme [30].
- Picnic – signature scheme based on zero knowledge proof from multiparty computation in the head paradigm [31].
- SPHINCS+ – hash based signature scheme [32].

4.1 Lattice-based cryptography

Lattices are considered to be one of the best tools to build post-quantum cryptosystems. Among all the submissions to the NIST PQC competition the majority of schemes belongs to the lattice-based family [7]. Additionally, it was mentioned in [7] that NIST considers lattice-based schemes to be the most promising general-purpose algorithms for the public-key encryption, key establishment and digital signatures. The following section introduces the main definitions about lattice-based cryptography that are used in this work.

The following definitions are adopted from [33].

Definition 14. Let $\mathbf{B} = (\mathbf{b}_1, ..., \mathbf{b}_n)$ be a basis, then n-dimensional **lattice** generated by $\mathbf{B}$ is defined as a set of all integer combinations of n linearly independent vectors $\mathbf{b}_1, ..., \mathbf{b}_n$:

$$\Lambda = \mathcal{L}(\mathbf{B}) = \{\textstyle\sum_{i=1}^{n} x_i \mathbf{b}_i | x_i \in \mathbb{Z}\}.$$

Definition 15. A set of vectors $\mathbf{B} = (\mathbf{b}_1, ..., \mathbf{b}_n)$ is a **basis** of a lattice Λ if the following holds:

1. vectors $\mathbf{b}_1, ..., \mathbf{b}_n$ are linearly independent,
2. $\Lambda = \mathcal{L}(\mathbf{B})$.

Intuitively, a lattice can be viewed as a set of all intersection points of n-dimensional grid. Figure below illustrates 2-dimensional lattice.

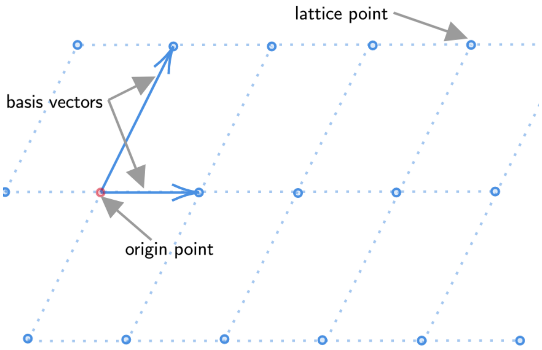


Figure 1: 2-dimensional lattice

Basis is a compact way of representing lattice, lattices of dimension at least two have infinite number of bases, it means that basis of lattice is not unique. This is important as some of the bases are easy to work with and some are not. For example, if the lattice is represented using short vectors, it allows using less storage to describe the lattice.

There are several fundamentally hard problems in lattices, one of them is called Shortest Vector Problem (SVP). The following definitions are adopted from [33] and [34].

Definition 16. SVP: given a basis that describes a lattice, find a non-zero vector in the lattice with minimal length.

Let the length of the shortest nonzero vector in the lattice be denoted as λ_1.

Definition 17. Approximate SVP: given a basis that describes a lattice and an approximation factor γ, find a non-zero vector in the lattice with length at most $\gamma\lambda_1$.

SVP problem can be turned into a decisional problem that is called GapSVP.

Definition 18. GapSVP: given a basis that describes a lattice and a parameter β, decide whether the lattice contains a non-zero vector of length at most 1 or if the shortest non-zero vector has length larger than $\beta > 1$.

The second fundamental problem in the lattices is called Closest Vector Problem (CVP).

Definition 19. CVP: given a basis that describes a lattice and a target vector that does not belong to the lattice, find a vector in the lattice that is the closest to the target vector.

Similarly to SVP, CVP problem can also be turned into a decisional problem called GapCVP.

Finally, there is a Bounded Distance Decoding (BDD) problem, that is very similar to CVP.

Definition 20. BDD: given a basis that describes a lattice and a target vector, find a vector in the lattice that is the closest to the target vector, where target is guaranteed to be d-close to the lattice, where $d < \lambda_1/2$.

However, problems described above cannot be used directly to build cryptographic algorithms on them. One of the hard lattice problems that is used to construct cryptosystems is called Short Integer Solution (SIS) and was introduced by M. Ajtai, in the work [35].

Definition 21. SIS: given a uniformly random matrix $\mathbf{A}_q^{n\times m}$ and a coefficient bound v, find a non-zero vector $\mathbf{y} \in \mathbb{Z}^m$ such that $\mathbf{Ay} \equiv 0 \pmod q$ and $0 <\parallel \mathbf{y} \parallel\leq v$.

This problem can be seen as a problem to find a short non-zero vector in the lattice generated by the solutions to $\mathbf{Ay} \equiv 0 \pmod q$, hence it is an instance of approximate SVP for some unknown approximation factor.

The second problem widely used in the lattice-based cryptography is Learning with Errors (LWE), it was introduced by O. Regev [34].

Definition 22. Computational LWE(n,m,q,χ):

- Let $(n, m, q) \in \mathbb{N}$ be positive integers and let χ be an error distribution defined over $\mathbb{Z}_q$.
- Let $\mathbf{A} \leftarrow \mathbb{Z}_q^{n\times m}$ be a uniformly random $n \times m$ matrix and let $\mathbf{e} \in \mathbb{Z}_q^n$ be an error vector chosen according to χ.

An algorithm solves computational LWE for parameters (n,m,q,χ) if, for any $\mathbf{s} \in \mathbb{Z}_q^m$, given $(\mathbf{A}, \mathbf{b} := \mathbf{As} + \mathbf{e})$ it outputs $\mathbf{s}$ (with non-negligible probability).

The Figure 2 graphically illustrates SIS and LWE problems.

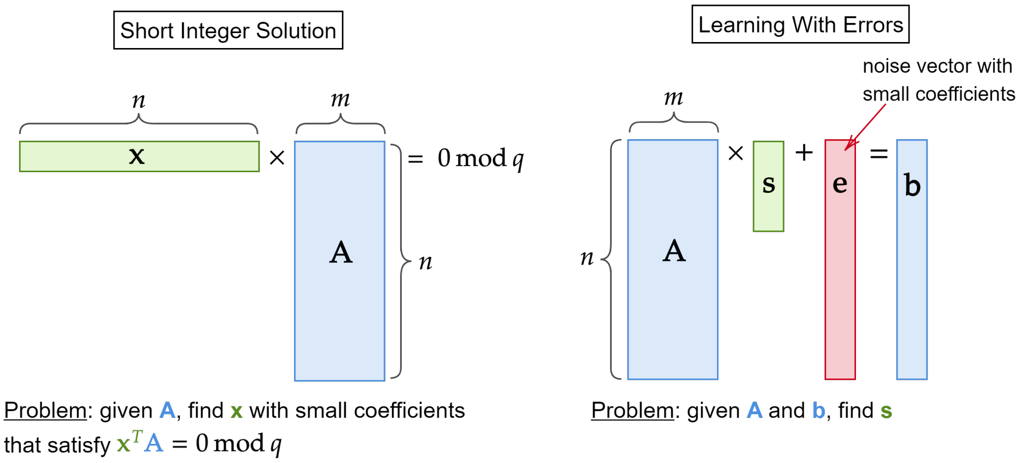

Short Integer Solution
n
m
$\mathbf{x}$ × A $= 0 \bmod q$
n
Problem: given A, find x with small coefficients that satisfy $\mathbf{x}^T\mathrm{A} = 0 \bmod q$
Learning With Errors
noise vector with small coefficients
m
n
A × s + e = b
Problem: given A and b, find s

Figure 2: SIS and LWE problems

Definition 23. Decisional LWE asks given a pair $(\mathbf{A}, \mathbf{b})$ to distinguish if the vector $\mathbf{b}$ is of the form $\mathbf{As} + \mathbf{e}$ or the vector $\mathbf{b}$ is chosen uniformly at random from $\mathbb{Z}_q^n$.

It can be seen that this problem is closely related to the BDD problem with the lattice given by $\{\mathbf{y} \in \mathbb{Z}^n : \mathbf{y} = \mathbf{As} \mod q,$ for some $\mathrm{s} \in \mathbb{Z}^m\}$ and the target vector $\mathbf{b}$.

The problems defined above use unstructured matrix $\mathbf{A}$, where each element is sampled uniformly at random from $\mathbb{Z}_q$. Therefore, cryptographic schemes based directly on SIS and LWE problems typically require large key sizes and expensive computations that makes them impractical. To store less amount of data and perform more efficient computations matrix $\mathbf{A}$ should be structured, the difference between structured and unstructured matrices is illustrated in Figure 3.

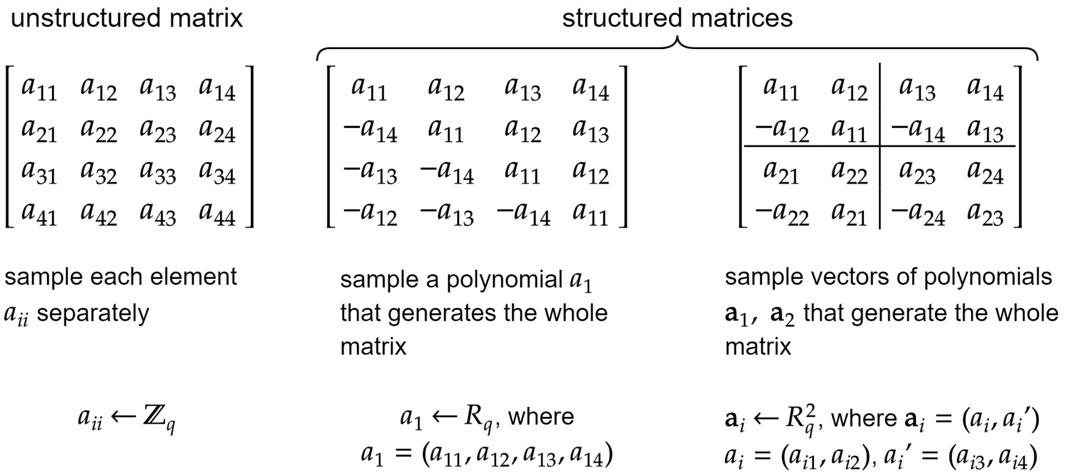

unstructured matrix
structured matrices
$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$
$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ -a_{14} & a_{11} & a_{12} & a_{13} \\ -a_{13} & -a_{14} & a_{11} & a_{12} \\ -a_{12} & -a_{13} & -a_{14} & a_{11} \end{bmatrix}$
$\left[\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{14} \\ -a_{12} & a_{11} & -a_{14} & a_{13} \\ \hline a_{21} & a_{22} & a_{23} & a_{24} \\ -a_{22} & a_{21} & -a_{24} & a_{23} \end{array}\right]$
sample each element a_{ii} separately
sample a polynomial a_1 that generates the whole matrix
sample vectors of polynomials $\mathbf{a}_1$, $\mathbf{a}_2$ that generate the whole matrix
$a_{ii} \leftarrow \mathbb{Z}_q$
$a_1 \leftarrow R_q$, where $a_1 = (a_{11}, a_{12}, a_{13}, a_{14})$
$\mathbf{a}_i \leftarrow R_q^2$, where $\mathbf{a}_i = (a_i, a_i')$ $a_i = (a_{i1}, a_{i2}), a_i' = (a_{i3}, a_{i4})$

Figure 3: Structured and unstructured matrices

The first matrix from Figure 3 corresponds to choosing each element uniformly at random from $\mathbb{Z}_q$.

The second matrix from Figure 3 corresponds to sampling one polynomial from R_q for the first row and then generating a square matrix using the first row. This matrix is constructed by rotating each row one element to the right relative to the preceding row. Elements that are shifted to the beginning of the row are negated. Hard problems that correspond to this matrix structure are called Ring-SIS and Ring-LWE.

The third matrix from Figure 3 corresponds to sampling a vector of polynomials for the first row and a vector of polynomials for the third row from R_q^2 and then generating square matrices out of them. Matrices are constructed similarly to the previous ones: by rotating each row and negating elements that are shifted to the beginning of the row. Hard problems that correspond to this matrix structure are called Module-SIS and Module-LWE.

The following definitions are adopted from [9].

Definition 24. Module-SIS(q, n, m, β): given a uniformly random matrix $\mathbf{A} \leftarrow R_q^{n\times m}$ and a coefficient bound β, find a non-zero vector $\mathbf{y} \in R_q^m$ such that $\mathbf{Ay} \equiv 0 \pmod q$ and $0 <\| \mathbf{y} \| \leqslant \beta$.

Definition 25. Decisional Module-LWE(q, n, m, η, χ): Let χ be an error distribution, given a pair $(\mathbf{A}, \mathbf{t}) \in (R_q^{n\times m} \times R_q^n)$ decide whether it was generated uniformly at random from $R_q^{n\times m} \times R_q^n$ or it was generated as: $\mathbf{A} \leftarrow R_q^{n\times m}$, $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow \chi(S_\eta^m \times S_\eta^n)$ and $\mathbf{t} := \mathbf{As}_1 + \mathbf{s}_2$.

Advantage of adversary$\mathcal{A}$ in breaking decisional Module-LWE for the set of parameters (q, n, m, η, χ) can be defined as follows:

$$\mathrm{Adv}^{\text{Dec-MLWE}}_{(q,n,m,\eta,\chi)}(\mathcal{A}) := |\Pr[b = 1 : \mathbf{A} \leftarrow R_q^{n\times m}, (\mathbf{s}_1, \mathbf{s}_2) \leftarrow \chi(S_\eta^m \times S_\eta^n), \mathbf{t} := \mathbf{As}_1 + \mathbf{s}_2, b \leftarrow \mathcal{A}(\mathbf{A}, \mathbf{t})] - \Pr[b = 1 : \mathbf{A} \leftarrow R_q^{n\times m}, \mathbf{t} \leftarrow R_q^n, b \leftarrow \mathcal{A}(\mathbf{A}, \mathbf{t})]|.$$

A graphical illustration of the decisional module-LWE game between challenger and adversary is represented in Figure 4.

Definition 26. Computational Module-LWE(q, n, m, η, χ): Let χ be an error distribution, given a pair $(\mathbf{A}, \mathbf{t}) \in (R_q^{n\times m} \times R_q^n)$, where $\mathbf{A} \leftarrow R_q^{n\times m}$, $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow \chi(S_\eta^m \times S_\eta^n)$ and $\mathbf{t} := \mathbf{As}_1 + \mathbf{s}_2$, find a vector $\mathbf{s}_1$.

Advantage of adversary$\mathcal{A}$ in breaking computational Module-LWE for the set of parameters (q, n, m, η, χ) can be defined as follows:

$$\mathrm{Adv}^{\text{Com-MLWE}}_{(q,n,m,\eta,\chi)}(\mathcal{A}) := \Pr[\mathbf{s}_1 = \mathbf{s}_1' : \mathbf{A} \leftarrow R_q^{n\times m}, (\mathbf{s}_1, \mathbf{s}_2) \leftarrow \chi(S_\eta^m \times S_\eta^n), \mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2, \mathbf{s}_1' \leftarrow \mathcal{A}(\mathbf{A}, \mathbf{t})].$$

A graphical illustration of the computational module-LWE game between challenger and adversary is represented in Figure 4.

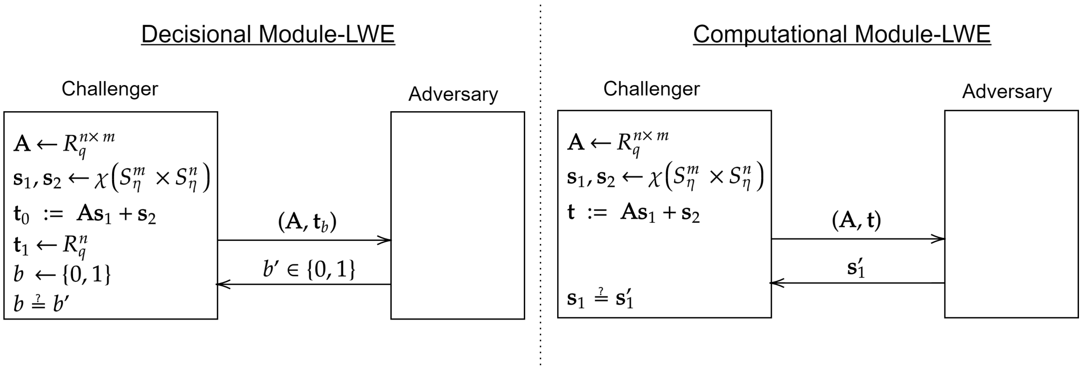

Decisional Module-LWE
Challenger
Adversary
$\mathbf{A} \leftarrow R_q^{n\times m}$
$\mathbf{s}_1, \mathbf{s}_2 \leftarrow \chi(S_\eta^m \times S_\eta^n)$
$\mathbf{t}_0 := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
$\mathbf{t}_1 \leftarrow R_q^n$
$b \leftarrow \{0,1\}$
$b \overset{?}{=} b'$
$(\mathbf{A}, \mathbf{t}_b)$
$b' \in \{0,1\}$
Computational Module-LWE
Challenger
Adversary
$\mathbf{A} \leftarrow R_q^{n\times m}$
$\mathbf{s}_1, \mathbf{s}_2 \leftarrow \chi(S_\eta^m \times S_\eta^n)$
$\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
$\mathbf{s}_1 \overset{?}{=} \mathbf{s}_1'$
$(\mathbf{A}, \mathbf{t})$
$\mathbf{s}_1'$

Figure 4: Decisional Module-LWE and Computational Module-LWE games between challenger and adversary

5 Crystals-Dilithium Digital Signature

A number of lattice-based signatures are constructed from the identification schemes using Fiat-Shamir transform. The following definition is adopted from [36].

Definition 27. An **identification scheme** ID is defined as a tuple of algorithms $ID := (IGen, P, C, V)$.

- The key generation algorithm $IGen$ takes as input system parameters par and returns public key and secret key as output (pk, sk). Public key pk defines the set of challenges C, the set of commitments W and the set of responses Z.
- The prover algorithm $P = (P_1, P_2)$ consists of two sub-algorithms. P_1 takes as input the secret key and returns a commitment $w \in W$ and a state st. P_2 takes as input the secret key, a commitment, a challenge, and a state and returns a response $z \in Z \cup \{\perp\}$, the symbol $\perp$ indicates a failure.
- The verifier algorithm V takes as input the public key and the conversation transcript and outputs a decision bit $b = 1$ (accepted) or $b = 0$ (rejected).

Figure 5 illustrates graphically an identification scheme and the message exchange between the prover and verifier.

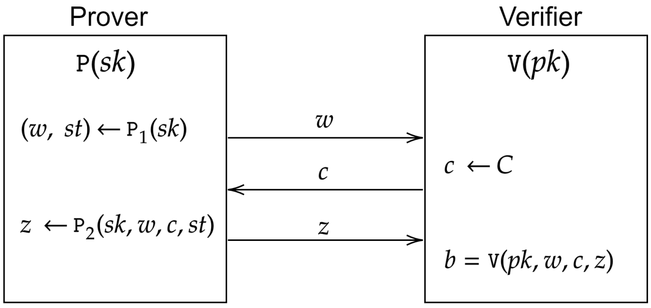


Figure 5: An identification scheme

A **transcript** of identification scheme consists of messages obtained from the interaction between prover and verifier $(w, c, z) \in W \times C \times Z \cup \{\perp, \perp, \perp\}$. Algorithm 1 defines a transcript oracle $Trans$ that returns a real interaction between prover and verifier. In case of failure (or abort) $Trans$ outputs $(\perp, \perp, \perp)$.

Algorithm 1 $Trans(sk)$

```
$(w, st) \leftarrow P_1(sk)$
$c \leftarrow C$
$z \leftarrow P_2(sk, w, c, st)$
if $z = \perp$ then return $(\perp, \perp, \perp)$
otherwise, return $(w, c, z)$
```

The Fiat-Shamir transform technique introduced in [37] allows the creation of a digital signature scheme by combining an identification scheme with a hash function. The signing algorithm generates a transcript (w, c, z), where a challenge is derived from a commitment w and the message to be signed m as follows $c := H(w||m)$. The signature $\sigma = (w, z)$ is valid if the transcript (w, c, z) passes the verification algorithm with $b = 1$. The work [38] introduced a generalisation to this technique called Fiat-Shamir with Aborts transformation that takes into consideration aborting provers.

The Crystals-Dilithium signature scheme is also constructed from an identification scheme using the Fiat-Shamir with Aborts technique. Crystals-Dilithium is one of the finalists of the NIST PQC competition. This is a lattice-based signature scheme based on the hardness of the Module-LWE and Module-SIS problems. The construction of the scheme submitted to the NIST competition uses uniform sampling instead of widely used discrete Gaussian sampling. It is known that generating samples from the discrete Gaussian distribution such that the implementation is secure under side-channel attacks is highly non-trivial [27]. Therefore, the usage of uniform distribution makes it easier to implement the scheme securely. Crystals-Dilithium is a modified and optimised version of the basic schemes proposed in [39] and [40], it has more supporting algorithms that make the scheme more efficient, but the main framework stays the same. This section describes slightly modified versions of both basic schemes [39], [40], because both of them are used during this work to construct the two-party signature scheme.

The Table 1 describes parameters that are used in the both basic signature schemes described in this section.

Table 1: Parameters for the basic lattice-based signature schemes

Parameter	Description
n	degree bound of the polynomials in the ring
q	modulus
(k, l)	dimension of matrix and vectors used in the scheme
γ_1	size bound of coefficients in masking vector, that is generated in the signing process. It should be large enough so the final signature will not leak information about the secret key and yet small enough so that it still remains hard to forge the signature
γ_2	low-order rounding range, used only in the scheme 5.2
η	size bound of coefficients in secret key vectors. It should be small enough so that it is hard to compute secret key out of the public key
β	parameter used in rejection sampling, it denotes maximum possible coefficient of $c\mathbf{s}_i$
B_τ	set of polynomials in R_q that have τ coefficients that are either -1 or 1 and the rest are 0

Both signature schemes that are described below use a special hash function which produces vector of size n with elements $\in \{-1, 0, 1\}$, this helps to ensure that the coefficients of signature are small [8]. The hashing algorithm starts with applying a collision-resistant hash function (e.g. SHAKE-256) to the input to obtain a vector $\mathbf{s} \in \{0, 1\}^\tau$ from the first τ bits of the hash function's output. Then SampleInBall algorithm (Algorithm 2) is invoked to create a vector $\mathbf{c}$ in $\{-1, 0, 1\}^n$ with exactly τ non-zero elements. In each iteration of the for loop SampleInBall algorithm generates an element $j \in \{0, ..., i\}$ using the output of a collision-resistant hash function. Then algorithm performs shuffling of the elements in the vector $\mathbf{c}$ and takes an element from the vector $\mathbf{s}$ to generate -1 or 1. For a more in-depth overview of the algorithm, refer to the original paper [8].

