Delft University of Technology

Master Thesis

Towards Quantum-Safe Smart Contracts on Hyperledger Fabric

Author: Jaylan Lee (6104223)

thesis advisor: Georgios Smaragdakis
daily supervisor: Kaitai Liang

A thesis submitted in fulfillment of the requirements for the degree of Master of Science in Computer Science

August 24, 2025

"Have a pleasant thousand years"
from Land of the Lustrous, *by Haruko Ichikawa*

Abstract

Blockchain technology has had a big impact on digital transactions and data management, providing a decentralized, transparent, and immutable ledger. *Private blockchains*, unlike public blockchains, are restricted to a pre-selected group of participants, making them more suitable for controlled environments such as enterprises, governments, or academic institutions. Hyperledger Fabric (HLF) is a widely used framework for private blockchain technology, designed for enterprise use.

With quantum computers on the rise, commonly used cryptographic algorithms are increasingly at risk of becoming obsolete. Blockchain networks rely extensively on these primitives. This reliance makes them particularly vulnerable to advances in quantum computing. In order to counter this vulnerability, *post-quantum* algorithms have seen rising popularity in the cryptographic community.

This thesis focuses on securing private blockchains built on HLF against potential quantum adversaries using post-quantum cryptographic primitives. We implement ML-DSA, Vesper, and TDUE as smart contracts for digital signing, zero-knowledge proofs, and updatable encryption, respectively, and report on their performance. Furthermore, we build on top of Fabric Private Chaincode to keep the contract application state confidential. While overall performance is not yet competitive with classical cryptographic primitives, we show that post-quantum primitives have promising potential for use in private blockchains.

Acknowledgements

The author would like to express gratitude to the following individuals and organizations for their invaluable support and contributions to this thesis:

- **Daily Supervisor:** Dr. Kaitai Liang, for their insights and constructive feedback.
- **Thesis Advisor:** Prof. Dr. Georgios Smaragdakis, for serving as the official advisor for this thesis.
- Dr. Marcus Brandenburger, for their valuable assistance and guidance in the troubleshooting of Hyperledger Fabric Private Chaincode.
- Huanhuan Chen, for their assistance and clarifications regarding the details of TDUE.

Contents

List of Figures

List of Tables

1 Introduction

With quantum computers on the rise, cryptographic algorithms that are currently deemed as the standard are increasingly at risk of becoming obsolete. Current algorithms such as RSA, ECC, and ECDSA could be cryptanalytically broken with a sufficiently powerful quantum computer using Shor's algorithm (7, 8). While quantum computing is still in its early stages, its eventual advancement will pose a vulnerability risk on classical cryptographic primitives. Even without an immediate quantum threat, "harvest now, decrypt later" programs still pose a risk on the long-term security of encrypted storage and communication. Data that is sensitive today, may remain sensitive in the future.

To expand on this topic, *blockchain* networks make extensive use of cryptographic primitives and rely almost exclusively on the hardness of currently used algorithms for their security. This reliance makes them particularly vulnerable to advances in quantum computing. Generally, blockchain networks use cryptographic primitives such as encryption and digital signing algorithms to manage identities, execute and endorse transactions, distribute new blocks, query the ledger, etc. Blockchain technology has seen widespread applications due to the qualities of its architecture, which inherently grant immutability, decentralisation, and transparency. The emergence of quantum computing poses a threat to these qualities.

In order to counter the vulnerability posed by these algorithms, a new suite of *post-quantum* algorithms (also referred to as *quantum-resistant* algorithms) has seen rising popularity in the cryptographic community. These algorithms are currently thought to remain secure against cryptanalytic attacks, even under the assumption of the existence of quantum computers. One approach to post-quantum algorithms can be reduced to lattice-based cryptography, with hard problems such as LWE (9) and SIS (10). In 2024, NIST has released three final versions of the first post-quantum cryptographic standards, two of which utilise lattice-based cryptography (11).

This thesis focuses on securing *private blockchains*. Unlike *public blockchains*, private blockchains are restricted to a pre-selected group of participants, making them more suitable for controlled environments such as enterprises, governments, or academic institutions. A widely used framework for private blockchain technology is Hyperledger Fabric, which is an open-source permissioned blockchain platform designed for enterprise use. We aim to secure private blockchains, particularly those built on Hyperledger Fabric, against the potential threat of quantum computers.

To illustrate the practical relevance of cryptography on blockchains, consider the following scenario:

Motivating Example: Blockchain Library with Subscription Payments

Consider a blockchain-based library system called Rainforest, where users can borrow books by making monthly subscription payments. Rainforest uses a private blockchain to host their books and manage payments.

Books are stored as encrypted files on the ledger, all under the same key, and users get the decryption keys upon successful payment. Every month, the keys get updated to ensure that users can only access the books they have paid for in the current month.

Suppose a user named Alice wants to apply for a subscription. She needs to prove that she has sufficient funds to make the payment without revealing her account balance or transaction history. To do this, Alice uses a zero-knowledge proof to demonstrate that she owns at least the required amount of funds. This proof is verified by the Rainforest blockchain network, allowing Alice to proceed with the payment without exposing her (sensitive) financial details. The payment request is signed by her private key, preventing unauthorized financial transactions.

This example illustrates how a private blockchain can be used to manage sensitive data, such as financial transactions and personal information, while maintaining user privacy. The use of *zero-knowledge proofs* allows users to prove their eligibility for actions (like payments) without revealing sensitive information, while *digital signatures* ensure the integrity and authenticity of transactions. At its core, the Rainforest library relies on *updatable encryption* to manage access to the books.

Building on this motivating example, we now formalize the research question addressed in this thesis:

RQ. *How can post-quantum cryptographic modules be integrated into Hyperledger Fabric while preserving confidentiality, and how do they perform in practice?*

Contributions. We propose replacements for three cryptographic primitives, including the digital signing algorithm present in the architecture of Hyperledger Fabric. We implement ML-DSA (signing algorithm), Vesper (zero-knowledge proof), and TDUE (updatable encryption) as smart contracts on top of the existing architecture, and report on their performance to demonstrate their feasibility for use in practical scenarios. The novel aspect of this research lies in the practical integration of multiple post-quantum cryptographic primitives into a real-world private blockchain framework, while using trusted computing technology to hide the application state.

1.1 Organization

First, we cover related work in Section 2. In Chapter 3, we provide the necessary background on the cryptographic primitives used in this thesis. Chapters 4, 5, and 6 present a brief overview of the three cryptographic primitives we implement, namely ML-DSA, TDUE, and Vesper, respectively. Chapter 7 covers the architecture of blockchain systems, with a focus on Hyperledger Fabric. In Chapter 8, we present a high-level overview of our design for three modules that implement the cryptographic primitives in Hyperledger

Fabric. In Chapter 9, we describe the implementation details of these modules. We benchmark the performance of our implementations in Chapter 10, and discuss the results in Chapter 11. Finally, we conclude our work in Chapter 12.

2

Related Work

In this chapter, we discuss related (scientific) works that share similar goals and/or methodology to ours.

Post-quantum distributed ledger technology: a systematic survey. Parida *et al.* (2023) (12) investigate the status quo of post-quantum cryptosystems within the framework of (public) blockchains. The article classifies approaches aimed at fortifying blockchains with post-quantum cryptography. The article also identifies specific (quantum) weaknesses in current blockchain systems, such as speeding up hash collision using Grover's algorithm (13) and breaking classical encryption using Shor's algorithm (7). The article does not focus on private blockchains, such as Hyperledger Fabric, and does not provide a concrete architectural design for integrating post-quantum cryptography into such systems.

Algorand. A practical example of post-quantum distributed ledger technology (PQDLT) is Algorand (14), first proposed in a whitepaper in 2017. In 2022, Algorand announced its transition to a post-quantum secure blockchain. They started this endeavour by replacing elliptic curve signing with FALCON (15), a post-quantum signature scheme based on lattice trapdoors. The consensus mechanism of Algorand still relies on the quantum-vulnerable Verifiable Randomness Function (VRF) (16), but there are plans to replace it with a post-quantum secure VRF in the future.

PQFabric: A Permissioned Blockchain Secure from Both Classical and Quantum Attacks. There have been multiple previous attempts at replacing classical primitives within Hyperledger Fabric with post-quantum primitives. These efforts have mostly focused on replacing ECDSA with either hybrid schemes (17, 18, 19) or non-standardized post-quantum algorithms (20). Since the NIST standardized post-quantum algorithms in 2024 were released, there have been works that have started to replace ECDSA with the standardized ML-DSA (21). What separates our thesis from these works, is that we focus on providing a more complete post-quantum platform for HLF by not only replacing the crucial elements, but also providing two additional cryptographic primitives. Combined with our efforts to keep the application state confidential by using FPC, our work can be viewed as more of a foundational set to develop complete applications for private blockchains.

3 Prerequisites

We denote the set of real numbers by $\mathbb{R}$ and the set of integer numbers by $\mathbb{Z}$. We follow the convention of using bold symbols to represent (column-)vectors and matrices, while standard (non-bold) symbols are used to denote scalars. The transpose of a vector $\mathbf{v}$ is given by $\mathbf{v}^t$, and the transpose of a matrix $\mathbf{A}$ is given by $\mathbf{A}^t$. We represent the 2-norm of a vector $\mathbf{v}$ with $\|\mathbf{v}\| := \sqrt{\sum_i v_i^2}$, where v_i are the components of $\mathbf{v}$. We denote the largest singular value of a matrix $\mathbf{A}$ by $s_1(\mathbf{A}) := \max_{\mathbf{u}} \|\mathbf{A}^t\mathbf{u}\|$, where the maximum is taken over all unit vectors $\mathbf{u}$. We use standard asymptotic notation for O, ω, and Θ. Let λ be the security parameter, which is a positive integer that determines the security level of cryptographic schemes.

3.1 Lattices

An m-dimensional lattice Λ is a discrete subgroup of Euclidean space. The simplest lattice in m-dimensional space is the integer lattice $\Lambda = \mathbb{Z}^m$, which encapsulates the discrete vector space. Other lattices are obtained by applying a linear transformation $\Lambda = \mathbf{B}\mathbb{Z}^m$. Lattices are thus generally defined as:

$$\Lambda = \left\{ \sum_{i=1}^{k} z_i \mathbf{b}_i \mid z_i \in \mathbb{Z} \right\} = \mathbf{B}\mathbb{Z}^k \tag{3.1}$$

for some $k \leq m$ linearly independent basis vectors $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_k\}$. Equivalently, an m-dimensional lattice Λ is a discrete additive subgroup of $\mathbb{R}^m$. The basis is commonly used to represent a lattice. The same lattice has multiple bases. The *dual* of the lattice is defined as:

$$\Lambda^* := \{\mathbf{x} \in \operatorname{span}(\Lambda) \mid \langle \mathbf{x}, \Lambda \rangle \subseteq \mathbb{Z}\} \tag{3.2}$$

and can be interpreted as the lattice consisting of all the vectors that produce integer dot products with vectors in the primary lattice.

This thesis exclusively discusses full-rank lattices, where $k = m$. Moreover, we concern ourselves with q-ary lattices specifically, which contain $q\mathbb{Z}^m$ as a sublattice for some integer q. In other words, a q-ary lattice is a lattice derived from integer vectors modulo q. For

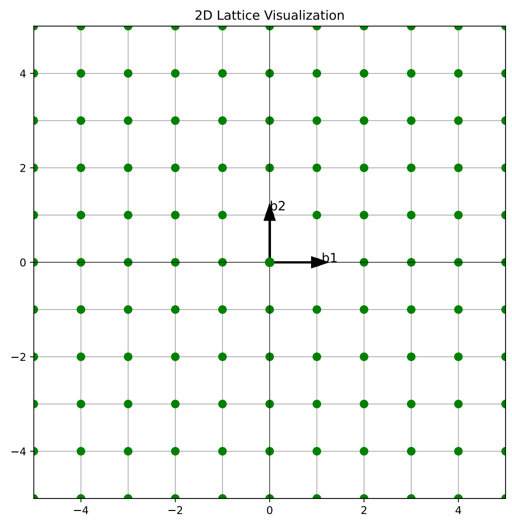

Figure 3.1: The simplest 2D lattice, with basis vectors $\mathbf{b}_1 = (1, 0)$ and $\mathbf{b}_2 = (0, 1)$

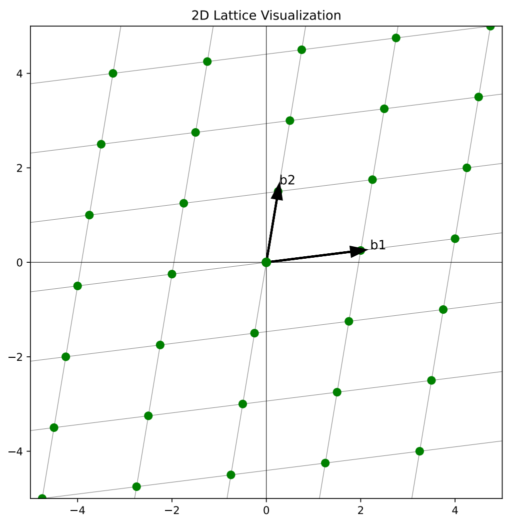

Figure 3.2: A more interesting 2D lattice, with basis vectors $\mathbf{b}_1 = (2, 0.25)$ and $\mathbf{b}_2 = (0.25, 1.5)$

positive integers n and q, let $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$ be arbitary and define the following full-rank m-dimensional q-ary lattices:

$$\Lambda^{\perp}(\mathbf{A}) := \{\mathbf{z} \in \mathbb{Z}^m : \mathbf{A}\mathbf{z} = \mathbf{0} \mod q\} \tag{3.3}$$

$$\Lambda(\mathbf{A}^t) := \left\{\mathbf{z} \in \mathbb{Z}^m : \exists \mathbf{s} \in \mathbb{Z}_q^n \quad \text{s.t.} \quad \mathbf{z} = \mathbf{A}^t\mathbf{s} \mod q\right\} \tag{3.4}$$

Note that $\Lambda^{\perp}(\mathbf{A})$ and $\Lambda(\mathbf{A}^t)$ are dual lattices, up to a scaling factor q: $q * \Lambda^{\perp}(\mathbf{A})^* = \Lambda(\mathbf{A}^t)$. For any $\mathbf{u}$ for which $\mathbf{A}\mathbf{x} = \mathbf{u} \mod q$, define the coset:

$$\Lambda_u^{\perp} = \{\mathbf{z} \in \mathbb{Z}^m : \mathbf{A}\mathbf{z} = \mathbf{u} \mod q\} = \Lambda^{\perp}(\mathbf{A}) + \mathbf{x} \tag{3.5}$$

Note that this new lattice can be considered simply as the original lattice shifted by $\mathbf{x}$. Importantly for our purposes is the following property shown (implicitly) by Agrawal *et al.* (22): for any invertible matrix $\mathbf{H} \in \mathbb{Z}_q^{n\times n}$, we have

$$\Lambda^{\perp}(\mathbf{H} \cdot \mathbf{A}) = \Lambda^{\perp}(\mathbf{A}) \tag{3.6}$$

Lattice problems are mostly formulated on the premise of computing some result given a basis. Some problems will be harder or easier to solve for the same lattice depending on the given basis of the lattice. Two important lattice problems are the *Shortest Integer Solution* (SIS) and *Learning With Errors* (LWE) problem which are covered in Sections 3.1.2 and 3.1.3 respectively.

3.1.1 Gaussians

Consider the n-dimensional Gaussian function $\rho : \mathbb{R}^n \to (0, 1]$, defined by

$$\rho(\mathbf{x}) \triangleq \exp(-\pi \cdot \|\mathbf{x}\|^2) = \exp(-\pi \cdot \langle \mathbf{x}, \mathbf{x}\rangle).$$

When applying a linear transformation via a matrix $\mathbf{B}$ with linearly independent columns (not necessarily square), we obtain a generalized or potentially degenerate Gaussian function. This function, denoted $\rho_{\mathbf{B}}$, is defined as:

$$\rho_{\mathbf{B}}(\mathbf{x}) \triangleq \begin{cases} \rho(\mathbf{B}^{+}\mathbf{x}) = \exp\left(-\pi \cdot \mathbf{x}^t \Sigma^{+} \mathbf{x}\right) & \text{if } \mathbf{x} \in \operatorname{span}(\mathbf{B}) = \operatorname{span}(\Sigma), \\ 0 & \text{otherwise,} \end{cases}$$

where $\Sigma = \mathbf{B}\mathbf{B}^t \geq 0$ denotes a positive semi-definite matrix.

Since $\rho_{\mathbf{B}}$ is uniquely determined only up to the span of Σ, we may simplify notation by writing $\rho_{\sqrt{\Sigma}}$. Normalizing this function over span(Σ) yields the continuous Gaussian distribution $D_{\sqrt{\Sigma}}$. We refer to Σ as the *covariance matrix* of the Gaussian distribution $D_{\sqrt{\Sigma}}$.

The *discrete Gaussian* (23) distribution $D_{\Lambda+\mathbf{c},\sqrt{\Sigma}}$ is defined as the distribution over $\Lambda + \mathbf{c}$, where $\Lambda \subset \mathbb{R}^n$, $\mathbf{c} \in \mathbb{R}^n$, and $\Sigma > \mathbf{0}$ is a positive semi-definite matrix such that $(\Lambda + \mathbf{c}) \cap \operatorname{span}(\Sigma) \neq \emptyset$. This is formally defined below.

Definition 1 (23) *For a positive semi-definite matrix Σ, the discrete Gaussian distribution over a lattice Λ, centered around c, is defined by the density function*

$$D_{\Lambda+\mathbf{c},\sqrt{\Sigma}}(\mathbf{x}) = \frac{\rho_{\sqrt{\Sigma}}(\mathbf{x})}{\rho_{\sqrt{\Sigma}}(\Lambda + \mathbf{c})} \propto \rho_{\sqrt{\Sigma}}(\mathbf{x}). \tag{3.7}$$

Specifically, when $\Sigma = s^2\mathbf{I}_n$ for some $s > 0$, the discrete Gaussian distribution is defined as:

$$D_{\Lambda,s}(x) := \frac{\rho_s(x)}{\sum_{y\in\Lambda} \rho_s(y)}, \tag{3.8}$$

where $\rho_s(x) = \exp(-\pi\|x\|^2/s^2)$.

3.1.2 Short Integer Solution

The Module Short Integer Solution (MSIS) problem is an *average*-case hard lattice problem that was first introduced by Ajtai in 1996 (10). For $\beta > 0$, the MSIS$_{q,\beta}$ problem is formulated as follows. Given a uniformly random matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$ for any $m = \text{poly}(n)$, find a relatively 'short' $\mathbf{z} \in \mathbb{Z}_q^m$, *s.t.* $\mathbf{A}\mathbf{z} = \mathbf{0} \mod q$ and $\|\mathbf{z}\| \leq \beta$. The Inhomogeneous MSIS problem, or I-MSIS$_{q,\beta}(\mathbf{u})$, is a variant of the MSIS problem, where we are tasked to find a small nonzero solution $\mathbf{z} \in \mathbb{Z}_q^m$ such that $\mathbf{A}\mathbf{z} = \mathbf{u} \mod q$ for some $\mathbf{u} \in \mathbb{Z}_q^n$.

The SIS problem is conjectured to be quantum-resistant, as it is believed to be hard to solve even with quantum computers. Under the assumption that SIS is hard in the average case, the following function is a (surjective) *collision-resistant hash function* (CRHF) (10):

$$f_{\mathbf{A}}(\mathbf{x}) := \mathbf{A}\mathbf{x} \mod q \in \mathbb{Z}_q^n, \tag{3.9}$$

3.1.3 Learning With Errors

The LWE$_{q,\alpha}$ problem was first introduced by Regev in 2005 (9). It has since seen various cryptographic applications in algorithms with conjectured quantum-resistance (24) (25).

For parameters $n \geq 1$, modulo $q \geq 2$, and error distribution function $D_{\mathbb{Z}^m,\alpha q}$ on $\mathbb{Z}_q$, the search-LWE$_{q,\alpha}$ problem is formulated as follows: recover arbitrary secret $\mathbf{s} \in \mathbb{Z}_q^n$ given the function

$$g_{\mathbf{A}}(\mathbf{s}, \mathbf{e}) = \mathbf{s}^t\mathbf{A} + \mathbf{e}^t \mod q, \tag{3.10}$$

where $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$ is a uniformly random matrix and $\mathbf{e} \leftarrow D_{\mathbb{Z}^m,\alpha q}$ is a small error vector. We can think of the problem as recovering an arbitary point in the lattice, perturbed by some error $\mathbf{e}$. The error should be within the unique decoding radius, *s.t.* the solution is uniquely determined. Note that if the error $\mathbf{e} = \mathbf{0}$ then $\mathbf{As} + \mathbf{e} = \mathbf{As} \in \Lambda(\mathbf{A}^t)$, in which case $\mathbf{s}$ can be recovered in polynomial time by using Gaussian elimination. We say that an algorithm solves search-LWE$_{q,\alpha}$ if given $g_{\mathbf{A}}(\mathbf{s}, \mathbf{e})$ for any $\mathbf{s} \in \mathbb{Z}_q^n$, it outputs $(\mathbf{s}, \mathbf{e})$ with high probability.

The decision-LWE$_{q,\alpha}$, or $\mathsf{D\text{-}LWE}_{q,\alpha}$, problem is to distinguish between $\mathbf{b}$ and a uniformly random sample from $\mathbb{Z}_q^m$.

3.2 Lattice trapdoors

Lattice-based cryptographic protocols that rely solely on hard problems tend to result mainly in collision-resistant hash functions and public key encryption schemes, with few other applications (26). More advanced applications, such as "hash-and-sign" digital signatures, IBE, ABE, and conjunction obfuscation, rely on the concept of "strong lattice trapdoors."

Lattice trapdoors, first introduced by Gentry *et al.* (27), essentially allow one to efficiently invert one-way lattice functions by providing a secret key that can be used to recover solutions that would otherwise be computationally infeasible to find. This technique requires sampling from a discrete Gaussian distribution over a lattice, which is usually a computationally intensive process.

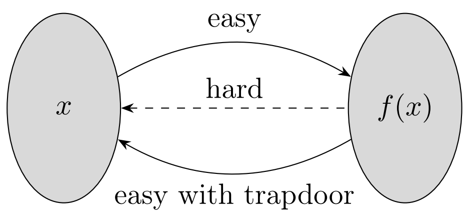

Figure 3.3: Simple visualization of trapdoor functions. Inversion of $f(x)$ is hard, but with a trapdoor, it becomes easy.

Micciancio and Peikert (26) introduce simple and efficient methods for generating and using lattice trapdoors. This section provides a brief overview of their construction, which is used in the TDUE scheme (25). Although there have since been various improvements to the original construction (28, 29), the basic principles remain the same.

