The Vesper Protocol

Leveraging Zero-Knowledge Proofs and SGX Enclaves in Hyperledger Fabric Smart Contracts

Msc Thesis EEMCS Cybersecurity
Lesley Franschman

TU Delft

Date: Saturday 31st August, 2024

The Vesper Protocol

Leveraging Zero-Knowledge Proofs and SGX Enclaves in Hyperledger Fabric Smart Contracts

by

Lesley Franschman
Supervised by Kaitai Liang and Huanhuan Chen

To obtain the degree of Master of Science
in Cybersecurity
at Delft University of Technology

to be defended on September 12th, 2024

Student number: 4853377
Project duration: November 1st, 2023 – september 12th, 2024

Thesis committee:
Dr. Kaitai Liang, The Cybersecurity Group, TU Delft
Professor Georgios Smaragdakis, The Cybersecurity Group, TU Delft
Dr. Jérémie Decouchant, The Delft Blockchain Lab, TU Delft

Faculty of EEMCS · Delft University of Technology

Preface

This preface provides context and background for the research conducted in this thesis, titled "The Vesper Protocol: Leveraging Zero-Knowledge Proofs and SGX Enclaves in Hyperledger Fabric Smart Contracts". The work presented in this thesis was carried out under the supervision of Dr. Kaitai Liang and Huanhuan Chen at the Cybersecurity Group, TU Delft. The research project commenced on November 1st, 2023, and culminated in the completion of this thesis on September 11th, 2024.

The motivation for this research stems from the increasing importance of robust security measures in the digital age. The Vesper Protocol aims to address these concerns by providing a thorough analysis of current security practices and proposing innovative solutions to enhance cybersecurity.

I would like to express my deepest gratitude to my supervisors, Dr. Kaitai Liang and Huanhuan Chen, for their guidance and support throughout this research. Their expertise and insights have been instrumental in shaping the direction and outcomes of this study. I would also like to extend my sincere thanks to Zeshun Shi, whose advice and support were always available when needed. Although he was not an officially assigned supervisor, his assistance was greatly appreciated and made a significant difference.

Additionally, I would like to thank my mother, Cécille Franschman, for her unwavering support and encouragement during this challenging yet rewarding journey. Her belief in my abilities has been a constant source of motivation. I am also grateful for her feedback and grammar checks on this paper.

Lesley Franschman
Delft, September 2024

Abstract

This work explores the feasibility of combining zero-knowledge proofs with SGX enclave protection technology, using the Hyperledger fabric, as the testing environment. The focus is on assessing the viability of this combination in real-world scenarios where post-quantum security is crucial. To this end, a new zero-knowledge proof, called Vesper, has been developed. This is a lattice-based zk-SNARK with a Regev commitment scheme. Vesper aims to provide a novel approach to developing lattice-based zero-knowledge proofs suitable for blockchain environments.

Vesper features a proof size of 128 bytes, an average verification time of 0.14 ms, requires no trusted setup, while achieving at least 128-bit and 256-bit security. Only the proof geberation time increases when the LWE dimension increases.

To analyze and explore the potential of combining Intel SGX with such a ZKP and Hyperledger Fabric, two projects have been developed:

1. `Vesper Smart Contract`: This project uses Vesper in combination with a lattice-based digital signing scheme that has been NIST-approved. The proof generation benefits from SGX enclave protection, while the verifier for Vesper is deployed in a permissioned blockchain as a smart contract. Middleware using Fabric Peer command line interface (CLI) tools facilitates communication between the prover and verifier sides.

2. `Vesper-FPC`: This project explores the feasibility of protecting Hyperledger Fabric chaincode with an SGX enclave. It combines Vesper with a simplified lattice-based digital signing function to accommodate a restricted environment. Both the proof generation and the deployed verifier are SGX-protected. Communication is managed via custom peer CLI commands specifically designed to handle the interaction between the additional SGX protection layer and the blockchain infrastructure.

The simplified digital signing scheme of Vesper-FPC has a fast average verification time of 1.46 ms and an even faster average signing time of 0.49 ms while achiving 128-bit security. This is quicker than the NIST-approved digital signing scheme of Vesper Smart Contract, which has average signing and verification times of approximately 76.52 ms and 13.25 ms, respectively. The simplified version features a private key size of 512 bytes, a signature size of 768 bytes, and a public key size of 48 KB. In contrast, the digital signing scheme of Vesper Smart Contract has a private key size of 2528 bytes, a signature size of 2420 bytes, and a public key size of 1312 bytes.