Algorithm 2 SampleInBall

```
Initialize $\mathbf{c}$ as zero vector of length $n$
for $i := n - \tau$ to $n - 1$
    1. $j \leftarrow \{0, 1, ..., i\}$
    2. $s \leftarrow \{0, 1\}$
    3. $c_i := c_j$
    4. $c_j := (-1)^s$
return $\mathbf{c}$
```

5.1 Basic scheme based on Module-LWE

The following signature scheme (further referred to as the 1st basic scheme) is a slightly modified version of the scheme [39], the description below is based on a version described in [36] (appendix B). All the algebraic operations performed in the scheme are done over the quotient ring R_q. The security of this signature scheme relies on the hardness of solving the Module-LWE problem.

Key Generation

Key generation is parametrised with the set of public parameters par, listed in Table 1. Key generation starts with generating $k \times k$ matrix that consists of polynomials from the ring R_q. Matrix $\mathbf{A}$ is the first part of the public key. Then two secret key vectors are sampled, the vectors consist of polynomials from the ring R_q. The coefficients of polynomials in these vectors are of size at most η. Finally, the second part of the public key is computed as $\mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$, the result is a vector of polynomials in the ring R_q.

Signing

H_0 is a special hash function defined as a combination of SHAKE-256 and SampleInBall algorithm (Algorithm 2). Signing starts with the sampling of two masking vectors of polynomials with coefficients less than γ_1. Then a challenge polynomial c is generated, it has exactly τ coefficients that are either -1 or 1 and the rest are 0. Then a potential signature is computed, and rejection sampling is performed. Rejection sampling is needed to ensure that the final signature does not leak information about the secret key. If at least one of the checks fails, the signing process starts again from the beginning. While loop is repeated until both conditions in the rejection sampling are satisfied.

Verification

To verify a signature $\sigma = (\mathbf{z}_1, \mathbf{z}_2, c)$, verifier reconstructs $\mathbf{w}'$. Then the infinity norm of the signature vectors $(\mathbf{z}_1, \mathbf{z}_2)$ is checked. Finally, verifier ensures that c (from the signature) is indeed the hash of $\mathbf{w}'$ and the message m.

A more formal definition of the key generation, signing, and verification is presented in Algorithm 3, Algorithm 5, Algorithm 4.

Algorithm 3 KeyGen(par)

```
$\mathbf{A} \leftarrow R_q^{k \times k}$
$(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^k$
$\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
return $pk = (\mathbf{A}, \mathbf{t}), sk = (\mathbf{A}, \mathbf{t}, \mathbf{s}_1, \mathbf{s}_2)$
```

Algorithm 4 Verify(pk, m, σ)

```
$\mathbf{w}' := \mathbf{A}\mathbf{z}_1 + \mathbf{z}_2 - c\mathbf{t}$
if $c = H_0(m||\mathbf{w}')$ and $||\mathbf{z}_1||_\infty < \gamma_1 - \beta$ and $||\mathbf{z}_2||_\infty < \gamma_1 - \beta$, return 1 (success).
else: return 0.
```

Algorithm 5 Sign(sk, m)

```
1: $\mathbf{z} := \perp$
2: while $(\mathbf{z}_1, \mathbf{z}_2) = (\perp, \perp)$ do:
    1. $\mathbf{y}_1, \mathbf{y}_2 \leftarrow S^k_{\gamma_1 - 1}$
    2. $\mathbf{w} := \mathbf{A}\mathbf{y}_1 + \mathbf{y}_2$
    3. $c := H_0(m||\mathbf{w}) \in B_\tau$
    4. $\mathbf{z}_1 := \mathbf{y}_1 + c\mathbf{s}_1$ and $\mathbf{z}_2 := \mathbf{y}_2 + c\mathbf{s}_2$
    5. if $||\mathbf{z}_1||_\infty \geq \gamma_1 - \beta$ or $||\mathbf{z}_2||_\infty \geq \gamma_1 - \beta$, then $(\mathbf{z}_1, \mathbf{z}_2) := (\perp, \perp)$
3: return $\sigma = (\mathbf{z}_1, \mathbf{z}_2, c)$
```

Correctness

Since $\mathbf{w} = \mathbf{A}\mathbf{y}_1 + \mathbf{y}_2$, $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$, $\mathbf{z}_1 = \mathbf{y}_1 + c\mathbf{s}_1$ and $\mathbf{z}_2 = \mathbf{y}_2 + c\mathbf{s}_2$ it holds that:

$$\mathbf{A}\mathbf{z}_1 + \mathbf{z}_2 - c\mathbf{t} = \mathbf{A}(\mathbf{y}_1 + c\mathbf{s}_1) + \mathbf{z}_2 - c(\mathbf{A}\mathbf{s}_1 + \mathbf{s}_2) =$$
$$\mathbf{A}\mathbf{y}_1 + \mathbf{A}c\mathbf{s}_1 + \mathbf{y}_2 + c\mathbf{s}_2 - c\mathbf{A}\mathbf{s}_1 - c\mathbf{s}_2 = \mathbf{A}\mathbf{y}_1 + \mathbf{y}_2.$$

Therefore, if signature was generated correctly, it will successfully pass the verification.

The identification scheme on which the described signature scheme is based satisfies an important property: No-Abort Honest-Verifier Zero Knowledge (naHVZK), more precisely the identification scheme is perfectly naHVZK. Intuitively, it means that the distribution of the real transcript of identification scheme (generated by $Trans$ using secret key as input) is the same as the distribution of the simulated transcript (produced by Sim using only public key). The following definition is adopted from [36].

Definition 28. An identification scheme is said to be ϵ_{ZK}-**naHVZK** if there exists a probabilistic expected polynomial-time algorithm Sim that is given only the public key pk and that outputs (w, c, z) such that the following holds:

- The distribution of the simulated transcript produced by Sim $((w, c, z) \leftarrow Sim(pk))$ has statistical distance at most ϵ_{ZK} from the real transcript produced by the transcript algorithm $(w', c', z') \leftarrow Trans(sk)$.
- The distribution of c from the output $(w, c, z) \leftarrow Sim(pk)$ conditioned on $c \neq \perp$ is uniformly random over the set C.

$Trans$ and Sim for the discussed identification scheme are defined in Algorithm 6 and Algorithm 7. By the proof in [36], these $Trans$ and Sim algorithms give perfect naHVZK property for the identification scheme with $\epsilon_{ZK} = 0$.

Algorithm 6 *Trans*(sk)

```
$\mathbf{A} \leftarrow R_q^{k \times k}$
$\mathbf{y}_1, \mathbf{y}_2 \leftarrow S_{\gamma-1}^k$
$\mathbf{w} := \mathbf{A}\mathbf{y}_1 + \mathbf{y}_2$
$c \leftarrow C$
$\mathbf{z}_1 := \mathbf{y}_1 + c\mathbf{s}_1$, $\mathbf{z}_2 := \mathbf{y}_2 + c\mathbf{s}_2$
if $||\mathbf{z}_1||_\infty \geq \gamma_1 - \beta$ and $||\mathbf{z}_2||_\infty \geq \gamma_1 - \beta$, then return $\perp$
otherwise, return $(c, (\mathbf{z}_1, \mathbf{z}_2))$
```

Algorithm 7 *Sim*(pk)

```
$\mathbf{A} \leftarrow R_q^{k \times k}$
with probability $1 - \left(1 - \frac{|S_{\gamma-\beta-1}^k|}{|S_{\gamma-1}^k|}\right)^2$, return $\perp$
$\mathbf{z}_1, \mathbf{z}_2 \leftarrow S_{\gamma-\beta-1}^k$
$c \leftarrow C$
return $(c, (\mathbf{z}_1, \mathbf{z}_2))$
```

The 1st basic signature scheme has simple linear operations and does not use additional algorithms to optimise the performance of the scheme. This facilitates converting the scheme into a two-party version. Additionally, the security of the signature scheme relies only on one hard problem (Module-LWE), this facilitates security analysis and the choice of parameters for the scheme. Therefore, the 1st basic signature scheme is used to construct a final version of the two-party signature protocol in this work.

5.2 Basic scheme based on Module-LWE and Module-SIS

The following section defines a version of the scheme [39] that was described in [8] (further referred to as the 2nd basic scheme). All the algebraic operations in the scheme are performed over the quotient ring R_q. The security of this signature scheme relies on two hard problems Module-LWE and Module-SIS. Module-LWE guarantees that adversary cannot compute secret keys out of public key and Module-SIS guarantees difficulty of forging signatures.

The signature scheme below makes use of supporting algorithms to extract high-order bits and low-order bits out of each coefficient of polynomial. The algorithm that is used to break up an element into high-order and low-order bits is called Decompose$_q$ and is defined in Algorithm 8. As a result, input r is broken up to $r = r_1 \cdot \alpha + r_0$. The output of Decompose$_q$ consists of two integers r_0, r_1 such that $0 \leq r_1 < \frac{(q-1)}{\alpha}$ and $||r_0||_\infty \leq \frac{\alpha}{2}$. **HighBits**$_q(r, \alpha)$ denotes r_1 part of Decompose$_q$ output and **LowBits**$_q(r, \alpha)$ denotes r_0 part of Decompose$_q$ output.

Algorithm 8 $\text{Decompose}_q(r, \alpha)$

```
$r := r \bmod q$
$r_0 := r \bmod^{\pm} \alpha$
if $(r - r_0) = q - 1$:
    • $r_1 := 0$
    • $r_0 := r_0 - 1$
else $r_1 := (r - r_0)/\alpha$
output $(r_1, r_0)$
```

Key Generation

Key generation is parametrised with the set of public parameters par, listed in Table 1. Key generation starts with generating $k \times l$ matrix that consists of polynomials from the ring R_q. Matrix $\mathbf{A}$ is the first part of the public key. Then two secret key vectors $(\mathbf{s}_1, \mathbf{s}_2)$ are generated. These vectors consist of polynomials from the ring R_q, the coefficients of polynomials are at most η. Finally, the second part of public key is computed as $\mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$, the result is a vector of polynomials in the ring R_q.

Signing

H_0 is a special hash function defined as a combination of SHAKE-256 and SampleInBall algorithm (Algorithm 2). The signing process starts with generating a masking vector of polynomials $\mathbf{y}$ with coefficients less than γ_1. Then, $\mathbf{A}\mathbf{y}$ is computed, and high order bits are extracted from its coefficients. HighBits_q algorithm is applied to each coefficient of polynomial in the input vector separately. According to the Algorithm 8, every coefficient of polynomial from the vector $\mathbf{w} = \mathbf{A}\mathbf{y}$ can be represented as $w = w_1 \cdot 2\gamma_2 + w_0$, where $|w_0| \leq \gamma_2$, w_1 denotes high-order bits and w_0 denotes low-order bits.

The signing process proceeds with generating challenge polynomial c with exactly τ coefficients that are either -1 or 1 and the rest are 0. Then, the potential signature is computed and rejection sampling is performed. Rejection sampling consists of two checks: the first check is needed for security, the second check is needed for both security and correctness. If at least one of the checks fails, the signing process starts again from the beginning. While loop is repeated until both conditions in the rejection sampling are satisfied.

Verification

Verifier reconstructs the high order bits of $\mathbf{w}$. Then, the verifier checks the infinity norm of the signature vector $\mathbf{z}$ and checks whether c (from the signature) is indeed the hash of $\mathbf{w}'_1$ and the message m.

A more formal definition of the key generation, signing, and verification is presented in Algorithm 9, Algorithm 11, Algorithm 10.

Algorithm 9 KeyGen(par)

```
$\mathbf{A} \leftarrow R_q^{k \times l}$
$(\mathrm{s}_1, \mathrm{s}_2) \leftarrow S_\eta^l \times S_\eta^k$
$\mathbf{t} := \mathbf{A}\mathrm{s}_1 + \mathrm{s}_2$
return $pk = (\mathbf{A}, \mathbf{t}), sk = (\mathbf{A}, \mathbf{t}, \mathrm{s}_1, \mathrm{s}_2)$
```

Algorithm 10 Verify(pk, m, σ)

```
$\mathbf{w}_1' := \mathrm{HighBits}_q(\mathbf{Az} - c\mathbf{t}, 2\gamma_2)$
if $c = H_0(m||\mathbf{w}_1')$ and $||\mathbf{z}||_\infty \geq \gamma_1 - \beta_2$, then return 1 (success).
else: return 0.
```

Algorithm 11 Sign(sk, m)

```
$\mathbf{z} := \perp$
while $\mathbf{z} = \perp$ do:
    $\mathbf{y} \leftarrow S_{\gamma_1 - 1}^l$
    $\mathbf{w}_1 := \mathrm{HighBits}_q(\mathbf{Ay}, 2\gamma_2)$
    $c := H_0(m||\mathbf{w}_1) \in B_\tau$
    $\mathbf{z} := \mathbf{y} + c\mathrm{s}_1$
    if $||\mathbf{z}||_\infty \geq \gamma_1 - \beta$ or $||\mathrm{LowBits}_q(\mathbf{w} - c\mathrm{s}_2, 2\gamma_2)||_\infty \geq \gamma_2 - \beta$, then $\mathbf{z} := \perp$
return $\sigma = (\mathbf{z}, c)$
```

Correctness

Since $\mathbf{w} = \mathbf{Ay}$, $\mathbf{t} = \mathbf{A}\mathrm{s}_1 + \mathrm{s}_2$ and $\mathbf{z} = \mathbf{y} + c\mathrm{s}_1$ it holds that:

$$\mathbf{Az} - c\mathbf{t} = \mathbf{A}(\mathbf{y} + c\mathrm{s}_1) - c(\mathbf{A}\mathrm{s}_1 + \mathrm{s}_2) = \mathbf{Ay} + \mathbf{A}c\mathrm{s}_1 - c\mathbf{A}\mathrm{s}_1 - c\mathrm{s}_2 = \mathbf{Ay} - c\mathrm{s}_2.$$

It means that the following should hold for a valid signature: $\mathrm{HighBits}_q(\mathbf{Ay}, 2\gamma_2) = \mathrm{HighBits}_q(\mathbf{Ay} - c\mathrm{s}_2, 2\gamma_2)$. During rejection sampling it was checked that $||\mathrm{LowBits}_q(\mathbf{w} - c\mathrm{s}_2, 2\gamma_2)||_\infty < \gamma_2 - \beta$. It means that coefficients of $c\mathrm{s}_2$ are small enough and do not cause bit carry. Therefore, if signature was generated correctly, it will successfully pass the verification.

The signature scheme is more similar to the Crystals-Dilithium signature scheme, which is submitted to the NIST PQC competition, than the 1st basic scheme. This is an optimised version of the 1st basic scheme and has smaller signatures. As a result of the changes in the scheme, its security relies on two hard problems (Module-LWE and Module-SIS). Thus, it is more complicated to choose the correct parameters for the scheme, as parameters should be chosen such that both problems are hard to solve. Additionally, the scheme uses a bit decomposition protocol. The process of designing a two-party protocol in this work started from this signature scheme. However, it appeared that due to the use of bit decomposition protocol it is hard to convert this signature scheme into a two-party version.

6 Designing Two-party Protocol

This section describes how the two-party protocol was designed. In order to show the process of designing the protocol and explain the problems that were faced in the process, three versions of the protocol are described in the following sections.

6.1 Version 1

This section describes the first version of the two-party signature protocol that was created. Firstly, three protocols (KeyGen(), Sign(), Verify()) are defined. Additionally, problems and limitations of the first version are explained. The following protocol was designed based on the 2nd basic signature scheme (defined in the section 5.2). As it is a simplified version of the signature scheme submitted to the NIST PQC competition it was decided to start the designing process based on this version. The initial idea was to design a simpler protocol and then add additional functionality needed for optimisation.

The table 2 describes parameters that are used in the two-party signature scheme.

Table 2: Parameters for the two-party protocol

Parameter	Description
n	degree bound of the polynomials in the ring
q	modulus
(k, l)	dimension of matrix and vectors used in the scheme
γ_1	size bound of the coefficients in the masking vector share, that is generated in the signing process
γ_2	low-order rounding range
γ_3	size bound of coefficients in the composed masking vector
η	size bound of coefficients in the secret key share
β	maximum possible coefficient of the client's and server's shares of $c\mathbf{s}_i$, where $i \in \{1, 2\}$
β_2	maximum possible coefficient of $c\mathbf{s}_i$, where $i \in \{1, 2\}$
τ	number on non-zero elements in the output of a special hash function H_0

6.1.1 Signature scheme

Key generation

The client starts with generating a public matrix $\mathbf{A}$, sampling two secret vectors ($\mathbf{s}_1^c$, $\mathbf{s}_2^c$) for its secret key, and generating a public key share $\mathbf{t}_c$. Then public matrix is sent to the server and the server can generate its secret key share ($\mathbf{s}_1^s$, $\mathbf{s}_2^s$) and a public key share $\mathbf{t}_s$. Finally, both public key shares are added together to create a composed public key $\mathbf{t}$. A more formal definition of the key generation protocol is presented in Protocol 1.

Signing

The client's inputs to the signing protocol are its secret key share sk_c and a message m to be signed. The server's input is its secret key share sk_s. CRH is a collision-resistant hash function. H_0 is a special hash function defined as a combination of SHAKE-256 and SampleInBall algorithm (Algorithm 2) that outputs a vector in $\{-1, 0, 1\}^n$ with exactly τ nonzero elements.

The client starts the signing process by generating a masking vector $\mathbf{y}$, deriving a challenge c using a message m and $\text{HighBits}_q(\mathbf{Ay}, 2\gamma_2)$. Then the client computes its signature share as $\mathbf{z}_c := \mathbf{y} + c\mathbf{s}_1^c$ that looks exactly like the signature from the 2nd basic scheme. Client proceeds by computing $\mathbf{v}_c = \mathbf{w} - c\mathbf{s}_2^c$ and performing rejection sampling on the signature share. The client must perform rejection sampling before sending signature share to the server, otherwise, the signature share may leak information about the client's secret key. Finally, client computes hash $b_c = \text{CRH}(\mathbf{w}_1)$ and sends message $(\mathbf{z}_c, c, \mathbf{v}_c, b_c)$ to the server. It can be seen that for now, the signature does not contain the server's secret key. Therefore, the client cannot output signature share as the final signature.

Upon receiving the client's message, the server computes its signature share but differently compared to the client. The server adds $c\mathbf{s}_1^s$ to the client's signature share, resulting signature $\mathbf{z}$ now contains everything like the original signature in the 2nd basic scheme $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$. Then the server performs rejection sampling on the composed signature, and if the signature passes the rejection sampling, then the server outputs it as a final signature.

For the correctness, in the original scheme it was required that $\text{HighBits}_q(\mathbf{Ay}, 2\gamma_2) = \text{HighBits}_q(\mathbf{Ay} - c\mathbf{s}_2, 2\gamma_2)$. In other words, it is needed to check that component $c\mathbf{s}_2$ has small enough coefficients for successful signature verification. As in two party version of the scheme $\mathbf{s}_2$ is shared between client and server this check is not straightforward, the following approach is proposed:

- Client computes a hash of high-order bits of $\mathbf{Ay}$ and sends it to the server along with $\mathbf{v}_c$ =($\mathbf{Ay} - c\mathbf{s}_2^c$).
- Server computes $(\mathbf{Ay} - c\mathbf{s}_2)$ as $\mathbf{v}_c - c\mathbf{s}_2^s$. Then the server computes a hash of $\text{HighBits}_q(\mathbf{Ay} - c\mathbf{s}_2)$ and compares the result with the value received from the client.

If the high-order bits are not the same they will differ only a little, as a collision-resistant hash function is used for this step different high-order bits are not likely to result in the same hash. If at least one of the rejection samplings fails, then both client and server should start signing process from the beginning. The number of rejections depends on the exact parameter selection. A more formal definition of the signing protocol is presented in Protocol 2.

Verification

The verification process is the same as in the original signature scheme. A formal definition of the verification algorithm is presented in Algorithm 12.

Correctness

Since $\mathbf{w} = \mathbf{A}\mathbf{y}$, $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ and $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$ it holds that:

$$\mathbf{A}\mathbf{z} - c\mathbf{t} = \mathbf{A}(\mathbf{y} + c\mathbf{s}_1) - c(\mathbf{A}\mathbf{s}_1 + \mathbf{s}_2) = \mathbf{A}(\mathbf{y} + c(\mathbf{s}_1^c + \mathbf{s}_1^s)) - c(\mathbf{A}(\mathbf{s}_1^c + \mathbf{s}_1^s) + (\mathbf{s}_2^c + \mathbf{s}_2^s)) = \mathbf{A}\mathbf{y} + \mathbf{A}c\mathbf{s}_1^c + \mathbf{A}c\mathbf{s}_1^s - c\mathbf{A}\mathbf{s}_1^c - c\mathbf{A}\mathbf{s}_1^s - c\mathbf{s}_2^c - c\mathbf{s}_2^s = \mathbf{A}\mathbf{y} - c\mathbf{s}_2^c - c\mathbf{s}_2^s = \mathbf{A}\mathbf{y} - c\mathbf{s}_2 = \mathbf{w} - c\mathbf{s}_2.$$

Carefully chosen parameters and rejection samplings help to guarantee that the following holds: HighBits($\mathbf{A}\mathbf{y}, 2\gamma_2$) = HighBits($\mathbf{A}\mathbf{y} - c\mathbf{s}_2, 2\gamma_2$).

Protocol 1: KeyGen()

1. **Client's first message**:
   (a) Generate matrix $\mathbf{A} \leftarrow R_q^{k \times l}$.
   (b) Sample two shares of secret vectors: $(\mathbf{s}_1^c, \mathbf{s}_2^c) \leftarrow S_\eta^l \times S_\eta^k$.
   (c) Compute share of public key: $\mathbf{t}_c := \mathbf{A}\mathbf{s}_1^c + \mathbf{s}_2^c$.
   (d) Send $(\mathbf{A}, \mathbf{t}_c)$ to the server.
2. **Server's first message**
   (a) Sample two shares of secret vectors: $(\mathbf{s}_1^s, \mathbf{s}_2^s) \leftarrow S_\eta^l \times S_\eta^k$.
   (b) Upon receiving client's first message, compute share of public key: $\mathbf{t}_s := \mathbf{A}\mathbf{s}_1^s + \mathbf{s}_2^s$.
   (c) Compute shared public key: $\mathbf{t} := \mathbf{t}_c + \mathbf{t}_s$.
   (d) Send out shared public key $pk = (\mathbf{A}, \mathbf{t})$.
3. **Output**
   (a) Client's share of secret key $sk_c = (\mathbf{A}, \mathbf{s}_1^c, \mathbf{s}_2^c)$.
   (b) Server's share of secret key $sk_s = (\mathbf{A}, \mathbf{s}_1^s, \mathbf{s}_2^s)$.
   (c) Shared public key $pk = (\mathbf{A}, \mathbf{t})$.