The one-way functions $f_{\mathbf{A}}(\mathbf{x})$ and $g_{\mathbf{A}}(\mathbf{s},\mathbf{e})$ that we aim to invert are defined based on the SIS and LWE problems (see equations 3.9 and 3.10). Since $f_{\mathbf{A}}(\mathbf{x})$ is a "compressing" CRHF (i.e., surjective), there are many possible preimages $\mathbf{x}$ for a given output $f_{\mathbf{A}}(\mathbf{x})$. Given $\mathbf{u} = f_{\mathbf{A}}(\mathbf{x}) = \mathbf{A}\mathbf{x} \bmod q$ for some $\mathbf{x} \in \mathbb{Z}_q^m$, our goal is to sample a preimage $\mathbf{x}' = f_{\mathbf{A}}^{-1}(\mathbf{u})$ with probability proportional to $\exp(-\|\mathbf{x}'\|^2/\sigma^2)$. This is shown by (26). For the function $g_{\mathbf{A}}(\mathbf{s},\mathbf{e})$, we want to efficiently recover the unique pair $(\mathbf{s},\mathbf{e})$ corresponding to a given $g_{\mathbf{A}}(\mathbf{s},\mathbf{e})$.

The approach can be broadly summarized in three steps:

1. **Inversion For Special Structures:** find functions $f_{\mathbf{G}}^{-1} : \mathbb{Z}_q^n \to \mathbb{Z}_q^m$ and $g_{\mathbf{G}}^{-1} : \mathbb{Z}_q^m \to (\mathbb{Z}_q^n, \mathbb{Z}_q^m)$ for some special matrix $\mathbf{G}$.
2. **Trapdoor Generation:** randomize matrix $\mathbf{G}$ to some matrix $\mathbf{A}$.
3. **Trapdoor Operation:** find functions $f_{\mathbf{A}}^{-1} : \mathbb{Z}_q^n \to \mathbb{Z}_q^m$ and $g_{\mathbf{A}}^{-1} : \mathbb{Z}_q^m \to (\mathbb{Z}_q^n, \mathbb{Z}_q^m)$ for $\mathbf{A}$.

We'll describe these steps in more detail below.

3.2.1 Inversion For Special Structures

The construction of trapdoors is based on a special family of lattices, which allow for fast and parallelizable decoding algorithms. These lattices are defined by *primitive matrices.* A matrix $\mathbf{G} \in \mathbb{Z}_q^{n\times m}$ is called a primitive matrix if its columns generate all of $\mathbb{Z}_q^n$, *i.e.,* $\mathbf{G} \cdot \mathbb{Z}^m = \mathbb{Z}_q^n$.

Micciancio and Peikert (26) start by constructing a *primitive vector* $\mathbf{g} \in \mathbb{Z}_q^k$ *s.t.* $\gcd(g_1, \ldots, g_k, q) = 1$, where q is an integer modulus and $k \geq 1$ is an integer dimension. Concretely, they consider the following structure for $\mathbf{g}$:

$$\mathbf{g}^t := \begin{bmatrix} 1 \; 2 \; 4 \; \ldots \; 2^{k-1} \end{bmatrix} \in \mathbb{Z}_q^k, \quad k = \lceil \log_2 q \rceil, \tag{3.11}$$

which generates the lattice $\Lambda^{\perp}(\mathbf{g}^t) \subset \mathbb{Z}^k$.

Let $\mathbf{S}_k \in \mathbb{Z}^{k\times k}$ be a basis of $\Lambda^{\perp}(\mathbf{g}^t)$, such that $\mathbf{g}^t \cdot \mathbf{S}_k = \mathbf{0} \in \mathbb{Z}_q^{1\times k}$. Now $\mathbf{g}$ and $\mathbf{S}_k$ are used to construct the gadget matrix $\mathbf{G}$ and basis $\mathbf{S}$ as $\mathbf{G} := \mathbf{I}_n \otimes \mathbf{g}^t \in \mathbb{Z}_q^{n\times nk}$ and $\mathbf{S} := \mathbf{I}_n \otimes \mathbf{S}_k \in \mathbb{Z}^{nk\times nk}$, illustrated below:

$$\mathbf{G} := \begin{bmatrix} \cdots & \mathbf{g}^t & \cdots & & & & \\ & \cdots & \mathbf{g}^t & \cdots & & & \\ & & & \ddots & & & \\ & & & & \cdots & \mathbf{g}^t & \cdots \end{bmatrix} \in \mathbb{Z}_q^{m\times nk}, \quad \mathbf{S} := \begin{bmatrix} \mathbf{S}_k & & & \\ & \mathbf{S}_k & & \\ & & \ddots & \\ & & & \mathbf{S}_k \end{bmatrix} \in \mathbb{Z}_q^{nk\times nk}.$$

Consequently, $\mathbf{G}$ is primitive and the lattice $\Lambda^{\perp}(\mathbf{G})$ has basis $\mathbf{S}$.

Sampling preimages of $f_{\mathbf{G}}(\mathbf{x})$ and inverting $g_{\mathbf{G}}(\mathbf{s},\mathbf{e})$ can be accomplished by solving n parallel instances of $f_{\mathbf{g}^t}(\mathbf{x})$ and $g_{\mathbf{g}^t}(s,\mathbf{e})$, respectively. For the former, if each of the $f_{\mathbf{g}^t}(\mathbf{x})$ preimages has Gaussian parameter $\sqrt{\Sigma}$, then by independence, their concatenation has parameter $\mathbf{I}_n \otimes \sqrt{\Sigma}$. Similarly for the latter, if each of the $g_{\mathbf{g}^t}(s,\mathbf{e})$-inversion subproblems are solved, then we can invert $g_{\mathbf{G}}(\mathbf{s},\mathbf{e})$. In the case that $q = 2^k$ is a power of 2, these functions can be efficiently sampled and inverted using the following methods.

Let $\mathbf{g}$ be the primitive vector defined above, and let $\mathbf{S}_k$ be defined as follows:

$$\mathbf{S}_k := \begin{bmatrix} 2 & & & & \\ -1 & 2 & & & \\ & -1 & \ddots & & \\ & & & 2 & \\ & & & -1 & 2 \end{bmatrix} \in \mathbb{Z}^{k \times k}.$$

For $g_{\mathbf{g}^t}(s, \mathbf{e})$, Micciancio and Peikert (26) describe a simple and efficient algorithm to find an unknown scalar $s \in \mathbb{Z}_q$ given the vector

$$\begin{aligned} \mathbf{b}^t &= [b_0, b_1, \dots, b_{k-1}] \\ &= s \cdot \mathbf{g}^t + \mathbf{e}^t \\ &= \left[s + e_0, 2s + e_1, \dots, 2^{k-1}s + e_{k-1}\right] \mod q, \end{aligned}$$

with small error $\mathbf{e} \in \mathbb{Z}^k$. An iterative algorithm which recovers the binary digits of s from least to most significant is illustrated in Algorithm 1. This algorithm is based on Babai's nearest plane algorithm (30), adapted to the scaled dual lattice $q(\mathbf{S}_k)^{-t}$ of the basis $\mathbf{S}_k$, which is a basis for $\Lambda(\mathbf{g})$.

Algorithm 1 Iterative Inversion of $g_{\mathbf{g}^t}$ (26)

```
Require: Vector $\mathbf{b}^t = [b_0, \dots, b_{k-1}] \in \mathbb{Z}_q^{1 \times k}$
Ensure: $s \in \mathbb{Z}_q$ and $\mathbf{e} = (e_0, \dots, e_{k-1}) \in [-\frac{q}{4}, \frac{q}{4})^k$
1: $s \leftarrow 0$
2: for $i = k-1$ down to $0$ do
3:     $temp \leftarrow b_i - 2^i \cdot s \mod q$
4:     if $temp \notin [-\frac{q}{4}, \frac{q}{4})$ then
5:         $s \leftarrow s + 2^{k-1-i}$
6:     end if
7:     $e_i \leftarrow b_i - 2^i \cdot s \mod q$                ▷ Ensure $e_i \in [-\frac{q}{4}, \frac{q}{4})$
8: end for
9: return $s, \mathbf{e}$
```

For the sampling of preimages for $f_{\mathbf{g}^t}$, we would like to sample with some Gaussian distribution a vector from the set $\Lambda_u^{\perp}(\mathbf{g}^t) = \{\mathbf{x} \in \mathbb{Z}^k : \langle \mathbf{g}, \mathbf{x} \rangle = u \mod q\}$ for some $u \in \mathbb{Z}_q$. The algorithm itself is a simple iterative process, which is illustrated in Algorithm 2. This is a simplified version of the randomized nearest-plane algorithm presented in (27, 31).

The functions $f_{\mathbf{G}}^{-1}(\mathbf{u})$ and $g_{\mathbf{G}}^{-1}(\mathbf{b})$ can now be trivially reduced to n parallel and offline calls to $f_{\mathbf{g}^t}^{-1}(u)$ and $g_{\mathbf{g}^t}^{-1}(b)$, respectively.

3.2.2 Trapdoor Generation

Let $m \geq nk \geq n$ be integer dimensions. Define a $\mathbf{G}$-trapdoor for a matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ as a matrix $\mathbf{R} \in \mathbb{Z}_q^{(m-nk) \times nk}$ such that $\mathbf{A} \begin{bmatrix} \mathbf{R} \\ \mathbf{I} \end{bmatrix} = \mathbf{HG}$ for some invertible matrix $\mathbf{H} \in \mathbb{Z}_q^{n \times n}$.

Algorithm 2 Iterative Preimage Sampling for $f_{\mathbf{g}^t}$ (26)

```
Require: Syndrome $u \in \{0, \dots, q-1\}$
1: for $i = 0, \dots, k-1$ do
2:     $x_i \leftarrow D_{2\mathbb{Z}+u,s}$
3:     $u \leftarrow (u - x_i)/2 \in \mathbb{Z}$
4: end for
5: return $\mathbf{x} = (x_0, \dots, x_{k-1})$
```

Concretely, a choice for $\mathbf{A}$ could be $\mathbf{A} = [\mathbf{A}_0 | -\mathbf{A}_0\mathbf{R} + \mathbf{HG}]$, where $\mathbf{A}_0 \in \mathbb{Z}_q^{n\times m}$ is a uniform matrix, $\mathbf{H} \in \mathbb{Z}_q^{n\times n}$ is an invertible matrix, and $\mathbf{G}$ is the gadget matrix defined above with $k = \lceil \log_2 q \rceil$ (26). It follows from the leftover hash lemma (23) that $-\mathbf{A}_0\mathbf{R}+\mathbf{HG}$ is (very close) to uniformly random.

3.2.3 Trapdoor Operation

Recall that $\mathbf{A}\begin{bmatrix}\mathbf{R}\\ \mathbf{I}\end{bmatrix} = \mathbf{HG}$. We can reduce an LWE instance $\mathbf{b}^t = \mathbf{s}^t\mathbf{A} + \mathbf{e}^t$ to an instance of $g_{\mathbf{G}}(\mathbf{s}, \mathbf{e})$ by multiplying the equation by $\begin{bmatrix}\mathbf{R}\\ \mathbf{I}\end{bmatrix}$, yielding

$$\mathbf{b}^t\begin{bmatrix}\mathbf{R}\\ \mathbf{I}\end{bmatrix} = \mathbf{s}^t(\mathbf{HG}) + \mathbf{e}^t\begin{bmatrix}\mathbf{R}\\ \mathbf{I}\end{bmatrix}, \tag{3.12}$$

if $\mathbf{R}$ is a $\mathbf{G}$-trapdoor for $\mathbf{A}$. It is clear that this increases the error, so care must be taken to ensure that $\mathbf{e}^t\begin{bmatrix}\mathbf{R}\\ \mathbf{I}\end{bmatrix} \in [-q/4, q/4)$.

Recall Equation 3.6. We can invert Equation 3.12 to recover $(\hat{\mathbf{s}}, \hat{\mathbf{e}})$ *s.t.* $\mathbf{s} = \mathbf{H}^{-t}\hat{\mathbf{s}}$ and $\mathbf{e} = \mathbf{b} - \mathbf{A}^t\mathbf{s}$. This process is encapsulated in a single function in the following lemma.

Lemma 1 ((26), Theorem 5.4) *Let $\mathbf{R}$ be a $\mathbf{G}$-trapdoor for $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$ with an invertible tag matrix $\mathbf{H}$, and consider an LWE sample $\mathbf{b}^t = \mathbf{s}^t\mathbf{A} + \mathbf{e}^t$. If $\|\mathbf{R}^t\mathbf{I}\|_\infty \le q/4$, then there exists an efficient procedure Invert$(\mathbf{R}, \mathbf{A}, \mathbf{H}, \mathbf{b})$ that efficiently recovers $(\mathbf{s}, \mathbf{e})$ from $\mathbf{b}^t = \mathbf{s}^t\mathbf{A} + \mathbf{e}^t$.*

In order to sample preimages of $\mathbf{u} = f_{\mathbf{A}}(\mathbf{x}) = \mathbf{Ax}$ with some spherical Gaussian distribution, we first sample a vector $\mathbf{z} \leftarrow f_{\mathbf{G}}^{-1}(\mathbf{u})$. Naively, we would then compute $\mathbf{x} = \mathbf{R}^t\mathbf{z}$, which would yield

$$\mathbf{Ax} = \mathbf{A}\cdot\begin{bmatrix}\mathbf{R}\\ \mathbf{I}\end{bmatrix}\cdot\mathbf{z} = \mathbf{Gz} = \mathbf{u} \mod q.$$

However, this leads to a grave problem; the covariance with which $\mathbf{x}$ is sampled leaks information about the trapdoor $\mathbf{R}$. The covariance of $\mathbf{x}$ is given by

$$\Sigma = \mathbb{E}[\mathbf{xx}^t] = \mathbb{E}[\mathbf{R}\cdot\mathbf{zz}^t\mathbf{R}^t] \approx s^2\mathbf{RR}^t,$$

Ideally, this should be a spherical Gaussian distribution with covariance $s^2\mathbf{I}$.

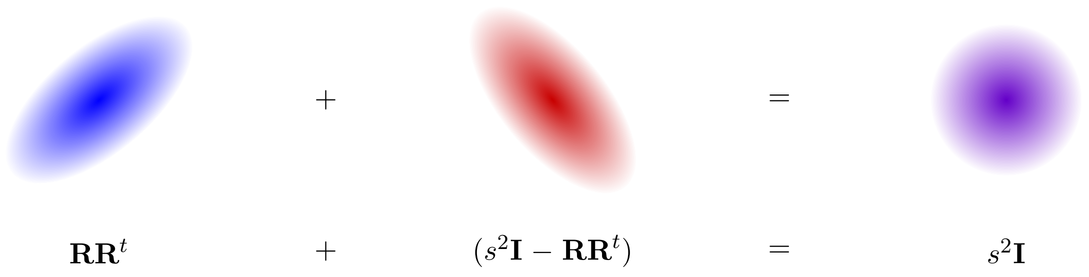

Figure 3.4: Perturbation technique to correct covariance leakage. The leftmost distribution leaks information about the trapdoor $\mathbf{R}$. The middle distribution is the perturbation, which is added to the left distribution to yield the right-hand spherical Gaussian distribution. Figure adapted from Agrawal (1).

To correct for this, we use the perturbation technique described in (32). First, we generate a perturbation vector $\mathbf{p}$ with covariance $s^2\mathbf{I} - \mathbf{RR}^t$. Then, we sample a vector $\mathbf{z} \leftarrow f_{\mathbf{G}}^{-1}(\mathbf{u} - \mathbf{Ap})$ and compute the final vector as $\mathbf{x} = \mathbf{p} + \begin{bmatrix}\mathbf{R}\\ \mathbf{I}\end{bmatrix} \cdot \mathbf{z}$. This yields

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{Ap} + \mathbf{A} \cdot \begin{bmatrix}\mathbf{R}\\ \mathbf{I}\end{bmatrix} \cdot \mathbf{z} = \mathbf{Ap} + \mathbf{Gz} = \mathbf{u}.$$

Figure 3.4 illustrates the perturbation technique. The complete sampling process is summarized in the following lemma, which encapsulates the above steps in a single function.

Lemma 2 ((26), Theorem 5.5) *Let $\mathbf{R}$ be a $\mathbf{G}$-trapdoor for $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$, with an invertible matrix $\mathbf{H}$, and let $\mathbf{u} \in \mathbb{Z}_q^n$ be arbitrary. There exists an efficient algorithm SampleD$(\mathbf{R}, \mathbf{A}, \mathbf{H}, \mathbf{u}, s)$, which outputs a vector $\mathbf{x}$ with probability distribution $D_{\mathbb{Z}^m,s}$, such that $\mathbf{Ax} = \mathbf{u}$. s can be as small as $\sqrt{s_1(\mathbf{R})^2 + 1} \cdot \sqrt{s_1(\Sigma_{\mathbf{G}}) + 1} \cdot \sqrt{\log n}$, where $s_1(\Sigma_{\mathbf{G}})$ is a constant depending on $\mathbf{G}$ (specifically, 4 if q is a power of 2, and 5 otherwise).*

3.3 Digital Signature

Digital signatures can be used to verify authenticity of digital messages or documents. They are used in applications ranging from file distribution to financial transactions and are vital in preventing tampering of data. Digital signature schemes ensure that an entity that has signed a message cannot deny having signed the message at a later time. This property is known as *non-repudiation*.

In general, a digital signature scheme consists of three algorithms:

- **Key Generation**: This algorithm outputs a pair of keys, a public key **pk** and a secret key **sk**. The secret key is kept private by the signer, while the public key is distributed to anyone who needs to verify signatures. The secret key should be sampled uniformly at random from a set of possible secret keys.

- **Signing**: This algorithm takes the secret key **sk** and a message m as input, and outputs a signature σ. The signature is typically attached to the message, and can be used to verify that the message was signed by the holder of the secret key.
- **Verification**: This algorithm takes the public key **pk**, a message m, and a signature σ as input, and outputs either *accept* or *reject*.

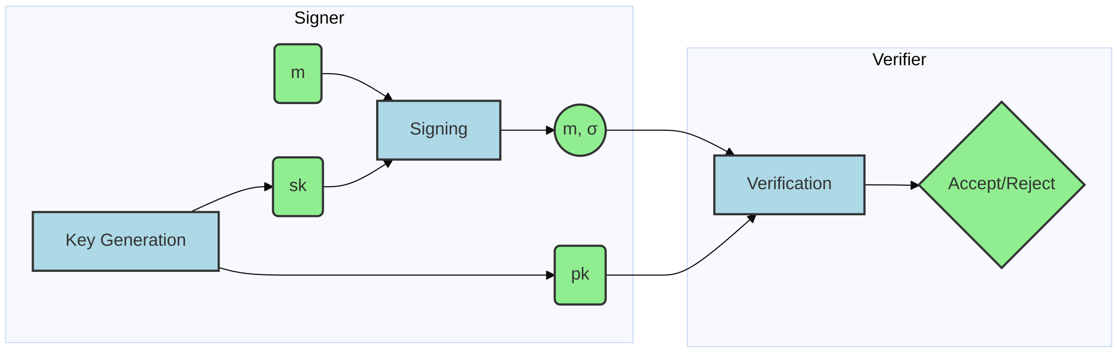

Figure 3.5: Simple visualisation of a digital signature scheme. After key generation, the signer uses their secret key (sk) to sign a message (m), producing a signature (σ). The verifier then uses the public key (pk) to verify the signature on the message.

A digital signature scheme is secure if it satisfies the following properties:

- the authenticity of the signature can be verified by anyone who has access to the public key, and
- it should be infeasible for an adversary to forge a valid signature for a message that was not signed by the legitimate signer.

As an example of a digital signature scheme, consider the Schnorr signature scheme (2). Its security is based on the hardness of the discrete logarithm problem. The Schnorr signature scheme defines the following algorithms:

- **Key Generation**: Let p be a large prime number, q a prime divisor of $p-1$, and g a generator of the cyclic group $\mathbb{Z}_p^*$. Choose uniformly at random $x \in_R \mathbb{Z}_q$. The public key is then $y = g^{-x} \mod p$. Output $(\mathsf{pk}, \mathsf{sk}) = (y, x)$.
- **Signing**: Choose random nonce $k \in \mathbb{Z}_q$ and compute the *challenge* $r = g^k \mod p$. Then, compute the hash *commitment* $e = H(r||m) \in \mathbb{Z}_q$, where H is a cryptographic hash function. Finally, the signature (*response*) is computed as $\sigma = k + xe \mod (p-1)$. The output is the signature (e, σ). Refer to Section 3.5.1 for details on this structure.
- **Verification**: To verify a signature σ for a message m, the verifier computes $r' = g^\sigma y^e \mod p$ and checks if $H(m||r') = e$. If this holds, the signature is valid; otherwise, it is invalid.

It is easy to see that the Schnorr signature scheme is complete; if the (honest) signer uses their secret key x to sign a message m, then the verifier computes $r' = g^\sigma y^e = g^{k+xe}g^{-xe} = g^k = r$, and thus $H(m||r') = H(m||r)$, which is equal to e.

Proving the security of the Schnorr signature scheme is more involved, but a proof by Seurin (33) shows that the scheme is secure (in the *random oracle model*) under the assumption that the discrete logarithm problem is hard.

3.4 Updatable encryption

Data stored on a semi-honest remote (cloud) server is typically encrypted by the user (client-side). Frequently changing encryption keys is widely regarded as good security practice to reduce the risk of key compromise (NIST recommends rotating keys at least once a year (34)). *Updatable encryption*, first defined by Boneh *et al.* in 2013 (35), enables the server to update ciphertexts to a new encryption key without sending it back to the user. Given the old and new keys, the user generates a *key-switching token* and sends it to the server. The server applies the token to the ciphertext, which becomes fully encrypted under the new key. The server learns nothing about the plaintext, and the scheme conserves bandwidth, which is especially useful for large ciphertexts. We distinguish two paradigms of updatable encryption: ciphertext-dependent (c-d UE) and ciphertext-independent (c-i UE). C-i UE generates tokens independently of the ciphertext, while c-d UE uses a portion of the ciphertext (the 'header') in token generation.

We briefly summarize the syntax of c-d UE below.

Definition 2 (25, 36) *A ciphertext-dependent UE scheme consists of the following algorithms which operate in epochs starting from 0:*

- $\mathsf{KG}(1^\lambda)$*: Generate an epoch key* $\mathsf{k_e}$.
- $\mathsf{Enc}(\mathsf{k_e}, \mathsf{msg})$*: Encrypt a message* msg *under the key* $\mathsf{k_e}$*, producing a ciphertext* c.
- $\mathsf{TokenGen}(\mathsf{k_e}, \mathsf{k_{e+1}}, \hat{\mathsf{ct}}_\mathsf{e})$*: Take two epoch keys* $\mathsf{k_e}$ *and* $\mathsf{k_{e+1}}$*, and a ciphertext header* $\hat{\mathsf{ct}}_\mathsf{e}$*, and output either a key-switching token* $\Delta_{\mathsf{e+1},\hat{\mathsf{ct}}_\mathsf{e}}$ *or* $\perp$.
- $\mathsf{Update}(\Delta_{\mathsf{e+1},\hat{\mathsf{ct}}_\mathsf{e}}, (\hat{\mathsf{ct}}_\mathsf{e}, \mathsf{ct_e}))$*: Take a token* $\Delta_{\mathsf{e+1},\hat{\mathsf{ct}}_\mathsf{e}}$ *and a ciphertext* $(\hat{\mathsf{ct}}_\mathsf{e}, \mathsf{ct_e})$*, and output a new ciphertext* $(\hat{\mathsf{ct}}_\mathsf{e}, \mathsf{ct_e}')$ *or* $\perp$*. This essentially re-encrypts the ciphertext under the new epoch key* $\mathsf{k_{e+1}}$.
- $\mathsf{Dec}(\mathsf{k_e}, (\hat{\mathsf{ct}}_\mathsf{e}, \mathsf{ct_e}))$*: Decrypt a ciphertext* $(\hat{\mathsf{ct}}_\mathsf{e}, \mathsf{ct_e})$ *using the epoch key* $\mathsf{k_e}$*, returning the plaintext message* msg' *or* $\perp$.