Protocol 2: Sign(m)

1. **Client's first message**:
   - (a) Sample a masking vector of polynomials $\mathbf{y} \leftarrow S^l_{\gamma_1 - 1}$.
   - (b) Compute $\mathbf{w} := \mathbf{A}\mathbf{y}$ and extract high-order bits: $\mathbf{w}_1$:= HighBits$_q(\mathbf{w}, 2\gamma_2)$.
   - (c) Compute challenge polynomial $c \in B_\tau := H_0(m||\mathbf{w}_1)$.
   - (d) Compute potential signature share $\mathbf{z}_c := \mathbf{y} + c\mathbf{s}^c_1$.
   - (e) Break up $\mathbf{v}_c = (\mathbf{w} - c\mathbf{s}^c_2)$ into high-order and low-order bits: $(\mathbf{r}_1, \mathbf{r}_0)$:= Decompose$_q(\mathbf{w} - c\mathbf{s}^c_2, 2\gamma_2)$.
   - (f) Perform rejection sampling step:
     if $||\mathbf{z}_c||_\infty \geq \gamma_1 - \beta$ or $||\mathbf{r}_0||_\infty \geq \gamma_2 - \beta$, then $\mathbf{z} :=\perp$
     else:
   - (g) Compute hash of high-order bits of $\mathbf{w}$: b_c := CRH($\mathbf{w}_1$).
   - (h) Send out $(\mathbf{z}_c, c, \mathbf{v}_c, b_c)$.
2. **Server's first message**
   - (a) Upon receiving client's first message, compute $\mathbf{v} := \mathbf{v}_c - c\mathbf{s}^s_2$ and corresponding high order bits $\mathbf{v}_1$:= HighBits$_q(\mathbf{v}, 2\gamma_2)$.
   - (b) Compute hash b_s := CRH($\mathbf{v}_1$).
   - (c) Compute signature share $\mathbf{z}_s := c\mathbf{s}^s_1$.
   - (d) Compute potential composed signature $\mathbf{z} := \mathbf{z}_c + \mathbf{z}_s$.
   - (e) Perform rejection sampling step:
     if $||\mathbf{z}||_\infty \geq \gamma_3 - \beta_2$ or $b_c \neq b_s$, then restart signing process
     else:
   - (f) Output the final signature on message m: $\sigma = (\mathbf{z}, c)$.

Algorithm 12 Verify(pk, σ, m)

```
Compute $\mathbf{w}'_1$ := HighBits$_q(\mathbf{A}\mathbf{z} - c\mathbf{t}, 2\gamma_2)$.
if $c = H_0(m||\mathbf{w}'_1)$ and $||\mathbf{z}||_\infty < \gamma_3 - \beta_2$: return 1 (success).
else: return 0.
```

6.1.2 Analysis of version 1

When the first version of the protocol was designed several major problems were found during its analysis. Firstly, after the shared public key is output by the server, the client cannot be sure that its share was indeed used to compute $\mathbf{t}$. This would allow a malicious server to use only its public key share $\mathbf{t}_s$ to compute the final public key $\mathbf{t}$. Therefore, the server would be able to create signatures on behalf of the client using only the server's secret key share. These signatures will be successfully verified against the public key because it contains only the server's share.

Secondly, after seeing the client's public key share the server would be able to adaptively choose a malicious public key share. This again may lead to the scenario where the server can create valid signatures on behalf of the client using only its secret key share. To solve both problems, in the second version of the scheme, commitments are introduced to the key generation protocol.

Additionally, there are several problems with the signing protocol that would allow the client to learn the secret key share of the server. While setting up parameters for the scheme, prime modulus q may be chosen such that in the underlying ring R_q all elements with coefficients less than $\sqrt{q/2}$ have an inverse [36]. Therefore, after seeing the final signature $\mathbf{z}$ and knowing $\mathbf{z}_c$ and c, the client would be able to reconstruct the server's secret key share. To solve this issue, in the second version of the scheme, both client's and server's shares are used to create a masking vector $\mathbf{y}$ and a value $\mathbf{w}$. As a result, client's and server's signature shares will be masked by randomly chosen vectors $\mathbf{y}_c$ and $\mathbf{y}_s$ respectively.

6.2 Version 2

This section describes an improved version of the first distributed signature protocol. Firstly, distributed signature scheme consisting of three protocols (KeyGen(), Sign(), Verify()) is defined. Then the problems and limitations of the second version are described. The parameters for this version stay the same as described in Table 2.

6.2.1 Signature scheme

Key generation

H_1 is some collision resistant hash functions. The client starts with generating a public matrix $\mathbf{A}$ and proceeds by generating two secret vectors ($\mathbf{s}_1^c$, $\mathbf{s}_2^c$) and computing its share of the public key $\mathbf{t}_c$. Instead of publishing public key share at this point, the client computes and sends commitment $com_c = H_1(\mathbf{t}_c)$ together with the public matrix $\mathbf{A}$. Hiding property of commitment guarantees that, when the server sees com_c it will not be able to adaptively choose a malicious share of public key $\mathbf{t}_s$ as the server will not be able to see what is inside the commitment.

The server samples two secret vectors ($\mathbf{s}_1^s$, $\mathbf{s}_2^s$) and upon receiving $\mathbf{A}$ generates its public key share $\mathbf{t}_s$. Then the server sends a commitment com_s to the public key share to the client. Once the client and server received commitments from each other, they exchange their public key shares. Then they both locally check if the commitments were opened correctly. If these checks succeed, then they both locally compute the final public key $\mathbf{t}$.

A more formal definition of the key generation protocol is presented in Protocol 3.

Signing

The client's inputs to the signing protocol are its secret key share sk_c and a message m to be signed. The server's input is its secret key share sk_s. H_2 is a collision-resistant hash function. H_0 is a special hash function defined as a combination of SHAKE-256 and SampleInBall algorithm (Algorithm 2) that outputs a vector in $\{-1, 0, 1\}^n$ with exactly τ nonzero elements.

The client starts the signing process by generating its share of masking vector $\mathbf{y}_c$. Then the client computes $\mathbf{w}_c = \mathbf{A}\mathbf{y}_c$ as its share of $\mathbf{w}$ and sends commitment to it com_c. The server, in turn, generates its share of masking vector $\mathbf{y}_s$ and sends commitment com_s to its share of $\mathbf{w}$. After receiving commitments from each other, the client and server open the commitments, by exchanging the shares of $\mathbf{w}$.

The client proceeds by checking if the server opened its commitment correctly. If the check succeeds, the client computes $\mathbf{w} = \mathbf{w}_c + \mathbf{w}_s$, extracts its high-order bits, and derives challenge c. Then the client computes its potential signature share $\mathbf{z}_c$ and performs rejection sampling. If all the conditions in rejection sampling were satisfied, the client sends its signature share to the server.

Server checks if the client opened its commitment correctly. If the check succeeds, the server computes composed $\mathbf{w}$, extracts its high order bits, and derives challenge c. Then the server computes its potential signature share $\mathbf{z}_s$ and performs rejection sampling. If all the conditions in rejection sampling were satisfied, then the server proceeds by computing the composed signature (once the client's share was received). Then server performs rejection sampling on the composed signature and if it passes, the server outputs the final signature.

It is important to note that if at least one of the rejection samplings fails, then both client and server should start their signing process from the beginning. The number of rejections depends on the exact parameter selection.

It can be seen that now both client's and server's shares of the signature are constructed similarly to the 2nd basic scheme. Therefore, to ensure the correctness of the protocol it is enough to perform rejection sampling on the signature shares and the composed signature. A more formal definition of the signing protocol is presented in Protocol 4.

Verification

The verification process is the same as in the original Dilithium scheme. A formal definition of the verification algorithm is presented in Algorithm 12.

Correctness

Since $\mathbf{w} = \mathbf{A}\mathbf{y}$, $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ and $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$ it holds that:

$\mathbf{w} - c\mathbf{s}_2 = (\mathbf{w}_c + \mathbf{w}_s) - c(\mathbf{s}_2^c + \mathbf{s}_2^s) = (\mathbf{A}\mathbf{y}_c + \mathbf{A}\mathbf{y}_s) - c(\mathbf{s}_2^c + \mathbf{s}_2^s) = \mathbf{A}(\mathbf{z}_c - c\mathbf{s}_1^c) + \mathbf{A}(\mathbf{z}_s - c\mathbf{s}_1^s) - c\mathbf{s}_2^c - c\mathbf{s}_2^s = \mathbf{A}\mathbf{z}_c - \mathbf{A}c\mathbf{s}_1^c + \mathbf{A}\mathbf{z}_s - \mathbf{A}c\mathbf{s}_1^s - c\mathbf{s}_2^c - c\mathbf{s}_2^s = \mathbf{A}\mathbf{z}_c - c(\mathbf{A}\mathbf{s}_1^c + \mathbf{s}_2^c) + \mathbf{A}\mathbf{z}_s - c(\mathbf{A}\mathbf{s}_1^s + \mathbf{s}_2^s) = \mathbf{A}\mathbf{z}_c - c\mathbf{t}_c + \mathbf{A}\mathbf{z}_s - c\mathbf{t}_s = \mathbf{A}(\mathbf{z}_c + \mathbf{z}_s) - c(\mathbf{t}_c + \mathbf{t}_s) = \mathbf{A}\mathbf{z} - c\mathbf{t}$.

Carefully chosen parameters and all the rejection samplings steps help to guarantee that HighBits($\mathbf{A}\mathbf{y}, 2\gamma_2$) = HighBits($\mathbf{A}\mathbf{y} - c\mathbf{s}_2, 2\gamma_2$).

Protocol 3: KeyGen()

1. **Client's first message**:
   (a) Generate matrix $\mathbf{A} \leftarrow R_q^{k \times l}$.
   (b) Sample two shares of secret vectors: $(\mathbf{s}_1^c, \mathbf{s}_2^c) \leftarrow S_\eta^l \times S_\eta^k$.
   (c) Compute share of public key: $\mathbf{t}_c := \mathbf{A}\mathbf{s}_1^c + \mathbf{s}_2^c$.
   (d) Send $(\mathbf{A}, com_c = H_1(\mathbf{t}_c))$ to the server.
2. **Server's first message**
   (a) Sample two shares of secret vectors: $(\mathbf{s}_1^s, \mathbf{s}_2^s) \leftarrow S_\eta^l \times S_\eta^k$.
   (b) Upon receiving client's first message, compute share of public key: $\mathbf{t}_s := \mathbf{A}\mathbf{s}_1^s + \mathbf{s}_2^s$.
   (c) Send $com_s = H_1(\mathbf{t}_s)$ to the client.
3. **Client's second message**
   (a) Upon receiving com_s, send out $\mathbf{t}_c$.
4. **Server's second message**
   (a) Upon receiving com_c, send out $\mathbf{t}_s$.
5. **Client's verification**
   (a) Upon receiving $\mathbf{t}_s$, check if $H_1(\mathbf{t}_s) = com_s$. Send out ABORT message if the check fails.
   (b) Compute shared public key: $\mathbf{t} := \mathbf{t}_c + \mathbf{t}_s$.
6. **Server's verification**
   (a) Upon receiving $\mathbf{t}_c$, check if $H_1(\mathbf{t}_c) = com_c$. Send out ABORT message if the check fails.
   (b) Compute shared public key: $\mathbf{t} := \mathbf{t}_c + \mathbf{t}_s$.
7. **Output**
   (a) Client's share of secret key $sk_c = (\mathbf{A}, \mathbf{s}_1^c, \mathbf{s}_2^c)$.
   (b) Server's share of secret key $sk_s = (\mathbf{A}, \mathbf{s}_1^s, \mathbf{s}_2^s)$.
   (c) Shared public key $pk = (\mathbf{A}, \mathbf{t})$.

Protocol 4: Sign(m)

1. **Client's first message**
   - (a) Sample a masking vector of polynomials $\mathbf{y}_c \leftarrow S^l_{\gamma_1 - 1}$.
   - (b) Compute $\mathbf{w}_c := \mathbf{A}\mathbf{y}_c$. Send $com_c = H_2(\mathbf{w}_c)$ to the server.
2. **Server's first message**
   - (a) Sample a masking vector of polynomials $\mathbf{y}_s \leftarrow S^l_{\gamma_1 - 1}$.
   - (b) Compute $\mathbf{w}_s := \mathbf{A}\mathbf{y}_s$. Send $com_s = H_2(\mathbf{w}_s)$ to the client.
3. **Client's second message**
   - (a) Upon receiving first message from the server, send $\mathbf{w}_c, m$ to the server.
4. **Client's second message**
   - (a) Upon receiving first message from the client, send $\mathbf{w}_s$ to the client.
5. **Client's third message**
   - (a) Upon receiving server's share $\mathbf{w}_s$, check if $H_2(\mathbf{w}_s) = h_s$. Send out ABORT message if the equality does not hold.
   - (b) Compute $\mathbf{w} = \mathbf{w}_c + \mathbf{w}_s$ and extract high-order bits: $\mathbf{w}_1 := \text{HighBits}_q(\mathbf{w}, 2\gamma_2)$.
   - (c) Compute challenge polynomial $c \in B_\tau := H_0(m||\mathbf{w}_1)$.
   - (d) Compute potential signature share $\mathbf{z}_c := \mathbf{y}_c + c\mathbf{s}_1^c$.
   - (e) Perform rejection sampling step:

     if $||\mathbf{z}_c||_\infty \geq \gamma_1 - \beta$ or $||\text{LowBits}_q(\mathbf{w} - c\mathbf{s}_2^c, 2\gamma_2)||_\infty \geq \gamma_2 - \beta$, then send out RESTART message.

     else:
   - (f) Send $\mathbf{z}_c$ to server.
6. **Server's third message**
   - (a) Upon receiving client's share $\mathbf{w}_c$, check if $H_2(\mathbf{w}_c) = h_c$. Send out ABORT message if the equality does not hold.
   - (b) Compute $\mathbf{w} = \mathbf{w}_c + \mathbf{w}_s$ and extract high-order bits: $\mathbf{w}_1 := \text{HighBits}_q(\mathbf{w}, 2\gamma_2)$.
   - (c) Compute challenge polynomial $c \in B_\tau := H_0(m||\mathbf{w}_1)$.
   - (d) Compute potential signature share $\mathbf{z}_s := \mathbf{y}_s + c\mathbf{s}_1^s$.
   - (e) Perform rejection sampling step:

     if $||\mathbf{z}_s||_\infty \geq \gamma_1 - \beta$ or $||\text{LowBits}_q(\mathbf{w} - c\mathbf{s}_2^s, 2\gamma_2)||_\infty \geq \gamma_2 - \beta$, then send out RESTART message.

     else:
   - (f) Upon receiving client's signature share, compute composed signature $\mathbf{z} = \mathbf{z}_c + \mathbf{z}_s$.
   - (g) Perform rejection sampling on composed signature

     if $||\mathbf{z}||_\infty \geq \gamma_3 - \beta_2$ or $||\text{LowBits}_q(\mathbf{A}\mathbf{z} - c\mathbf{t}, 2\gamma_2)||_\infty \geq \gamma_2 - \beta_2$, then then send out RESTART message.

     else:
   - (h) Output the final signature on message m: $\sigma = (\mathbf{z}, c)$.
7. **Upon receiving RESTART message**
   - (a) Client and server start signing process again from the beginning.

Algorithm 13 Verify(pk, σ, m)

```
Compute $\mathbf{w}_1' := \text{HighBits}_q(\mathbf{Az} - c\mathbf{t}, 2\gamma_2)$.
if $c = H_0(m||\mathbf{w}_1')$ and $||\mathbf{z}||_\infty \geq \gamma_3 - \beta_2$: return 1 (success).
else: return 0.
```

6.2.2 Analysis of version 2

There is still one more problem with the key generation protocol. Currently, the public matrix $\mathbf{A}$ is generated fully by the client. This allows the client to choose a malicious matrix that will leak information about the server's secret key once the server's public key share gets published. Therefore, in the next version of the scheme matrix $\mathbf{A}$ is generated jointly by client and server. Additionally, before exchanging the shares of the matrix $\mathbf{A}$, client and server need to exchange commitments of these shares. This helps to prevent choosing adaptively a malicious matrix share after seeing other party's matrix share.

The main problem that occurred during the analysis of this version of the scheme is connected to revealing $\mathbf{w}$. It can be seen that by the end of the protocol both parties obtain $\mathbf{w}$. By correctness, it holds that $\mathbf{Az} - c\mathbf{t} = \mathbf{w} - c\mathbf{s}_2$, both parties are able to reconstruct shared secret key $\mathbf{s}_2$ and use it to create signatures without communicating to each other. This means that the values $\mathbf{w}$ should not be revealed during the protocol. Additionally, HighBits algorithm is not homomorphic, that is, it does not hold that

$$\text{HighBits}_q(\mathbf{w}_c, 2\gamma_2) + \text{HighBits}_q(\mathbf{w}_s, 2\gamma_2) = \text{HighBits}_q(\mathbf{w}_c \oplus \mathbf{w}_s, 2\gamma_2),$$

where "+" is operation on the outputs of HighBits an algorithm and "$\oplus$" is an operation on the inputs. Homomorphism does not hold as during the addition of HighBits outputs, bit carry may occur.

Therefore, to solve the problem of deriving high order bits of $\mathbf{w}$ a special two-party protocol is needed. Possible solutions include:

- a two-party protocol to compute the high order bits of $\mathbf{w}_1 + \mathbf{w}_2$ without revealing the inputs.
- a two-party protocol for integer comparison to make sure low bits are in the right range and there will be no bit carry if parties compute

$$\text{HighBits}_q(\mathbf{w}_1, 2\gamma_2) + \text{HighBits}_q(\mathbf{w}_2, 2\gamma_2).$$

As the goal of this work is to construct a post-quantum scheme, the choice of a two-party protocol is also limited to the quantum-secure techniques. Currently, there were no good options found that could satisfy all the requirements needed for this scheme. Therefore, in the next version of the protocol HighBits algorithm is not used anymore as the underlying signature scheme is changed to the 1st basic scheme.

Finally, even if a suitable two-party protocol is found, the amount of communication between parties will increase. Keeping in mind that there are also rejection samplings that require repeating the signing process several times, the amount of communication needed to create a single signature may not be very practical.

6.3 Version 3 (final version)

It was decided to switch the underlying digital signature scheme to the 1st basic signature to create a post-quantum two-party signature scheme. The reason for this decision is that it was concluded that there are no straightforward approaches to modify the signature scheme submitted to the NIST PQC competition to the distributed version. Solutions will require using a two-party computation protocol, which will increase not only the signing time but the communication complexity as well. The 1st basic scheme is easier to work with because there are no additional algorithms like HighBits, LowBits used, and its security relies only on one problem – Module-LWE.

As was mentioned earlier, for the security proofs of the schemes based on the FSwA technique, a difficult question to answer is whether intermediate computations can be published before the rejection sampling has been accomplished. Therefore, it is important to make sure that the intermediate values containing secret information are not published before the rejection sampling step. A solution to this problem was proposed in the paper [9], where homomorphic commitments are used and the values inside the commitments are opened only after the successful rejection sampling. Current work follows the approach from [9], however, instead of homomorphic commitments, a homomorphic hash function is used.

6.3.1 SWIFFT hash function

It was decided to use a homomorphic hash function proposed in [41] that is called SWIFFT. This section starts with defining cryptographic and statistical properties that are desired from some of the hash functions. The following definitions are adopted from [41].

Definition 29. A hash function $f : X \to R$ is **one-way** if given the output of hash function $y = f(x)$, where $x \in X$ is chosen uniformly at random, the probability that adversary finds $x' \in X$ such that $f(x') = y$ is negligible.

Advantage of adversary $\mathcal{A}$ in breaking one-wayness of a hash function $f : X \to R$ can be defined as follows:

$$\text{Adv}^{\text{OW}}(\mathcal{A}) := \Pr[f(x) = f(x') : x \leftarrow X, y \leftarrow f(x), x' \leftarrow \mathcal{A}(y)].$$

Definition 30. A hash function $f : X \to R$ is **second preimage resistant** if given a uniformly random input $x \in X$ and corresponding output $y = f(x)$, the probability that adversary finds $x' \in X$ such that $x' \neq x$ and $f(x') = f(x)$ is negligible.

Additionally, there are some statistical properties, the following definitions are adopted from [42, 41]:

Definition 31. Let $\mathcal{F} = \{f_a\}_{a \in A}$, where $f_a : X \to R$ be a collection of functions indexed by a set A. A family of hash functions $\mathcal{F}$ is called **ϵ-regular** if the statistical distance between its output distribution $\{(a, f_a(x)) : a \leftarrow A, x \leftarrow X\}$ and the uniform distribution $\{(a, r) : a \leftarrow A, r \leftarrow R\}$ is at most ϵ.

Definition 32. A hash function $f : X \to R$ is called **ϵ-regular** if the statistical distance between its output distribution $\{f(x) : x \leftarrow X\}$ and the uniform distribution over the range $\{r : r \leftarrow R\}$ is at most ϵ.

Definition 33. A family of functions is called **universal** if for any fixed $x \neq x'$ the probability that $f(x) = f(x')$ is $1/|R|$, where $f : X \to R$ is chosen randomly from the family and $|R|$ is the size of the range.

Definition 34. A family of functions is said to be **randomness extractor** if for an input $x \in X$ taken from a weak randomness source, function $f : X \to R$ produces an output distributed uniformly in the range (or as close to uniform as possible), where f is chosen randomly from the family.

SWIFFT is a collection of compression functions that are provably one-way and collision-resistant [41]. SWIFFT compression functions are a special case of function proposed in [43], [42], [44] that achieves more practical and efficient implementation.

The SWIFFT hashing algorithm takes as an input a binary string of length $a \cdot b$ that is then interpreted as $b \times a$ binary matrix. SWIFFT uses the FFT as one of the main building blocks that helps to achieve diffusion (to mix all the input bits). A linear combination is performed over $\mathbb{Z}_p$ (for a suitable modulus p) and helps to achieve compression and confusion. The output is a vector of length b that belongs to $\mathbb{Z}_p^b$.

Additionally, SWIFFT has several statistical properties that can be proven unconditionally: universal hashing, regularity, and randomness extraction. Unconditional security means that there were no assumptions used about the computational power and resources available for the adversary [45].

The provable security features of the SWIFFT functions are equivalent to the ones presented in [43], [42]. Therefore, according to the proofs from [43], [42] SWIFFT family of compression functions is provably collision-resistant and one-way under the assumption about the worst-case difficulty of finding short vectors on a certain kind of lattices. However, due to the linearity SWIFFT functions are not pseudorandom. This means that if an adversary is given oracle access to a function f, it can efficiently distinguish between the two cases:

- case 1: f is chosen at random from the function family.
- case 2: every output of f is uniformly random and independent of other outputs.

It follows that the function is not a suitable instantiation of a random oracle [41]. Therefore, in the security proofs of the two-party signature protocol, SWIFFT is not used as a random oracle. Security proof makes use of such provable properties as one-wayness, regularity, and collision-resistance.

The function was designed to achieve a very efficient implementation since it is highly parallelizable. SWIFFT was implemented and tested on a 3.2GHz Intel Pentium 4. The implementation was written in C and compiled using a PC running under Linux kernel 2.6.18. Tests showed that the basic compression function could be evaluated in 1.5 µs on the above system, achieving a throughput close to 40 MB/s in a standard chaining mode of operation. The efficiency of implementation was compared with SHA256 on the same system that achieves a throughput of 47 MB/s when run on 8KB blocks.

6.3.2 Signature scheme

The table 3 describes parameters that are used in the version 3 of two-party signature scheme.

Parameter setup

Assume that before starting the key generation and signing protocols, parties invoke a Setup(1^λ) function that based on the security parameter λ outputs a set of public parameters par that are described in Table 3.

Key generation

H_1 and H_2 are some collision resistant hash functions. The client begins the key generation process by sampling a share of matrix $\mathbf{A}_c$ and sending out the commitment to this

Table 3: Parameters for the two-party protocol

Parameter	Description
n	degree bound of the polynomials in the ring
q	modulus
(k, k)	dimension of matrix and vectors used in the scheme
γ	size bound of the coefficients in the masking vector share, that is generated in the signing process
γ_2	size bound of coefficients in the composed masking vector
η	size bound of coefficients in the secret key share
τ	number on non-zero elements in the output of special hash function H_0
β	maximum possible coefficient of the client's and server's shares of $c\mathbf{s}_i$, where $i \in \{1, 2\}$
β_2	maximum possible coefficient of $c\mathbf{s}_i$, where $i \in \{1, 2\}$
(a, b, p)	parameters for the homomorphic hash function: $a \cdot b$ is input length, b is output length and p is modulus.

share hk_c. The server generates its matrix share $\mathbf{A}_s$ and sends commitment hk_s to the client. Upon receiving commitments, the client and server exchange matrix shares and check if the openings for the commitments were correct.

Then client proceeds by generating two secret vectors (s_1^c, s_2^c) and computing its share of the public key $\mathbf{t}_c$. The client sends out commitment to the public key share $comk_c$. The server samples its secret vectors (s_1^s, s_2^s) and uses them to compute its public key share $\mathbf{t}_s$. Then the server sends commitment to the public key share $comk_s$ to the client.

Once the client and server received commitments from each other, client and server exchange public key shares. Then client and server both locally check if the commitments were opened correctly. If these checks succeed, then the client and the server locally compute the final public key.

It is needed to include the server's public key share $\mathbf{t}_s$ to the client's secret key sk_c and vice versa. During the signing process the client needs to use the server's public key share to verify the correctness of a commitment.

Figure 6 describes the key generation process and illustrates communication between client and server. Protocol 5 describes two-party key generation in the more formal way.

Signing

Inputs of the client are secret key share sk_c, and a message m to be signed. The server's input is secret key share sk_s. $HomH$ is a homomorphic hash function from the SWIFFT family. H_0 is a special hash function defined as a combination of SHAKE-256 and

SampleInBall algorithm (Algorithm 2) that outputs a vector of length n with exactly τ coefficients being either -1 or 1 and the rest being 0. H_3 is a collision-resistant hash function.

Client starts signing process with generating its shares of masking vectors $(\mathbf{y}_1^c, \mathbf{y}_2^c)$ and computing a share of $\mathbf{w}$. Now, $\mathbf{w}$ is an instance of the Module-LWE problem, therefore it is needed to generate two masking vectors instead of one. Then client uses a homomorphic hash function to compute $com_c = HomH(\mathbf{w}_c)$ and hashes it using some collision-resistant hash function. The composed output of the homomorphic hash function $com = com_c + com_s$ will be later used to derive a challenge. Therefore, it is crucial to ensure that com_c, com_s were not chosen maliciously. Thus, before publishing these shares client and server should exchange commitments to these shares h_c, h_s.

Server, in turn, generates its shares of masking vectors $(\mathbf{y}_1^s, \mathbf{y}_2^s)$ computes its share of $\mathbf{w}$ and sends commitment to the $com_s = HomH(\mathbf{w}_s)$. After receiving commitments h_c, h_s from each other, the client and server open the commitments by sending out shares com_s, com_c.

The client proceeds by checking if the server opened its commitment correctly. If the check succeeds, the client computes $com = com_c + com_s$ and derives challenge c. Then the client computes potential signature shares $(\mathbf{z}_1^c, \mathbf{z}_2^c)$ and performs rejection sampling. If all the conditions in rejection sampling were satisfied, the client sends its signature share to the server.

Server checks if the client opened its commitment correctly. If the check succeeds, the server computes composed com and derives challenge c. Then the server computes its potential signature shares $(\mathbf{z}_1^s, \mathbf{z}_2^s)$ and performs rejection sampling. If all the conditions in rejection sampling were satisfied, the server sends its signature share to the client.

Finally, the client performs verification if com_s indeed contains $\mathbf{w}_s$. Client reconstructs $\mathbf{w}_s$ using $(\mathbf{z}_1^s, \mathbf{z}_2^s)$ and $\mathbf{t}_s$ and checks if it is a valid opening for com_s. If the check succeeds, the client computes the final signature $(\mathbf{z}_1, \mathbf{z}_2)$. The server performs the same verification that com_c indeed contains $\mathbf{w}_c$ using $(\mathbf{z}_1^c, \mathbf{z}_2^c)$ and $\mathbf{t}_c$. If the check succeeds, the server computes and outputs the final signature.

Figure 7 describes the signing process and illustrates communication between client and server. Protocol 6 describes two-party signing process in the more formal way.

Verification

Verification is almost the same as in the original scheme except verifier needs to apply homomorphic hash function on reconstructed $\mathbf{w}'$ in order to check the correctness of challenge. Algorithm 14 describes verification in the more formal way.

Correctness

Since $\mathbf{w} = \mathbf{A}\mathbf{y}_1 + \mathbf{y}_2$, $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$, $\mathbf{z}_1 = \mathbf{y}_1 + c\mathbf{s}_1$ and $\mathbf{z}_2 = \mathbf{y}_2 + c\mathbf{s}_2$ it holds that:

$\mathbf{A}\mathbf{z}_1 + \mathbf{z}_2 - c\mathbf{t} = \mathbf{A}(\mathbf{y}_1 + c\mathbf{s}_1) + (\mathbf{y}_2 + c\mathbf{s}_2) - c(\mathbf{A}\mathbf{s}_1 + \mathbf{s}_2) = \mathbf{A}\mathbf{y}_1 + \mathbf{A}c\mathbf{s}_1 + \mathbf{y}_2 + c\mathbf{s}_2 - c\mathbf{A}\mathbf{s}_1 - c\mathbf{s}_2 = \mathbf{A}\mathbf{y}_1 + \mathbf{y}_2$.

Furthermore, by triangle inequality it holds that if $||\mathbf{z}_1^s||_\infty < \gamma - \beta$ and $||\mathbf{z}_1^c||_\infty < \gamma - \beta$ then $||\mathbf{z}_1||_\infty = ||\mathbf{z}_1^s + \mathbf{z}_1^c||_\infty < ||\mathbf{z}_1^s||_\infty + ||\mathbf{z}_1^c||_\infty = 2\gamma - 2\beta$. The same holds for the second signature component $\mathbf{z}_2$. This means that γ_2 can be defined as $\gamma_2 = 2\gamma$ and $\beta_2 = 2\beta$. Therefore, if signature was generated correctly, verification will always succeed.

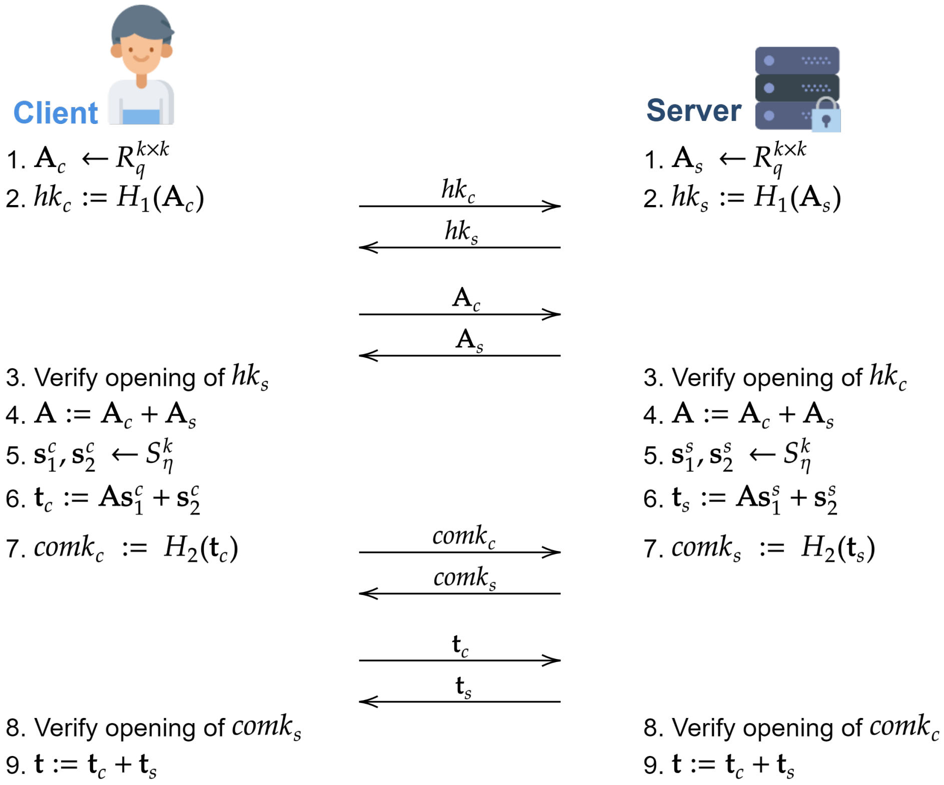
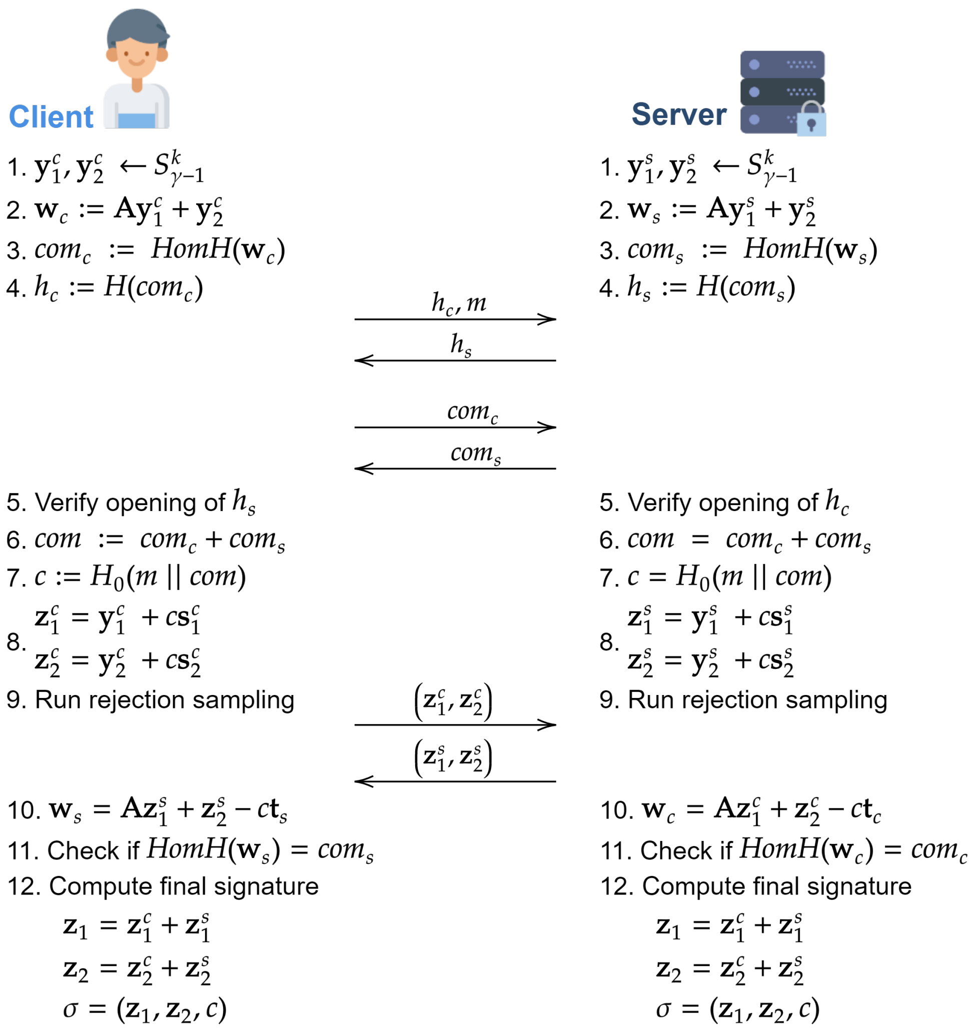

Client
Server
1. $\mathbf{A}_c \leftarrow R_q^{k\times k}$
2. $hk_c := H_1(\mathbf{A}_c)$
1. $\mathbf{A}_s \leftarrow R_q^{k\times k}$
2. $hk_s := H_1(\mathbf{A}_s)$
hk_c
hk_s
$\mathbf{A}_c$
$\mathbf{A}_s$
3. Verify opening of hk_s
4. $\mathbf{A} := \mathbf{A}_c + \mathbf{A}_s$
5. $\mathbf{s}_1^c, \mathbf{s}_2^c \leftarrow S_\eta^k$
6. $\mathbf{t}_c := \mathbf{A}\mathbf{s}_1^c + \mathbf{s}_2^c$
7. $comk_c := H_2(\mathbf{t}_c)$
3. Verify opening of hk_c
4. $\mathbf{A} := \mathbf{A}_c + \mathbf{A}_s$
5. $\mathbf{s}_1^s, \mathbf{s}_2^s \leftarrow S_\eta^k$
6. $\mathbf{t}_s := \mathbf{A}\mathbf{s}_1^s + \mathbf{s}_2^s$
7. $comk_s := H_2(\mathbf{t}_s)$
$comk_c$
$comk_s$
$\mathbf{t}_c$
$\mathbf{t}_s$
8. Verify opening of $comk_s$
9. $\mathbf{t} := \mathbf{t}_c + \mathbf{t}_s$
8. Verify opening of $comk_c$
9. $\mathbf{t} := \mathbf{t}_c + \mathbf{t}_s$

Figure 6: Distributed key generation protocol (final version)



Client

1. $\mathbf{y}_1^c, \mathbf{y}_2^c \leftarrow S_{\gamma-1}^k$
2. $\mathbf{w}_c := \mathbf{A}\mathbf{y}_1^c + \mathbf{y}_2^c$
3. $com_c := HomH(\mathbf{w}_c)$
4. $h_c := H(com_c)$

Server

1. $\mathbf{y}_1^s, \mathbf{y}_2^s \leftarrow S_{\gamma-1}^k$
2. $\mathbf{w}_s := \mathbf{A}\mathbf{y}_1^s + \mathbf{y}_2^s$
3. $com_s := HomH(\mathbf{w}_s)$
4. $h_s := H(com_s)$

$\xrightarrow{h_c, m}$

$\xleftarrow{h_s}$

$\xrightarrow{com_c}$

$\xleftarrow{com_s}$

Client

5. Verify opening of h_s
6. $com := com_c + com_s$
7. $c := H_0(m \,||\, com)$
8. $\mathbf{z}_1^c = \mathbf{y}_1^c + c\mathbf{s}_1^c$, $\mathbf{z}_2^c = \mathbf{y}_2^c + c\mathbf{s}_2^c$
9. Run rejection sampling

Server

5. Verify opening of h_c
6. $com = com_c + com_s$
7. $c = H_0(m \,||\, com)$
8. $\mathbf{z}_1^s = \mathbf{y}_1^s + c\mathbf{s}_1^s$, $\mathbf{z}_2^s = \mathbf{y}_2^s + c\mathbf{s}_2^s$
9. Run rejection sampling

$\xrightarrow{(\mathbf{z}_1^c, \mathbf{z}_2^c)}$

$\xleftarrow{(\mathbf{z}_1^s, \mathbf{z}_2^s)}$

Client

10. $\mathbf{w}_s = \mathbf{A}\mathbf{z}_1^s + \mathbf{z}_2^s - c\mathbf{t}_s$
11. Check if $HomH(\mathbf{w}_s) = com_s$
12. Compute final signature

$$\mathbf{z}_1 = \mathbf{z}_1^c + \mathbf{z}_1^s$$
$$\mathbf{z}_2 = \mathbf{z}_2^c + \mathbf{z}_2^s$$
$$\sigma = (\mathbf{z}_1, \mathbf{z}_2, c)$$

Server

10. $\mathbf{w}_c = \mathbf{A}\mathbf{z}_1^c + \mathbf{z}_2^c - c\mathbf{t}_c$
11. Check if $HomH(\mathbf{w}_c) = com_c$
12. Compute final signature

$$\mathbf{z}_1 = \mathbf{z}_1^c + \mathbf{z}_1^s$$
$$\mathbf{z}_2 = \mathbf{z}_2^c + \mathbf{z}_2^s$$
$$\sigma = (\mathbf{z}_1, \mathbf{z}_2, c)$$

Figure 7: Distributed signing protocol (final version)

Protocol 5: KeyGen(par)

1. **Client's first message**:
   (a) Generate matrix share $\mathbf{A}_c \leftarrow R_q^{k \times k}$ and send $hk_c := H_1(\mathbf{A}_c)$ to the server.
2. **Server's first message**:
   (a) Generate matrix share $\mathbf{A}_s \leftarrow R_q^{k \times k}$ and send $hk_s := H_1(\mathbf{A}_s)$ to the client.
3. **Client's second message**:
   (a) Upon receiving hk_s, send out matrix share $\mathbf{A}_c$.
4. **Server's second message**:
   (a) Upon receiving hk_c, send out matrix share $\mathbf{A}_s$.
5. **Client's third message**
   (a) Upon receiving $\mathbf{A}_s$, verify if $H_1(\mathbf{A}_s) = hk_s$. Send out ABORT message if equality does not hold.
   (b) Compute combined public matrix $\mathbf{A} := \mathbf{A}_c + \mathbf{A}_s$.
   (c) Sample two secret vectors: $\mathbf{s}_1^c, \mathbf{s}_2^c \leftarrow S_\eta^k$.
   (d) Compute share of the public key: $\mathbf{t}_c := \mathbf{A}\mathbf{s}_1^c + \mathbf{s}_2^c$ and send $comk_c := H_2(\mathbf{t}_c)$ to the server.
6. **Server's third message**
   (a) Upon receiving $\mathbf{A}_c$, verify if $H_1(\mathbf{A}_c) = hk_c$. Send out ABORT message if equality does not hold.
   (b) Compute combined public matrix $\mathbf{A} := \mathbf{A}_c + \mathbf{A}_s$.
   (c) Sample two secret vectors: $\mathbf{s}_1^s, \mathbf{s}_2^s \leftarrow S_\eta^k$.
   (d) Compute share of the public key: $\mathbf{t}_s := \mathbf{A}\mathbf{s}_1^s + \mathbf{s}_2^s$ and send $comk_s := H_2(\mathbf{t}_s)$ to the client.
7. **Client's fourth message**
   (a) Upon receiving $comk_s$, send out public key share $\mathbf{t}_c$.
8. **Server's fourth message**
   (a) Upon receiving $comk_c$, send out public key share $\mathbf{t}_s$.
9. **Client's verification**
   (a) Upon receiving $\mathbf{t}_s$, verify if $H_2(\mathbf{t}_s) = comk_s$. Send out ABORT message if equality does not hold.
   (b) Compute combined public key as $\mathbf{t} := \mathbf{t}_c + \mathbf{t}_s$.
10. **Server's verification**
    (a) Upon receiving $\mathbf{t}_c$, verify if $H_2(\mathbf{t}_c) = comk_c$. Send out ABORT message if equality does not hold.
    (b) Compute combined public key as $\mathbf{t} := \mathbf{t}_c + \mathbf{t}_s$.
11. **Output**
    (a) Client's share of secret key $sk_c = (\mathbf{A}, \mathbf{t}_s, \mathbf{s}_1^c, \mathbf{s}_2^c)$.
    (b) Server's share of secret key $sk_s = (\mathbf{A}, \mathbf{t}_c, \mathbf{s}_1^s, \mathbf{s}_2^s)$.
    (c) Shared public key $pk = (\mathbf{A}, \mathbf{t})$.

Protocol 6: Sign(m)

1. **Client's first message**
   (a) Sample two masking vectors of polynomials $\mathbf{y}_1^c, \mathbf{y}_2^c \leftarrow S_{\gamma-1}^k$.
   (b) Compute $\mathbf{w}_c := \mathbf{A}\mathbf{y}_1^c + \mathbf{y}_2^c$.
   (c) Compute $com_c := HomH(\mathbf{w}_c)$ and send out $(h_c := H_3(com_c), m)$.
2. **Server's first message**
   (a) Sample two masking vectors of polynomials $\mathbf{y}_1^s, \mathbf{y}_2^s \leftarrow S_{\gamma-1}^k$.
   (b) Compute $\mathbf{w}_s := \mathbf{A}\mathbf{y}_1^s + \mathbf{y}_2^s$.
   (c) Compute $com_s := HomH(\mathbf{w}_s)$ and send out $h_s := H_3(com_s)$.
3. **Client's second message**
   (a) Upon receiving h_s, send out com_c.
4. **Server's second message**
   (a) Upon receiving h_c, send out com_s.
5. **Client's third message**
   (a) Upon receiving com_s, verify if $H_3(com_s) = h_s$. Send out ABORT message if equality does not hold.
   (b) Compute $com := com_c + com_s$ and derive challenge polynomial $c \in B_\tau := H_0(m||com)$.
   (c) Compute potential signature share $\mathbf{z}_1^c := \mathbf{y}_1^c + c\mathbf{s}_1^c$ and $\mathbf{z}_2^c := \mathbf{y}_2^c + c\mathbf{s}_2^c$.
   (d) Perform rejection sampling:
   if $||\mathbf{z}_1^c||_\infty \geq \gamma - \beta$ or $||\mathbf{z}_2^c||_\infty \geq \gamma - \beta$, then send out RESTART message.
   else:
   (e) Send out $(\mathbf{z}_1^c, \mathbf{z}_2^c)$.
6. **Server's third message**
   (a) Upon receiving com_c, verify if $H_3(com_c) = h_c$. Send ABORT message if equality does not hold.
   (b) Compute $com := com_c + com_s$ and derive challenge polynomial $c \in B_\tau := H_0(m||com)$.
   (c) Compute potential signature share $\mathbf{z}_1^s := \mathbf{y}_1^s + c\mathbf{s}_1^s$ and $\mathbf{z}_2^s := \mathbf{y}_2^s + c\mathbf{s}_2^s$.
   (d) Perform rejection sampling:
   if $||\mathbf{z}_1^s||_\infty \geq \gamma - \beta$ or $||\mathbf{z}_2^s||_\infty \geq \gamma - \beta$, then send out RESTART message.
   else:
   (e) Send out $(\mathbf{z}_1^s, \mathbf{z}_2^s)$.
7. **Client's verification**
   (a) Upon receiving $(\mathbf{z}_1^s, \mathbf{z}_2^s)$, reconstruct $\mathbf{w}_s := \mathbf{A}\mathbf{z}_1^s + \mathbf{z}_2^s - c\mathbf{t}_s$.
   (b) Check if $HomH(\mathbf{w}_s) = com_s$ and send out ABORT message if check fails.
   (c) Compute final signature on message m as $\mathbf{z}_1 := \mathbf{z}_1^c + \mathbf{z}_1^s$ and $\mathbf{z}_2 := \mathbf{z}_2^c + \mathbf{z}_2^s$, $\sigma = (\mathbf{z}_1, \mathbf{z}_2, c)$.
8. **Server's verification**
   (a) Upon receiving client's message $(\mathbf{z}_1^c, \mathbf{z}_2^c)$, reconstruct $\mathbf{w}_c := \mathbf{A}\mathbf{z}_1^c + \mathbf{z}_2^c - c\mathbf{t}_c$.
   (b) Check if $HomH(\mathbf{w}_c) = com_c$ and and send out ABORT message if check fails.
   (c) Compute final signature on message m as $\mathbf{z}_1 := \mathbf{z}_1^c + \mathbf{z}_1^s$ and $\mathbf{z}_2 := \mathbf{z}_2^c + \mathbf{z}_2^s$, $\sigma = (\mathbf{z}_1, \mathbf{z}_2, c)$.
9. **Upon receiving RESTART message**
   (a) Client and server start signing process again from the beginning.