Ciphertext-independent updatable encryption (c-i UE) is a special case of c-d UE where the ciphertext header $\hat{\mathsf{ct}}_\mathsf{e}$ is empty.

A UE scheme is correct if for any epoch e, any message msg, and any ciphertext $(\hat{\mathsf{ct}}_\mathsf{e}, \mathsf{ct_e})$ generated by the scheme, the output of $\mathsf{Dec}(\mathsf{k_e}, (\hat{\mathsf{ct}}_\mathsf{e}, \mathsf{ct_e}))$ equals msg with overwhelming probability.

3.5 Zero-Knowledge Proof

Zero-Knowledge Proofs (ZKP) allow one to prove knowledge of a *statement* without revealing any information beyond the statement's truth. The concept of ZKPs was introduced by Goldwasser, Micali, and Rackoff in 1985 (37). To qualify as a zero-knowledge proof, the proof must satisfy three properties:

1. **Completeness**: If the statement is true, the verifier will invariably accept the proof.
2. **Soundness**: If the statement is not true, a (cheating) prover will not be able to convince the verifier of its validity, except with negligible probability.
3. **Zero-knowledge**: There exists a *simulator* that, given only the statement, can produce transcripts of an interaction between a prover and a verifier that are indistinguishable from real transcripts. Informally, this means if the statement is true, then the verifier gains no information beyond the truth of the statement.

Zero-knowledge proofs enable secure, private verification across various domains. In cryptocurrencies like Zcash, they allow transaction validation without revealing amounts or parties (38). In digital identity, they enable users to prove age or credentials without exposing personal data (39). Voting systems use them to ensure vote validity while preserving voter anonymity. ZKPs also enhance blockchain scalability through zk-rollups, allowing off-chain computation with on-chain verification.

3.5.1 Interactive vs. Non-Interactive Zero-Knowledge Proofs

Zero-knowledge proofs can be broadly categorized into interactive and non-interactive types, distinguished by the communication requirements between the prover and the verifier.

Interactive Zero-Knowledge Proofs (IZK): The original concept of ZKPs (37) involve multiple rounds of interaction between the prover and the verifier. Interactive ZKPs typically require 3 phases: commitment, challenge, and response. During the commitment phase, the prover sends a commitment to the statement they want to prove. In the challenge phase, the verifier sends a random challenge to the prover, which is typically a random value or a question related to the statement. Finally, in the response phase, the prover sends a response based on their commitment and the verifier's challenge. The verifier then checks if the response is valid according to the rules of the protocol.

Non-Interactive Zero-Knowledge Proofs (NIZK): In contrast, NIZKs require only a single message from the prover to the verifier. This makes NIZKs highly desirable for applications where real-time interaction is impractical or impossible, such as in blockchain technologies. The challenge in designing NIZKs lies in achieving the zero-knowledge property without the interactive element. This is often accomplished by relying on a common reference string (CRS) or by employing cryptographic hashes in a specific way, such as through the Fiat-Shamir heuristic.

This technique, proposed by Amos Fiat and Adi Shamir in 1987 (40), is used to transform interactive ZKPs into non-interactive ones. The core idea behind the Fiat-Shamir heuristic is to replace the verifier's challenges, which are typically random values chosen by the verifier in an interactive protocol, with challenges derived from a cryptographic hash function. Since the hash function is assumed to be a *random oracle* (a theoretical ideal hash function that produces truly random outputs for any input), the prover cannot predict the challenge before generating its initial commitments. This effectively simulates the verifier's random challenge without actual interaction.

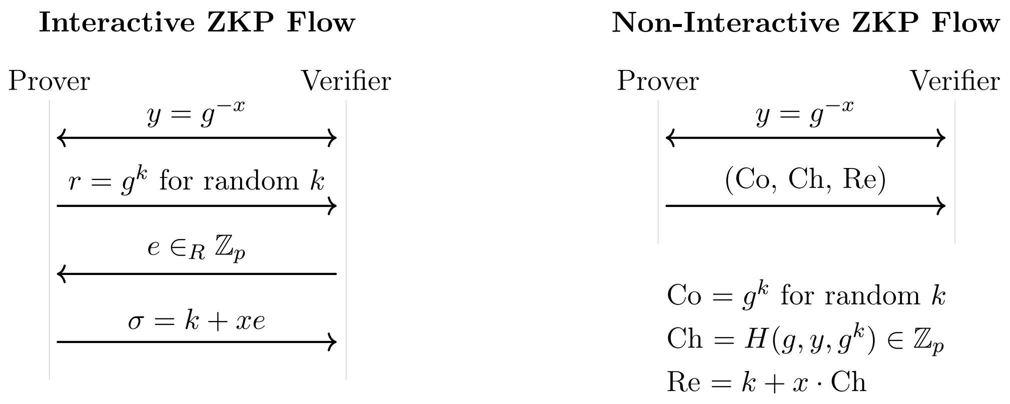

Figure 3.6: Comparison of Interactive and Non-Interactive Zero-Knowledge Proofs using Schnorr's scheme (2). The interactive ZKP requires multiple rounds of communication, while the non-interactive ZKP uses a single message with a cryptographic hash function to simulate the verifier's challenge.

3.5.2 zk-SNARKs

Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zk-SNARK) was first introduced by Bitansky *et al.* in 2012 (41). It is a specific type of non-interactive zero-knowledge proof that has gained significant attention due to its small proof sizes and fast verification times in various cryptographic systems, particularly in blockchain and cryptocurrency. zk-SNARKS use arithmetic circuits to represent statements. We'll first define arithmetic circuits, then explain the zk-SNARK structure.

Definition 3 *(Arithmetic Circuits) Let $\mathbb{F} = 0, \dots, p-1$ be a finite field for some prime $p > 2$. An arithmetic circuit $C : \mathbb{F}^n \to \mathbb{F}$ is a directed acyclic graph (DAG) where each node represents either an input or an arithmetic operation (addition or multiplication), and the edges represent the flow of data. Each internal node, or* gate*, has two inputs and one output, while the input nodes have no incoming edges. All $n+1$ input nodes are labeled with variables $1, x_1, \dots, x_n$. We can use arithmetic circuits to represent n-variate polynomial functions and its evaluation. The size of a circuit C is given by $|C|$, which is the number of internal nodes in the circuit.*

An argument system is a tuple of functions (S, P, V) for the setup, prover, and verifier, respectively. In the context of argument systems, we construct a public arithmetic circuit $C(x, w) \to \mathbb{F}$, where x is the statement and w is the witness. A prover might want to convince a verifier, who is exclusively in possession of a statement x, that they hold knowledge of a witness w such that $C(x, w) = 0$. We've seen the difference between interactive and non-interactive argument systems in the previous Section 3.5.1.

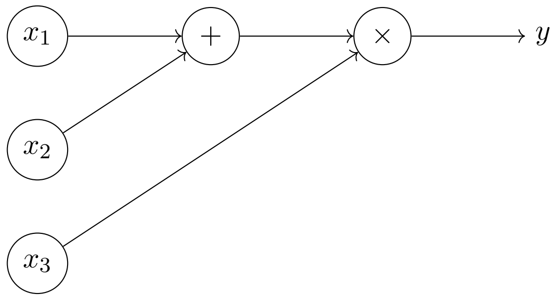

Figure 3.7: Example of a 2-gate, 3-input arithmetic circuit for $y = (x_1 + x_2) \cdot x_3$.

In the case of Non-Interactive argument systems, we typically preprocess the arithmetic circuit. This setup phase $S(C) \to (S_p, S_v)$ generates some public parameters (*e.g.,* CRS) for both the prover and the verifier. These are then used by the prover to generate a proof $P(S_p, x, w) \to \pi$ for a witness w such that $C(x, w) = 0$. The verifier can then check the validity of the proof by running $V(S_v, x, \pi) \to \{\text{accept}, \text{reject}\}$. Informally, an argument system is called an Argument of Knowledge for a circuit C if, whenever the verifier V accepts a proof π, it is guaranteed that there exists an efficient way to extract a witness w from the prover P.

Succinctness of the system is characterized by the proof size and verification time. In zk-SNARKs, the proof size and verification time are typically logarithmic in the size of the circuit (42, 43). This is in part possible due to the preprocessing of the circuit performed during the setup phase.

There are generally three types of setup phases, listed below in decreasing trust requirements:

1. **Trusted Setup**: A trusted party generates the public parameters, which are then used by both the prover and the verifier. The security of the system relies on the assumption that this trusted party does not collude with any malicious prover.

2. **(Trusted) Universal Setup**: A universal setup phase generates parameters that can be used for any circuit of a certain class, without needing to know the specific circuit in advance. This allows for greater flexibility and reusability of the parameters across different circuits.

3. **Transparent Setup**: In this case, the setup phase is performed in a way that does not require trusted parties. Instead, it relies on publicly verifiable randomness or cryptographic techniques to ensure that the parameters are secure and cannot be manipulated by any party.

As a brief example of preprocessing in zk-SNARKs, consider the Groth16 protocol (43), which is a constant proof size and verification time zk-SNARK construction based on elliptic curve pairings. Groth16 is a widely used zk-SNARK protocol in the blockchain space, particularly for its extremely small proof size and fast verification. In Groth16, arithmetic circuits are first transformed into their Rank 1 Constraint System (R1CS) representation.

From Arithmetic Circuits to R1CS. Let m be the number of gates in the arithmetic circuit. Define the set of variables in the arithmetic circuit as all input variables, all output variables (for each gate), and a constant dummy variable 1. Let the size of this set be n. Each constraint is represented as a tuple of vectors $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ and a witness vector $\mathbf{s}$ such that $\langle \mathbf{a}, \mathbf{s} \rangle \cdot \langle \mathbf{b}, \mathbf{s} \rangle = \langle \mathbf{c}, \mathbf{s} \rangle$ holds. Each element of the vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}$ corresponds to a variable in the arithmetic circuit, while the witness vector $\mathbf{s}$ contains the concrete values of these variables. The vectors are grouped into matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$, *s.t.* $\mathbf{As} \cdot \mathbf{Bs} = \mathbf{Cs}$ holds. This essentially encodes the arithmetic circuit into a set of m linear constraints with n variables that must be satisfied by the witness vector $\mathbf{s}$. Figure 3.8 illustrates a concrete example of R1CS conversion.

Step 1: Defining Constraints

$$t_1 = x_1 + x_2$$

$$t_2 = t_1 \times x_3$$

$$\downarrow$$

Step 2: Vector Representation

Variables: $[1, x_1, x_2, x_3, t_1, t_2]$

$$\swarrow \qquad \searrow$$

For $t_1 = x_1 + x_2$:
$\mathbf{a} = [0, 1, 1, 0, 0, 0]$
$\mathbf{b} = [1, 0, 0, 0, 0, 0]$
$\mathbf{c} = [0, 0, 0, 0, 1, 0]$

For $t_2 = t_1 \times x_3$:
$\mathbf{a} = [0, 0, 0, 0, 1, 0]$
$\mathbf{b} = [0, 0, 0, 1, 0, 0]$
$\mathbf{c} = [0, 0, 0, 0, 0, 1]$

$$\searrow \qquad \swarrow$$

Step 3: Matrix Representation

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \quad \mathbf{C} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 3.8: Conversion of the arithmetic circuit seen in Figure 3.7 into a Rank 1 Constraint System (R1CS) representation.

From R1CS to QAP. The R1CS representation is transformed into a quadratic arithmetic program (QAP) representation, which is a polynomial representation of the constraints. The m constraint vectors of length n are transformed into n polynomials of degree m. A polynomial $a_i(j)$ for $i \in \{1, \ldots, m\}, j \in \{1, \ldots, n\}$ should evaluate to the j-th element of the a component for the i-th constraint. Lagrange interpolation is used to construct these polynomials. Polynomials $a_i(x)$ are grouped into $\mathbf{A}(x) = [a_1(x), \ldots, a_m(x)]$. When evaluated at a point x, $\mathbf{A}(x)$ returns the values of the a components for constraint x.

The same process is applied to the b and c components, resulting in $\mathbf{B}(x)$ and $\mathbf{C}(x)$. Verifying the R1CS constraints can then be reduced to checking $\langle \mathbf{A}(x), \mathbf{s} \rangle \cdot \langle \mathbf{B}(x), \mathbf{s} \rangle = \langle \mathbf{C}(x), \mathbf{s} \rangle$ for all $x \in \{1, \ldots, n\}$. This is equivalent to verifying

$$\langle \mathbf{A}(x), \mathbf{s} \rangle \cdot \langle \mathbf{B}(x), \mathbf{s} \rangle - \langle \mathbf{C}(x), \mathbf{s} \rangle = H(x) \cdot T(x), \tag{3.13}$$

where typically $T(x) = (x-1)(x-2)\cdots(x-n)$, a polynomial that vanishes at all points $x \in \{1, \ldots, n\}$. For conciseness, we rephrase Equation 3.13 as

$$L_{\mathbf{s}}(x) \cdot R_{\mathbf{s}}(x) - O_{\mathbf{s}}(x) = H(x) \cdot T(x). \tag{3.14}$$

To check this equality efficiently, we rely on the Schwartz-Zippel lemma (44), which guarantees that if two polynomials agree at a randomly chosen point, then with high probability the polynomials are identical.

Proving knowledge of a witness. Let $[\cdot]_E$ be a *homomorphic hiding* function, which is a function that hides the input while allowing additive and multiplicative operations to be performed on the hidden values (bilinear elliptic curve pairings, in the case of Groth16). The intuition behind the scheme is that the verifier chooses a random point $t \in \mathbb{F}$ unknown to the prover and comoputes $T(t)$. The verifier sends t to the prover, who then evaluates the polynomials at that point and sends back the hidden evaluations $[L_{\mathbf{s}}(t)]_E$, $[R_{\mathbf{s}}(t)]_E$, $[O_{\mathbf{s}}(t)]_E$, and $[H(t)]_E$. The prover shifts the evaluations of the left-hand side equations by a random offset of $T(t)$ to ensure zero-knowledge over multiple runs. The verifier checks that $[L_{\mathbf{s}}(t)R_{\mathbf{s}}(t) - O_{\mathbf{s}}(t)]_E = [T(t)]_E\,[H(t)]_E$. The Schwartz-Zippel lemma ensures that if the prover sends the correct evaluations, then with high probability the prover knows the witness.

In practice, the random point t is chosen from a common reference string (CRS) that is generated during a trusted setup phase. While trusted setups have been a point of concern, ongoing research and new constructions like zk-STARKs (45) aim to eliminate or minimize this requirement.

4

ML-DSA

The Module-Lattice-Based Digital Signature Algorithm (ML-DSA), originally published as 'CRYSTALS-Dilithium' (3), is a digital signing scheme standardized by NIST under FIPS204 as part of their most recent post-quantum cryptography suite (4). ML-DSA is existentially unforgeable against chosen-message attacks assuming the D-MLWE and MSIS problems are intractable. The standard describes the following functions: KeyGen, Sign, and Verify. Advantages of ML-DSA include its relatively small key sizes, fast signing and verification times, and resistance to quantum attacks. The scheme also exclusively uses uniform random sampling as opposed to Gaussian sampling, which is difficult to implement securely as evident from findings by multiple sources (46) (47) (48). A summary of these functions is given below. We follow the latest version of the scheme published in FIPS-204 (4).

4.1 Simplified Scheme

Let Alice be the signer and Bob be the verifier. The symmetric modulo operator is denoted by mod$^{\pm}$. The infinity norm is denoted by $\|\cdot\|_\infty$ and represents the maximum absolute value of the coefficients of a polynomial. Define $\mathbf{R}_q$ as the ring of polynomials in x, formally $\mathbf{R}_q = \mathbb{Z}_q[x]/(x^n+1)$. Define $\mathbf{S}_\eta$ as the set of polynomials in $\mathbf{R}_q$ with (mods q) coefficients in $[-\eta, \eta]$. Define $\tilde{\mathbf{S}}_{\gamma_1}$ as the set of polynomials in $\mathbf{R}_q$ with (mods q) coefficients in $(-\gamma_1, \gamma_1]$. Let $\tilde{\mathbf{B}}_\tau$ be the polynomials in $\mathbf{S}_1$, of which exactly τ coefficients are ± 1 and $\beta = \tau\eta$. Lastly, define size parameters k, l with $k \geq l$. For illustration, table 4.1 shows the parameters used in the ML-DSA standard.

Classifier	q	n	ζ	(k,ℓ)	η	d	γ_1	τ	β	γ_2	λ	ω
ML-DSA-44	$2^{23}-2^{13}+1$	256	1753	(4,4)	2	13	2^{17}	39	78	$(q-1)/88$	128	80
ML-DSA-65	$2^{23}-2^{13}+1$	256	1753	(6,5)	4	13	2^{19}	49	196	$(q-1)/32$	192	55
ML-DSA-87	$2^{23}-2^{13}+1$	256	1753	(8,7)	2	13	2^{19}	60	120	$(q-1)/32$	256	75

Table 4.1: Parameter sets for ML-DSA-44, ML-DSA-65, and ML-DSA-87 (4).

Before we get to the details of the scheme, we first introduce some functions that are used in the scheme.

```
Decompose_q(r, α)
1: $r \leftarrow r \bmod^{+} q$
2: $r_0 \leftarrow r \bmod^{\pm} \alpha$
3: if $r - r_0 = q - 1$ then
4:     $r_1 \leftarrow 0$; $r_0 \leftarrow r_0 - 1$
5: else
6:     $r_1 \leftarrow (r - r_0)/\alpha$
7: end if
8: return $(r_1, r_0)$
```

```
HighBits_q(r, α)
9: $(r_1, r_0) := \mathrm{Decompose}_q(r, \alpha)$
10: return $r_1$
```

```
LowBits_q(r, α)
11: $(r_1, r_0) := \mathrm{Decompose}_q(r, \alpha)$
12: return $r_0$
```

Figure 4.1: Functions used in ML-DSA, adapted from (3).

For every $\mathsf{Decompose}_q(r, \alpha) = (r_1, r_0)$, we have that $r = r_1\alpha + r_0$ with $r_0 \in (-\alpha/2, \alpha/2]$. When applying the $\mathsf{HighBits}_q(r, \alpha)$ and $\mathsf{LowBits}_q(r, \alpha)$ functions to a polynomial r, we apply them to each coefficient of the polynomial. See Figure 4.1 for the corresponding pseudo-code.

A simplified version of the scheme is explained below.

- KeyGen
  Alice generates a polynomial matrix $\mathbf{A} \in \mathbf{R}_q^{k \times l}$, and secrets $\mathbf{s}_1 \in \mathbf{S}_\eta^l$, $\mathbf{s}_2 \in \mathbf{S}_\eta^k$. Alice then computes $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$. Her verifiying public key is now $(\mathbf{A}, \mathbf{t})$, and her secret signing key is $(\mathbf{s}_1, \mathbf{s}_2)$. Note that recovering $\mathbf{s}_1$ from $(\mathbf{A}, \mathbf{t})$ is an instance of LWE.

- Sign
  Alice generates random $\mathbf{y} \in \tilde{\mathbf{S}}_{\gamma_1}$ and computes the commitment $\mathbf{w} = \mathbf{A}\mathbf{y}$. Note that the coefficients of $\mathbf{y}$ lie within a power-of-two range, enabling efficient uniform sampling using bitstrings. She then computes the 'high-order' bits of $\mathbf{w}$ and the 'low-order' bits of $\mathbf{w}$ such that $\mathbf{w}_1 = \mathsf{HighBits}_q(\mathbf{w}, \gamma_2)$ and $\mathbf{w}_0 = \mathsf{LowBits}_q(\mathbf{w}, \gamma_2)$. Using $\mathbf{w}_1$, she computes the challenge $c = H(M||\mathbf{w}_1)$. c is formed as a polynomial in $\tilde{\mathbf{B}_\tau}$ using the SampleInBall function. Finally, she computes the potential response $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$. This response, however, reveals information about the secret key $\mathbf{s}_1$.

  Coefficients in $\mathbf{y}$ are in a range of $(-\gamma_1, \gamma_1]$, while coefficients in $c\mathbf{s}_1$ are in a range of $[-\beta, \beta]$. Thus, the coefficients in $\mathbf{z}$ are in a range of $(-\gamma_1 - \beta, \gamma_1 + \beta]$. Suppose now that some coefficient in $\mathbf{z}$ is exactly equal to $\gamma_1 + \beta$. The corresponding coefficient in $c\mathbf{s}_1$ must then be equal to β, which leaks some information about $\mathbf{s}_1$. In general, if a coefficient in $\mathbf{z}$ is $\gamma_1 + a$ for some $-\beta + 1 \leq a \leq \beta$, then the corresponding coefficient in $c\mathbf{s}_1$ must be in $[a, \beta]$. Similarly, if a coefficient in $\mathbf{z}$ is $-\gamma_1 + a$ for some $-\beta < a \leq \beta$, then the corresponding coefficient in $c\mathbf{s}_1$ must be in $[-\beta, a-1]$. ML-DSA solves this security issue by rejection sampling. Alice computes c and $\mathbf{z}$ repeatedly until $||\mathbf{z}||_\infty < \gamma_1 - \beta$. She also verifies that the low-order bits of $\mathbf{w} - c\mathbf{s}_2$ are smaller than $\gamma_2 - \beta$.

  The signature is comprised of $\sigma = (c, \mathbf{z})$.

- Verify
  Bob verifies the signature $\sigma = (c, \mathbf{z})$ by obtaining an authentic copy of Alice's public key $(\mathbf{A}, \mathbf{t})$, (re)computing the high-order bits of commitment, and verifying that $c = H(M\|\mathbf{w}_1')$. Bob computes the high-order bits of $\mathbf{Az} - c\mathbf{t} = \mathbf{Ay} - c\mathbf{s}_2$. Because we know that $\|\mathsf{LowBits}_q(\mathbf{Ay} - c\mathbf{s}_2, 2\gamma_2)\|_\infty$ are smaller than $\gamma_2 - \beta$, and that the coefficients of $c\mathbf{s}_2$ are in $[-\beta, \beta]$, we can conclude that adding $c\mathbf{s}_2$ to $\mathbf{Ay}$ does not change the high-order bits of the polynomial. Thus, $\mathsf{HighBits}_q(\mathbf{Ay}, 2\gamma_2) = \mathsf{HighBits}_q(\mathbf{Ay} - c\mathbf{s}_2, 2\gamma_2)$ and Bob verifies the signature correctly.

This scheme works, but has several inefficiencies which we will discuss next. After all the optimizations have been covered, we summarize the final scheme.

4.2 Public Key and Signature Size Reduction

Firstly, the public key size can be reduced. We generate the matrix $\mathbf{A}$ from some seed ρ using SHAKE-128. For $M \in \{0,1\}^*$ and $d \geq 1$, $\mathsf{SHAKE128}(M, d)$ is the d-bit cryptographic hash of M. SHAKE-128 is an eXtendable Output Function (XOF), which means that the first d' bits of $\mathsf{SHAKE128}(M, d')$ are equal to the first d' bits of $\mathsf{SHAKE128}(M, d)$ for any $d' \leq d$.