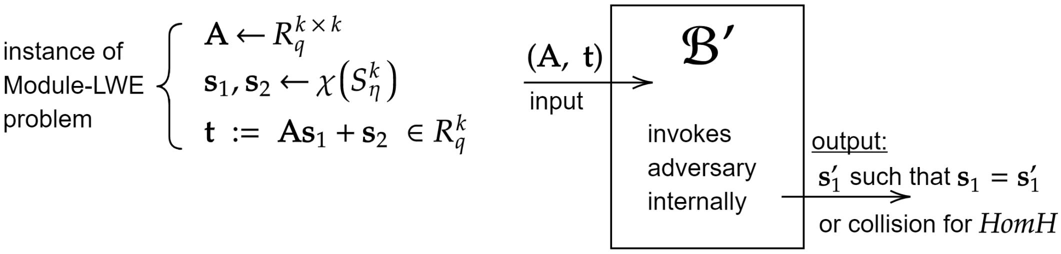
Algorithm 14 Verify(pk, σ, m)

```
1: Compute $\mathbf{w}' := \mathbf{A}\mathbf{z}_1 + \mathbf{z}_2 - c\mathbf{t}$.
2: if $c = \mathrm{H}(m||HomH(\mathbf{w}'))$ and $||\mathbf{z}_1||_\infty < \gamma_2 - \beta_2$ and $||\mathbf{z}_2||_\infty < \gamma_2 - \beta_2$: return 1 (success).
3: else: return 0.
```

The signature protocol presented in this section may fit the Smart-ID framework. As future work, a clone detection mechanism, similar to the one currently used in Smart-ID, may be introduced. Switching the underlying signature scheme from RSA to the scheme presented in this section would allow using Smart-ID even in the quantum computing era.

7 Security of two-party protocol

This section presents security proof for version 3 of the two-party signature scheme (Section 6.3.2). The proof considers only the classical adversary and relies on the forking lemma (Lemma 1). The forking lemma helps to obtain two valid signature forgeries from the adversary such that commitments in both forgeries are the same and challenges are distinct. A proof against the quantum adversary is left for future work. The difference between classical and quantum adversary is that the quantum adversary has access to a quantum computer. This allows the quantum adversary to perform different computations and make different queries using special properties of quantum mechanics. To prove the security of the signature scheme against the quantum adversary, it will be needed to change a considerable part of the proof below because forking lemma cannot be used in the quantum proofs. The idea of the proof below is, given adversary $\mathcal{A}$ against the distributed signature scheme, to construct an algorithm $\mathcal{B}'$ around it that can be used to solve computational Module-LWE problem or to break a collision resistance of the homomorphic hash function. Figure 8 graphically illustrates the main idea of the algorithm $\mathcal{B}'$.



instance of Module-LWE problem
$\mathbf{A} \leftarrow R_q^{k \times k}$
$\mathbf{s}_1, \mathbf{s}_2 \leftarrow \chi(S_\eta^k)$
$\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2 \in R_q^k$
(A, t)
input
$\mathfrak{B}'$
invokes adversary internally
output:
$\mathbf{s}_1'$ such that $\mathbf{s}_1 = \mathbf{s}_1'$
or collision for $HomH$

Figure 8: The construction of algorithm $\mathcal{B}'$

The following forking lemma is adopted from [10]. There exists another version of forking lemma, defined in [46]. The difference between these two versions is that lemma defined below does not mention explicitly signature schemes or random oracles. Lemma 1 concentrates on the behavior of the output of an algorithm that was run twice on the inputs that are related to each other. Therefore, Lemma 1 is easily applicable for the analysis of non-standard signature schemes [47]. In the definition below, x can be viewed as a public key of the signature scheme, and $h_1, ..., h_q$ can be viewed as replies to the random oracle queries.

Lemma 1. ***General forking lemma***. *Fix an integer $q \geq 1$ to be the number of queries. Fix set C of size $|C| \geq 2$. Let $\mathcal{B}$ be a randomised algorithm that takes as input $x, h_1, ..., h_q$, where $(h_1, ..., h_q) \in C$ and returns a pair with the first element being index i (integer in the range $0, ..., q$) and the second element being side output* *out. Let $\mathcal{IG}$ be a randomized input generation algorithm. Let accepting probability of $\mathcal{B}$ be denoted as acc. This is the probability that i $\neq 0$ in the following experiment:*

- $x \leftarrow \mathcal{IG}$
- $h_1, ..., h_q \leftarrow C$
- $(i, out) \leftarrow \mathcal{B}(x, h_1, ..., h_q)$

The forking algorithm $\mathcal{F}_B$ connected with $\mathcal{B}$ is defined in Figure 15.

Algorithm 15 $\mathcal{F}_B(x)$

```
Pick random coins $\rho$ for $\mathcal{B}$
Generate $h_1, ..., h_q \leftarrow C$
$(i, out) \leftarrow \mathcal{B}(x, h_1, ..., h_q; \rho)$
If $i = 0$ then return $(0, \perp, \perp)$
Regenerate $h'_i, ..., h'_q \leftarrow C$
$(i', out') \leftarrow \mathcal{B}(x, h_1, ..., h_{i-1}, h'_i, ..., h'_q; \rho)$
If $i = i'$ and $h_i \neq h'_i$ then return $(1, out, out')$
Otherwise return $(0, \perp, \perp)$
```

Let define frk probability as

$$frk \ = \ Pr[b = 1 : x \leftarrow \mathcal{IG}; (b, out, out') \leftarrow \mathcal{F}_B(x)].$$

Then

$$frk \geq acc \cdot (\frac{acc}{q} - \frac{1}{|C|}).$$

Alternatively

$$acc \leq \frac{q}{|C|} + \sqrt{q \cdot frk}.$$

Definition 35. Distributed signature protocol is called DS-UF-CMA (distributed signature unforgeability under chosen message attacks) secure if for any probabilistic polynomial time adversary $\mathcal{A}$, its advantage of creating successful signature forgery is negligible. Advantage of adversary is defined as probability of winning in the experiment $\mathrm{Exp}^{\text{DS-UF-CMA}}$:

$$\mathrm{Adv}^{\text{DS-UF-CMA}}(\mathcal{A}) := \Pr[\mathrm{Exp}^{\text{DS-UF-CMA}}(\mathcal{A}) \rightarrow 1].$$

Experiment 1: $\text{Exp}^{\textbf{DS-UF-CMA}}(\mathcal{A})$

```
1. $\mathcal{M} \leftarrow \emptyset$
2. $kgen \leftarrow$ false
3. $par \leftarrow \text{Setup}(1^\lambda)$
4. $(m^*, \sigma^*) \leftarrow \mathcal{A}^{\text{KeyGen}\mathcal{O}(.,.), \text{ Sign}\mathcal{O}(.,.)}(\text{par})$
5. $b \leftarrow \text{Verify}(m^*, \sigma^*, pk)$
6. if $b = 1$ and $m^* \notin \mathcal{M}$: return 1
7. otherwise, return 0
```

The experiment defined above starts with creating a set of messages $\mathcal{M}$ that are queried by the adversary during the signing process. At the beginning of the experiment, the set $\mathcal{M}$ is empty. Then $kgen$ flag is set to false, it is needed to ensure that key generation was done before the adversary starts querying the signing oracle. Then Setup() algorithm is invoked to generate public parameters par and then the adversary tries to produce forgery on message m^* that has never been queried through the signing oracle before. If verification succeeds and the message has never been queried, the adversary wins. The figure 9 graphically illustrates experiment $\text{Exp}^{\text{DS-UF-CMA}}(\mathcal{A})$.

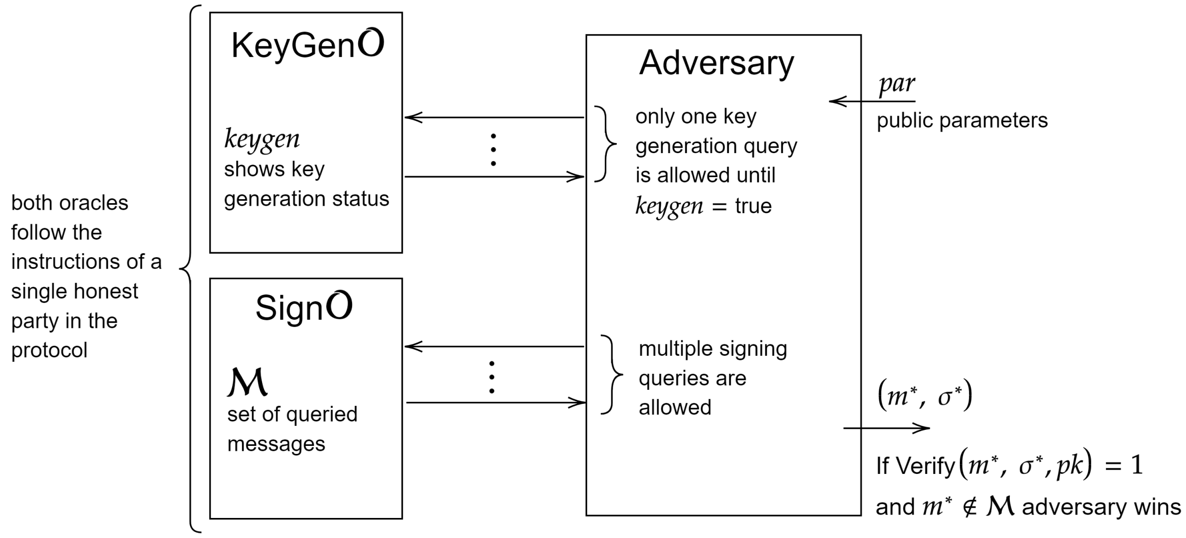

both oracles follow the instructions of a single honest party in the protocol
KeyGenO
keygen
shows key generation status
SignO
M
set of queried messages
Adversary
only one key generation query is allowed until keygen = true
multiple signing queries are allowed
par
public parameters
(m*, σ*)
If Verify(m*, σ*, pk) = 1 and m* ∉ M adversary wins

Figure 9: DS-UF-CMA experiment

Key generation and signing oracles follow the instructions of a single honest party P_n in the protocol, the other party is corrupted by the adversary. Both oracles receive messages in the form (sid, msg), where sid is a session identifier and msg a message that was supposed to be sent by the protocol. A session identifier is needed for the oracle to keep in track of the state of the protocol as the adversary is allowed to execute many signing sessions concurrently. KeyGen$\mathcal{O}(sid, msg)$ and Sign$\mathcal{O}(sid, msg)$ are defined more formally in Oracle 1 and Oracle 2.

The key generation oracle follows the instructions of the key generation protocol. When key generation is done, flag $kgen$ is set to true, which means that the adversary can start querying the signing oracle. If $kgen$ has already been set to $true$ then abort state $\perp$ is returned as the adversary is allowed to query the key generation oracle only once.

Signing oracle follows the instructions of the signing protocol. The adversary can query signing oracle once key generation is done (flag $kgen$ is set to true), otherwise signing oracle ignores the incoming queries. When adversary queries signing oracle on message m, m is added to the set $\mathcal{M}$.

In the random oracle model, adversary and the KeyGen, Sign, Verify algorithms, additionally have access to a random oracle.

The security proofs that involve using forking lemma typically follow the same steps. Firstly, given an adversary $\mathcal{A}$ against the signature scheme, another algorithm $\mathcal{B}$ is constructed around $\mathcal{A}$. $\mathcal{B}$ is constructed such that it fits the assumptions of the forking lemma. $h_1, ..., h_q$ are used as replies to the random oracle queries made by $\mathcal{A}$ that is invoked by $\mathcal{B}$. The important part is that both executions of $\mathcal{B}$ performed using the forking algorithm $\mathcal{F}_B$ should use the same randomness ρ. [47]

Oracle 1: KeyGen$\mathcal{O}(par, sid, msg)$

The oracle is initialised with the set of public parameters par generated by Setup(1^λ) algorithm.

1. Upon receiving $(0, msg)$ if the flag $kgen =$ true then return $\perp$.
2. Upon receiving query with $sid = 0$ for the first time:
   (a) Initialise a machine $\mathcal{M}_0$. $\mathcal{M}_0$ uses the instructions of the party P_n in the key generation protocol KeyGen(par).
   (b) If P_n sends the first message according to the key generation protocol then oracle returns this message.
3. If machine $\mathcal{M}_0$ has been already initialised:
   (a) Oracle gives the next incoming message msg to the $\mathcal{M}_0$.
   (b) Oracle returns reply that was received from $\mathcal{M}_0$.
   (c) If $\mathcal{M}_0$ finished the protocol with a local output (sk_n, pk) then oracle sets the flag $kgen =$ true.

Oracle 2: Sign$\mathcal{O}(sid, msg)$

1. Upon receiving (sid, msg) if the flag $kgen =$ false and $sid \neq 0$ then return $\perp$.
2. Upon receiving query with sid for the first time:
   (a) Parse incoming message msg as message to be signed m.
   (b) Initialise a machine $\mathcal{M}_{sid}$. $\mathcal{M}_{sid}$ uses the instructions of the party P_n in the signing protocol Sign(par, sk_n, pk, m^*).
   (c) The message to be signed m is included in the set of all queried messages $\mathcal{M}$.
   (d) If P_n sends the first message according to the signing protocol then oracle returns this message.
3. If machine $\mathcal{M}_{sid}$ has been already initialised:
   (a) Oracle gives the next incoming message msg to the $\mathcal{M}_{sid}$.
   (b) Oracle returns reply that was received from $\mathcal{M}_{sid}$.
   (c) If $\mathcal{M}_{sid}$ finished the protocol with a local output σ then the oracle returns this output.

Theorem 2. *Assume a homomorphic hash function $HomH : \{0,1\}^{a \cdot b} \rightarrow \mathbb{Z}_p^b$ is provably collision-resistant and ϵ-regular then, for any probabilistic polynomial time adversary $\mathcal{A}$ that makes a single query to the key generation oracle, q_s queries to the signing oracle and q_h queries to the random oracles H_0, H_1, H_2, H_3, the distributed signature protocol is DS-UF-CMA secure in the random oracle model under Module-LWE assumption.*

The idea of the following proof relies on the proofs from [47, 10, 9, 3].

The proof below consists of two major steps. The first step involves constructing an algorithm $\mathcal{B}$ around $\mathcal{A}$ that simulates the behavior of the single honest party P_n without using its actual secret keys. It should be noted that the instructions of the key generation and signing protocols are the same for the client and server. Therefore it is assumed in the proof that one of them is corrupted (party P_i) and one of them behaves honestly (party P_n) without explicitly mentioning who plays which role. In the second step, forking algorithm $\mathcal{F}_B$ associated with $\mathcal{B}$ is invoked to obtain two forgeries with distinct challenges and the same commitments. This, in turn, allows to construct a solution to the computational Module-LWE problem or to break the collision resistance of the homomorphic hash function $HomH$.

It is important to note that the forking algorithm only receives a public key as input but to show that the adversary can be used to break the underlying computational Module-LWE problem, it is needed to simulate the whole view of the adversary, who makes signing queries. This means that $\mathcal{B}$ is needed to simulate the interaction of the adversary with the signing oracle and all the messages that the adversary sends and receives during the protocol.

Proof

Given an adversary $\mathcal{A}$ that succeeds in breaking distributed signature protocol with advantage $\text{Adv}^{\text{DS-UF-CMA}}(\mathcal{A})$, a simulator $\mathcal{B}$ is constructed. $\mathcal{B}$ simulates the behaviour of the single honest party without using honestly generated secret keys for the computation. Algorithm $\mathcal{B}$ is constructed such that it fits all the assumptions of the forking lemma defined above. By the definition of forking algorithm, it was required that $\mathcal{B}$ is given a public key and a random oracle query replies as input. $\mathcal{B}$ simulates the behaviour of honest party P_n, party P_i is corrupted by the adversary. The algorithm $\mathcal{B}$ is defined in Algorithm 16.

Algorithm 16 $\mathcal{B}(pk, h_1, ..., h_{q_h+q_s+1})$

```
Create empty hash tables $HT_i$ for $i \in \{0, ..., 3\}$.
Create a set of queried messages $\mathcal{M} = \emptyset$.
Simulate honest party oracle as follows:
    • Upon receiving a query from $\mathcal{A}$ of the form $(sid, msg)$, reply to the query as described in $Sim\mathcal{O}_{KeyGen}$ (Oracle 3) and $Sim\mathcal{O}_{Sign}$ (Oracle 4).
    • If one of the oracles terminates with output of the form $(0, \perp)$ then $\mathcal{B}$ also terminates with the same output $(0, \perp)$.
Simulate random oracles as follows:
    • Upon receiving a query from $\mathcal{A}$ to the random oracle, reply to the query as described in the Random oracle simulation algorithm.
Upon receiving a forgery $\sigma = (\mathbf{z}_1, \mathbf{z}_2, c)$ on message $m^*$ from $\mathcal{A}$:
    • If $m^* \in \mathcal{M}$ then $\mathcal{B}$ terminates with output $(0, \perp)$.
    • Compute $com^* := HomH(\mathbf{A}\mathbf{z}_1 + \mathbf{z}_2 - c\mathbf{t})$.
    • Make query $c^* \leftarrow H_0(m^*||com')$.
    • If $c \neq c'$ or $||\mathbf{z}_1||_\infty \geq \gamma_2 - \beta_2$ or $||\mathbf{z}_2||_\infty \geq \gamma_2 - \beta_2$ then $\mathcal{B}$ terminates with output $(0, \perp)$.
    • Find index $i_f \in [q_h + q_s + 1]$ such that $c^* = h_{i_f}$. $\mathcal{B}$ terminates with the output $(i_f, out = (com^*, c^*, \mathbf{z}_1, \mathbf{z}_2, m^*))$
```

Oracle 3: $Sim\mathcal{O}_{KeyGen}(par, sid, msg)$

The oracle is initialised with the set of public parameters par generated by Setup(1^λ) algorithm.

1. Upon receiving $(0, msg)$ if the flag $kgen =$ true then return $\perp$.
2. Upon receiving query with $sid = 0$ for the first time:
   (a) Initialise a machine $\mathcal{M}_0$. $\mathcal{M}_0$ uses the instructions of $Sim_{KeyGen}(par, \mathbf{A}, \mathbf{t})$.
   (b) If P_n sends the first message according to the key generation protocol then oracle returns this message.
3. If machine $\mathcal{M}_0$ has been already initialised:
   (a) Oracle gives the next incoming message msg to the $\mathcal{M}_0$.
   (b) Oracle returns reply that was received from $\mathcal{M}_0$.
   (c) If $\mathcal{M}_0$ finished the protocol with a local output $(\mathbf{t}_n, pk)$ then oracle sets the flag $kgen =$ true.

Oracle 4: $Sim\mathcal{O}_{Sign}(sid, msg)$

1. Upon receiving (sid, msg) if the flag $kgen =$ false and $sid \neq 0$ then return $\perp$.
2. Upon receiving query with sid for the first time:
   (a) Parse incoming message msg as message to be signed m.
   (b) Initialise a machine $\mathcal{M}_{sid}$. $\mathcal{M}_{sid}$ uses the instructions of $Sim_{Sign}(sid, \mathbf{t}_n, pk, m)$.
   (c) The message to be signed m is included in the set of all queried messages $\mathcal{M}$.
   (d) If P_n sends the first message according to the signing protocol then oracle returns this message.
3. If machine $\mathcal{M}_{sid}$ has been already initialised:
   (a) Oracle gives the next incoming message msg to the $\mathcal{M}_{sid}$.
   (b) Oracle returns reply that was received from $\mathcal{M}_{sid}$.
   (c) If $\mathcal{M}_{sid}$ finished the protocol with a local output σ then the oracle returns this output.

Random oracle simulation

There are several random oracles that need to be simulated:

1. $H_0 : \{0,1\}^* \to C$

   [C is a set of all vectors of size n with exactly $\tau \pm 1$ and other elements being zeros]

2. $H_1 : \{0,1\}^* \to \{0,1\}^{l_1}$

3. $H_2 : \{0,1\}^* \to \{0,1\}^{l_2}$

4. $H_3 : \{0,1\}^* \to \{0,1\}^{l_3}$

All of the random oracles are simulated similarly as described in Algorithm 17. Additionally, there is a searchHash(HT, h) algorithm for searching entries from the hash table defined in Algorithm 18.

Algorithm 17 $H_i(x)$

```
$HT_i$ is a hash table that is initially empty.
1: On a query $x$ return element $HT_i[x]$ if it was previously defined.
2: Otherwise sample output $y$ uniformly at random from the range of $H_i$ and return $HT_i[x] := y$
```

Algorithm 18 searchHash(HT, h)

```
1: For value $h$ find its preimage $m$ in the hash table such that $HT[m] = h$.
2: If preimage of value $h$ does not exist, set flag $alert$ and set preimage $m = \perp$.
3: If for value $h$ more than one preimage exists in hash table $HT$, set flag $bad$.
4: Output: $(m, alert, bad)$
```

The games described below illustrate how the simulators for the key generation process and the signing process are constructed. Let Pr[$\mathbf{G}_i$] denote the probability that $\mathcal{B}$ does not output (0,$\perp$) in the game $\mathbf{G}_i$. This means that adversary must have created a valid forgery (as defined in Algorithm 16). Then Pr[$\mathbf{G}_0$] $=$ Adv$^{\text{DS-UF-CMA}}(\mathcal{A})$.

Game 0

In Game 0, $\mathcal{B}$ simulates honest party behaviour using the same instructions as in the original KeyGen() and Sign() protocols. In latter games, key generation will be modified such that additionally to the parameters par it will take pre-generated public key $(\mathbf{A}, \mathbf{t})$ as input. The simulation of signing process will be modified to use only message m, public key share $\mathbf{t}_n$ and composed public key pk as input.

Algorithm 19 $Sim_{KeyGen}(par)$

```
1: $\mathbf{A}_n \leftarrow R_q^{k \times k}$, send out $h_n = H_1(\mathbf{A}_n)$.
2: Upon receiving $h_i$, send out $\mathbf{A}_n$.
3: Upon receiving $\mathbf{A}_i$, check that $H_1(\mathbf{A}_i) = h_i$. If not: send out ABORT.
4: $\mathbf{A} := \mathbf{A}_n + \mathbf{A}_i$
5: $\mathbf{s}_1^n$, $\mathbf{s}_2^n \leftarrow S_\eta^k$.
6: $\mathbf{t}_n := \mathbf{A}\mathbf{s}_1^n + \mathbf{s}_2^n$, send out $com_n := H_2(\mathbf{t}_n)$.
7: Upon receiving $com_i$, send out $\mathbf{t}_n$.
8: Upon receiving $\mathbf{t}_i$, check that $H_2(\mathbf{t}_i) = com_i$. If not: send out ABORT.
9: Otherwise, set $\mathbf{t} := \mathbf{t}_n + \mathbf{t}_i$, $pk := (\mathbf{A}, \mathbf{t})$ and $sk := (\mathbf{A}, \mathbf{t}_i, \mathbf{s}_1^n, \mathbf{s}_2^n)$.
```

Algorithm 20 $Sim_{Sign}(sk_n, pk, m)$

```
1: $\mathbf{y}_1^n, \mathbf{y}_2^n \leftarrow S_{\gamma-1}^k$.
2: $\mathbf{w}_n := \mathbf{A}\mathbf{y}_1^n + \mathbf{y}_2^n$.
3: $com_n \leftarrow HomH(\mathbf{w}_n)$, send out $h_n \leftarrow H_3(com_n)$.
4: Upon receiving $h_i$ send out $com_n$.
5: Upon receiving $com_i$ check that $H_3(com_i) = h_i$, if not: send out ABORT.
6: Otherwise, compute $com = com_n + com_i$.
7: $c \leftarrow H_0(com, m)$.
8: $\mathbf{z}_1^n = \mathbf{y}_1^n + c\mathbf{s}_1^n$, $\mathbf{z}_2^n = \mathbf{y}_2^n + c\mathbf{s}_2^n$
9: Run rejection sampling, if it did not pass: send out RESTART and go to the step 1.

10: Otherwise, send out $(\mathbf{z}_1^n, \mathbf{z}_2^n)$. Upon receiving RESTART, go to step 1.
11: Upon receiving $(\mathbf{z}_1^i, \mathbf{z}_2^i)$, reconstruct $\mathbf{w}_i := \mathbf{A}\mathbf{z}_1^i + \mathbf{z}_2^i - c\mathbf{t}_i$ and check that $HomH(\mathbf{w}_i) = com_i$, if not: send out ABORT.
12: Otherwise, set $\mathbf{z}_1 := \mathbf{z}_1^n + \mathbf{z}_1^i$ and $\mathbf{z}_2 := \mathbf{z}_2^n + \mathbf{z}_2^i$ and output composed signature $\sigma := (\mathbf{z}_1, \mathbf{z}_2, c)$.
```

Game 1

In Game 1 only signing process is changed with respect to the previous game. Challenge c is now sampled uniformly at random and signature shares are computed out of it without communicating with adversary. Changes with respect to the previous game are highlighted.

Algorithm 21 $Sim_{Sign}(sk_n, pk, m)$

```
1: $c \leftarrow C$.
2: $\mathbf{y}_1^n, \mathbf{y}_2^n \leftarrow S_{\gamma-1}^k$.
3: $\mathbf{w}_n := \mathbf{A}\mathbf{y}_1^n + \mathbf{y}_2^n$.
4: $\mathbf{z}_1^n = \mathbf{y}_1^n + c\mathbf{s}_1^n$ and $\mathbf{z}_2^n = \mathbf{y}_2^n + c\mathbf{s}_2^n$.
5: $com_n \leftarrow HomH(\mathbf{w}_n)$, send out $h_n \leftarrow H_3(com_n)$.
6: Upon receiving $h_i$ search for $(com_i, alert, bad_7) \leftarrow \text{searchHash}(HT_3, h_i)$.
7: If the flag $bad_7$ is set, then simulation fails with output $(0, \perp)$.
   If the flag $alert$ is set, then send out $com_n$.
8: $com = com_n + com_i$.
9: Program random oracle $H_0$ to respond queries $(com, m)$ with $c$.
   Set $HT_0[(com, m)] := c$. If $HT_0[(com, m)]$ has been already set, set flag $bad_8$
   and simulation fails with output $(0, \perp)$.
10: Send out $com_n$. Upon receiving $com_i$:
      • if $H_3(com_i) \neq h_i$: send out ABORT.
      • if the flag $alert$ is set and $H_3(com_i) = h_i$: set the flag $bad_9$ and simulation fails
        with output $(0, \perp)$ with output $(0, \perp)$.
11: Otherwise, run rejection sampling, if it did not pass: send out RESTART and go to
    the step 1.
12: Otherwise, send out $(\mathbf{z}_1^n, \mathbf{z}_2^n)$. Upon receiving RESTART, go to step 1.
13: Upon receiving $(\mathbf{z}_1^i, \mathbf{z}_2^i)$, reconstruct $\mathbf{w}_i := \mathbf{A}\mathbf{z}_1^i + \mathbf{z}_2^i - c\mathbf{t}_i$ and check that $HomH(\mathbf{w}_i) =$
    $com_i$, if not: send out ABORT.
14: Otherwise, set $\mathbf{z}_1 := \mathbf{z}_1^n + \mathbf{z}_1^i$, $\mathbf{z}_2 := \mathbf{z}_2^n + \mathbf{z}_2^i$ and output composed signature $\sigma :=$
    $(\mathbf{z}_1, \mathbf{z}_2, c)$.
```

In case of successful forgery, hash tables used in the signing process should contain the values as shown in Table 4 (note that ⋮ denotes all the other entries in the table):

Table 4: Hash tables for the oracles H_0, H_3

	Query	Output
HT_0:	(com, m)	c
	⋮	⋮

	Query	Output
HT_3:	com_n	h_n
	com_i	h_i
	⋮	⋮

Event bad_7 happens if at the step 6 it occurs that there is more than one preimage for the value h_i, in this case hash table of the oracle H_3 contains the values as in Table 5.

Flag $alert$ is set if at the step 6 it occurs that there is no preimage for the value h_i, in this case hash table of the oracle H_3 contains the values as in Table 6.

Event bad_8 happens if at the step 9 it occurs that the output for the query (com, m) has been already set, in this case hash table of the oracle H_0 before the step 10 contains the values as in Table 7.

Table 5: Hash table for the oracle H_3 in case of bad_7

Query	Output
com_n	h_n
com_i	h_i
com'_i	h_i
$\vdots$	$\vdots$

Table 6: Hash table for the oracle H_3 in case of $alert$ is set

Query	Output
com_n	h_n
$\vdots$	$\vdots$

Table 7: Hash table for the oracle H_0 in case of bad_8

Query	Output
(com, m)	$c*$
$\vdots$	$\vdots$

Game 0 → Game 1:

The difference between Game 0 and Game 1 can be expressed using the bad events that can happen with the following probabilities:

- Pr[bad_7] is the probability that at least one collision occurs during at most $q_h + 2q_s$ queries to the random oracle H_3 made by adversary or simulator. This means that two values $com_j \neq com'_j$ were found such that $h_j = HT_3[com_j] = HT_3[com'_j]$. As all the responses of H_3 are chosen uniformly at random from $\{0,1\}^{l_3}$ and there are at most $q_h + 2q_s$ queries to the random oracle H_3, the probability of at least one collision occurring can be expressed as $\frac{\big((q_h + 2q_s)(q_h + 2q_s + 1)\big)/2}{2^{l_3}} \leq \frac{(q_h + 2q_s + 1)^2}{2^{l_3+1}}$, where l_3 is the length of H_3 output.
- Pr[bad_8] is the probability that programming random oracle H_0 fails at least once during q_s queries. This event can happen in the following two cases: $H_3(com_n)$ was previously queried by the adversary or it was not queried by the adversary:
  - Case 1: $H_3(com_n)$ has been already asked by adversary during at most $q_h + 2q_s$ queries to H_3. This means that the adversary knows com and may have queried $H_0(com, m)$ before. This event corresponds to guessing the value of com_n.

    Let the uniform distribution over $\mathbb{Z}_p^b$ be denoted as X and the distribution of $HomH$ output be denoted as Y. As $HomH$ is ϵ-regular (for some negligibly small ϵ) it holds that SD$(X, Y) \leq \epsilon$. Then for any (guessing) subset T of $\mathbb{Z}_p^b$, by the definition of statistical distance (Definition 13), it holds that Pr$[X \in T] \leq$ Pr$[Y \in T] + \epsilon$. Therefore, for a uniform distribution X, the probability of guessing Y by T is bounded by $\frac{1}{|\mathbb{Z}_p^b|} + \epsilon$.

    Since com_n was produced by $\mathcal{B}$ in the beginning of the signing protocol completely independently from $\mathcal{A}$, the probability that $\mathcal{A}$ queried $H_3(com_n)$ is at most $\frac{1}{|\mathbb{Z}_p^b|} + \epsilon$ for each query.

- Case 2: $HT_0[com, m]$ has been set by adversary or simulator by chance during at most $q_h + q_s$ prior queries to the H_0. Since $\mathcal{A}$ has not queried $H_3(com_n)$, adversary does not know com_n and the view of $\mathcal{A}$ is completely independent of com. The probability that com occurred by chance in one of the previous queries to H_0 is at most $(q_h + q_s)\left(\frac{1}{|\mathbb{Z}_p^b|} + \epsilon\right)$.

- Pr[bad_9] is the probability that the adversary predicted at least one of two outputs of the random oracle H_3 without making a query to it. In this case, there will be no record in the hash table HT_3 that corresponds to the preimage com_j. This can happen with probability at most $\frac{2}{2^{l_3}}$ for each signing query.

Therefore the difference between two games is

$$|\mathrm{Pr}[\mathbf{G}_1] - \mathrm{Pr}[\mathbf{G}_0]| \leq \mathrm{Pr}[bad_7] + \mathrm{Pr}[bad_8] + \mathrm{Pr}[bad_9] \leq$$

$$\frac{(q_h + 2q_s + 1)^2}{2^{l_3+1}} + q_s\left((q_h + 2q_s)\left(\frac{1}{|\mathbb{Z}_p^b|} + \epsilon\right) + (q_h + q_s)\left(\frac{1}{|\mathbb{Z}_p^b|} + \epsilon\right) + \frac{2}{2^{l_3}}\right) =$$

$$\frac{(q_h + 2q_s + 1)^2}{2^{l_3+1}} + q_s\left(\left(\frac{1}{|\mathbb{Z}_p^b|} + \epsilon\right) \cdot (2q_h + 3q_s) + \frac{2}{2^{l_3}}\right).$$

Game 2

In Game 2, only the signing process is changed. When the signature share gets rejected, simulator commits to a uniformly random vector $\mathbf{w}_n$ from the ring R_q instead of committing to a vector computed as $\mathbf{A}\mathbf{y}_1^n + \mathbf{y}_2^n$. Hash tables of the random oracles are the same as described for the Game 1.

Algorithm 22 $Sim_{Sign}(sk_n, pk, m)$

```
1: $c \leftarrow C$.
2: $\mathbf{y}_1^n, \mathbf{y}_2^n \leftarrow S_{\gamma-1}^k$.
3: $\mathbf{z}_1^n = \mathbf{y}_1^n + c\mathbf{s}_1^n$ and $\mathbf{z}_2^n = \mathbf{y}_2^n + c\mathbf{s}_2^n$.
4: Run rejection sampling, if it did not pass do the following:
    1. $\mathbf{w}_n \leftarrow R_q^k$.
    2. $com_n \leftarrow HomH(\mathbf{w}_n)$, send out $h_n \leftarrow H_3(com_n)$.
    3. Upon receiving $h_i$ search for $(com_i, alert, bad_7) \leftarrow \text{searchHash}(HT_3, h_i)$.
    4. If the flag $bad_7$ is set, then simulation fails with output $(0, \perp)$. If the flag $alert$
       is set, then send out $com_n$.
    5. $com = com_n + com_i$.
    6. Program random oracle $H_0$ to respond queries $(com, m)$ with $c$. Set
       $HT_0[(com, m)] := c$. If $HT_0[(com, m)]$ has been already set, set flag $bad_8$
       and simulation fails with output $(0, \perp)$.
    7. Send out $com_n$. Upon receiving $com_i$:
        • if $H_3(com_i) \neq h_i$: send out ABORT.
        • if the flag $alert$ is set and $H_3(com_i) = h_i$: set the flag $bad_9$ and simulation
          fails with output $(0, \perp)$.
    8. Otherwise, send out RESTART and go to step 1.
5: If rejection sampling passes do the following:
    1. $\mathbf{w}_n := \mathbf{A}\mathbf{y}_1^n + \mathbf{y}_2^n$.
    2. $com_n \leftarrow HomH(\mathbf{w}_n)$, send out $h_n \leftarrow H_3(com_n)$.
    3. Upon receiving $h_i$ search for $(com_i, alert, bad_7) \leftarrow \text{searchHash}(HT_3, h_i)$.
    4. If the flag $bad_7$ is set, then simulation fails with output $(0, \perp)$. If the flag $alert$
       is set, then continue.
    5. $com = com_n + com_i$.
    6. Program random oracle $H_0$ to respond queries $(com, m)$ with $c$. Set
       $HT_0[(com, m)] := c$. If $HT_0[(com, m)]$ has been already set, set flag $bad_8$
       and simulation fails with output $(0, \perp)$.
    7. Send out $com_n$.Upon receiving $com_i$:
        • if $H_3(com_i) \neq h_i$: send out ABORT.
        • if the flag $alert$ is set and $H_3(com_i) = h_i$: set the flag $bad_9$ and simulation
          fails with output $(0, \perp)$.
    8. Otherwise, send out $(\mathbf{z}_1^n, \mathbf{z}_2^n)$. Upon receiving RESTART, go to step 1.
    9. Upon receiving $(\mathbf{z}_1^i, \mathbf{z}_2^i)$, reconstruct $\mathbf{w}_i := \mathbf{A}\mathbf{z}_1^i + \mathbf{z}_2^i - c\mathbf{t}_i$ and check that
       $HomH(\mathbf{w}_i) = com_i$, if not: send out ABORT.
    10. Otherwise, set $\mathbf{z}_1 := \mathbf{z}_1^n + \mathbf{z}_1^i$, $\mathbf{z}_2 := \mathbf{z}_2^n + \mathbf{z}_2^i$ and output composed signature
        $\sigma := (\mathbf{z}_1, \mathbf{z}_2, c)$.
```

Game 1 → Game 2:

The difference between Game 1 and Game 2 can be expressed with the probability that adversary can distinguish simulated commitment with random $\mathbf{w}_n$ from the real one.

Let assume that there exists an adversary $\mathcal{D}$ who succeeds in distinguish simulated commitment with random $\mathbf{w}_n$ from the real one with non-negligible probability:

$$\begin{aligned}\text{Adv}(\mathcal{D}) = \text{Pr}[\mathbf{A} \leftarrow R_q^{k\times k}, \mathbf{y}_1, \mathbf{y}_2 \leftarrow S_{\gamma-1}^k, \mathbf{w}_0 \leftarrow \mathbf{A}\mathbf{y}_1 + \mathbf{y}_2, \mathbf{w}_1 \leftarrow R_q^k,\\ b \leftarrow \{0,1\}, h_b \leftarrow HomH(\mathbf{w}_b), b' \leftarrow \mathcal{D}(\mathbf{A}, \mathbf{w}_b) : b = b'].\end{aligned}$$

Then the adversary $\mathcal{D}$ can be used to construct an adversary $\mathcal{A}_{MLWE}$ who solves the decisional Module-LWE for parameters $(q, k, k, \gamma - 1, U)$, where U is the uniform distribution.

Algorithm 23 $\mathcal{A}_{MLWE}(\mathbf{A}, \mathbf{w}_b)$

```
$h_b \leftarrow HomH(\mathbf{w}_b)$
$b' \leftarrow \mathcal{D}(\mathbf{A}, h_b)$
output $b'$
```

As a result the difference between the two games is bounded by:

$$|\text{Pr}[\mathbf{G}_2] - \text{Pr}[\mathbf{G}_1]| \leq q_s \cdot \text{Adv}^{\text{Dec-MLWE}}_{(q,k,k,\gamma-1,U)}$$

Game 3

In Game 3 simulator does not generate the signature shares honestly and, thus, does not perform rejection sampling honestly. Rejection sampling is simulated as follows:

- Rejection case: with probability $1 - \left(1 - \frac{|S^k_{\gamma-\beta-1}|}{|S^k_{\gamma-1}|}\right)^2$ simulator generates commitment to the random $\mathbf{w}_n$ as in the previous game.
- Otherwise, sample signature shares from the set $S_{\gamma-\beta-1}$ and compute $\mathbf{w}_n$ out of it.

Hash tables of the random oracles are the same as described for the Game 1.

Algorithm 24 $Sim_{Sign}(\mathbf{t}_n, pk, m)$

```
1: With probability $1 - \left(1 - \frac{|S^k_{\gamma-\beta-1}|}{|S^k_{\gamma-1}|}\right)^2$ do the following:
    1. $c \leftarrow C$.
    2. $\mathbf{w}_n \leftarrow R^k_q$.
    3. $com_n \leftarrow HomH(\mathbf{w}_n)$, send out $h_n \leftarrow H_3(com_n)$.
    4. Upon receiving $h_i$ search for $(com_i, alert, bad_7) \leftarrow \text{searchHash}(HT_3, h_i)$.
    5. If the flag $bad_7$ is set, then simulation fails with output $(0, \perp)$. If the flag $alert$
       is set, then send out $com_n$.
    6. $com = com_n + com_i$.
    7. Program random oracle $H_0$ to respond queries $(com, m)$ with $c$. Set
       $HT_0[(com, m)] := c$. If $HT_0[(com, m)]$ has been already set, set flag $bad_8$
       and simulation fails with output $(0, \perp)$.
    8. Send out $com_n$. Upon receiving $com_i$:
        • if $H_3(com_i) \neq h_i$: send out ABORT.
        • if the flag $alert$ is set and $H_3(com_i) = h_i$: set the flag $bad_9$ and simulation
          fails with output $(0, \perp)$.
    9. Otherwise, send out RESTART and go step 1.
2: Otherwise do the following:
    1. $c \leftarrow C$.
    2. $\mathbf{z}^n_1 \leftarrow S^k_{\gamma-\beta-1}$ and $\mathbf{z}^n_2 \leftarrow S^k_{\gamma-\beta-1}$.
    3. $\mathbf{w}_n = \mathbf{A}\mathbf{z}^n_1 + \mathbf{z}^n_2 - c\mathbf{t}_n$.
    4. $com_n \leftarrow HomH(\mathbf{w}_n)$, send out $h_n \leftarrow H_3(com_n)$.
    5. Upon receiving $h_i$ search for $(com_i, alert, bad_7) \leftarrow \text{searchHash}(HT_3, h_i)$.
    6. If the flag $bad_7$ is set, then simulation fails with output $(0, \perp)$. If the flag $alert$
       is set, then continue.
    7. $com = com_n + com_i$.
    8. Program random oracle $H_0$ to respond queries $(com, m)$ with $c$. Set
       $HT_0[(com, m)] := c$. If $HT_0[(com, m)]$ has been already set, set flag $bad_8$
       and simulation fails with output $(0, \perp)$.
    9. Send out $com_n$. Upon receiving $com_i$:
        • if $H_3(com_i) \neq h_i$: send out ABORT.
        • if the flag $alert$ is set and $H_3(com_i) = h_i$: set the flag $bad_9$ and simulation
          fails with output $(0, \perp)$.
    10. Otherwise, send out $(\mathbf{z}^n_1, \mathbf{z}^n_2)$. Upon receiving RESTART, go to step 1.
    11. Upon receiving $(\mathbf{z}^i_1, \mathbf{z}^i_2)$, reconstruct $\mathbf{w}_i := \mathbf{A}\mathbf{z}^i_1 + \mathbf{z}^i_2 - c\mathbf{t}_i$ and check that
        $HomH(\mathbf{w}_i) = com_i$, if not: send out ABORT.
    12. Otherwise, set $\mathbf{z}_1 := \mathbf{z}^n_1 + \mathbf{z}^i_1$, $\mathbf{z}_2 := \mathbf{z}^n_2 + \mathbf{z}^i_2$ and output composed signature
        $\sigma := (\mathbf{z}_1, \mathbf{z}_2, c)$.
```

Game 2 → Game 3:

The signature shares generated this way are indistinguishable from the real ones because of the ϵ_{ZK}-naHVZK property of the underlying identification scheme from [36], appendix B. Therefore, the difference between Game 2 and Game 3 can be defined as:

$$|\Pr[\mathbf{G}_3] - \Pr[\mathbf{G}_2]| \leq \epsilon_{ZK}$$

According to the proof from [36], $\epsilon_{ZK} = 0$ for the underlying identification scheme.

Game 4

Now, the signing process does not rely on the actual secret keys of the honest party P_n. In the next games, the key generation process is changed such that it does not use secret keys as well. In this game, the simulator is given a predefined uniformly random matrix $\mathbf{A} \leftarrow R_q^{k\times k}$ and the simulator defines its own matrix share out of it. By the definition, the algorithm $\mathcal{B}$ (Algorithm 16) receives a pre-generated public key pk as input. Therefore, the simulator in this game is given a matrix $\mathbf{A}$, in the later games simulator will be changed such that it receives the whole public key and uses it to compute its shares $\mathbf{A}_n, \mathbf{t}_n$.