The size of $\mathbf{t}$ can be reduced by dropping the low-order bits of the coefficients of $\mathbf{t}$ using the $\mathsf{Power2Round}_q(r, d)$ function. This is not to be confused with the $\mathsf{HighBits}_q(r, \alpha)$ and $\mathsf{LowBits}_q(r, \alpha)$ functions. The $\mathsf{Power2Round}_q(r, d)$ function can be extended to polynomials and vectors of polynomials by applying it each coefficient-wise. We calculate $\mathsf{Power2Round}_q(\mathbf{t}, d) = (\mathbf{t}_1, \mathbf{t}_0)$, such that $\mathbf{t} = \mathbf{t}_1 2^d + \mathbf{t}_0$. Instead of $\mathbf{t}$, we now include $\mathbf{t}_1$ in the public key. The concrete reduction in public key size can be seen in Table 4.2.

Parameter Set	Original t Size (bytes)	Compressed $\mathbf{t}_1$ Size (bytes)
ML-DSA-44	2944	1280
ML-DSA-65	4416	1920
ML-DSA-87	5888	2560

Table 4.2: Public key size reduction by compressing t to $\mathbf{t}_1$ for each ML-DSA parameter set.

Now in order to verify the signature, Bob computes $\mathbf{Az} - c\mathbf{t}_1 2^d = \mathbf{Az} - c(\mathbf{t} - \mathbf{t}_0) = \mathbf{Az} - c\mathbf{t} + c\mathbf{t}_0 = \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0$. We would like to correct for $c\mathbf{t}_0$ by adding $-c\mathbf{t}_0$ to the equation. With high probability, we know that $\|-c\mathbf{t}_0\|_\infty \leq \gamma_2$. This is because the coefficients of c are in $\tilde{\mathbf{B}}_\tau$ and the coefficients of $\mathbf{t}_0$ are in $[-2^{d-1}, 2^{d-1}]$. The coefficients of $-c\mathbf{t}_0$ are thus in $[-2^{d-1}\tau, 2^{d-1}\tau]$. The ML-DSA parameters are chosen such that with high probability, $\|-c\mathbf{t}_0\|_\infty \leq \gamma_2$. For ML-DSA-65 and ML-DSA-87, this condition always holds, whereas for ML-DSA-44 it holds with probability 0.596.

The signer verifies this is the case. Then adding $-c\mathbf{t}_0$ to $\mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0$ changes the 'high-order' bits of the polynomial by -1, 0, or +1. Alice includes 'hint bits' in the signature to enable Bob to correct for this change.

Take $r \in [0, q-1]$ and let $r_1 = \mathsf{HighBits}_q(r, 2\gamma_2)$ and $r_0 = \mathsf{LowBits}_q(r, 2\gamma_2)$. Notice that when adding a $z \in [-\gamma_2, \gamma_2]$ to r, the 'high-order' bits of the polynomial change by at most 1. This is made possible due to the condition in line 3 of the $\mathsf{Decompose}_q(r, \alpha)$ function, where we essentially 'merge' the last group of 'high-order' bits with the first group. This ensures that no group of 'high-order' bits contains less than than $2\gamma_2$ elements, which in turn allows us to produce a 1-bit hint to compute $\mathsf{HighBits}_q(r+z, 2\gamma_2)$ from r and h without knowledge of z. This procedure is defined in functions $\mathsf{MakeHint}_q(z, r, \alpha)$ and $\mathsf{UseHint}_q(h, r, \alpha)$.

```
UseHint_q(h, r, α)
13: $m := \frac{(q-1)}{\alpha}$
14: $(r_1, r_0) := \text{Decompose}_q(r, \alpha)$
15: if $h = 1$ and $r_0 > 0$ then
16:     return $(r_1 + 1) \bmod^+ m$
17: else if $h = 1$ and $r_0 \leq 0$ then
18:     return $(r_1 - 1) \bmod^+ m$
19: end if
20: return $r_1$
```

```
MakeHint_q(z, r, α)
21: $r_1 := \text{HighBits}_q(r, \alpha)$
22: $v_1 := \text{HighBits}_q(r + z, \alpha)$
23: return $[\![r_1 \neq v_1]\!]$
```

```
Power2Round_q(r, d)
24: $r := r \bmod^+ q$
25: $r_0 := r \bmod^\pm 2^d$
26: return $\left(\frac{r - r_0}{2^d}, r_0\right)$
```

Figure 4.2: Functions used to compute the hint bits, adapted from (3).

Now, it is easy to see that $\mathsf{UseHint}_q(\mathsf{MakeHint}_q(z, r, a), r, a) = \mathsf{HighBits}_q(r + z, a)$ These functions extend to polynomials by applying them to each coefficient of the polynomial. The FIPS-204 standard specifies that the signer verifies that $\|-c\mathbf{t}_0\|_\infty < \gamma_2$ (slightly stricter than the previously mentioned constraint). Alice calculates the hint bits for $\mathsf{MakeHint}_q(-c\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0, 2\gamma_2)$ and includes them in the signature. Alice also verifies that the number of 1s in the hint bit vector is less than or equal to ω.

4.3 Number Theoretic Transform

Finally, multiplications in the polynomial ring $\mathbf{R}_q$ can be done more efficiently using the Number-Theoretic Transform (NTT). The NTT is a version of Fast Fourier Transform (FFT) that operates in the finite field $\mathbb{Z}_q$. The NTT is a function $\mathsf{NTT} : \mathbf{R}_q \to \mathbb{Z}_q^n$ that transforms a polynomial into a vector of its coefficients in the NTT domain. This function is invertible, *s.t.* $\mathsf{NTT}^{-1} : \mathbb{Z}_q^n \to \mathbf{R}_q$. Classical multiplications of two polynomials in $\mathbf{R}_q$ take $O(n^2)$ time, while NTT-based multiplications take $O(n \log n)$ time. For two polynomials $a(x), b(x) \in \mathbf{R}_q$, the product $c(x) = a(x)b(x) \bmod (x^n + 1)$ is classically computed as follows:

1. **Compute the polynomial product**: Multiply each coefficient of $a(x) = \sum_{i=0}^{n-1} a_i x^i$ by every coefficient of $b(x) = \sum_{j=0}^{n-1} b_j x^j$, forming

$$d_k = \sum_{\substack{i,j \\ i+j=k}} a_i b_j \quad \text{for } 0 \leq k \leq 2n-2.$$

2. **Reduce modulo** $x^n + 1$: Use the relation $x^n = -1$ to rewrite terms with powers x^m for $m \geq n$ as
$$x^m = -x^{m-n},$$
folding higher-degree terms back into degree less than n.

3. **Add coefficients modulo** q: For each coefficient index, sum the folded coefficients and reduce each
$$c_i = d_i \bmod q,$$
to obtain the final polynomial coefficients c_i for $0 \leq i < n$.

The bottleneck of this method is the polynomial multiplication step, which takes $O(n^2)$ time. Using the NTT, we can compute the same polynomial product $c(x) = a(x)b(x)$ as follows:

1. **Transform to NTT domain**: Compute the $\hat{a} = \mathsf{NTT}(a)$ and $\hat{b} = \mathsf{NTT}(b)$.

2. **Pointwise multiplication**: Compute $\hat{c} = \hat{a} \circ \hat{b}$, which is an element-wise multiplication.

3. **Inverse NTT**: Compute $c = \mathsf{NTT}^{-1}(\hat{c})$.

In this case the bottleneck is the NTT transform (and its inverse), which takes $O(n \log n)$ time. The ML-DSA parameters are chosen such that the NTT can be computed efficiently. To enable the NTT, we select $n = 2^k$ and modulus q as a prime such that $q - 1$ divisible by $2n$. Let $\zeta \in \mathbb{Z}_q^*$ be an element of order $2n$, such that $\zeta^{2n} = 1$ and $\zeta^n = -1$. For a polynomial $a(x) \in \mathbf{R}_q$, the NTT is defined as *polynomial evaluation* of $a(x)$ at the odd powers of ζ, or
$$\mathsf{NTT}(a) = \hat{a} = \left(a(\zeta^1), a(\zeta^3), a(\zeta^5), \ldots, a(\zeta^{2n-1})\right).$$

Multiplication in the NTT domain is computed component-wise. Let $c(x) = a(x)b(x) \bmod (x^n+1)$ for $a(x), b(x) \in \mathbf{R}_q$, or equivalently $c(x) = a(x)b(x) + \ell(x)(x^n+1)$. For odd integers i we have $c(\zeta^i) = a(\zeta^i)b(\zeta^i) + \ell(\zeta^i)(\zeta^{ni} + 1)$. Thus, $\hat{c} = \hat{a} \circ \hat{b}$, where $\circ$ is component-wise multiplication. Addition is also done component-wise, $\hat{c} = \hat{a} + \hat{b}$.

4.3.1 NTT Transform

We'll be using the polynomial remainder theorem (49) for the NTT transform, which states that for a polynomial $a(x)$, the value of $a(\zeta^i)$ is equal to the remainder of $a(x)$ when divided by $x - \zeta^i$. From this theorem naturally follows the following property: if $f(x)$ divides $g(x)$, then $a(x) \bmod f(x) = (a(x) \bmod g(x)) \bmod f(x)$. This property is used to recursively compute the NTT of a polynomial $a(x)$ by dividing it into smaller polynomials:
$$\begin{aligned} x^n + 1 &= x^n + \zeta^n \\ &= (x^{n/2} + \zeta^{n/2})(x^{n/2} - \zeta^{n/2}) \\ &\vdots \end{aligned}$$

To compute the NTT of a polynomial $a(x)$, we need to reduce the polynomial modulo its smallest factors.

For example, to compute NTT($a(x)$), we first compute the results upon division of a by the degree 1 factors of x^8+1, which are $a_0 = a \bmod (x^4 - \zeta^4)$ and $a_1 = a \bmod (x^4 + \zeta^4)$. We then compute the results upon division of a_0 and a_1 by the degree 2 factors of x^8+1, which are $a_{00} = a_0 \bmod (x^2 - \zeta^2)$, similarly for a_{01}, a_{10}, a_{11}. This is repeated for the next level of recursion, computing the last eight remainders $a_{000}, a_{001}, \ldots, a_{111}$.

To compute $a_0(x)$ and $a_1(x)$ from $a(x)$ (and similarly for lower levels of recursion), we proceed as follows:

$$\begin{aligned} a(x) &= a_L(x) + x^{n/2} a_R(x), \quad \text{where } \deg(a_L), \deg(a_R) < n/2 \\ a_0(x) &= a(x) \bmod (x^n - \zeta^n) \\ &= a_L(x) + \zeta^{n/2} a_R(x) \\ a_1(x) &= a(x) \bmod (x^n + \zeta^n) \\ &= a_L(x) - \zeta^{n/2} a_R(x) \end{aligned}$$

For example, for $n = 8$:

$$\begin{aligned} a(x) &= a_L(x) + x^4 a_R(x) \\ a_0(x) &= a_L(x) + \zeta^4 a_R(x) \\ a_1(x) &= a_L(x) - \zeta^4 a_R(x) \end{aligned}$$

Dividing $a(x)$ into $a_0(x)$ and $a_1(x)$ takes n time. This operation is repeated for each level of recursion, which gives us a running time of $T(n) = 2T(n/2) + O(n)$. Thus, the NTT can be computed in $O(n \log n)$ time.

4.3.2 NTT Inversion

Inverting the NTT (*polynomial interpolation*) is done similarly, but in reverse order. For example, we use the remainder of $a(x) \bmod (x - \zeta)$ and $a(x) \bmod (x + \zeta)$ to compute the remainder of $a(x)$ when divided by $x^2 - \zeta^2$. This is repeated for each level of recursion, until we reach the original polynomial $a(x)$. To reconstruct the higher-level polynomial $a(x)$ (degree $< n$) from its lower-level remainders $a_0(x)$ and $a_1(x)$ (each of degree $< n/2$), use:

$$\begin{aligned} a_L(x) &= \frac{a_0(x) + a_1(x)}{2} \\ a_R(x) &= \frac{a_0(x) - a_1(x)}{2\zeta^4} \\ a(x) &= a_L(x) + x^{n/2} a_R(x) \end{aligned}$$

Similarly, this can be computed in $O(n \log n)$ time.

4.3.3 NTT Domain Representation

ML-DSA modifies the order in which the NTT is stored. Instead of sequentially storing the evaluation of the polynomial at the odd powers of ζ, it stores the evaluation in the order it is computed. For example, for $n = 8$, the NTT is stored as

$$\mathrm{NTT}(a) = (a(\zeta^1), a(\zeta^9), a(\zeta^5), a(\zeta^{13}), a(\zeta^3), a(\zeta^{11}), a(\zeta^7), a(\zeta^{15})).$$

Formally, let $\text{brv}(v)$ be the bit-reversal of an 8-bit integer v and let $\zeta_i = \zeta^{brv(128+i)}$. Then, for $n = 256$, the NTT is stored as

$$\text{NTT}(a) = (a(\zeta_0), a(-\zeta_0), \ldots, a(\zeta_{127}), a(-\zeta_{127})).$$

This structure allows for 'in-place' NTT computation and avoids the permutation step that is usually required after the NTT computation.

The NTT is naturally extended to polynomials in $\mathbf{R}_q$ by applying it to each coefficient of the polynomial. During the key generation phase, Alice generates a random matrix in NTT form from a seed $\hat{\mathbf{A}} = \mathsf{ExpandA}(\rho)$. Note that elements of $\hat{\mathbf{A}}$ are in $\mathbb{Z}_q^{256}$ ($n = 256$ for all parameter sets), which enables uniform random sampling using SHAKE-128. She then generates random $\mathbf{s}_1, \mathbf{s}_2$ and computes $\hat{\mathbf{s}_1} = \mathsf{NTT}(\mathbf{s}_1)$. She computes $\hat{\mathbf{b}} = \hat{\mathbf{A}}\hat{\mathbf{s}_1}$, after which she inverts the NTT to obtain $\mathbf{b} = \mathsf{NTT}^{-1}(\hat{\mathbf{b}})$. Finally, $\mathbf{t} = \mathbf{b} + \mathbf{s}_2$ is computed. Note that $\mathbf{A}$ is never explicitly computed, only its NTT form $\hat{\mathbf{A}}$ is used. During the signature generation, NTT is applied to the computation of $\mathbf{w} = \mathbf{Ay}$.

4.4 Complete Scheme

With all the optimizations in place, the complete ML-DSA scheme is given below. We refer to SHAKE-256 as H and omit the details of NTT domain representation in this summary. The expected number of iterations for signing is 1.5, 2.5, and 3.5 for ML-DSA-44, ML-DSA-65, and ML-DSA-87 respectively (4).

To show completeness, we show that if Alice generates a valid signature, then Bob can verify it. Bob computes

$$\begin{aligned}\mathbf{Az} - c\mathbf{t}_1 \cdot 2^d &= \mathbf{A}(\mathbf{y} + c\mathbf{s}_1) - c(\mathbf{t} - \mathbf{t}_0) \\ &= \mathbf{Ay} + c\mathbf{As}_1 - c(\mathbf{As}_1 + \mathbf{s}_2) + c\mathbf{t}_0 \\ &= \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0.\end{aligned}$$

Because we know that $\|-c\mathbf{t}_0\|_\infty \leq \gamma_2$, we can conclude that

$$\begin{aligned}\mathbf{w}_1' &= \mathsf{UseHint}_q(h, \mathbf{Az} - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2) \\ &= \mathsf{HighBits}_q(\mathbf{Az} - c\mathbf{t}, 2\gamma_2) \\ &= \mathsf{HighBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2).\end{aligned}$$

Since $\|\mathsf{LowBits}_q(\mathbf{Ay} - c\mathbf{s}_2, 2\gamma_2)\|_\infty < \gamma_2 - \beta$, we have that $\mathsf{HighBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2) = \mathsf{HighBits}_q(\mathbf{w}, 2\gamma_2) = \mathbf{w}_1$. Thus $\mathbf{w}_1' = \mathbf{w}_1$ and Bob verifies the signature correctly.

An adversary that can forge a signature must be able to find a polynomial $\mathbf{z}$ such that $\mathbf{Az} - c\mathbf{t}$ is close to the commitment $\mathbf{w}$. This is an instance of the MSIS problem, which is believed to be hard. The scheme is thus secure against existential forgery under chosen-message attacks, assuming the D-MLWE and MSIS problems are intractable.

Not discussed in this summary are the implementation details of the functions $\mathsf{ExpandA}$, $\mathsf{ExpandS}$, $\mathsf{ExpandMask}$, $\mathsf{SampleInBall}$, and the hint vector representation. These functions are covered in detail in the FIPS-204 standard (4).

Algorithm 3 Dilithium Key Generation, Signing, and Verification (4) (3)

```
1: function Gen
2:     $\xi \leftarrow \{0,1\}^{256}$
3:     $(\rho, \rho', K) \in (\{0,1\}^{256}, \{0,1\}^{512}, \{0,1\}^{256}) := H(\xi, 256+512+256)$
4:     $(s_1, s_2) \in S_\eta^k \times S_\eta^l \leftarrow \mathsf{ExpandS}(\rho')$
5:     $A \in R_q^{k\times l} \leftarrow \mathsf{ExpandA}(\rho)$                    ▷ $A$ is stored in NTT representation
6:     $t := As_1 + s_2$
7:     $(t_1, t_0) := \mathsf{Power2Round}(t, d)$
8:     $\mathrm{tr} \in \{0,1\}^{512} := \mathsf{H}(\rho \| t_1, 512)$
9:     return $(pk = (\rho, t_1), sk = (\rho, K, tr, s_1, s_2, t_0))$
10: end function
11: function Sign$(sk, M)$
12:     $A \in R_q^{k\times l} \leftarrow \mathsf{ExpandA}(\rho)$                    ▷ $A$ is stored in NTT representation
13:     $\mu \in \{0,1\}^{512} := H(\mathsf{tr} \| M, 512)$
14:     $\rho'' := H(\mathsf{tr} \| \mathsf{rnd} \| \mu, 512)$      ▷ rnd is either det. ($0^{256}$) or hedged ($\mathsf{rnd} \in_R \{0,1\}^{256}$)
15:     $\kappa := 0, (z, h) := \perp$
16:     while $(z, h) = \perp$ do
17:         $y \in S_\gamma^l := \mathsf{ExpandMask}(\rho'' \| \kappa)$
18:         $w := Ay$
19:         $w_1 := \mathsf{HighBits}_\gamma(w, 2\gamma_2)$
20:         $\tilde{c} := H(\mu \| w_1)$
21:         $c \in B_\tau := \mathsf{SampleInBall}(\tilde{c})$
22:         $z := y + cs_1$
23:         $(r_1, r_0) := \mathsf{Decompose}_\gamma(w - cs_2, 2\gamma_2)$
24:         if $\|z\|_\infty \geq \gamma_1 - \beta$ or $\|r_0\|_\infty \geq \gamma_2 - \beta$ or $r_1 \neq w_1$ then
25:             $(z, h) := \perp$
26:         else
27:             $h := \mathsf{MakeHint}_\gamma(-ct_0, w - cs_2 + ct_0, 2\gamma_2)$
28:             if $\|ct_0\|_\infty \geq \gamma_2$ or #1's in $h > \omega$ then
29:                 $(z, h) := \perp$
30:             end if
31:         end if
32:         $\kappa := \kappa + 1$
33:     end while
34:     return $\sigma = (z, h, \tilde{c})$
35: end function
36: function Verify$(pk, M, \sigma = (z, h, c))$
37:     $A \in R_q^{k\times l} \leftarrow \mathsf{ExpandA}(\rho)$                    ▷ $A$ is stored in NTT representation
38:     $\mu \in \{0,1\}^{512} := H(H(\rho \| t_1, 512) \| M, 512)$
39:     $w_1' := \mathsf{UseHint}(h, Az - ct_1 \cdot 2^d, 2\gamma_2)$
40:     return $(\|z\|_\infty < \gamma_1 - \beta) \wedge [c = H(\mu \| w_1', 2\lambda)] \wedge [\#1's \text{ in } h \leq \omega]$
41: end function
```

5

TDUE

Chen, Jiang, and Liang (25) introduce TDUE, a lattice trapdoor-based c-d UE algorithm. They also introduce a packing technique that allows for simultaneous encryption and updating of multiple messages within a single ciphertext. This helps further reduce the cost of c-d UE.

Aside from this novel algorithm, the paper also defines a new confidentiality notion for c-d UE and proves TDUE to be *Chosen Ciphertext Attack-1* (CCA-1) secure with adaptive security. Further details on CCA-1 security for updatable encryption are provided later in this section.

Importantly, under the assumption that the LWE problem is intractable, the scheme is quantum-resistant as well. TDUE is based on a new public key encryption scheme TDP, which is a lattice-based trapdoor public key encryption scheme.

5.1 TDP

We use the same notation as defined in Section 3.1.1.

The TDP scheme assumes the existence of functions to invert the LWE problem and sample a preimage to an SIS problem using trapdoors. Specifically, it assumes that given a $\mathbf{G}$-trapdoor $\mathbf{R}$ for $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$ there is a function $\mathsf{Invert}(\mathbf{R}, \mathbf{A}, \mathbf{H}, \mathbf{b})$ that recovers $\mathbf{s}$ and $\mathbf{e}$ from the LWE instance $\mathbf{b}^t = \mathbf{s}^t\mathbf{A}^t + \mathbf{e}^t$, if $\left\| \left[\mathbf{R}^t\, \mathbf{I}\right] \right\|_\infty \leq \frac{q}{4}$.

It also assumes that given a $\mathbf{G}$-trapdoor $\mathbf{R}$ for $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$ with invertible matrix $\mathbf{H}$ and any $\mathbf{u} \in \mathbb{Z}_q^n$, there is a function $\mathsf{SampleD}(\mathbf{R}, \mathbf{A}, \mathbf{H}, \mathbf{u}, s)$ that samples a Gaussian vector $\mathbf{x}$ from a distribution $D_{\mathbb{Z}^m,s}$ such that $\mathbf{Ax} = \mathbf{u}$.

Refer to Section 3.2 for a more detailed description of lattice trapdoors.

The TDP scheme uses the following parameters:

- $q = \mathrm{poly}(\lambda)$, (modulus)
- n, $k = \lceil \log_2 q \rceil = O(\log n)$, $\bar{m} = O(nk)$, (matrix dimensions)
- $m = \bar{m} + 2nk$

Let $\mathcal{D} = D_{\mathbb{Z}^{\bar{m}\times nk},\omega(\sqrt{\log n})}$ be a discrete Gaussian distribution. Let encode : $\{0,1\}^{nk} \to \Lambda(\mathbf{G}^t)$ be a function that encodes a message $\mathbf{m}$ as $\mathbf{Bm} \in \mathbb{Z}^{nk}$, where $\mathbf{B}$ is any basis of $\Lambda(\mathbf{G}^t)$. Lastly, define the LWE error rate α as $1/\alpha = 4 \cdot O(nk) \cdot \omega(\sqrt{\log n})$.