Algorithm 25 $Sim_{KeyGen}(par, \mathbf{A})$

```
Send out $hk_n \leftarrow \{0,1\}^{l_1}$.
Upon receiving $hk_i$:
    • search for $(\mathbf{A}_i, alert, bad_1) \leftarrow$ searchHash$(HT_1, hk_i)$.
    • if the flag $bad_1$ is set then simulation fails with output $(0, \perp)$.
    • if the flag $alert$ is set then sample $\mathbf{A}_n \leftarrow R_q^{k\times k}$.
      Otherwise, define $\mathbf{A}_n := \mathbf{A} - \mathbf{A}_i$.
Program random oracle $H_1$ to respond queries $\mathbf{A}_n$ with $hk_n$. Set $HT_1[\mathbf{A}_n] := hk_n$.
If $HT_1[\mathbf{A}_n]$ has been already set, then set the flag $bad_2$ and simulation fails with output $(0, \perp)$.
Send out $\mathbf{A}_n$. Upon receiving $\mathbf{A}_i$:
    • if $H_1(\mathbf{A}_i) \neq hk_i$: send out ABORT.
    • if the flag $alert$ is set and $H_1(\mathbf{A}_i) = hk_i$: set the flag $bad_3$ and simulation fails
      with output $(0, \perp)$ with output $(0, \perp)$.
$(\mathbf{s}_1^n, \mathbf{s}_2^n) \leftarrow S_\eta^k \times S_\eta^k$.
$\mathbf{t}_n := \mathbf{A}\mathbf{s}_1^n + \mathbf{s}_2^n$, send out $comk_n := H_2(\mathbf{t}_n)$.
Upon receiving $comk_i$, send out $\mathbf{t}_n$.
Upon receiving $\mathbf{t}_i$, check that $H_2(\mathbf{t}_i) = comk_i$. If not: send out ABORT.
Otherwise $\mathbf{t} := \mathbf{t}_n + \mathbf{t}_i$, $pk := (\mathbf{A}, \mathbf{t})$ and $sk := (\mathbf{A}, \mathbf{t}_i, \mathbf{s}_n, \mathbf{s}'_n)$.
```

In case of successful forgery, hash tables used in key generation and signing processes should contain the values as shown in Table 8.

Table 8: Hash tables for the oracles H_0, H_1, H_2, H_3

HT_0:

Query	Output
(com, m)	c
⋮	⋮

HT_1:

Query	Output
$\mathbf{A}_i$	hk_i
$\mathbf{A}_n$	hk_n
⋮	⋮

HT_2:

Query	Output
$\mathbf{t}_i$	$comk_i$
$\mathbf{t}_n$	$comk_n$
⋮	⋮

HT_3:

Query	Output
com_n	h_n
com_i	h_i
⋮	⋮

Event bad_1 happens if at the step 2 it occurs that there is more than one preimage for the value hk_i, in this case, hash table of the oracle H_1 contains the values as in Table 9.

Flag $alert$ is set if at the step 2 it occurs that there is no preimage for the value hk_i, in this case, hash table of the oracle H_1 contains the values as in Table 10.

Event bad_2 happens if at the step 3 it occurs that the output for the query $\mathbf{A}_n$ has been already set, in this case, hash table of the oracle H_1 before the step 4 contains the values as in Table 11.

Table 9: Hash table for the oracles H_1 in case of bad_1

Query	Output
$\mathbf{A}_i$	hk_i
$\mathbf{A}'_i$	hk_i
⋮	⋮

Table 10: Hash table for the oracles H_1 in case of $alert$ is set

Query	Output
⋮	⋮

Table 11: Hash table for the oracles H_1 in case of bad_2

Query	Output
$\mathbf{A}_n$	hk_n^*
⋮	⋮

Game 3 → Game 4:

The distribution of public matrix $\mathbf{A}$ does not change between Game 3 and Game 4. The difference between Game 3 and Game 4 can be expressed using bad events that happen with the following probabilities:

- Pr[bad_1] is the probability that at least one collision occurs during at most q_h queries to the random oracle H_1 made by adversary or simulator. This can happen with probability at most $\frac{q_h(q_h+1)/2}{2^{l_1+1}}$, where l_1 is the length of H_1 output.
- Pr[bad_2] is the probability that programming random oracle H_1 fails which happens if $H_1(\mathbf{A}_n)$ has been previously asked by adversary during at most q_h queries to the random oracle H_1. This event corresponds to guessing random $\mathbf{A}_n$, for each query the probability of this event is bounded by $\frac{1}{q^{n \cdot k \cdot k}}$.

- $\Pr[bad_3]$ is the probability that adversary predicted at least one of two outputs of the random oracle H_1 without making a query to it. This can happen with probability at most $\frac{2}{2^{l_1}}$.

Therefore the difference between the two games is

$$|\Pr[\mathbf{G}_4] - \Pr[\mathbf{G}_3]| \leq \Pr[bad_1] + \Pr[bad_2] + \Pr[bad_3] \leq \frac{(q_h+1)q_h}{2^{l_1+1}} + \frac{q_h}{q^{n\cdot k\cdot k}} + \frac{2}{2^{l_1}}$$

Game 5

In Game 5, the simulator picks public key share $\mathbf{t}_n$ randomly from the ring, instead of computing it using secret keys. Hash tables of the random oracles are the same as described for the Game 4.

Algorithm 26 $Sim_{KeyGen}(par, \mathbf{A})$

```
Sample $hk_n \leftarrow \{0,1\}^{l_1}$. Send out $hk_n$.
Upon receiving $hk_i$:
    • search for $(\mathbf{A}_i, alert, bad_1) \leftarrow \text{searchHash}(HT_1, hk_i)$.
    • if the flag $bad_1$ is set then simulation fails with output $(0, \perp)$.
    • if the flag $alert$ is set then sample $\mathbf{A}_n \leftarrow R_q^{k\times k}$. Otherwise, define $\mathbf{A}_n := \mathbf{A} - \mathbf{A}_i$.
Program random oracle $H_1$ to respond queries $\mathbf{A}_n$ with $hk_n$. Set $HT_1[\mathbf{A}_n] := hk_n$. If
  $HT_1[\mathbf{A}_n]$ has been already set, then set the flag $bad_2$ and simulation fails with output
  $(0, \perp)$.
Send out $\mathbf{A}_n$. Upon receiving $\mathbf{A}_i$:
    • if $H_1(\mathbf{A}_i) \neq hk_i$: send out ABORT.
    • if the flag $alert$ is set and $H_1(\mathbf{A}_i) = hk_i$: set the flag $bad_3$ and simulation fails
      with output $(0, \perp)$.
$\mathbf{t}_n \leftarrow R_q^k$, send out $comk_n = H_2(\mathbf{t}_n)$.
Upon receiving $comk_i$, send out $\mathbf{t}_n$.
Upon receiving $\mathbf{t}_i$, check that $H_2(\mathbf{t}_i) = comk_i$. If not: send out ABORT.
Otherwise $\mathbf{t} := \mathbf{t}_n + \mathbf{t}_i$, $pk := (\mathbf{A}, \mathbf{t})$.
```

Game 4 → Game 5:

In Game 5, public key share $\mathbf{t}_n$ is sampled uniformly at random from R_q^k, instead of computing it as $\mathbf{A}\mathrm{s}_n + \mathrm{s}'_n$, where $\mathrm{s}_n, \mathrm{s}'_n$ are random elements from S_η^k. As matrix $\mathbf{A}$ follows the uniform distribution over $R_q^{k\times k}$ if adversary can distinguish between Game 3 and Game 4 this adversary can be used as a distinguisher that breaks the decisional Module-LWE problem for parameters (q, k, k, η, U), where U is the uniform distribution.

Therefore, the difference between two games is bounded by the advantage of breaking decisional Module-LWE:

$$|\Pr[\mathbf{G}_5] - \Pr[\mathbf{G}_4]| \leq \mathrm{Adv}^{\text{Dec-MLWE}}_{(q,k,k,\eta,U)}$$

Game 6

In Game 6, the simulator uses as input a random resulting public key $\mathbf{t} \in R_q^k$ to compute its own share out of it.

Algorithm 27 $Sim_{KeyGen}(par, \mathbf{A}, \mathbf{t})$

```
Sample $hk_n \leftarrow \{0,1\}^{l_1}$. Send out $hk_n$.
Upon receiving $hk_i$:
    • search for $(\mathbf{A}_i, alert, bad_1) \leftarrow \mathrm{searchHash}(HT_1, hk_i)$.
    • if the flag $bad_1$ is set then simulation fails with output $(0, \perp)$.
    • if the flag $alert$ is set then sample $\mathbf{A}_n \leftarrow R_q^{k\times k}$. Otherwise, define $\mathbf{A}_n := \mathbf{A} - \mathbf{A}_i$.
Program random oracle $H_1$ to respond queries $\mathbf{A}_n$ with $hk_n$. Set $HT_1[\mathbf{A}_n] := hk_n$. If $HT_1[\mathbf{A}_n]$ has been already set, then set the flag $bad_2$ and simulation fails with output $(0, \perp)$.
Send out $\mathbf{A}_n$. Upon receiving $\mathbf{A}_i$:
    • if $H_1(\mathbf{A}_i) \neq hk_i$: send out ABORT.
    • if the flag $alert$ is set and $H_1(\mathbf{A}_i) = hk_i$: set the flag $bad_3$ and simulation fails with output $(0, \perp)$.
Send out $comk_n \leftarrow \{0,1\}^{l_2}$.
Upon receiving $comk_i$, search for $(\mathbf{t}_i, alert, bad_4) \leftarrow \mathrm{searchHash}(HT_2, comk_i)$.
If the flag $bad_4$ is set, then simulation fails with output $(0, \perp)$.
Compute public key share:
    • If the flag $alert$ is set, $\mathbf{t}_n \leftarrow R_q^k$.
    • Otherwise, $\mathbf{t}_n := \mathbf{t} - \mathbf{t}_i$.
Program random oracle $H_2$ to respond queries $\mathbf{t}_n$ with $comk_n$. Set $HT_2[\mathbf{t}_n] := comk_n$. If $HT_2[\mathbf{t}_n]$ has been already set, set flag $bad_5$ and simulation fails with output $(0, \perp)$.

Send out $\mathbf{t}_n$. Upon receiving $\mathbf{t}_i$:
    • if $H_2(t_i) \neq comk_i$: send out ABORT.
    • if the flag $alert$ is set and $H_2(\mathbf{t}_i) = comk_i$: set the flag $bad_6$ and simulation fails with output $(0, \perp)$.
Otherwise $\mathbf{t} := \mathbf{t}_n + \mathbf{t}_i$, $pk := (\mathbf{A}, \mathbf{t})$.
```

In case of successful forgery, hash tables should contain the values as presented in Table 12.

Table 12: Hash tables for the oracles H_0, H_1, H_2, H_3

HT_0:

Query	Output
(com, m)	c
$\vdots$	$\vdots$

HT_2:

Query	Output
$\mathbf{t}_i$	$comk_i$
$\mathbf{t}_n$	$comk_n$
$\vdots$	$\vdots$

HT_1:

Query	Output
$\mathbf{A}_i$	hk_i
$\mathbf{A}_n$	hk_n
$\vdots$	$\vdots$

HT_3:

Query	Output
com_n	h_n
com_i	h_i
$\vdots$	$\vdots$

Event bad_4 happens if at the step 6 it occurs that there is more than one preimage for the value $comk_i$, in this case hash table of the oracle H_2 contains the values as in Table 13.

Flag $alert$ is set if at the step 6 it occurs that there is no preimage for the value $comk_i$, in this case hash table of the oracle H_2 contains the values as in Table 14.

Event bad_5 happens if at the step 9 it occurs that the output for the query $\mathbf{t}_n$ has been already set, in this case hash table of the oracle H_2 before the step 11 contains the values as in Table 15.

Table 13: Hash table for the oracles H_2 in case of bad_4

Query	Output
$\mathbf{t}_i$	$comk_i$
$\mathbf{t}'_i$	$comk_i$
$\vdots$	$\vdots$

Table 14: Hash table for the oracles H_2 in case of $alert$ is set

Query	Output
$\vdots$	$\vdots$

Table 15: Hash table for the oracles H_2 in case of bad_5

Query	Output
$\mathbf{t}_n$	$comk_n*$
$\vdots$	$\vdots$

Game 5 → Game 6:

The distributions of $\mathbf{t}, \mathbf{t}_n$ do not change. The difference between Game 4 and Game 5 can be expressed using bad events that happen with the following probabilities:

- Pr[bad_4] is the probability that at least one collision occurs during at most q_h queries to the random oracle H_2 made by adversary or simulator. This can happen with probability at most $\dfrac{q_h(q_h+1)/2}{2^{l_2+1}}$, where l_2 is the length of H_2 output.
- Pr[bad_5] is the probability that programming random oracle H_2 fails which happens if $H_2(\mathbf{t}_n)$ was previously asked by adversary during at most q_h queries to the random oracle H_2. This event corresponds to guessing a uniformly random $\mathbf{t}_n \in R_q^k$, for each query the probability of this event is bounded by $\dfrac{1}{q^{n \cdot k}}$.

- Pr$[bad_6]$ is the probability that adversary predicted at least one of two outputs of the random oracle H_2 without making a query to it. This can happen with probability at most $\frac{2}{2^{l_2}}$.

Therefore the difference between the two games is

$$|\text{Pr}[\mathbf{G}_6] - \text{Pr}[\mathbf{G}_5]| \leq \text{Pr}[bad_4] + \text{Pr}[bad_5] + \text{Pr}[bad_6] \leq \frac{(q_h+1)q_h}{2^{l_2+1}} + \frac{q_h}{q^{n\cdot k}} + \frac{2}{2^{l_2}}$$

Forking Lemma

Now, both key generation and signing do not rely on the actual secret keys of the honest party P_n. In order to conclude the proof, it is needed to invoke forking lemma to receive two valid forgeries from the adversary that are constructed using the same commitment $com = com'$, but different challenges $c \neq c'$.

Define an input generation algorithm $\mathcal{IG}$ such that it produces the following input: $(\mathbf{A}, \mathbf{t})$ for the $\mathcal{F}_B$. Now $\mathcal{B}'$ is constructed around the previously defined simulator $\mathcal{B}$. $\mathcal{B}'$ invokes the forking algorithm $\mathcal{F}_B$ on the input $(\mathbf{A}, \mathbf{t})$.

As a result with probability frk two valid forgeries are obtained $out = (com, c, \mathbf{z}_1, \mathbf{z}_2, m)$ and $out' = (com', c', \mathbf{z}'_1, \mathbf{z}'_2, m')$. Here by the construction of $\mathcal{F}_B$ the challenges are different ($c \neq c'$), but the commitments and messages are the same ($com = com', m = m'$). The probability frk satisfies

$$\text{Pr}[\mathbf{G}_6] = \text{acc} \leq \frac{q_h + q_s + 1}{|C|} + \sqrt{(q_h + q_s + 1) \cdot \text{frk}}$$

Since both signatures are valid it holds that

- $c =$ H$(m||HomH(\mathbf{A}\mathbf{z}_1 + \mathbf{z}_2 - c\mathbf{t}))$ and $||\mathbf{z}_1||_\infty < \gamma_3 - \beta_2$, $||\mathbf{z}_2||_\infty < \gamma_3 - \beta_2$
- $c' =$ H$(m'||HomH(\mathbf{A}\mathbf{z}'_1 + \mathbf{z}'_2 - c'\mathbf{t}))$ and $||\mathbf{z}'_1||_\infty < \gamma_3 - \beta_2$, $||\mathbf{z}'_2||_\infty < \gamma_3 - \beta_2$
- $HomH(\mathbf{A}\mathbf{z}_1 + \mathbf{z}_2 - c\mathbf{t}) = com = com' = HomH(\mathbf{A}\mathbf{z}'_1 + \mathbf{z}'_2 - c'\mathbf{t})$

Let examine the following cases:

Case 1: $\mathbf{A}\mathbf{z}_1 + \mathbf{z}_2 - c\mathbf{t} \neq \mathbf{A}\mathbf{z}'_1 + \mathbf{z}'_2 - c'\mathbf{t}$, then $\mathcal{B}'$ is able to break the collision resistance of the hash function (that is hard under the worst-case difficulty of finding short vectors in cyclic/ideal lattices) as was proven in [43], [42].

Case 2: $\mathbf{A}\mathbf{z}_1+\mathbf{z}_2-c\mathbf{t} = \mathbf{A}\mathbf{z}'_1+\mathbf{z}'_2-c'\mathbf{t}$. This implies that $\mathbf{A}(\mathbf{z}_1-\mathbf{z}'_1)+(\mathbf{z}_2-\mathbf{z}'_2) = (c-c')\mathbf{t}$. Let denote $\mathbf{r}_1 = \mathbf{z}_1-\mathbf{z}'_1$, $\mathbf{r}_2 = \mathbf{z}_2-\mathbf{z}'_2$ and $d = (c-c')$, where $d \neq 0$ since $c \neq c'$. It can be seen that for a uniformly random $\mathbf{A} \in R_q^{k\times k}$ algorithm $\mathcal{B}'$ found two vectors $\mathbf{r}_1, \mathbf{r}_2$ with small coefficients such that $\mathbf{A}\mathbf{r}_1 + \mathbf{r}_2 = d\mathbf{t}$. This solves computational Module-LWE problem for (q, k, k, ξ, χ), where $\xi \leq 2(\gamma_2 - \beta_2)$ and χ is the normal distribution with standard deviation σ that depends on the parameter choice.

Therefore, the probability $\texttt{frk}$ is the following:

$$\texttt{frk} \leq \text{Adv}^{\text{Com-MLWE}}_{(q,k,k,\xi,\chi)} + \text{Adv}^{\text{CR}}$$

Finally, taking into account that the underlying identification scheme has perfect naHVZK (i.e. $\epsilon_{ZK} = 0$), the advantage of the adversary is bounded by the following:

$$\text{Adv}^{\text{DS-UF-CMA}}(\mathcal{A}) \leq \frac{(q_h + 2q_s + 1)^2}{2^{l_3+1}} + q_s \cdot \left(\left(\frac{1}{|\mathbb{Z}_p^b|}\right) \cdot (2q_h + 3q_s) + \frac{2}{2^{l_3}}\right) +$$

$$q_s \cdot \text{Adv}^{\text{Dec-MLWE}}_{(q,k,k,\gamma-1,U)} + \frac{(q_h + 1)\, q_h}{2^{l_1+1}} + \frac{q_h}{q^{n \cdot k \cdot k}} + \frac{2}{2^{l_1}} + \text{Adv}^{\text{Dec-MLWE}}_{(q,k,k,\eta,U)} +$$

$$\frac{(q_h + 1)\, q_h}{2^{l_2+1}} + \frac{q_h}{q^{n \cdot k}} + \frac{2}{2^{l_2}} + \frac{q_h + q_s + 1}{|C|} + \sqrt{(q_h + q_s + 1) \cdot \left(\text{Adv}^{\text{Com-MLWE}}_{(q,k,k,\xi,\chi)} + \text{Adv}^{\text{CR}}\right)}$$

The most influential parts in the formula above are the advantage of the adversary in breaking decisional and computational Module-LWE and the advantage of the adversary in breaking collision-resistance of the homomorphic hash function. These advantages are negligible for the correct choice of parameters. Therefore, if the parameters for the scheme are chosen such that the instance of Module-LWE is hard to solve and it is hard to find collision given $HomH$ output, the advantage of the adversary in forging a signature is negligible.

The final versions of the key generation and signing simulators are defined in Algorithm 28 and Algorithm 29.

Algorithm 28 $Sim_{KeyGen}(par, pk = (\mathbf{A}, \mathbf{t}))$

The protocol is parameterised by the public parameters par that were generated by the Setup algorithm and relies on the random oracles H_1, H_2.

```
Sample $hk_n \leftarrow \{0,1\}^{l_1}$. Send out $hk_n$.
Upon receiving $hk_i$:
    • search for $(\mathbf{A}_i, alert, bad_1) \leftarrow \text{searchHash}(HT_1, hk_i)$.
    • if the flag $bad_1$ is set then simulation fails with output $(0, \perp)$.
    • if the flag $alert$ is set then sample $\mathbf{A}_n \leftarrow R_q^{k \times k}$. Otherwise, define $\mathbf{A}_n := \mathbf{A} - \mathbf{A}_i$.
Program random oracle $H_1$ to respond queries $\mathbf{A}_n$ with $hk_n$. Set $HT_1[\mathbf{A}_n] := hk_n$. If $HT_1[\mathbf{A}_n]$ has been already set, then set the flag $bad_2$ and simulation fails with output $(0, \perp)$.
Send out $\mathbf{A}_n$. Upon receiving $\mathbf{A}_i$:
    • if $H_1(\mathbf{A}_i) \neq hk_i$: send out ABORT.
    • if the flag $alert$ is set and $H_1(\mathbf{A}_i) = hk_i$: set the flag $bad_3$ and simulation fails with output $(0, \perp)$.
Send out $comk_n \leftarrow \{0,1\}^{l_2}$.
Upon receiving $comk_i$, search for $(\mathbf{t}_i, alert, bad_4) \leftarrow \text{searchHash}(HT_2, comk_i)$.
If the flag $bad_4$ is set, then simulation fails with output $(0, \perp)$.
Compute public key share:
    • If the flag $alert$ is set, $\mathbf{t}_n \leftarrow R_q^k$.
    • Otherwise, $\mathbf{t}_n := \mathbf{t} - \mathbf{t}_i$.
Program random oracle $H_2$ to respond queries $\mathbf{t}_n$ with $comk_n$. Set $HT_2[\mathbf{t}_n] := comk_n$. If $HT_2[\mathbf{t}_n]$ has been already set, set flag $bad_5$ and simulation fails with output $(0, \perp)$.
Send out $\mathbf{t}_n$.
Upon receiving $\mathbf{t}_i$:
    • if $H_2(t_i) \neq comk_i$: send out ABORT.
    • if the flag $alert$ is set and $H_2(\mathbf{t}_i) = comk_i$: set the flag $bad_6$ and simulation fails with output $(0, \perp)$.
Otherwise $\mathbf{t} := \mathbf{t}_n + \mathbf{t}_i$, $pk := (\mathbf{A}, \mathbf{t})$.
```

Algorithm 29 $Sim_{Sign}(\mathbf{t}_n, pk, m)$

The protocol is parameterised by the public parameters par that were generated by the Setup algorithm and relies on the random oracles H_0, H_3. It is assumed that $Sim_{KeyGen}(par)$ has been previously invoked (keypair has been successfully generated).

```
1: With probability $1 - \left(1 - \frac{|S^k_{\gamma-\beta-1}|}{|S^k_{\gamma-1}|}\right)^2$ do the following:
    1. $c \leftarrow C$.
    2. $\mathbf{w}_n \leftarrow R^k_q$.
    3. $com_n \leftarrow HomH(\mathbf{w}_n)$.
    4. Send out $h_n \leftarrow H_3(com_n)$.
    5. Upon receiving $h_i$ search for $(com_i, alert, bad_7) \leftarrow \text{searchHash}(HT_3, h_i)$.
    6. If the flag $bad_7$ is set, then simulation fails with output $(0, \perp)$. If the flag $alert$
       is set, then send out $com_n$.
    7. $com = com_n + com_i$.
    8. Program random oracle $H_0$ to respond queries $(com, m)$ with $c$. Set
       $HT_0[(com, m)] := c$. If $HT_0[(com, m)]$ has been already set, set flag $bad_8$
       and simulation fails with output $(0, \perp)$.
    9. Send out $com_n$. Upon receiving $com_i$:
        • if $H_3(com_i) \neq h_i$: send out ABORT.
        • if the flag $alert$ is set and $H_3(com_i) = h_i$: set the flag $bad_9$ and simulation
          fails with output $(0, \perp)$.
    10. Otherwise, send out RESTART and go to step 1.
2: Otherwise do the following:
    1. $c \leftarrow C$.
    2. $\mathbf{z}^n_1 \leftarrow S^k_{\gamma-\beta-1}$ and $\mathbf{z}^n_2 \leftarrow S^k_{\gamma-\beta-1}$.
    3. $\mathbf{w}_n = \mathbf{A}\mathbf{z}^n_1 + \mathbf{z}^n_2 - c\mathbf{t}_n$.
    4. $com_n \leftarrow HomH(\mathbf{w}_n)$.
    5. Send out $h_n \leftarrow H_3(com_n)$.
    6. Upon receiving $h_i$ search for $(com_i, alert, bad_7) \leftarrow \text{searchHash}(HT_3, h_i)$.
    7. If the flag $bad_7$ is set, then simulation fails with output $(0, \perp)$. If the flag $alert$
       is set, then continue.
    8. $com = com_n + com_i$.
    9. Program random oracle $H_0$ to respond queries $(com, m)$ with $c$. Set
       $HT_0[(com, m)] := c$. If $HT_0[(com, m)]$ has been already set, set flag $bad_8$
       and simulation fails with output $(0, \perp)$.
    10. Send out $com_n$. Upon receiving $com_i$:
        • if $H_3(com_i) \neq h_i$: send out ABORT.
        • if the flag $alert$ is set and $H_3(com_i) = h_i$: set the flag $bad_9$ and simulation
          fails with output $(0, \perp)$.
    11. Otherwise, send out $(\mathbf{z}^n_1, \mathbf{z}^n_2)$. Upon receiving RESTART, go to step 1.
    12. Upon receiving $(\mathbf{z}^i_1, \mathbf{z}^i_2)$, reconstruct $\mathbf{w}_i := \mathbf{A}\mathbf{z}^i_1 + \mathbf{z}^i_2 - c\mathbf{t}_i$ and check that
        $HomH(\mathbf{w}_i) = com_i$, if not: send out ABORT.
    13. Otherwise, set $\mathbf{z}_1 := \mathbf{z}^n_1 + \mathbf{z}^i_1$ and $\mathbf{z}_2 := \mathbf{z}^n_2 + \mathbf{z}^i_2$.
    14. Output composed signature $\sigma := (\mathbf{z}_1, \mathbf{z}_2, c)$.
```

8 Performance estimation

This section gives the performance estimations of the proposed scheme according to the following metrics:

- number of communication rounds in key generation and signing protocols,
- size of the keys and signatures,
- number of rejection sampling rounds.