Algorithm 4 Functions in TDP (25)

```
1:  function TDP.KG($1^\lambda$)
2:      $\mathbf{A}_0 \leftarrow \mathcal{U}(\mathbb{Z}_q^{n\times\bar{m}})$
3:      $\mathbf{R}_1, \mathbf{R}_2 \leftarrow \mathcal{D}$
4:      $\mathbf{A} = \begin{bmatrix}\mathbf{A}_0 & -\mathbf{A}_0\mathbf{R}_1 & -\mathbf{A}_0\mathbf{R}_2\end{bmatrix} = \begin{bmatrix}\mathbf{A}_0 & \mathbf{A}_1 & \mathbf{A}_2\end{bmatrix} \in \mathbb{Z}_q^{n\times m}$
5:      return $(\mathrm{pk} = \mathbf{A}, \mathrm{sk} = \mathbf{R}_1)$
6:  end function

7:  function TDP.Enc($\mathrm{pk} = \mathbf{A}, \mathbf{m} \in \{0,1\}^{nk}$)
8:      Choose invertible tag matrix $\mathbf{H}_\mu \in \mathbb{Z}_q^{n\times n}$
9:      $\mathbf{A}_\mu = \begin{bmatrix}\mathbf{A}_0 & \mathbf{A}_1 + \mathbf{H}_\mu\mathbf{G} & \mathbf{A}_2\end{bmatrix}$
10:     $d^2 = (\|\mathbf{e}_0\|^2 + \bar{m}\cdot(\alpha q)^2)\cdot w\log n^2$
11:     $\mathbf{s} \leftarrow \mathcal{U}(\mathbb{Z}_q^n)$
12:     $\mathbf{e} = (\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2) \in D_{\mathbb{Z}^{\bar{m}},\alpha q} \times D_{\mathbb{Z}^{nk},d} \times D_{\mathbb{Z}^{nk},d}$
13:     $\mathbf{b}^T = \mathbf{s}^T\mathbf{A}_\mu^T + \mathbf{e}^T + (\mathbf{0}_{\bar{m}}, \mathbf{0}_{nk}, \mathrm{encode}(\mathbf{m}))^T \pmod q$
14:     return $c = (\mathbf{H}_\mu, \mathbf{b})$
15: end function

16: function TDP.Dec($\mathrm{sk} = \mathbf{R}_1, c = (\mathbf{H}_\mu, \mathbf{b})$)
17:     if $c$ or $\mathbf{b}$ is malformed, or $\mathbf{H}_\mu = \mathbf{0}$ then
18:         return $\perp$
19:     end if
20:     $\mathbf{A}_\mu = \begin{bmatrix}\mathbf{A}_0 & \mathbf{A}_1 + \mathbf{H}_\mu\mathbf{G} & \mathbf{A}_2\end{bmatrix}$
21:     Parse $\mathbf{b}^T = (\mathbf{b}_0^T, \mathbf{b}_1^T, \mathbf{b}_2^T)$
22:     $(\mathbf{s}, (\mathbf{e}_0, \mathbf{e}_1)) \leftarrow \mathrm{Invert}(\mathbf{R}_1, \begin{bmatrix}\mathbf{A}_0 & \mathbf{A}_1 + \mathbf{H}_\mu\mathbf{G}\end{bmatrix}, [\mathbf{b}_0, \mathbf{b}_1], \mathbf{H}_\mu)$
23:     if the call fails then
24:         return $\perp$
25:     end if
26:     $(\mathbf{b}_2, \mathbf{e}_2) \leftarrow g_{\mathbf{G}}^{-1}(\mathbf{b}_2^T - \mathbf{s}^T\mathbf{A}_2)$        ▷ $s.t.$ $\mathbf{b}_2^t - \mathbf{s}^t\mathbf{A}_2 = \mathbf{u}^t\mathbf{G} + \mathbf{e}_2^t$
27:     if $\|\mathbf{e}_0\| \geq \alpha q\sqrt{\bar{m}}$ or $\|\mathbf{e}_j\| \geq \alpha q\sqrt{2\bar{m}nk}\cdot w\log n$ for $j \in \{1,2\}$ then
28:         return $\perp$        ▷ Error bounds exceeded
29:     end if
30:     $\mathbf{m} = \mathrm{encode}^{-1}(\mathbf{b}_2^T - \mathbf{s}^T\mathbf{A}_2 - \mathbf{e}_2^T) \in \{0,1\}^{nk}$
31:     if decoding fails then
32:         return $\perp$
33:     end if
34:     return $\mathbf{m}$
35: end function
```

Functions in TDP are described in Algorithm 4.

TDP is correct and CCA-1 secure under the assumption that the LWE problem is intractable (25). The runtime of KG is bounded by the matrix multiplications, which gives us $O(n \cdot m \cdot nk)$. Assuming $\mathbf{H}_\mu\mathbf{G}$ is precomputed, Enc is bounded by the vector-matrix multiplication, which gives $O(n \cdot m)$. Dec is bounded by Invert, which takes $O(nk)$ work (26).

5.2 Full Scheme

TDUE builds on top of TDP by adding two new functions: TokenGen and Update. The same parameters as in TDP are used, with the exception of:

- $1/\alpha = 4l \cdot \omega(\sqrt{\log n})^{2l+2} O(\sqrt{nk})^{3l+3}$ where l is the number of updates the scheme supports.
- $\tau = \sqrt{s_1(\mathbf{R})^2 + 1} \cdot \sqrt{s_1(\Sigma_G) + 1} \cdot \omega(\sqrt{\log n})$ where Σ_G is either 4 if q is a power of 2, or 5 otherwise. The value τ is the smallest Gaussian parameter for the discrete Gaussian distribution from which the function SampleD can sample a vector.

The TDUE scheme is described in Algorithm 5.

Algorithm 5 Functions in TDUE (25)

```
36: function TDUE.TokenGen(pk, sk, pk', $\mathbf{H}_\mu$)
37:     Parse pk = $\begin{bmatrix}\mathbf{A}_0 & \mathbf{A}_1 & \mathbf{A}_2\end{bmatrix}$ and $sk = \mathbf{R}_1$
38:     Parse $pk' = \begin{bmatrix}\mathbf{A}'_0 & \mathbf{A}'_1 & \mathbf{A}'_2\end{bmatrix}$
39:     Choose invertible matrix $\mathbf{H}'_\mu$
40:     $\mathbf{A}'_\mu = \begin{bmatrix}\mathbf{A}'_0 & \mathbf{A}'_1 + \mathbf{H}'_\mu\mathbf{G} & \mathbf{A}'_2\end{bmatrix}$
41:     $(\mathbf{X}_{00}, \mathbf{X}_{10}) \leftarrow \mathsf{SampleD}(\mathbf{R}_1, [\mathbf{A}_0 \mid -\mathbf{A}_0\mathbf{R}_1 + \mathbf{H}_\mu\mathbf{G}], \mathbf{H}_\mu, \mathbf{A}'_0, \tau)$
42:     $(\mathbf{X}_{01}, \mathbf{X}_{11}) \leftarrow \mathsf{SampleD}(\mathbf{R}_1, [\mathbf{A}_0 \mid -\mathbf{A}_0\mathbf{R}_1 + \mathbf{H}_\mu\mathbf{G}], \mathbf{H}_\mu, \mathbf{A}'_1, \tau\sqrt{\bar{m}/2})$
43:     $(\mathbf{X}_{02}, \mathbf{X}_{12}) \leftarrow \mathsf{SampleD}(\mathbf{R}_1, [\mathbf{A}_0 \mid -\mathbf{A}_0\mathbf{R}_1 + \mathbf{H}_\mu\mathbf{G}], \mathbf{H}_\mu, \mathbf{A}'_2 - \mathbf{A}_2, \tau\sqrt{\bar{m}/2})$
44:     $\mathbf{M} \leftarrow \begin{bmatrix}\mathbf{X}_{00} & \mathbf{X}_{01} & \mathbf{X}_{02} \\ \mathbf{X}_{10} & \mathbf{X}_{11} & \mathbf{X}_{12} \\ \mathbf{0} & \mathbf{0} & \mathbf{I}\end{bmatrix}$
45:     Sample $\mathbf{s}', \mathbf{e}'$ and compute $\mathbf{b}_0^T = (\mathbf{s}')^T\mathbf{A}'_\mu + (\mathbf{e}')^T \bmod q$
46:     return $\Delta = (\mathbf{M}, \mathbf{b}_0, \mathbf{H}'_\mu)$
47: end function
48: function TDUE.Update($\Delta = (\mathbf{M}, \mathbf{b}_0, \mathbf{H}'_\mu), c = (\mathbf{H}_\mu, \mathbf{b})$)
49:     $(\mathbf{b}')^T \leftarrow \mathbf{b}^T \cdot \mathbf{M} + \mathbf{b}_0^T \bmod q$
50:     return $c' = (\mathbf{H}'_\mu, \mathbf{b}')$
51: end function
```

TDUE is correct with overwhelming probability. Showing that $(\mathbf{0}_{\bar{m}}, \mathbf{0}_{nk}, \text{encode}(\mathbf{m}))^t$ stays the same after the update is straightforward:

$$\begin{aligned}(\mathbf{b}')^t &= \mathbf{b}^t \cdot \mathbf{M} + \mathbf{b}_0^t \\ &= \left[\mathbf{s}^t \mathbf{A}_\mu + \mathbf{e}^t + (0, 0, \text{encode}(\mathbf{m}))^t\right] \mathbf{M} + (\mathbf{s}')^t \mathbf{A}'_\mu + (\mathbf{e}')^t \\ &= (\mathbf{s} + \mathbf{s}')^t \mathbf{A}'_\mu + (\mathbf{e}^t \mathbf{M} + (\mathbf{e}')^t) + (0, 0, \text{encode}(\mathbf{m}))^t \mod q.\end{aligned}$$

Note, however, that the error term grows significantly from $\mathbf{e}^t$ to $\mathbf{e}^t\mathbf{M} + (\mathbf{e}')^t$ after the update. Proving the error bound is rather involved. The full proof of correctness is given in (25). In conclusion, TDUE decrypts correctly except with $2^{-\Omega(n)}$ failure probability.

Under the new confidentiality notion for c-d UE, TDUE is IND-UE-CCA-1 secure:

Lemma 3 ((25), Theorem 3.1) *For any IND-UE-CCA-1 adversary $\mathcal{A}$ against TDUE, there exists a polynomial-time adversary $\mathcal{B}$ against LWE$_{n,q,\alpha}$ such that*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{IND-UE-CCA-1}}_{\mathsf{TDUE},\mathcal{A}}(1^\lambda) \leq 2(l+1)^3 \cdot \Big[&(l+2)\cdot \mathsf{negl}(\lambda) \\ &+ (n_{\mathsf{Dec}} + n_{\mathsf{Upd}}) \cdot 2^{-\Omega(n)} + \mathsf{Adv}^{LWE}_{n,q,\alpha}(\mathcal{B})\Big].\end{aligned}$$

where l is the maximum number of ciphertext updates the scheme supports, and n_{Dec} and n_{sUpd} are the number of queries to the oracles $\mathcal{O}_{\mathsf{Dec}}$ and $\mathcal{O}_{\mathsf{sUpd}}$ respectively.

The runtime of TokenGen is bounded by the SampleD function, which is quadratic in the n dimension for a single vector (26). This gives $O(\bar{m}n^2)$ for an $\mathbf{X}$ matrix. The Update function is bounded by a vector-matrix multiplication in $\mathbb{Z}_q$, which gives $O(m^2)$.

Plaintext messages are nk bits long. Ciphertexts are m elements in $\mathbb{Z}_q$, where $m = O(nk)$, resulting in $O(nk)$ elements in $\mathbb{Z}_q$. Update tokens are $m(\bar{m} + nk) + m$ elements in $\mathbb{Z}_q$, or $O(n^2k^2)$ elements in $\mathbb{Z}_q$. Note that the size of the update token is quadratic in the size of the ciphertext. Consequently, this implies that using TDUE requires at least a number of ciphertexts quadratic in the size of a ciphertext (or half of that) to be encrypted with the same key before using update tokens becomes more network-efficient than manually downloading the ciphertexts, updating them locally, and reuploading them.

5.3 Other lattice-based UE schemes

Aside from TDUE, other lattice-based UE schemes have been proposed with slightly different parameters. We'll briefly cover a few schemes below.

- *The Direction of Updatable Encryption Does Not Matter Much* by Jiang (50). Similar to TDUE, the proposed scheme is based on a PKE rather than a *Dec-then-Enc* structure. The PKE scheme is IND-CPA secure under the assumption that the D-LWE$_{q,\alpha}$ problem is intractable. The full scheme is randIND-UE-CPA secure.

  Let plaintext messages be t bits long, let q be the modulus, and let $\mathbf{A} \in \mathbb{Z}_q^{m\times n}$ be a uniformly randomly selected matrix (for the LWE problem). Ciphertexts are $nt + t$ elements in $\mathbb{Z}_q$, and update tokens are $n + m$ elements in $\mathbb{Z}_q$. This is a significant improvement over TDUE, as the size of the update token is linear in the ciphertext size.

The main difference with TDUE is the 'packing' functionality of TDUE, which allows for encrypting multiple messages in a single ciphertext. TDUE is also proven to be secure under a stronger security definition, namely IND-UE-CCA-1, while this scheme is only proven to be secure under the weaker IND-UE-CPA security definition (with old notions of directional key updates, as we'll cover next).

- *The Direction of Updatable Encryption Does Matter* by Nishimaki (51). This paper makes new distinctions in uni-directional key updates for updatable encryption, namely *backward-leak uni-directional* key updates, and introduces a new scheme based on the LWE problem. This lattice-based c-i UE scheme is also based on a PKE scheme, which is in turn inspired by Regev's LWE-based PKE scheme (9). This scheme is also randIND-UE-CPA secure under the newly established backward-leak uni-directional key updates.

  Let plaintext messages be l bits long, let q be the modulus, and let $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$ be a random parity-check matrix in the LWE problem (for the Regev cryptosystem variant). Ciphertexts are $n + l$ elements in $\mathbb{Z}_q$. Update tokens are $(n\eta \times n) + (m \times n)$ elements in $\mathbb{Z}_q$, where η is $\lceil \log_2 q \rceil$. Although the size of the update token is quadratic in the ciphertext size, the hidden constant is still smaller than in TDUE.

  Similar to the previous paper, the main difference with TDUE is the 'packing' functionality.

- *Improving Speed and Security in Updatable Encryption Schemes* by Boneh *et al.* (6). This paper introduces a new security notion of *compactness* for c-d UE and two new updatable encryption schemes, the first of which relies solely on symmetric cryptographic primitives, while the second is based on the Ring-LWE problem. The authors claim a 200x speedup over previous elliptic curve-based schemes for the Ring-LWE based scheme.

  The first scheme has extremely high encryption throughput approaching the performance of AES, and relies entirely on symmetric cryptographic primitives. Decryption throughput is higher for re-encryptions (updates) fewer than 50, after which the second scheme takes the lead.

  Similarly to TDUE, the second scheme is also considered post-quantum because its hardness relies on LWE. A notable asset over TDUE is that the second scheme supports a (nearly) unlimited number of updates. Compared to TDUE, the scheme is also more practical for larger plaintext messages, because of the relatively small ciphertext expansion of 19% at the lowest (the expansion is dependent on the modulus used). However, the second scheme is only proven to be secure under slightly loosened security definitions, namely a weaker version of the re-encryption (update) oracle in the security games.

  TDUE differs from these schemes in that it is based on a public-key encryption scheme, and that it supports a packing functionality that allows for encrypting multiple messages in a single ciphertext.

6 Vesper

This thesis builds on the work of Franschman (42), who introduces Vesper, a non-interactive zero-knowledge proof system (specifically, a zk-SNARK). The construction is inspired by Groth16 (43), but improves upon it by eliminating the need for a trusted setup phase. Additionally, Vesper replaces elliptic curve commitments with the Regev encryption scheme (9), making it suitable for post-quantum cryptography.

Proof generation. At its core, a Vesper prover seeks to demonstrate knowledge of a secret $\mathbf{s}$ such that $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}$ for some public $\mathbf{t}$ and $\mathbf{A}$, which is an instance of the LWE problem. The prover processes the corresponding arithmetic circuit, converting it into an R1CS and then a QAP. Verification of the QAP is performed using the equation $L_{\mathbf{s}}(x) \cdot R_{\mathbf{s}}(x) - O_{\mathbf{s}}(x) = H(x) \cdot T(x)$ (see Equation 3.14).

Both sides of this equation are encrypted using Regev's encryption scheme (9), which is IND-CPA secure and additively homomorphic. Let $[\cdot]_R$ denote the Regev encryption function. The prover computes:

$$\mathsf{Left} = [L_{\mathbf{s}}(x) \cdot R_{\mathbf{s}}(x) - O_{\mathbf{s}}(x)]_R \tag{6.1}$$

$$\mathsf{Right} = [H(x) \cdot T(x)]_R \,. \tag{6.2}$$

After encrypting both sides, the prover signs the encrypted QAP with ML-DSA, ensuring integrity and authenticity of the proof.

Proof verification. The verifier checks that the encrypted left-hand side equals the encrypted right-hand side, i.e., $\mathsf{Left} = \mathsf{Right}$. The signature is also verified to confirm the proof's origin and integrity.

Performance. The proof size is determined by the size of Left and Right, and by the expansion factor of the Regev encryption scheme. Both sides are single scalar values, and the encryption expands each to a ciphertext of size n (the dimension of the LWE instance for Regev). Franschman reports a constant proof size of 512 bytes, and claims a reduction to 128 bytes using hexadecimal representation—likely by hashing both sides of the equation to 64 bytes each. Further research is needed to assess the security of this optimization.

6.1 Alternative zk-SNARKS

Other post-quantum zk-SNARKs exist, though they are generally less efficient or practical than Groth16. Alternatives are based on cryptographic hash functions (52) (53) or lattice-based cryptography (54) (55). Two notable lattice-based zk-SNARKs are:

- (54) presents a lattice-based zk-SNARK with a proof size of around 16.4 KB (for an NP relation of size 2^{20}) and a verification time of about 1.5 seconds. Compared to Groth16, the proof size is 131 times larger, but proof generation and verification are 1.2 and 2.8 times faster, respectively. At an estimated 128-bit security level, setup takes 2240 seconds, proof generation 68 seconds, and the CRS size is 5.3 GB.
- (55) introduces a zk-SNARK construction based on LWE. For a security parameter $\lambda = 162$, the proof size is 0.64 MB, with average proof generation time of 53.6 seconds and verification time of 2.28 milliseconds. The setup phase takes 167 seconds, generating a CRS of 8.36 MB. Performance was evaluated on a single-threaded Intel Core i7-4770K CPU at 3.50 GHz, running Debian (kernel version 4.9.110).

While these alternatives show promising performance characteristics, they remain less practical than pre-quantum zk-SNARKs or the Vesper protocol combined with ML-DSA. Moreover, Vesper has a complete implementation ready for integration into Hyperledger Fabric.

7

Blockchain

In this section we briefly review blockchains, Hyperledger Fabric, and Fabric Private Chaincode. These technologies are used to develop the secure blockchain modules.

7.1 Blockchains

Blockchain technology enables secure transmission and storage through a decentralized and distributed database system. The system aims to provide *transparency*, *decentralization* and *immutability*, by employing various cryptographic protocols. However, the specific protocols may vary across different blockchain implementations and the aforementioned qualities may be present in varying degrees. The database ledger keeps track of a complete history of all exchanges, or *transactions*, between users on the network without central authority. This history is divided into *blocks*, collections of transaction records, which are linked via cryptographic hashes. Consensus mechanisms, which vary across blockchains, validate and agree upon the contents of each subsequent block that is recorded. Choice of consensus mechanism influences the performance and security of the system.

Blockchains are generally divided into two categories: *permissionless* (public) and *permissioned* blockchains. Notable examples of permissionless blockchains include Bitcoin (56) and Ethereum (57), where consensus is reached through protocols such as *Proof-of-Work* (PoW) and *Proof-of-Stake* (PoS). Anyone can join a permissionless blockchain network, which may be unsuitable for enterprise environments. Unlike permissionless blockchains, permissioned blockchains require every user in the network to be authenticated, restricting access to identifiable participants. Enterprise blockchain applications also benefit from higher transaction throughput and lower latency of transaction confirmation. Such performance benefits are more easily achieved in permissioned blockchains, because the requirements for consensus are less stringent than in permissionless blockchains.

A permissioned blockchain network run by a pre-selected group of organizations is known as a *consortium blockchain*, whereas a permissioned blockchain run by a single organization is referred to as a *private blockchain.*

7.2 Hyperledger Fabric

Hyperledger Fabric (HLF), also referred to as just *Fabric*, is a scalable, open-source, permissioned blockchain platform. HLF allows multiple organizations (and their clients) to participate in a distributed ledger network of nodes. Each node, also referred to as *peer* in the context of HLF, keeps a copy of the ledger containing all interactions, or *transactions*, between parties. A transaction invokes an application called *smart contract*, or *chaincode*, to interact with the ledger.

Chaincode is a program that defines the rules for how transactions are processed and validated. In HLF, chaincode is written in Go, Java, or JavaScript, and functions as a distributed application that runs on the blockchain network. HLF follows an *execute-order-validate* architecture, where transactions are first executed by endorsing peers, then ordered by a consensus mechanism, and finally validated by all peers in the network. This architecture allows for greater flexibility and scalability compared to traditional blockchain systems (*order-execute*), as it separates the execution of transactions from their ordering and validation.

HLF supports the concept of *channels*, which are private sub-networks within the larger blockchain network. Channels allow a subset of participants to have their own private ledger, enabling confidential transactions between specific parties. Alternative approaches to achieving privacy in HLF include *private data collections*, which allow for sharing of sensitive data among a subset of participants without exposing it to the entire network. Traditionally, confidential transactions may be achieved through encryption or ZKPs. The counterargument is that encrypted data is still visible to all peers in the network (which might be a problem in case of sensitive data), and ZKPs are computationally expensive. A simple overview of the HLF architecture is shown in Figure 7.1.

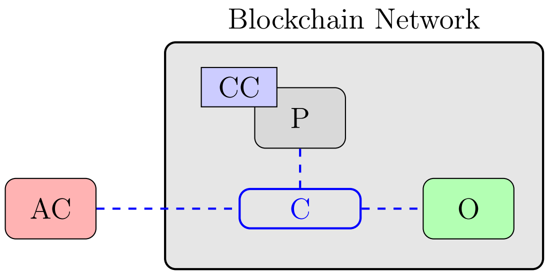

Figure 7.1: Simplified Hyperledger Fabric architecture overview (AC = client, CC = chaincode, C = channel, O = orderer). The client interacts with all peers, which execute chaincode and communicate with the orderer. In practice, the architecture supports multiple clients, channels, peers and orderers.

Being a permissioned blockchain, HLF requires some form of identity management. Identities are distributed by a Certificate Authority (CA), which is a trusted authority that issues digital certificates to participants. These certificates are based on the X.509 standard (58), which is widely used in public key infrastructure (PKI) systems. An X.509 certificate contains a public key, the identity of the participant, and a digital signature from the issuing authority. By default, the digital signatures in HLF are generated using the ECDSA algorithm, which is vulnerable to quantum attacks. After receiving an identity certificate, a participant can be added to the network by a Membership Service Provider (MSP).

In conclusion, in order to strengthen the security of HLF against quantum attacks, it is necessary to replace the ECDSA signatures with post-quantum cryptographic primitives. This can be achieved by revisiting the implementation of the CA, and any peers that require certificate verification.

7.3 Fabric Private Chaincode

HLF, through its inherently transparent nature, does not grant privacy of transactions. All transactions are executed over all peers in the network, and as a consequence the application state is not kept private. Hyperledger Fabric Private Chaincode (FPC) mitigates this problem by executing code in a Trusted Execution Environment (TEE) using Intel Software Guard Extensions (SGX). The TEE is also referred to as a *secure enclave.* By leveraging Intel SGX, the privacy of chaincode data and computation is concealed from executing peers. Any chaincode can now be executed in a secure enclave and any data can now be optionally encrypted (through SGX) before storage on the ledger. This concept was originally proposed by Brandenburger *et al.* (59). The paper covers the challenges associated with implementing TEEs for HLF and proposes the architecture currently known as FPC. Figure 7.2 shows a simplified architecture of FPC for a single peer.

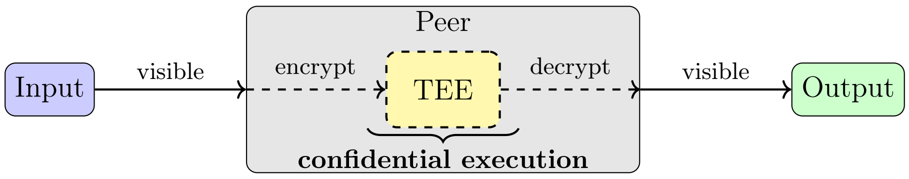

Figure 7.2: Simplified architecture of Fabric Private Chaincode (FPC) execution for a single peer. The chaincode is executed in a TEE, which ensures that the application state is kept private from the executing peer.

Note that a caveat of using FPC is that it requires all peers in the network to have SGX-compatible hardware. This may not be desirable or feasible in all scenarios. Another point to consider is that FPC does not match the performance of regular chaincode, as the overhead can be significant. For *submit* transactions (which entail encryption and storage of data on the ledger), the transaction throughput for FPC can range from 0.55x to 0.95x that of the native execution (59). The trade-off between performance and confidentiality of the application state is a key consideration when using FPC.

Fully Homomorphic Encryption. As an alternative to FPC, fully homomorphic encryption (FHE) can be used to achieve similar results, as it allows for computations on encrypted data without revealing the plaintext. The main disadvantage of FHE is that it is computationally expensive, and the performance overhead can be significant.

8

Design

We propose a quantum-resistant blockchain solution for secure data communication and storage by extending HLF with three modules: digital signing, zero-knowledge proofs, and updatable encryption. This section details the high-level design of these three blockchain modules for HLF and the structures in place to support them. These modules will provide a fundamental framework for post-quantum secure communication and storage on HLF. Implementation details are covered in Chapter 9.

We'll first provide a general outline of an interaction flow, followed by a description of the client application, the middleware server, and finally the chaincode.

8.1 Interaction Flow

Interactions with the HLF network are initiated by a user client, which communicates with a middleware server. The middleware server processes the user's request and relays it to a peer node in the HLF network. The peer node consequently executes the requested chaincode function. To ensure the security and privacy of these operations, the design makes use of FPC, which allows for private execution of chaincode. The application state is kept private from the executing peers, ensuring that sensitive data remains confidential. A simplified flow diagram of an example user request is shown in Figure 8.1.

8.2 Client

The client is responsible for all off-chain functionalities. The client must be able to perform the following operations:

- Generate key pairs for digital signing and updatable encryption.
- Retrieve ciphertexts from the middleware server.
- Generate update tokens for updatable encryption.
- Decrypt ciphertexts with the private key.
- Sign messages with the private key.
- Encrypt plaintexts into ciphertexts.

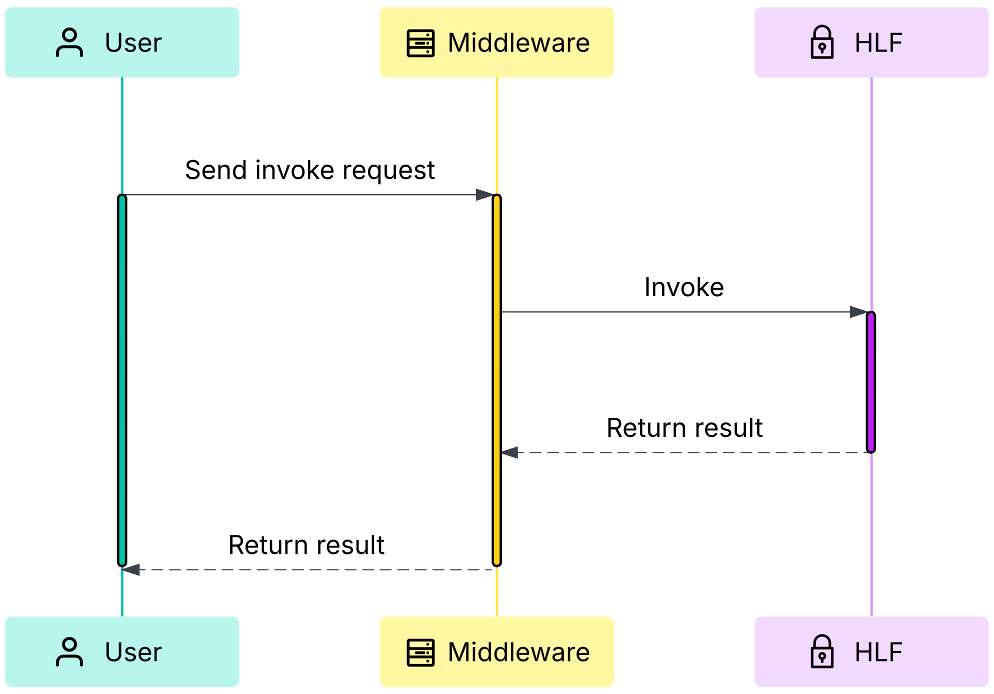

Figure 8.1: Simplified flow diagram of a user request

- Generate zk-SNARK proofs.
- Send requests to the middleware server with the relevant data (e.g. signatures, ciphertexts, proofs).

The client can be implemented in any programming language, as long as it can perform the above operations and send HTTP requests to the middleware server. As this thesis is mostly focused on the design and implementation of the chaincode, we do only provide a simple implementation of the client. UI/UX considerations are not covered in this thesis, but future work may consider implementing a user-friendly client with a graphical interface.

8.3 Middleware

The middleware server acts as an intermediary between the user client and the HLF network. The middleware server is solely responsible for parsing the user's request and forwarding it to the HLF network. This means that the middleware enforces a format for the user's request. Any malformed request will result in a failure message being returned to the user. Requests will be POST requests, with the request body containing a JSON object.

All requests, including the respective parameters, are sent to the middleware in plain. The middleware server must be trusted with any plaintext data sent by the user. Extra considerations must be made to ensure that the middleware server is secure. Ideally, the middleware server is self-hosted or hosted by a trusted party. The server should also be secured with HTTPS to prevent eavesdropping of the user's request.

8.4 Chaincode

All chaincode interactions are executed using FPC. The chaincode for all three modules is present in the same FPC chaincode package, which is deployed on the HLF network. The chaincode is responsible for handling the following functionalities:

- **Digital signing**: verifying signatures and their respective messages.
- **Zero-knowledge proof verification**: verifying zk-SNARK proofs and storing their signatures on the ledger.
- **Updatable encryption**: storing ciphertexts, updating ciphertexts with update tokens, and retrieving ciphertexts.

8.4.1 Digital Signing

We propose to implement a contract to verify digital signatures on HLF. This will allow contracts to verify the integrity and authenticity of transactions. Concretely, the following functionalities have to be present:

- Off-chain generation of key pairs and signatures (for arbitrary messages).
- On-chain verification of signatures (and their respective messages).

We implement ML-DSA (4) as the digital signature scheme for this module. This algorithm has been standardized by NIST as FIPS204, and has passed extensive cryptanalysis and public review.

Aside from the requirements listed above, some considerations must be made regarding the dissemination of the public key and the verification of signatures. We assume that a Public Key Infrastructure (PKI) for ML-DSA is in place, where each user is aware of their own public key and the public keys of other users in the network. Future work may consider integrating key dissemination/management into HLF's native public key infrastructure (i.e. key rotation, key revocation, etc.) to ensure the security of the digital signing module.

Currently, HLF manages identities through X.509 certificates, which are issued by a Certificate Authority (CA) and can be used to authenticate users in the network. Ideally, these certificates would be implemented with ML-DSA. This way, peers can directly verify signatures against a user's public key (identity). The user can then sign arbitrary messages with their private key, and send the signature and the message to the chaincode for verification. The chaincode will verify the signature against the public key stored on the ledger, and return either a "Success" or "Failure" status message. Additionally, a user must take care to keep their private key secret and never share it with anyone.

Lastly, on-chain signature verification is executed in an SGX enclave, ensuring the message is never revealed to the executing peers. This prevents the network from recovering the potentially sensitive message, while still allowing the chaincode to verify the signature.

8.4.2 Zero Knowledge Proof

This project builds on top of the Vesper protocol for its zk-SNARKs. The following features should be present for this protocol to work:

- The same requirements as listed for Digital Signing in Section 8.4.1.
- Off-chain generation of zk-SNARK proofs.
- On-chain verification of zk-SNARK proof.
- On-chain storage of verified signatures of zk-SNARK proofs.

After a user requests their proof from the proof generation server, the user can send their signed proof to the chaincode through a middleware server as a POST request. This server parses the request and redirects it to the blockchain, where chaincode will verify the signature (see previous Section 8.4.1), verify the proof, and store the signature on the ledger. This allows for a use case where tokens or credentials are distributed via zk-SNARKs, and subsequently put on the ledger to be verified for any member of the network. SGX prevents the network from recovering the (sensitive) proof message by masking the application state from the nodes in the network.

This process is illustrated in Figure 8.2. Importantly, the middleware must be trusted with the plain proof message. This can be ensured by sending requests exclusively to a self-hosted (or similarly trusted) middleware server. This is feasible in the context of a private (consortium) blockchain, where organizations can set up their own middleware and peer servers.

8.4.3 Updatable Encryption

Lastly, updatable encryption of files is implemented with the following requirements in mind:

- Off-chain generation of public and secret keys.
- Off-chain encryption of plaintexts into ciphertexts.
- Off-chain decryption of ciphertexts into plaintexts.
- Off-chain generation of update tokens.
- On-chain storage of ciphertexts.
- On-chain updating of ciphertext through update tokens.

All updatable encryption operations are implemented according to Chen *et al.* (25), which is based on the hardness of learning with errors (LWE).

The general process for updatable encryption starts with a user and their to-be-encrypted plaintext message. The user calls KeyGen to generate their key pair (private matrix $\mathbf{R}_1$ and public matrix $\mathbf{A}$). The user can then encrypt their message with the public key (via $\mathsf{Encrypt}$) and decrypt their message with the private key (via $\mathsf{Decrypt}$). Storing ciphertexts on the ledger is straight-forward: the user sends their ciphertext to the chaincode through

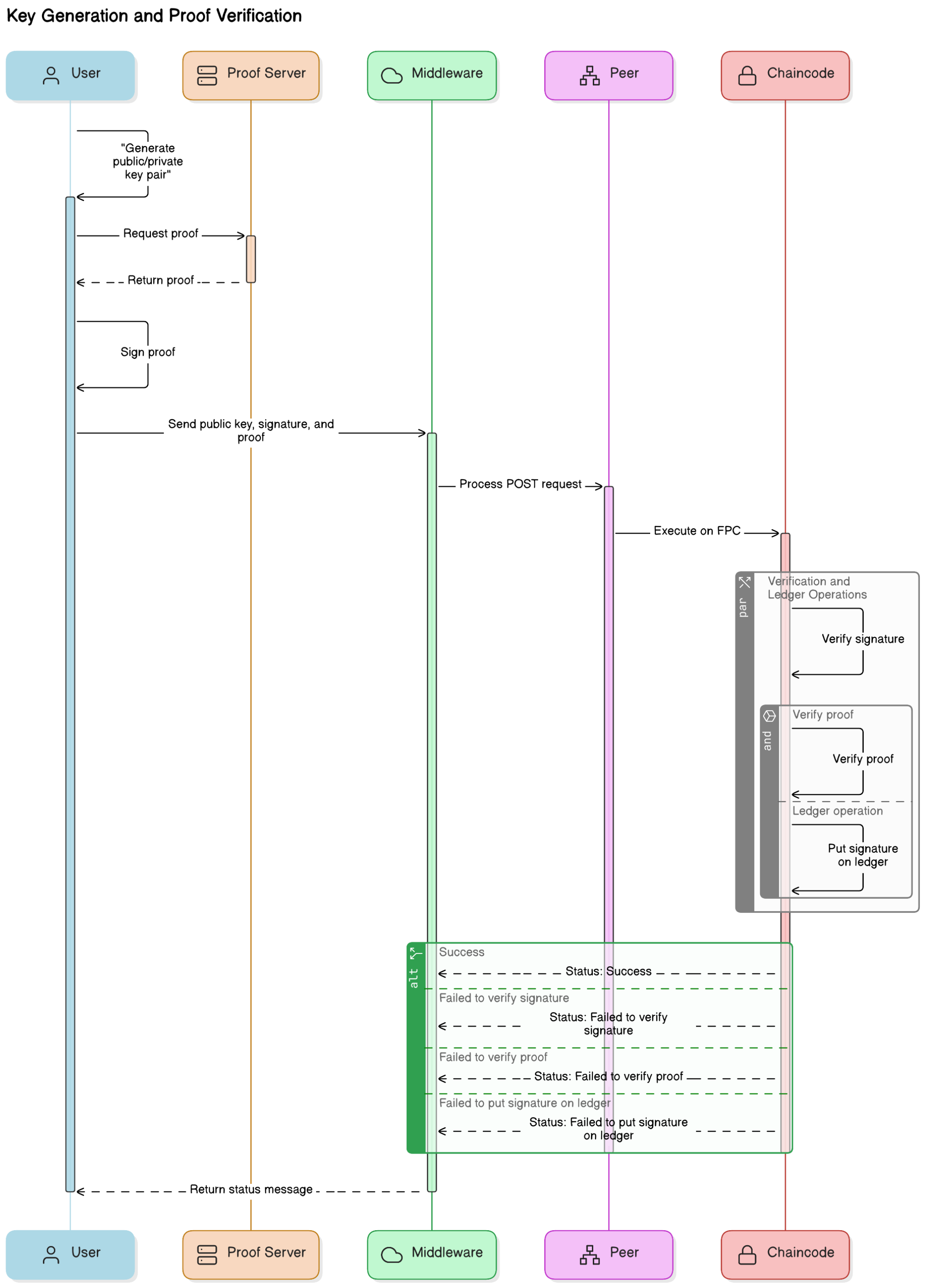
Key Generation and Proof Verification
User
Proof Server
Middleware
Peer
Chaincode
"Generate public/private key pair"
Request proof
Return proof
Sign proof
Send public key, signature, and proof
Process POST request
Execute on FPC
par
Verification and Ledger Operations
Verify signature
and
Verify proof
Verify proof
Ledger operation
Put signature on ledger
alt
Success
Status: Success
Failed to verify signature
Status: Failed to verify signature
Failed to verify proof
Status: Failed to verify proof
Failed to put signature on ledger
Status: Failed to put signature on ledger
Return status message
User
Proof Server
Middleware
Peer
Chaincode

Figure 8.2: ZKP Generation and Verification process

a middleware server, and receives either a ledger id to retrieve the ciphertext on success or a failure message. The ciphertext is stored as a key-value pair in the ledger, where the key is the ledger ID and the value is the ciphertext. Retrieving ciphertexts from the ledger is done similarly: the user requests a retrieval using the corresponding ledger ID and receives either the ciphertext or a failure message. Updating ciphertexts can be done by sending a request to the chaincode (through middleware) with a ledger ID and relevant update token.

Note that the middleware server is able to read any ciphertexts and update tokens sent to the chaincode. This is not a security issue, as the ciphertexts and update tokens by themselves do not leak any sensitive information (25). Nonetheless, if openly sharing ciphertexts and update tokens is not desired, an organization can choose to self-host the middleware server, which would prevent other organizations from having direct access.

9 Implementation

This section will discuss the implementation details of three blockchain modules. Chaincode for FPC is written in `C++11`. The middleware is written in Go. These languages are natively supported by FPC and are available in the default Docker development container, following the official installation and setup guide. Using languages already present in the container reduces dependencies and simplifies the setup process for users. This avoids the need to install additional languages or tools in the container. A guide for setting up and installing the modules can be found in the Appendix 12.1.

9.1 Client

Implementations of client-side operations for ML-DSA and Vesper are available from prior work (60, 61). However, Chen *et al.* (25) do not provide an implementation of the Updatable Encryption scheme. The paper also does not provide specific examples for parameter selection. We therefore implement only the required client-side operations for TDUE, and rely on the existing implementations for ML-DSA and Vesper.

9.1.1 ML-DSA Client

Pseudocode for client-side ML-DSA operations is given in Section 4.4 in Algorithm 3.

9.1.2 Vesper Client

Simplified pseudocode for client-side Vesper operations is given in Algorithm 6. This pseudocode only shows the Vesper parts of the proof generation and verification, and omits the ML-DSA signature generation and verification. It also omits the details of R1CS and QAP generation, which are given in detail in (42).

9.1.3 TDUE Client

We use the same notation and parameter names as in Section 5. For our implementation, q is a power of 2, giving exactly $k = \log_2 q$. This makes the LWE trapdoor inversion easier to implement (refer to Section 3.2.1). We implement a simple version of the TDUE scheme and use the 'lattice-estimator' tool (62) to estimate the parameters for the scheme.

Algorithm 6 Vesper Client Pseudocode

```
1: function GENERATEPROOF(x, w)                          ▷ Statement and witness resp.
2:     $(\mathbf{A}, \mathbf{B}, \mathbf{C}) \leftarrow$ Vesper.GenR1CS(x)
3:     qap $\leftarrow$ Vesper.GenQAP($\mathbf{A}, \mathbf{B}, \mathbf{C}$, w)
4:     (left, right) $\leftarrow$ qap
5:     proof $\leftarrow$ (Vesper.RegevEnc(left), Vesper.RegevEnc(right))
6:     return proof
7: end function
8: function VERIFYPROOF(proof)                           ▷ Used alongside ML-DSA
9:     proofIsValid $\leftarrow$ two halves of proof are equal
10:    return proofIsValid
11: end function
```

This tool estimates the parameters for the scheme based on the desired security level. We treat the ciphertext as an LWE sample of size $\bar{m} + 2nk$. The tool estimates the parameters $\bar{m} + 2nk = 571$, $\alpha q = 1.3$ and $q = 2^{10}$ to have a conjectured security level of 128 bits (equivalent to AES-128). An in-depth study has to be done to determine the actual security guarantees these parameters provide. Our TDUE parameters for multiple security levels are shown in Table 9.1. All parameter sets are chosen to encrypt messages of around 200 bits. We vary the $\bar{m}$ dimension to achieve this.

Parameter	**80-bit**	**128-bit**	**192-bit**
n (lattice dimension)	16	20	25
k (lattice dimension)	8	10	10
$\bar{m}$ (lattice dimension)	128	171	270
q (modulus)	2^8	2^{10}	2^{10}
αq (error std. dev.)	> 0.5	> 1.3	> 1.3

Table 9.1: TDUE parameters for different security levels.

We provide a static C++ library for the TDUE scheme, which is used by the client to generate key pairs, encrypt/decrypt messages, and generate update tokens. The code uses the Eigen library (63) internally for linear algebra operations and data structures. We omit tag matrices from the implementation, but do provide placeholders for them in the code. The TDUE library is a closely implemented version of the pseudocode given in Figure 5. We refer to the client-side TDUE functions as keygen, encrypt, decrypt, tokengen, and update_1.

The trapdoor functions Invert and SampleD are implemented as private functions in the library. It must be noted that these functions are implemented rather naively and are not optimized for performance. Future work may include optimizing these functions, for example, by incorporating (28) or (29).

Message encoding. We implement the function the bitstring message encoding function $\mathsf{encode} : \{0,1\}^{nk} \to \Lambda(\mathbf{G}^t)$ as a simple function that takes an array of bytes and greedily packs the bits into a vector of integers modulo q. Bytes are 8 bits long, while integers modulo q are k bits long. Due to the arbitrary nature of k, we require the message length to be less than or equal to $nk \cdot k$ bits. Any leftover bits are padded with zeros. For example, we can visualize encoding two (8-bit) bytes into two 10-bit numbers as follows:

$$[1010\ 1100]\ [0000\ 1111] \to [1010\ 1100\ 00]\ [0011\ 1100\ 00] \tag{9.1}$$

Note how the last four bits are padded with zeros. Decoding is the reverse operation, which takes a vector of integers modulo q and unpacks the bits into bytes.

Randomness. TDUE requires a lot of random number generation. Specifically:

- keygen uses randomness to sample $\mathbf{A}_0 \in \mathbb{Z}_q^{n\times\bar{m}}$ with uniform distribution and sample $\mathbf{R}_1, \mathbf{R}_2 \in \mathbf{Z}^{\bar{m}\times nk}$ with some discrete gaussian distribution,
- $\mathsf{encrypt}$ borrows randomness to generate error vectors with discrete gaussian distribution,
- $\mathsf{tokengen}$ employs $\mathsf{SampleD}$ to sample vectors (or entire matrices in our case) with some discrete gaussian distribution.

In our implementation, we borrow secure randomness from Linux's `/dev/urandom` to seed multiple Mersenne Twister pseudo-random number generators (PRNGs), which are used to sample both uniformly and with discrete gaussian distribution.

9.2 Middleware

The middleware is a simple Go HTTP server that acts as an interface between clients and the FPC chaincode. Table 9.2 shows the available endpoints.

Endpoint	**Method**	**Expected req. body (JSON)**
/verifySig	POST	`{"sig": ..., "msg": ..., "pk": ..., "ver": ...}`
/putVerificationResult	POST	`{"sig": ..., "msg": ..., "pk": ..., "ver": ...}`
/getVerificationResult	POST	`{"sig": ...}`
/putCiphertext	POST	`{"ciphertext": ...}`
/getCiphertext	POST	`{"cipherId": ...}`
/updateCipher	POST	`{"cipherId": ..., "token": ..., "ver": ...}`

Table 9.2: Middleware API Endpoints

All endpoints validate required fields and forward requests to the appropriate chaincode functions via the Fabric Private Chaincode SDK.

9.3 Chaincode

Requirements for developing the chaincode are:

- Docker, for the FPC development container, which includes the `C++` compiler and CMake.
- The reference implementation provided by the original 'CRYSTALS-Dilithium' paper by Ducas *et al.* (3), available at (61). A static library of the implementation is automatically downloaded, built, and included by the CMake file when building the chaincode.

```
contract/
|-- CMakeLists.txt
|-- Makefile
|-- test.sh
|-- client_app/
|   |-- CMakeLists.txt
|   |-- client_app.go
|   `-- client_app.h
|-- src/
    |-- b64.cpp
    |-- b64.h
    |-- ml_dsa_cc.cpp
    |-- tdue.cpp
    |-- tdue.h
    |-- verification.cpp
    `-- verification.h
```

Figure 9.1: Directory structure of the chaincode project

Figure 9.1 shows the directory structure of the project. The `src/` directory contains the main chaincode implementation, which is split into several files for clarity and modularity. The `Makefile` in the root directory is used to call CMake to build the chaincode and the client application. The `CMakeLists.txt` file automatically clones and builds the ML-DSA library, and compiles the source files to a static library. This project uses base-64 encoding to import and export data, and to store data on the ledger. The code for base-64 encoding/decoding is in `b64.cpp`. The `test.sh` file is a simple script to test the chaincode locally, which is useful for development and debugging purposes.