It should be noted that the choice of parameters such that output signature would result in a sufficiently hard instance of Module-LWE problem and implementation of the signature scheme are left for future work. Therefore, this section contains only estimations that depend on the parameters listed in Table 3.

Number of rejection samplings

To estimate the number repetitions in the signing process it is needed to compute the probability that the following holds for both parties: $||\mathbf{z}_1^n||_\infty < \gamma - \beta$ and $||\mathbf{z}_2^n||_\infty < \gamma - \beta$. This probability can be computed by examining each coefficient of $\mathbf{z}_i^n, i \in \{1, 2\}$ separately. Let σ be a coefficient of $c\mathbf{s}_i^n$. If coefficient of polynomial from the vector $\mathbf{y}_i^n$ is in the range $\{-\gamma + \beta + 1 - \sigma, ..., \gamma - \beta - 1 - \sigma\}$, then corresponding coefficient of $\mathbf{z}_i^n$ will be in the range $\{-\gamma + \beta + 1, ..., \gamma - \beta - 1\}$ that is exactly the requirement for the valid signature share. Therefore, the size of correct coefficient range for $\mathbf{y}_i^n$ is $2(\gamma - \beta) - 1$ and the coefficients of $\mathbf{y}_i^n$ have $2\gamma - 1$ possibilities. Then the probability that every coefficient of $\mathbf{y}_i^n$ is in the correct range is:

$$\left(\frac{2(\gamma - \beta) - 1}{2\gamma - 1}\right)^{n \cdot k}$$

As the client and server sample vectors $\mathbf{y}_i^n$ independently in the beginning of the signing protocol, the probability that the check succeeds for both signature components on the client and server side is the following:

$$\text{Pr[success]} = \left(\frac{2(\gamma - \beta) - 1}{2\gamma - 1}\right)^{n \cdot k \cdot 4}$$

Expected number of repetitions of the signing process can be computed as $N = \dfrac{1}{\text{Pr[success]}}$.

Signature and key sizes

The **public key** of the two-party scheme presented in this work consists of two components matrix $\mathbf{A} \in R_q^{k \times k}$ and vector $\mathbf{t} \in R_q^k$. Therefore, the size of the public key in bytes can be computed as

$$\frac{n \cdot k \cdot k \cdot \lceil \log(q) \rceil + n \cdot k \cdot \lceil \log(q) \rceil}{8 \text{ bytes}} = \frac{n \cdot k \cdot \lceil \log(q) \rceil \cdot (k+1)}{8 \text{ bytes}}.$$

It can be seen that storing the whole matrix $\mathbf{A}$ is not size-optimal. Therefore, the possible solution may include generating matrix $\mathbf{A}$ out of 256-bit seed using extendable output function as was proposed in the Crystals-Dilithium signature scheme [8]. While using this approach, only the seed that was used to generate the matrix needs to be stored. As both parties need to generate their matrix share, two seeds may be stored to represent matrix $\mathbf{A}$. Each seed will be converted to the matrix form using an extendable output function and then, two matrix shares can be added together. In this case, the size of the public key will be the following:

$$\frac{2 \cdot 256 + n \cdot k \cdot \lceil \log(q) \rceil}{8 \text{ bytes}}.$$

The **secret key** of party P_n consists of two vectors $\mathbf{s}_1^n$, $\mathbf{s}_2^n \in S_\eta^k$, matrix $\mathbf{A}$ and vector $\mathbf{t}_i \in R_q^k$. It should be noted that vectors $\mathbf{s}_1^n$, $\mathbf{s}_2^n$ may contain negative values as well, so one bit should be reserved for each coefficient to indicate the sign. Therefore, the size of the secret key in bytes can be computed as

$$\frac{2 \cdot n \cdot k \cdot (\lceil \log(\eta) \rceil + 1) + n \cdot k \cdot k \cdot \lceil \log(q) \rceil + n \cdot k \cdot \lceil \log(q) \rceil}{8 \text{ bytes}} = \frac{n \cdot k \cdot (2 \cdot (\lceil \log(\eta) \rceil + 1) + (k+1) \cdot \lceil \log(q) \rceil)}{8 \text{ bytes}}.$$

In case of generating matrix from the seed the size will change to

$$\frac{2 \cdot n \cdot k \cdot (\lceil \log(\eta) \rceil + 1) + 2 \cdot 256 + n \cdot k \cdot \lceil \log(q) \rceil}{8 \text{ bytes}} = \frac{n \cdot k \cdot (2 \cdot (\lceil \log(\eta) \rceil + 1) + \lceil \log(q) \rceil) + 2 \cdot 256}{8 \text{ bytes}}.$$

Finally, **signature** consists of three components $\mathbf{z}_1, \mathbf{z}_2 \in S_{\gamma_2 - \beta_2 - 1}^k$ and $c \in \{0,1\}^n$ with exactly $\tau \pm 1$. All the components may contain negative values, so for each coefficient of $\mathbf{z}_1, \mathbf{z}_2, c$ one bit should be reserved to indicate the sign. To store c it is possible to store only the positions of ± 1 in c. Therefore, the size of the signature in bytes can be computed as

$$\frac{2 \cdot n \cdot k \cdot (\lceil \log(\gamma_2 - \beta_2 - 1) \rceil + 1) + \tau \cdot (\lceil \log(n) \rceil + 1)}{8 \text{ bytes}}.$$

In order to better understand key and signature sizes, let's assume the choice of parameters defined in Table 16 (this example is illustrative, the security of parameters was not studied). Key and signature sizes corresponding to this choice of parameters are listed in Table 17. Furthermore, the probability that the checks in rejection sampling succeed for both signature components on the client and the server side is approximately $14,74\%$ for parameters from Table 16, this would lead to 6.8 repetitions of the signing process. The optimal choice of parameters for both security and efficiency is left for future work.

Table 16: Illustrative parameters

Parameter	Value
n	256
q	8380417
(k,k)	$(5,5)$
γ	2^{18}
γ_2	2^{19}
η	2
β	98
β_2	196
(a,b,p)	$(64,16,257)$

Table 17: Key and signature sizes in bytes

Public key pk	22080
Public key pk [matrix from seed]	3744
Secret key share sk_i	22720
Secret key share sk_i [matrix from seed]	4384
Signature σ	6456

Communication between client and server

From Figure 6, it can be seen that to generate a keypair four rounds of communication between the client and server are needed. Table 18 shows sizes of messages that are exchanged between the client and the server during the key generation process using illustrative parameters from Table 16.

From Figure 7, it can be seen that the no rejection signature generation process requires three rounds of communication between the client and server. If at least one of the signature shares gets rejected, then the signing process should start again, which means that the total number of communication rounds depends on the number of rejections. Table 19 shows sizes of messages that are exchanged between the client and the server during the signing process using illustrative parameters from Table 16. It is assumed that in the first message client sends a message to be signed as a 256-bit hash.

Table 18: Message sizes in the key generation process

First message hk_i	256 bits
Second message $\mathbf{A}_i$	18400 bytes
Second message as seed	256 bits
Third message $comk_i$	256 bits
Fourth message $\mathbf{t}_i$	3680 bytes

Table 19: Message sizes in the signing process

Client's first message (h_c, m)	512 bits
Server's first message h_s	256 bits
Second message com_i	528 bits
Third message $(\mathbf{z}_1^i, \mathbf{z}_2^i)$	6080 bytes

9 Conclusions

This research has demonstrated that there is no straightforward way to convert a Crystals-Dilithium signature scheme into a two-party version. Due to the use of the bit decomposition algorithm, it is needed to find a suitable two-party computation protocol that is post-quantum and offers security against the active adversary. Even if a suitable two-party protocol is found, the amount of communication between client and server will increase due to rejection samplings and messages that need to be exchanged according to the two-party computation protocol.

A new lattice-based two-party signature was proposed in this work. The signature scheme was analysed and proven to be DS-UF-CMA secure in the random oracle model under the Module-LWE assumption. Additionally, the sizes of keys and signatures were computed and the number of communication rounds estimated. The security proofs considered a classical active adversary who may not follow the rules of the protocol to create a signature forgery. With some minor modifications, such as clone detection, the proposed scheme may fit the Smart-ID framework. This would allow using Smart-ID even in the quantum computing era.

Compared to the scheme proposed in [9] this work does not use sampling from the discrete Gaussian distribution and does not use lattice-based homomorphic commitment schemes. It was decided to use a homomorphic hash function in this work to possibly achieve better performance.

Implementation of the proposed scheme and the exact choice of parameters for the implementation are left for future work. Additionally, future work may contain optimisation of the size of keys and signature and the security proof against the quantum adversary.

References

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Comput.*, vol. 26, no. 5, pp. 1484–1509, 1997. https://doi.org/10.1137/S0097539795293172.

[2] L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone, "Report on post-quantum cryptography," tech. rep., Apr. 2016. https://doi.org/10.6028/nist.ir.8105.

[3] Y. Lindell, "Fast Secure Two-Party ECDSA Signing." Cryptology ePrint Archive, Report 2017/552, 2017. https://eprint.iacr.org/2017/552.

[4] SK ID Solutions, "eid scheme: SMART-ID," tech. rep., Aug. 2019. https://www.ria.ee/sites/default/files/content-editors/EID/smart-id_skeemi_kirjeldus.pdf.

[5] A. Buldas, A. Kalu, P. Laud, and M. Oruaas, "Server-supported RSA signatures for mobile devices," in *Computer Security – ESORICS 2017* (S. N. Foley, D. Gollmann, and E. Snekkenes, eds.), (Cham), pp. 315–333, Springer International Publishing, 2017.

[6] M. Mosca, "Cybersecurity in an era with quantum computers: Will we be ready?," *IEEE Secur. Priv.*, vol. 16, no. 5, pp. 38–41, 2018. https://doi.org/10.1109/MSP.2018.3761723.

[7] D. Moody, G. Alagic, D. C. Apon, D. A. Cooper, Q. H. Dang, J. M. Kelsey, Y.-K. Liu, C. A. Miller, R. C. Peralta, R. A. Perlner, A. Y. Robinson, D. C. Smith-Tone, and J. Alperin-Sheriff, "Status report on the second round of the NIST post-quantum cryptography standardization process," tech. rep., July 2020. https://doi.org/10.6028/nist.ir.8309.

[8] V. Lyubashevsky, L. Ducas, E. Kiltz, T. Lepoint, P. Schwabe, G. Seiler, D. Stehle, and S. Bai, "Crystals-dilithium. algorithm specifications and supporting documentation," 2020. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[9] I. Damgård, C. Orlandi, A. Takahashi, and M. Tibouchi, "Two-round n-out-of-n and multi-signatures and trapdoor commitment from lattices." Cryptology ePrint Archive, Report 2020/1110, 2020. https://eprint.iacr.org/2020/1110.

[10] M. Bellare and G. Neven, "Multi-signatures in the plain public-key model and a general forking lemma," in *Proceedings of the 13th ACM conference on Computer and communications security - CCS '06*, ACM Press, 2006. https://doi.org/10.1145/1180405.1180453.

[11] D. Unruh, "Cryptology 1. Short notes, spring 2020." http://kodu.ut.ee/~unruh/courses/crypto1/2020/notes.pdf, 2020. Accessed: 25.01.2021.

[12] N. P. Smart, *Cryptography: an introduction*, p. 214–219. McGraw-Hill, 2003.

[13] I. Damgård and J. Nielsen, "Commitment schemes and zero-knowledge protocols (2007)," *Lecture Notes in Computer Science - LNCS*, 08 2008.

[14] R. Cramer, I. B. Damgård, and J. B. Nielsen, *Secure Multiparty Computation and Secret Sharing. Introduction*, p. 3–13. Cambridge University Press, 2015.

[15] D. Unruh, "Quantum cryptography. Short notes, spring 2021." `https://kodu.ut.ee/~unruh/courses/qc/2021/notes-old.pdf`, 2021. Accessed: 23.03.2021.

[16] R. E. Bansarkhani and J. Sturm, "An efficient lattice-based multisignature scheme with applications to bitcoins," in *Cryptology and Network Security - 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings* (S. Foresti and G. Persiano, eds.), vol. 10052 of *Lecture Notes in Computer Science*, pp. 140–155, 2016. `https://doi.org/10.1007/978-3-319-48965-0_9`.

[17] M. Fukumitsu and S. Hasegawa, "A tightly-secure lattice-based multisignature," in *Proceedings of the 6th on ASIA Public-Key Cryptography Workshop, APKC@AsiaCCS 2019, Auckland, New Zealand, July 8, 2019* (K. Emura and T. Mizuki, eds.), pp. 3–11, ACM, 2019. `https://doi.org/10.1145/3327958.3329542`.

[18] R. Tso, Z. Liu, and Y. Tseng, "Identity-based blind multisignature from lattices," *IEEE Access*, vol. 7, pp. 182916–182923, 2019. `https://doi.org/10.1109/ACCESS.2019.2959943`.

[19] C. Ma and M. Jiang, "Practical lattice-based multisignature schemes for blockchains," *IEEE Access*, vol. 7, pp. 179765–179778, 2019. `https://doi.org/10.1109/ACCESS.2019.2958816`.

[20] R. Toluee and T. Eghlidos, "An efficient and secure id-based multi-proxy multi-signature scheme based on lattice." Cryptology ePrint Archive, Report 2019/1031, 2019. `https://eprint.iacr.org/2019/1031`.

[21] M. Fukumitsu and S. Hasegawa, "A lattice-based provably secure multisignature scheme in quantum random oracle model," in *Provable and Practical Security - 14th International Conference, ProvSec 2020, Singapore, November 29 - December 1, 2020, Proceedings* (K. Nguyen, W. Wu, K. Lam, and H. Wang, eds.), vol. 12505 of *Lecture Notes in Computer Science*, pp. 45–64, Springer, 2020. `https://doi.org/10.1007/978-3-030-62576-4_3`.

[22] D. Cozzo and N. P. smart, "Sharing the LUOV: Threshold post-quantum signatures." Cryptology ePrint Archive, Report 2019/1060, 2019. `https://eprint.iacr.org/2019/1060`.

[23] J. Ding, M.-S. Chen, A. Petzoldt, D. Schmidt, B.-Y. Yang, M. Kannwischer, and J. Patarin, "Rainbow - algorithm specification and documentation. the 3rd round proposal," 2020. `https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions`.

[24] R. Bendlin, S. Krehbiel, and C. Peikert, "How to share a lattice trapdoor: Threshold protocols for signatures and (H)IBE," in *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28,*

2013. Proceedings (M. J. J. Jr., M. E. Locasto, P. Mohassel, and R. Safavi-Naini, eds.), vol. 7954 of *Lecture Notes in Computer Science*, pp. 218–236, Springer, 2013. https://doi.org/10.1007/978-3-642-38980-1_14.

[25] M. Kansal and R. Dutta, "Round optimal secure multisignature schemes from lattice with public key aggregation and signature compression," in *Progress in Cryptology - AFRICACRYPT 2020 - 12th International Conference on Cryptology in Africa, Cairo, Egypt, July 20-22, 2020, Proceedings* (A. Nitaj and A. M. Youssef, eds.), vol. 12174 of *Lecture Notes in Computer Science*, pp. 281–300, Springer, 2020. https://doi.org/10.1007/978-3-030-51938-4_14.

[26] L. Ducas, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehle, "Crystals – dilithium: Digital signatures from module lattices." Cryptology ePrint Archive, Report 2017/633, 2017. https://eprint.iacr.org/2017/633.

[27] P. Pessl, L. G. Bruinderink, and Y. Yarom, "To BLISS-B or not to be: Attacking strongswan's implementation of post-quantum signatures," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, eds.), pp. 1843–1855, ACM, 2017. https://doi.org/10.1145/3133956.3134023.

[28] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, and D. Smith-Tone, "Status report on the first round of the NIST post-quantum cryptography standardization process," tech. rep., Jan. 2019. https://doi.org/10.6028/nist.ir.8240.

[29] T. Prest, P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "Falcon: Fast-fourier lattice-based compact signatures over ntru," 2020. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[30] A. Casanova, J.-C. Faugere, G. Macario-Rat, J. Patarin, L. Perret, and J. Ryckeghem, "Gemss: A great multivariate short signature," 2020. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[31] G. Zaverucha, M. Chase, D. Derler, S. Goldfeder, C. Orlandi, S. Ramacher, C. Rechberger, D. Slamanig, J. Katz, X. Wang, V. Kolesnikov, and D. Kales, "The picnic signature scheme design document," 2020. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[32] A. Hulsing, D. J. Bernstein, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdag, P. Kampanakis, S. Kolbl, T. Lange, M. M. Lauridsen, F. Mendel, R. Niederhagen, C. Rechberger, J. Rijneveld, P. Schwabe, J.-P. Aumasson, B. Westerbaan, and W. Beullens, "Sphincs+. submission to the nist post-quantum project, v.3," 2020. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[33] O. Regev, "The learning with errors problem." https://cims.nyu.edu/~regev/papers/lwesurvey.pdf. Accessed: 25.01.2021.

[34] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005* (H. N. Gabow and R. Fagin, eds.), pp. 84–93, ACM, 2005. https://doi.org/10.1145/1060590.1060603.

[35] M. Ajtai, "Generating hard instances of lattice problems (extended abstract)," in *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996* (G. L. Miller, ed.), pp. 99–108, ACM, 1996. https://doi.org/10.1145/237814.237838.

[36] E. Kiltz, V. Lyubashevsky, and C. Schaffner, "A concrete treatment of fiat-shamir signatures in the quantum random-oracle model." Cryptology ePrint Archive, Report 2017/916, 2017. https://eprint.iacr.org/2017/916.

[37] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings* (A. M. Odlyzko, ed.), vol. 263 of *Lecture Notes in Computer Science*, pp. 186–194, Springer, 1986. https://doi.org/10.1007/3-540-47721-7_12.

[38] V. Lyubashevsky, "Fiat-shamir with aborts: Applications to lattice and factoring-based signatures," in *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* (M. Matsui, ed.), vol. 5912 of *Lecture Notes in Computer Science*, pp. 598–616, Springer, 2009. https://doi.org/10.1007/978-3-642-10366-7_35.

[39] T. Güneysu, V. Lyubashevsky, and T. Pöppelmann, "Practical lattice-based cryptography: A signature scheme for embedded systems," in *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings* (E. Prouff and P. Schaumont, eds.), vol. 7428 of *Lecture Notes in Computer Science*, pp. 530–547, Springer, 2012. https://doi.org/10.1007/978-3-642-33027-8_31.

[40] S. Bai and S. D. Galbraith, "An improved compression technique for signatures based on learning with errors." Cryptology ePrint Archive, Report 2013/838, 2013. https://eprint.iacr.org/2013/838.

[41] V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen, "SWIFFT: A modest proposal for FFT hashing," in *Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers*, vol. 5086 of *Lecture Notes in Computer Science*, pp. 54–72, Springer, 2008. https://iacr.org/archive/fse2008/50860052/50860052.pdf.

[42] D. Micciancio, "Generalized compact knapsacks, cyclic lattices, and efficient one-way functions from worst-case complexity assumptions," in *43rd Symposium on Foundations of Computer Science (FOCS 2002), 16-19 November 2002, Vancouver, BC, Canada, Proceedings*, pp. 356–365, IEEE Computer Society, 2002. https://doi.org/10.1109/SFCS.2002.1181960.

[43] V. Lyubashevsky and D. Micciancio, "Generalized compact knapsacks are collision resistant," in *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II* (M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, eds.), vol. 4052 of *Lecture Notes in Computer Science*, pp. 144–155, Springer, 2006. https://doi.org/10.1007/11787006_13.

[44] C. Peikert and A. Rosen, "Efficient collision-resistant hashing from worst-case assumptions on cyclic lattices," in *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings* (S. Halevi and T. Rabin, eds.), vol. 3876 of *Lecture Notes in Computer Science*, pp. 145–166, Springer, 2006. https://doi.org/10.1007/11681878_8.

[45] R. Rivest and Z. Ahmed, "6.857 Computer and Network Security. Lecture 2.." http://web.mit.edu/6.857/OldStuff/Fall97/lectures/lecture2.pdf, 1997. Accessed: 23.03.2021.

[46] D. Pointcheval and J. Stern, "Security arguments for digital signatures and blind signatures," *J. Cryptol.*, vol. 13, no. 3, pp. 361–396, 2000. https://doi.org/10.1007/s001450010003.

[47] M. Bellare and G. Neven, "New multi-signature schemes and a general forking lemma," 2005. https://soc1024.ece.illinois.edu/teaching/ece498ac/fall2018/forkinglemma.pdf.