All chaincode functions have access to the ledger. In our pseudocode, we represent the ledger as a key-value store, where data can be retrieved by key.

9.3.1 Digital Signing

Pseudocode for all ML-DSA operations is given in Section 4.4 in Algorithm 3. As far as we know, there are no publically available audited implementations of ML-DSA at the time of writing. We use the reference implementation provided by the original CRYSTALS-Dilithium paper by Ducas *et al.* (3), which is updated to adhere to the latest standards (4). Specifically, we use the 'dilithium' GitHub repository published under 'pq-crystals' (61) to implement ML-DSA in our smart contract. The repository offers both a reference implementation and an `avx2` vector accelerated implementation, which improves the efficiency of the algorithm, as confirmed in Section 10. Unfortunately, FPC does not support vector acceleration. The chaincode thus uses the slightly slower reference implementation.

The enclave also doesn't support shared libraries, so the 'dilithium' code is built and imported as a static library. This isn't an option made directly available by the original code, but is a simple change to the Makefile.

The chaincode pseudocode for ML-DSA is given in Algorithm 7. The functions $\mathsf{putVerificationResult}_1$ and getVerificationResult are used in tandem with the ZKP verification process, which is described in the next section.

Algorithm 7 ML-DSA Chaincode Functions

```
 1: function VERIFYSIG(sig, msg, pk, ver)
 2:     return Dilithium.Verify(sig, msg, pk, ver)
 3: end function
 4: function PUTVERIFICATIONRESULT1(sig, msg, pk, ver)
 5:     result ← verifySig(sig, msg, pk, ver)
 6:     if result = true then
 7:         ledger[sig] ← true
 8:     end if
 9:     return result
10: end function
11: function GETVERIFICATIONRESULT(sig)
12:     return ledger[sig]
13: end function
```

9.3.2 Zero Knowledge Proof

Franschman (42) provides a simple implementation for proof generation, verification, and simple code for middleware and chaincode. We build on top of the code for proof generation and verification, which we leave mostly untouched, but replace the middleware and chaincode to better suit our own requirements. The proof generation runs on a (local) Python server, which can be queried to retrieve a proof for a fixed statement and witness. The proof is encoded as a base-64 string. The public key to sign the proof is sent along.

Verifying the proof is straightforward: we verify the signature of the proof using the ML-DSA chaincode, and then verify the proof by simply comparing both halves of the proof. We reimplement this in the chaincode. We therefore extend $\mathsf{putVerificationResult}_1$ to include proof verification capabilities. The updated function is given in Algorithm 8.

9.3.3 Updatable Encryption

The chaincode exposes the below functions for updatable encryption. Note that the version parameter is used solely to select the appropriate TDUE parameter set for the update, which determines the required memory allocation.

The updatable encryption functions in the chaincode are written in pure C++ by encoding vectors and matrices as arrays. The chaincode re-implements the $\mathsf{update}_1(\mathsf{cipher}, \mathsf{token})$ function without the use of the Eigen library. This reduces the dependencies of the chaincode and also avoids having to include the Eigen library in the FPC development container.

Algorithm 8 Updated ZKP Verification Function

```
function putVerificationResult2(sig, proof, pk, ver)
    result ← verifySig(sig, proof, pk, ver)
    proofIsValid ← verifyProof(proof)
    if result = true and proofIsValid = true then
        ledger[sig] ← true
        return accept
    end if
    return reject
end function
```

Algorithm 9 Updatable Encryption Chaincode Functions

```
function putCiphertext(cipher)
    cipherId ← Hash(cipher)
    ledger[cipherId] ← cipher
    return cipherId
end function
function getCiphertext(cipherId)
    return ledger[cipherId]
end function
function update2(cipherId, token, version)
    cipher ← ledger[cipherId]
    cipher' ← applyUpdate(cipher, token, version)
    ledger[cipherId] ← cipher'
end function
function applyUpdate(cipher, token, version)          ▷ Version for mem. allocation
    $(\mathbf{X}_{00}, \mathbf{X}_{10}, \mathbf{X}_{01}, \mathbf{X}_{11}, \mathbf{X}_{02}, \mathbf{X}_{12}, \mathbf{b}_0) \leftarrow$ parse token
    $[\mathbf{b}_a \mid \mathbf{b}_b \mid \mathbf{b}_c] \leftarrow$ parse cipher
    $\mathbf{b}'_a \leftarrow \mathbf{b}_a \cdot \mathbf{X}_{00} + \mathbf{b}_b \cdot \mathbf{X}_{10}$
    $\mathbf{b}'_b \leftarrow \mathbf{b}_a \cdot \mathbf{X}_{01} + \mathbf{b}_b \cdot \mathbf{X}_{11}$
    $\mathbf{b}'_c \leftarrow \mathbf{b}_a \cdot \mathbf{X}_{02} + \mathbf{b}_b \cdot \mathbf{X}_{12} + \mathbf{b}_c$
    $\mathbf{b}' \leftarrow [\mathbf{b}'_a \mid \mathbf{b}'_b \mid \mathbf{b}'_c]$
    return $\mathbf{b}' + \mathbf{b}_0$
end function
```

The $\mathsf{update}_2(\mathsf{cipher}, \mathsf{token})$ function performs a single matrix multiplication and addition, which is simple enough to implement without the use of a library. In practice, the matrix multiplication is broken down into three independent matrix multiplications to save on memory usage and computational overhead. Chen *et al.* (25) describe a token update as effectively performing the following calculation (omitting the old and new tag matrices):

$$\mathsf{TDUE.Update}(\mathbf{M}, \mathbf{b}_0, \mathbf{b}) = \mathbf{b}^t \cdot \mathbf{M} + \mathbf{b}_0^t \tag{9.2}$$

where $\mathbf{M}$ is the key-switching matrix, $\mathbf{b}_0$ is an encryption of the zero message using the new key and serves as a re-randomization of the secret $\mathbf{s}$, and $\mathbf{b} = [\mathbf{b}_a|\mathbf{b}_b|\mathbf{b}_c] \in \mathbb{Z}_q^{\bar{m}+nk+nk}$ is the old ciphertext. The resulting ciphertext is a vector of the form $\mathbf{b}' = [\mathbf{b}'_a|\mathbf{b}'_b|\mathbf{b}'_c] \in \mathbb{Z}_q^{\bar{m}+nk+nk}$.

$\mathbf{M}$ can be represented as a matrix of the form:

$$\mathbf{M} = \begin{bmatrix} \mathbf{X}_{00} & \mathbf{X}_{01} & \mathbf{X}_{02} \\ \mathbf{X}_{10} & \mathbf{X}_{11} & \mathbf{X}_{12} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{bmatrix} \tag{9.3}$$

where I is the identity matrix and the $\mathbf{X}_{ij}$ matrices are the sampled matrices from the token generation process. It makes sense to exclusively consider the first two rows of $\mathbf{M}$ when sending the token over the network, as the third row doesn't contain any information. The matrix multiplication in 9.2 is performed as follows:

$$\mathbf{b}'_a = \mathbf{b}_a \cdot \mathbf{X}_{00} + \mathbf{b}_b \cdot \mathbf{X}_{10} \tag{9.4}$$

$$\mathbf{b}'_b = \mathbf{b}_a \cdot \mathbf{X}_{01} + \mathbf{b}_b \cdot \mathbf{X}_{11} \tag{9.5}$$

$$\mathbf{b}'_c = \mathbf{b}_a \cdot \mathbf{X}_{02} + \mathbf{b}_b \cdot \mathbf{X}_{12} + \mathbf{b}_c \tag{9.6}$$

$\mathbf{b}_0$ is added element-wise to the resulting ciphertext vector $\mathbf{b}'$ to complete the update.

Token encoding. In practice, the update token is sent as a base-64 encoded string of the $\mathbf{X}_{ij}$ matrices concatenated by the $\mathbf{b}_0$ vector. Each element is stored as a 32-bit integer (which is suboptimal, because an element in $\mathbb{Z}_q$ only requires $k = \lceil \log q \rceil$ bits), and the structures are flattened into a single array. The array is base-64 encoded, which, in the worst case, increases the size of the token by a factor of 4/3. This token can get quite large. For convenience, we refer to the raw, optimal packing as the *optimal encoding* and we refer to the base-64 encoded, suboptimally packed token as the *base-64 encoding.* Future work may consider replacing base-64 encoding with a more efficient network transmission format, such as Protocol Buffers, to reduce the size of the update token, although this might not be feasible without major changes to the enclave structure. Using the parameters in Table 9.1, the size of the base-64 encoded update token is approximately 1 MB. Adding a tag matrix would not drastically increase the size of the update token, as the tag matrix can theoretically be generated from a seed using a pseudo-random generator.

Key storage. Recall that the public key is a matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$ and that the secret key is a matrix $\mathbf{R}_1 \in \mathbb{Z}^{\bar{m}\times nk}$. In theory, this would imply storing $n \times m$ elements in $\mathbb{Z}_q$ and $\bar{m} \times nk$ elements in $\mathbb{Z}$ for the public and secret key, respectively. In practice, we can trivially improve on this by trading memory for resampling overhead. We can instead store $\mathbf{A}_0$, $\mathbf{R}_1$, and $\mathbf{R}_2$ as a 256-bit seed to a pseudo-random generator, which generates the matrices on the fly. Aside from the sampling overhead, the public key will also incur some computational overhead because of the two additional matrix multiplications $-\mathbf{A}_0\mathbf{R}_1$ and $-\mathbf{A}_0\mathbf{R}_2$.

When the overhead for sampling or the matrix multiplications is not desired, the keys can be stored locally in their raw binary form, because the matrices don't require network-friendly encoding (they are never sent over the network).

Handling large update tokens. The default parameters for the enclave cause the enclave to crash from a *buffer overflow* when trying to process large update tokens ($>$ 100 KB). The obvious first step is to increase the heap size of the enclave, because the update token size exceeds the default heap size. Although this is a step towards the right direction, it does not solve the problem. The enclave overflows when deserializing the (arguments of the) query. Unfortunately, we haven't been able to find a solution to the (buffer overflow) problem yet. As a temporary fix, we can use smaller (toy) parameter sets, which do not cause the enclave to crash.

10 Evaluation

This section evaluates the performance of the chaincode implementation described in Section 9. We focus on two main aspects: how fast operations run and how much data they produce.

We test four components of our system. First, we measure the 'dilithium' library's key and signature sizes, and execution time for key generation, signing, and verification. Second, we measure the execution time of Vesper proof generation and report its proof sizes. Third, we evaluate the TDUE execution time for all client-side operations (key generation, encryption, decryption, token generation, and updating) and the sizes of the ciphertexts and update tokens. Finally, we measure the round-trip time (RTT) for the chaincode functions `putVerificationResult` and `updateCipher`.

10.1 Test Bed

All performance evaluations were run on the same machine with the following specifications:

- **CPU**: AMD Ryzen 5 2500U (Raven Ridge) with Radeon Vega Mobile Gfx (8) @ 2.000GHz, 4 Cores, 8 Threads
- **GPU**: AMD ATI Radeon Vega Series / Radeon Vega Mobile Series
- **RAM**: 15613MiB
- **OS**: Arch Linux x86_64

Note that this machine does not have an Intel CPU. FPC is not natively supported on this machine, because it requires Intel SGX support. Tests were instead run in a simulated SGX environment. This is enabled by default by the FPC development container, but can be explicitly set in the FPC configuration file `config.override.mk` through the following parameter: `SGX_MODE=SIM`.

Python programs were run using `python` version 3.13.3. `C++` programs were compiled using `gcc` version 15.1.1. Rust programs were compiled using `rustc` version 1.88.0. Local (non-container) Go programs were compiled using `go` version 1.24.4. To further reduce variance in the benchmarks, we disabled CPU frequency scaling by setting the CPU governor to 'performance' using the `cpupower` tool (64). Further details regarding the experimental setup and additional configuration parameters can be found in the Appendix 12.2. All tests are run single-threaded, unless otherwise specified.

10.2 'dilithium' evaluation

First, we evaluate the 'dilithium' library (61) on its own for its execution time on key generation, signing, and verification. Furthermore, we summarize the sizes of public keys and signatures. We do this for three different parameter sets: ML-DSA-44, ML-DSA-65, and ML-DSA-87. Refer to Section 4 for a description of these parameter sets. Table 10.1 shows the median execution times of the 'dilithium' library on our test bed.

Function	**ML-DSA-44**	**ML-DSA-65**	**ML-DSA-87**
Keygen median time (μs)	164	298	454
Keygen median time (AVX2) (μs)	80	143	232
Sign median time (μs)	651	1072	1307
Sign median time (AVX2) (μs)	194	316	419
Verify median time (μs)	188	293	485
Verify median time (AVX2) (μs)	81	140	227

Table 10.1: Measured 'dilithium' execution times by ML-DSA parameter sets.

Recall that signing for ML-DSA is a rejection sampling process, which can take multiple rounds of sampling before a valid signature is found. We report the median execution time rather than the average to better represent typical performance.

Table 10.2 summarizes the sizes of the verifying keys, signing keys, and signatures for the ML-DSA parameter sets.

Parameter	**ML-DSA-44**	**ML-DSA-65**	**ML-DSA-87**	**ed25519**
Verifying key size (bytes)	1312	1952	2592	32
Signing key size (bytes)	2560	4032	4896	32
Signature size (bytes)	2420	3309	4627	64

Table 10.2: ML-DSA and ECDSA key and signature sizes per parameter set (4) (5).

We can compare this to the Go standard crypto implementation of ECDSA. Specifically, we use the 'crypto/ed25519' package for the 'ed25519' curve (65). This makes sense, because the Hyperledger Fabric framework uses the 'ed25519' curve for its ECDSA implementation. We measure the execution time of key generation, signing, and verification over 10,000 executions of each function. The results are shown in Table 10.3. Relevant sizes of the keys and signatures are also shown in Table 10.2.

10.3 Vesper evaluation

Vesper is divided into two components: proof generation and proof verification. We omit a separate evaluation of proof verification, as its performance is nearly identical to that

Function	ed25519
Keygen (ns)	51,725
Sign (ns)	62,160
Verify (ns)	154,619

Table 10.3: Measured 'crypto/ecdsa' average execution times for ed25519 curve.

of ML-DSA signature verification. The verification process in Vesper consists primarily of verifying an ML-DSA signature and comparing two byte strings, resulting in negligible differences in execution time. Refer to Table 10.1 for the performance of ML-DSA signature verification.

We evaluate the performance of proof generation by randomly generating witnesses and measuring the average time it takes to generate a proof over 20 executions. Before measuring, we run the setup procedure once to initialize the circuit. Benchmarks were taken over 4 different parameter sets as defined by Franschman (42). These parameter sets are conjectured to provide security levels of 80, 128, 192, and 256 bits (we include 256-bit for continuity). Table 10.4 shows the concrete parameters for each security level. We use Franschman's Vesper implementation (60) to generate the proofs. The results are reported in Table 10.5. The number of runs for the 256-bit security level was reduced to 5, due to the lengthy execution times.

Parameter	80-bit	128-bit	192-bit	256-bit
n (lattice dimension)	4	6	9	12
g (order)	701	701	701	701
q (modulo)	2^8	2^8	2^8	2^8

Table 10.4: Vesper parameters by conjectured security levels.

Function	80-bit	128-bit	192-bit	256-bit
Proof generation (s)	0.4871	2.8107	32.6992	249.6747

Table 10.5: Average execution time for 1000 Vesper proof generations by parameter set.

In order to illustrate the performance difference between Vesper and classical zk-SNARKs, we benchmark Groth16 (43) on the same test bed. We use the 'groth16' library (66) implemented in Rust. The library offers three choices of elliptic curves: BLS12-381 for 128-bit security, MNT4-298 for 80-bit security, and MNT6-298 for 80-bit security (67). Results are shown in Table 10.6. The library is well optimized for multiple CPU cores, so we run the benchmarks both with a dedicated CPU core and with no CPU affinity set (allowing the process to use all available cores as scheduled by the OS).

Function	**BLS12-381**	**MNT4-298**	**MNT6-298**
Proof generation single (s)	49.489	45.331	86.708
Proof generation multi (s)	16.360	15.823	28.532
Proof verification single (μs)	8334	8835	17,553
Proof verification multi (μs)	8124	8193	15,305

Table 10.6: Average execution times for Groth16. Proof generation times are reported for 5 executions, while proof verification times are reported for 100 executions. Both single core and multiple core execution times are reported.

Groth16 also features constant proof size. Vesper proofs are 128 bytes, while Groth16 proofs consist of 2 elements in $\mathbb{G}_1$ and 1 element in $\mathbb{G}_2$, where $\mathbb{G}$ means the elliptic curve group. Groth16 proof sizes for three elliptic curves are summarized in Table 10.7.

Parameter	**Vesper**	**BLS12-381**	**MNT4-298**	**MNT6-298**
Proof size (bytes)	128	192	152	190

Table 10.7: Proof sizes for Vesper and three elliptic curves for Groth16 (BLS12-381, MNT4-298, MNT6-298).

10.4 TDUE evaluation

In this section, we evaluate the performance of our TDUE client implementation. We measure the time it takes to generate a key pair, encrypt a message, decrypt a message, generate an update token, and (locally) update a ciphertext. We test three different security levels: 80-bit, 128-bit, and 192-bit.

Parameters for the different security levels were presented in Table 9.1. Corresponding plaintext, ciphertext, and update token sizes for each security level are shown in Table 10.8.

Some alternative approaches can be taken regarding the storage/representation of the ciphertext, tokens, and keys. Ciphertexts and update tokens can be stored optimally by encoding exactly the required number of bits for every element, which is $k = \lceil \log_2(q) \rceil$ bits per element in $\mathbb{Z}_q$. This gives us a ciphertext size of $m \cdot k$ bits, where m is the number of elements in the ciphertext $(\bar{m} + 2nk)$, and an update token size of $m \cdot (\bar{m} + nk) \cdot k$.

In our implementation, however, we use base-64 encoding to safely send ciphertexts and update tokens over the network as strings. We also use 32 bits per element in $\mathbb{Z}_q$ instead of the optimal number of bits. This results in a ciphertext size of $4 \cdot (m \cdot 32)/3$ bits, and an update token size of $4 \cdot (m \cdot (\bar{m} + nk) \cdot 32)/3$ bits. This is a trade-off between performance and storage efficiency.

Parameter	80-bit	128-bit	192-bit
Plaintext (bits)	128	200	250
Ciphertext optimal (bits)	3072	5710	7700
Ciphertext b64 (bits)	16,384	24,363	32,854
Update token optimal (bits)	789,504	2,124,120	4,011,700
Update token b64 (bits)	4,194,304	9,038,550	23,047,083
Public key seeded (bits)	768	768	768
Public key optimal (bits)	32,768	114,200	192,500
Public key suboptimal (bits)	131,072	365,440	616,000
Private key seeded (bits)	256	256	256
Private key optimal (bits)	131,072	342,000	675,000
Private key suboptimal (bits)	524,288	1,094,400	2,160,000

Table 10.8: TDUE plaintext, ciphertext, token, and key sizes for different security levels. The sizes for ciphertext and update token are given for both the optimal encoding and our base-64 encoding. Optimal and suboptimal key sizes refer to the naive implementation (refer to Section 9.3.3).

As for the keys, when stored naively (but with optimal bit representation), the public key $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ requires $n \cdot m \cdot k$ bits, and the private key $\mathbf{R}_1 \in \mathbb{Z}_q^{\bar{m} \times nk}$ requires $\bar{m} \cdot n \cdot k$ bits. With suboptimal bit representation (32-bit integers), this evaluates to $n \cdot m \cdot 32$ bits for the public key and $\bar{m} \cdot n \cdot 32$ bits for the private key. However, we can also use 256-bit PRNG seeds to generate the matrices for the public and private keys. Refer to Sections 5.2 and 9.3.3 for further details.

For each security level, we run 100 iterations of each operation and report the mean execution time. Execution times are shown in Table 10.9.

Function	80-bit	128-bit	192-bit
Keygen time (μs)	3010	6310	12,837
Encrypt time (μs)	129	197	252
Decrypt time (μs)	55	133	219
Tokengen time (μs)	32,929	126,841	272,912
Update time (μs)	150	480	1347

Table 10.9: Average time for TDUE functions by security level.

The results show that execution time increases with security level, which is expected due to the larger parameter sizes required for higher security levels. Both key generation and token generation take significantly longer than encryption and decryption. Key generation is largely a process of sampling random values, which might explain the longer execution time. As for token generation, it relies on the SampleD function to invert an SIS instance, which is computationally expensive (refer to Section 5.2). This is further exacerbated by the naive SampleD implementation, and the fact that the algorithm requires three calls to the trapdoor function.

As far as we know, there are no standardized or widely used updatable encryption schemes at the time of writing. Therefore, it is difficult to compare these benchmarks to an established community standard. In order to provide some anchor for comparison, we compare the TDUE performance to an alternative LWE-based updatable encryption scheme, namely the Ring-LWE-based scheme by Boneh *et al.* (6). To keep results comparable, we use a plaintext message size of 256 bits, which is approximately the same as the 192-bit security level of TDUE (250 bits). We also use their the highest security level (with smallest ciphertext expansion) of 128-bit security. Each function is run at least 1000 times, and the average execution time is reported. Results are shown in Table 10.10.

Function	**128-bit**
Keygen time (μs)	90
Encrypt time (μs)	11,377
Decrypt time (μs)	10,854
Tokengen time (μs)	548
Update time (μs)	10,929

Table 10.10: Average time for Ring-LWE based UE scheme by Boneh *et al.* (6) for highest security level.

The results show that the TDUE scheme is significantly slower for key generation and token generation, but faster for encryption, updating, and decryption.

10.5 Chaincode results

For the chaincode, we evaluate the performance of the functions putVerificationResult and updateCipher. We measure the RTT of these functions from the middleware to the chaincode and back. We run the chaincode in an FPC development container, which is configured to use the 'dilithium' library as described in Section 9. The middleware is run locally in the same container as the chaincode, and the chaincode is run in a simulated SGX environment.

We measure the average RTT of putVerificationResult by calling it 50 times for each of the three ML-DSA parameter sets. The message is a 128-byte Vesper proof, which is constant across all parameter sets. We also generate a new key pair for each call, and sign the message with the random key. The RTT excludes the time it takes to generate the keys, generate the proof, and sign the message, as these are client-side operations. Results are reported in Table 10.11.

Function	**ML-DSA-44**	**ML-DSA-65**	**ML-DSA-87**
putVerificationResult (μs)	22,848	23,578	25,272

Table 10.11: Average RTT for 50 runs of putVerificationResult by parameter set

We can see that there's a slight increase in RTT for putVerificationResult with increasing security level. This is expected, since the execution time of signature verification on its own also increases with the parameter sets (refer to Table 10.1) Note, however, that the differences are a magnitude larger than the differences in execution time of the 'dilithium' library (refer to Table 10.1). This difference likely comes from overhead in the FPC enclave, Hyperledger Fabric transaction processing, or the extra time needed to store larger signatures on the ledger.

Next, we evaluate the updateCipher function. We were unable to run updateCipher for the normal parameter sets due to the buffer overflow bug (refer to Section 9.3.3). Instead, we use a toy parameter set with a smaller token size of around 2.5 KB. The toy parameter set is not secure and is only used for testing purposes. Table 10.12 shows the parameters used for the toy parameter set.

Parameter	**Value**
n (lattice dimension)	2
k (lattice dimension)	5
$\bar{m}$ (lattice dimension)	10
q (modulus)	2^5
αq (error std. dev.)	–

Table 10.12: TDUE toy parameters (not secure).

With these toy parameters, updateCipher is called 50 times with update tokens generated by the TDUE client. New, random ciphertexts are used for each call, and the average RTT is measured. We measure an average RTT of 2078 ms. This is significantly higher than the RTT for putVerificationResult (which is approx. 25 ms for the strictest security parameters), which is expected due to the larger size of the update token and the complexity of the update operation. The larger size results in more data that needs to be transmitted, read, and stored on the ledger. The update operation itself is also implemented rather inefficiently, as it naively computes the matrix multiplication.

11 Discussion

In this chapter, we discuss the results of our measurements and future directions for research and development. We also offer insights into the limitations of our implementation and propose ways to improve it.

11.1 Contributions

We implemented and benchmarked three modules for signing, zero-knowledge proofs, and encryption in Hyperledger Fabric. We provide a full TDUE implementation, including the trapdoor functions, and a chaincode that integrates ML-DSA and zk-SNARKs built using FPC. We also provide TDUE parameter sets for three levels of conjectured security (80-bit, 128-bit, and 192-bit), although no in-depth analysis has been conducted to determine the actual security guarantees of these parameters. We hope this implementation can serve as a starting point for further development of TDUE and its integration into Hyperledger Fabric. We also propose ways to greatly reduce the size of public and private keys. Although there are still many points of improvement, the benchmarks for the TDUE library are promising, with execution times of the main operations in the order of milliseconds for the 192-bit security level.

Franschman (42) had initiated efforts to implement Vesper for FPC, although these efforts were not fully realized. Their work did not include ML-DSA verification for the FPC variant, and chaincode execution was tunneled through a makeshifted communication channel. We complete their work by implementing Vesper fully for FPC, allowing for the confidential verification of proofs.

Finally, we implemented ML-DSA signature verification (and storage) in the chaincode. This implementation supports all three standardized parameter sets (ML-DSA-44, ML-DSA-65, ML-DSA-87) and works in tandem with the Vesper protocol to provide zero-knowledge proofs. ML-DSA boasted great results for the chaincode environment, with RTTs in the order of milliseconds for the verification of signatures.

11.2 Limitations and future work

Identity management. As stated in Section 8.4.1, some form of identity management is necessary in order for ML-DSA to be effective. A network must know its participants and their public keys. Our current implementation assumes a PKI system is in place. HLF currently relies on ECDSA for its PKI. To fully arm against quantum attacks, we propose to use ML-DSA as a replacement for ECDSA in all X.509 certificates. This would entail changing the certificate generation process (in CAs) to use ML-DSA instead of ECDSA. The HLF framework would also have to be updated to support ML-DSA signatures in the certificate verification process.

Replacing cryptographic primitives. Our implementation builds on top of the existing architecture. Ideally, the cryptographic primitives that we implement should be available directly in the HLF framework. However, without major changes to the HLF codebase, it is not easy to replace the elliptic curve cryptographic elements with ML-DSA. Future work could focus on integrating ML-DSA into the HLF framework, allowing for a more seamless transition to post-quantum cryptography. Ideally, the framework's cryptographic components would be modular and pluggable, enabling 'plug-and-play' replacement of cryptographic primitives as needed. This would also allow flexible choice of specific schemes. We can imagine, for example, that a consortium may want to opt for a different post-quantum signature scheme than ML-DSA, such as the NIST standardized SLH-DSA (68).

To add to this, there is no need for a network to fully transition to post-quantum cryptography at once. A consortium may choose to use ML-DSA for some of its applications, while still relying on classical cryptography in others. Pluggable cryptography would more easily allow for such a hybrid approach.

TDUE implementation. A main point of improvement in our implementation is the performance of TDUE. Token sizes are very large, which causes issues with the FPC enclave. Unfortunately, this means we were not able to benchmark the performance of our main parameter sets. However, the toy parameters already show suboptimal performance for online (chaincode) operation. This is mainly due to the naiveté of the implementation, but can also be attributed to the large size of the update tokens (quadratic in the size of ciphertexts). Additionally, these sizes scale quadratically with the security parameter. Performance of real-world applications with higher security requirements may suffer from this. We propose to investigate the issue of large update tokens further, either by optimizing token generation or by changing the way update tokens are processed in the FPC enclave. Implementing the 'packing' functionality of TDUE (encrypting multiple messages in a single ciphertext) could also help reduce the relative on-chain runtime.

A further point of improvement for the TDUE implementation, is the (client-side) implementation of the trapdoor functions. The main bottleneck in the current implementation is the sampling function, which is evident through the execution time of $\mathsf{Tokengen}$. We propose to implement a more efficient version of Gaussian sampling, as detailed in (28).

Lastly, the current implementation is missing tag matrix functionality. Although this should have minor impact on performance, it is still a limitation.

12 Conclusion

In this thesis, we have explored the integration of post-quantum cryptographic primitives into Hyperledger Fabric, a prominent permissioned blockchain framework. With fully fledged quantum computers on the horizon, blockchain systems relying on classical cryptographic algorithms face increasingly serious security threats. Hyperledger Fabric, specifically, is at risk due to its reliance on the ECDSA algorithm for digital signatures in identity management[1]. Our key contribution for Hyperledger Fabric is the integration of ML-DSA, a post-quantum digital signing algorithm. Additionally, we implement and benchmark two other important post-quantum primitives: Vesper, a lattice-based zk-SNARK, and TDUE, a lattice-based updatable encryption scheme. We map out a design for integrating these primitives into FPC, which provides a secure enclave environment for executing chaincode.

Our implementation of signing relies on the assumption that a public key infrastructure is in place. Future work could focus on embedding ML-DSA into the Hyperledger Fabric framework. We also note that the TDUE implementation is not yet optimized for performance, and the size of update tokens can be prohibitively large. Despite these limitations, our benchmarks show promising performance for all three modules.

We hope that this thesis serves as a foundation for future development in securing private blockchains against quantum threats.

[1]There has been some community interest in officially adding post-quantum signing to HLF, although no concrete proposals have been made yet (69).

References

[1] Shweta Agrawal. **Lattice Trapdoors**. `https://www.youtube.com/live/fVen9vkFWlk?si=XNSzrG_KI9wyMuBy`, January 2020. Lecture, Lattices: Algorithms, Complexity, and Cryptography Boot Camp, Simons Institute. iv, 12

[2] C. P. Schnorr. **Efficient Identification and Signatures for Smart Cards**. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 239–252, New York, NY, 1990. Springer New York. iv, 13, 16

[3] Leo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. **CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme**. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 238–268, 02 2018. iv, 20, 21, 23, 27, 47

[4] National Institute of Standards and Technology. **Module-Lattice-Based Digital Signature Standard**. Technical Report FIPS 204, U.S. Department of Commerce, Washington, D.C., 2024. v, 20, 26, 27, 40, 47, 53

[5] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. **Ed25519: high-speed high-security signatures**. `https://ed25519.cr.yp.to/`, 2017. Accessed: 2025-07-09. v, 53

[6] Dan Boneh, Saba Eskandarian, Sam Kim, and Maurice Shih. **Improving Speed and Security in Updatable Encryption Schemes**. Cryptology ePrint Archive, Paper 2020/222, 2020. v, 32, 57

[7] Peter W. Shor. **Algorithms for Quantum Computation: Discrete Logarithms and Factoring**. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 124–134. IEEE, 1994. 1, 4

[8] Peter W. Shor. **Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer**. *SIAM Journal on Computing*, **26**(5):1484–1509, 1997. 1

[9] Oded Regev. **On Lattices, Learning with Errors, Random Linear Codes, and Cryptography**. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC)*, pages 84–93. ACM, 2005. 1, 8, 32, 33

[10] M. Ajtai. **Generating hard instances of lattice problems (extended abstract)**. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 99–108, New York, NY, USA, 1996. Association for Computing Machinery. 1, 7

[11] National Institute of Standards and Technology. **Post-Quantum Cryptography Project**. `https://csrc.nist.gov/projects/post-quantum-cryptography`. Accessed: 2025-05-26. 1

[12] Nikhil Kumar Parida, Chandrashekar Jatoth, V. Dinesh Reddy, Md. Muzakkir Hussain, and Jamilurahman Faizi. **Post-quantum distributed ledger technology: a systematic survey**. *Scientific Reports*, **13**(1):20729, 2023. 4

[13] Lov K. Grover. **A fast quantum mechanical algorithm for database search**. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, pages 212–219. ACM, 1996. 4

[14] Jing Chen and Silvio Micali. **Algorand**, 2017. 4

[15] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. **Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU**, 2018. Available at `https://falcon-sign.info/falcon.pdf`. 4

[16] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. **Verifiable Random Functions**. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 120–130. IEEE, 1999. 4

[17] Robert Campbell. **Transitioning to a hyperledger fabric quantum-resistant classical hybrid public key infrastructure**. *The Journal of The British Blockchain Association*, 2019. 4

[18] Amelia Holcomb, Geovandro Pereira, Bhargav Das, and Michele Mosca. **PQFabric: A Permissioned Blockchain Secure from Both Classical and Quantum Attacks**. In *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9, 2021. 4

[19] Shahroz Abbas, Ajmery Sultana, and Georges Kaddoum. **Quantum-Safe Blockchain in Hyperledger Fabric**. *IEEE Networking Letters*, **7**(1):61–65, 2025. 4

[20] Engin Zeydan, Yekta Turk, S. Bugrahan Ozturk, Hakan Mutlu, and A. Anil Dundar. **Post-Quantum Blockchain-Based Data Sharing for IoT Service Providers**. *IEEE Internet of Things Magazine*, **5**(2):96–101, 2022. 4

[21] Ali Abbas Hussain, Aamir Raza, Abdul Karim Sajid Ali, and Aashesh Kumar. *Annual Methodological Archive Research Review*, **2**(5):19–27, Dec. 2024. [link]. 4

[22] Shweta Agrawal, Dan Boneh, and Xavier Boyen. **Efficient Lattice (H)IBE in the Standard Model**. In *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, **6110** of *Lecture Notes in Computer Science*, pages 553–572. Springer, 2010. 6

[23] Shweta Agrawal, Craig Gentry, Shai Halevi, and Amit Sahai. **Discrete Gaussian Leftover Hash Lemma over Infinite Domains**. Cryptology ePrint Archive, Paper 2012/714, 2012. 7, 11

[24] Craig Gentry, Amit Sahai, and Brent Waters. **Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based**. Cryptology ePrint Archive, Paper 2013/340, 2013. 8

[25] Huanhuan Chen, Yao Jiang Galteland, and Kaitai Liang. **CCA-1 Secure Updatable Encryption with Adaptive Security**. Cryptology ePrint Archive, Paper 2022/1339, 2022. 8, 14, 28, 29, 30, 31, 41, 43, 44, 50

[26] Daniele Micciancio and Chris Peikert. **Trapdoors for Lattices: Simpler, Tighter, Faster, Smaller**. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pages 700–718, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. 8, 9, 10, 11, 12, 30, 31

[27] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. **Trapdoors for Hard Lattices and New Cryptographic Constructions**. Cryptology ePrint Archive, Paper 2007/432, 2007. 8, 10

[28] Nicholas Genise and Daniele Micciancio. **Faster Gaussian Sampling for Trapdoor Lattices with Arbitrary Modulus**. Cryptology ePrint Archive, Paper 2017/308, 2017. 8, 45, 60

[29] Pauline Bert, Gautier Eberhart, Lucas Prabel, Adeline Roux-Langlois, and Mohamed Sabt. **Implementation of Lattice Trapdoors on Modules and Applications**. In Jung Hee Cheon and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography*, pages 195–214, Cham, 2021. Springer International Publishing. 8, 45

[30] L Babai. **On Lovász' lattice reduction and the nearest lattice point problem**. *Combinatorica*, **6**(1):1–13, January 1986. 10

[31] Philip Klein. **Finding the closest lattice vector when it's unusually close**. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '00, page 937–941, USA, 2000. Society for Industrial and Applied Mathematics. 10

[32] Chris Peikert. **An Efficient and Parallel Gaussian Sampler for Lattices**. Cryptology ePrint Archive, Paper 2010/088, 2010. 12

[33] Yannick Seurin. **On the Exact Security of Schnorr-Type Signatures in the Random Oracle Model**. Cryptology ePrint Archive, Paper 2012/029, 2012. 14

[34] Elaine Barker. **NIST Special Publication 800-57 Part 1 Revision 4: Recommendation for Key Management**. Technical report, National Institute of Standards and Technology, 2016. 14

[35] Dan Boneh, Kevin Lewi, Hart Montgomery, and Ananth Raghunathan. **Key Homomorphic PRFs and Their Applications**. In *Advances in Cryptology – CRYPTO 2013*, **8042** of *Lecture Notes in Computer Science*, pages 410–428. Springer, 2013. 14

[36] Adam Everspaugh, Kenneth Paterson, Thomas Ristenpart, and Sam Scott. **Key Rotation for Authenticated Encryption**. Cryptology ePrint Archive, Paper 2017/527, 2017. 14

[37] S Goldwasser, S Micali, and C Rackoff. **The knowledge complexity of interactive proof-systems**. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, page 291–304, New York, NY, USA, 1985. Association for Computing Machinery. 15

[38] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. **Zerocash: Decentralized Anonymous Payments from Bitcoin**. Cryptology ePrint Archive, Paper 2014/349, 2014. 15

[39] Michael Rosenberg, Jacob White, Christina Garman, and Ian Miers. **zk-creds: Flexible Anonymous Credentials from zkSNARKs and Existing Identity Infrastructure**. Cryptology ePrint Archive, Paper 2022/878, 2022. 15

[40] Amos Fiat and Adi Shamir. **How To Prove Yourself: Practical Solutions to Identification and Signature Problems**. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg. 15

[41] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. **From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again**. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, page 326–349, New York, NY, USA, 2012. Association for Computing Machinery. 16

[42] Lesley Franschman. *The Vesper Protocol: Leveraging Zero-Knowledge Proofs and SGX Enclaves in Hyperledger Fabric Smart Contracts.* Master's thesis, Delft University of Technology, Delft, Netherlands, September 2024. Supervised by Kaitai Liang and Huanhuan Chen. 17, 33, 44, 48, 54, 59

[43] Jens Groth. **On the Size of Pairing-based Non-interactive Arguments**. Cryptology ePrint Archive, Paper 2016/260, 2016. 17, 33, 54

[44] J. T. Schwartz. **Fast Probabilistic Algorithms for Verification of Polynomial Identities**. *Journal of the ACM*, **27**(4):701–717, October 1980. 19

[45] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. **Scalable, transparent, and post-quantum secure computational integrity**. `https://eprint.iacr.org/2018/046.pdf`, March 2018. International Association for Cryptologic Research. 19

[46] Thomas Espitau, Pierre-Alain Fouque, Benoit Gérard, and Mehdi Tibouchi. **Side-Channel Attacks on BLISS Lattice-Based Signatures – Exploiting Branch Tracing Against StrongSwan and Electromagnetic Emanations in Microcontrollers**. Report 2017/505, Cryptology ePrint Archive, 2017. To appear in CCS 2017. 20

[47] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. **Flush, Gauss, and Reload – A Cache Attack on the BLISS Lattice-Based Signature Scheme**. Cryptology ePrint Archive, Paper 2016/300, 2016. 20

[48] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. **To BLISS-B or not to be - Attacking strongSwan's Implementation of Post-Quantum Signatures**. Cryptology ePrint Archive, Paper 2017/490, 2017. 20

[49] Piotr Rudnicki. **Little Bézout Theorem (Factor Theorem)**. *Formalized Mathematics*, **12**(1):49–58, 2004. 24

[50] Yujiang Jiang. **The Direction of Updatable Encryption Does Not Matter Much**. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020, Part III*, **12493** of *Lecture Notes in Computer Science*, pages 529–558. Springer, 2020. 31

[51] Ryo Nishimaki. **The Direction of Updatable Encryption Does Matter**. In Goichiro Hanaoka, Junji Shikata, and Yohei Watanabe, editors, *Public-Key Cryptography – PKC 2022, Part II*, **13178** of *Lecture Notes in Computer Science*, pages 194–224. Springer, 2022. 32

[52] Rishabh Bhadauria, Zhiyong Fang, Carmit Hazay, Muthuramakrishnan Venkitasubramaniam, Tiancheng Xie, and Yupeng Zhang. **Ligero++: A New Optimized Sublinear IOP**. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 2025–2038, New York, NY, USA, 2020. Association for Computing Machinery. 34

[53] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. **Fractal: Post-Quantum and Transparent Recursive Proofs from Holography**. Cryptology ePrint Archive, Paper 2019/1076, 2019. 34

[54] Yuval Ishai, Hang Su, and David J. Wu. **Shorter and Faster Post-Quantum Designated-Verifier zkSNARKs from Lattices**. Cryptology ePrint Archive, Paper 2021/977, 2021. 34

[55] Rosario Gennaro, Michele Minelli, Anca Nitulescu, and Michele Orrù. **Lattice-Based zk-SNARKs from Square Span Programs**. Cryptology ePrint Archive, Paper 2018/275, 2018. 34

[56] Satoshi Nakamoto. **Bitcoin: A Peer-to-Peer Electronic Cash System**. `https://bitcoin.org/bitcoin.pdf`, 2008. 35

[57] Vitalik Buterin. **Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform**. `https://ethereum.org/en/whitepaper/`, 2014. 35

[58] International Telecommunication Union. **ITU-T Recommendation X.509: Information technology — Open Systems Interconnection — The Directory: Public-key and attribute certificate frameworks**. `https://www.itu.int/rec/T-REC-X.509`, 2019. Series X: Data Networks, Open System Communications and Security. 36

[59] Marcus Brandenburger, Christian Cachin, Rüdiger Kapitza, and Alessandro Sorniotti. **Blockchain and Trusted Computing: Problems, Pitfalls, and a Solution for Hyperledger Fabric**. *CoRR*, **abs/1805.08541**, 2018. 37

[60] Lesley Franschman. **Post-Quantum Smart Contract**. `https://github.com/lfranschman/post-quantum-smart-contract`, 2024. Commit: ce29051, Accessed: 2025-05-31. 44, 54

[61] pq-crystals. **CRYSTALS-Dilithium: Reference Implementation**. `https://github.com/pq-crystals/dilithium`, 2024. Commit: 444cdcc, Accessed: 2025-05-29. 44, 47, 53

[62] Martin R. Albrecht, Rachel Player, and Sam Scott. **On the concrete hardness of Learning with Errors**. Cryptology ePrint Archive, Paper 2015/046, 2015. 44

[63] Gaël Guennebaud, Benoît Jacob, et al. **Eigen: C++ Template Library for Linear Algebra (Version 3.4.0)**. `http://eigen.tuxfamily.org`, 2021. Accessed: 2025-05-30. 45, 69

[64] Len Brown and Thomas Renninger. **cpupower: CPU frequency scaling tool**. `https://github.com/torvalds/linux/blob/master/tools/power/cpupower/utils/cpupower.c`, 2022. Linux kernel source code, Accessed: 2025-07-01. 52

[65] The Go Authors. **crypto/ed25519 — Go Standard Library**. `https://pkg.go.dev/crypto/ed25519`, 2025. Accessed: 2025-07-09. 53

[66] arkworks rs. **A Rust implementation of the Groth16 zkSNARK**. `https://github.com/arkworks-rs/groth16`, 2025. Commit: d570ee5, Accessed: 2025-07-09. 54

[67] Maria Corte-Real Santos, Craig Costello, and Michael Naehrig. **On cycles of pairing-friendly abelian varieties**. Cryptology ePrint Archive, Paper 2024/869, 2024. 54

[68] National Institute of Standards and Technology. **Stateless Hash-Based Digital Signature Standard**. Technical Report FIPS 205, U.S. Department of Commerce, Washington, D.C., 2024. 60

[69] bestbeforetoday. **Issue #3763: Support post-quantum crypto algorithms**. GitHub Issue, 2022. Accessed: 2025-07-04. 61

[70] Hyperledger Fabric. **Hyperledger Fabric Private Chaincode**, 2025. Commit: 5a3481d, Accessed: 2025-07-01. 69

[71] Jaylan Lee. **FPC ML-DSA Contract**, 2025. Commit: 6b3da52, Accessed: 2025-07-01. 69

[72] Jaylan Lee. **FPC ML-DSA Application Client**, 2025. Commit: f5a0417, Accessed: 2025-07-01. 69

[73] Jaylan Lee. **FPC ML-DSA Web App**, 2025. Commit: b176721, Accessed: 2025-07-01. 69

[74] Jaylan Lee. **TDUE library**, 2025. Commit: 7f6bceb, Accessed: 2025-07-01. 69

[75] Google. **Google Benchmark**, 2025. Accessed: 2025-06-29. 69

Appendix

12.1 Setting up the development environment

The full code base consists of three parts: the chaincode, the middleware application, and the TDUE library. This section guides the reader through the process of setting up the development environment for this code base.

Chaincode. Firstly, the FPC development environment must be set up. This is done by following the instructions in the `fabric-private-chaincode` repository (70). After the Docker container is available, our chaincode can be imported into the container. Our chaincode is available in the `fpc-ml-dsa-contract` repository (71). We recommend cloning this repository into the `samples/chaincode` directory of the container. Further instructions on how to run the chaincode are available in the repository.

Middleware application. The middleware application also runs in the FPC Docker container. The middleware is available in the `fpc-ml-dsa-application-client` repository (72). The application should be cloned into the `samples/application` directory of the container. In order to run the application, we first need a local deployment. The `samples/deployment/test-network` directory contains a guide on how to set up a local deployment of Hyperledger Fabric. After deployment of the network, the application is ready to be run. Further instructions on how to run the application are available in the repository.

Simple web app. A simple web application to interface with the middleware server is available in the `fpc-ml-dsa-web-app` repository (73). The web application uses React, Typescript, and Vite. The web application can be run by following the instructions in the repository. The interface only provides a simple way to generate key pairs, sign messages, and send them to the middleware server.

TDUE. The TDUE implementation is available in the `TDUE` repository (74). The TDUE library is written in C++ and depends on the Eigen library (63) for linear algebra operations. Further instructions on how to build and use the library are available in the repository.

12.2 Reducing variance in benchmarks

This section covers the measures that were taken to prevent or reduce variance in benchmarking the programs in this thesis. All benchmarking in this work was done with the help of Google's 'benchmark' tool (75). The relevant measures were mostly taken from guides posted in the corresponding GitHub repository.

CPU frequency scaling can be a source of noise while running benchmarks. The simple option to stabilize this noise is to disable the frequency scaling. We used the `cpupower` tool to disable frequency scaling as follows:

```
sudo cpupower frequency-set --governor performance
```

Further measures to reduce noise from CPU frequency scaling include forcing the same CPU to be used durig benchmarks and disabling turbo/boost. These measures can be applied as follows:

```
echo 0 | sudo tee /sys/devices/system/cpu/cpufreq/boost
taskset -c 0 ./mybenchmark
```