The execution time of Vesper Smart Contract without SGX is on average approximately 2086.83 ms for a 128-bit security level. Adding SGX protection on the prover side increased the average execution time to 26339 ms. Vesper-FPC, with SGX protection for both the prover and the deployed verifier, has an average execution time of approximately 51229 ms.

Contents

1 Introduction **4**

2 Preliminaries **5**
2.1 Zero-Knowledge Proof . . . 5
2.1.1 The Simulation Paradigm . . . 5
2.1.2 Interactive Zero-Knowledge Proofs . . . 6
2.1.3 Non-Interactive Zero-Knowledge Proofs . . . 8
2.1.4 Fiat-Shamir Heuristic . . . 9
2.2 Lattice Cryptography . . . 10
2.2.1 Shortest Vector Problem . . . 11
2.2.2 (Module) Short Integer Solution . . . 12
2.2.3 Learning With Errors . . . 13
2.2.4 Ring-LWE . . . 13
2.2.5 Modular-LWE . . . 14
2.3 Fully Homomorphic Encryption (FHE) . . . 14
2.3.1 Addition and Multiplication in FHE . . . 15
2.3.2 Bootstrapping . . . 16
2.3.3 An Regev's Encryption Security . . . 16
2.3.4 Applications of FHE . . . 17
2.3.5 Challenges and Future Directions . . . 17

3 Previous Work **19**
3.1 zk-SNARKs . . . 19
3.1.1 Sonic . . . 19
3.1.2 Fractal . . . 19
3.1.3 Halo . . . 20
3.1.4 SuperSonic . . . 20
3.1.5 Marlin . . . 20
3.1.6 Plonk . . . 20
3.1.7 zk-STARKs by StarkWare . . . 20
3.1.8 Understanding the Mathematical Basis of These Frameworks . . . 20
3.2 Groth16 . . . 21
3.2.1 Constructing Groth16 . . . 21
3.2.2 The Correctness . . . 27
3.2.3 Other Strengths and Drawbacks . . . 27
3.3 Lattice-Based Zero-Knowledge Proofs: Where It Started . . . 27
3.4 Lattice-Based Zero-Knowledge Proofs and Applications: Shorter, Simpler, and More General . . . 28
3.5 Lattice-Based Zero-Knowledge Proofs Under a Few Dozen Kilobytes . . . 29
3.6 OpenFHE . . . 32
3.7 Blockchain and Trusted Computing: Problems, Pitfalls, and a Solution for Hyperledger Fabric . . . 33
3.7.1 Proposed Architecture and System Initialization . . . 33
3.7.2 Chaincode Enclave Bootstrapping and Execution . . . 34
3.7.3 Security . . . 35
3.7.4 Chaincode verification and data confidentiality . . . 35

3.7.5 performance . . . 37

4 Vesper: A New zk-SNARK **39**
4.1 Flattening the Base Computation and R1CS Transform . . . 39
4.2 QAP Transformation . . . 39
4.3 Proof Generation . . . 42
4.3.1 Regev's Encryption Scheme . . . 43
4.4 Modular-LWE Hardness Assumption . . . 45
4.5 Parameter Selection . . . 46
4.6 Digital Signing . . . 48
4.6.1 Crystal-Dilithium . . . 48
4.7 Verification . . . 50
4.7.1 Dilithium Verification . . . 50

5 The Vesper Smart Contract and its Components **51**
5.1 Intel SGX . . . 51
5.1.1 SGX Usage and Experience . . . 53
5.2 Gramine . . . 54
5.2.1 Gramine Development . . . 55
5.2.2 Gramine: User Experience and Troubleshooting Challenges . . . 56
5.3 Hyperledger Fabric . . . 56
5.3.1 Hyperledger Fabric Performance Analyses . . . 59
5.3.2 Case I Results: The Impact of Transaction Rates . . . 60
5.3.3 Case II Results: The Impact of the Number of Transactions . . . 60
5.3.4 Case III Results: The Impact of Simultaneous Transactions . . . 61
5.3.5 Improving Performance Through Peers . . . 61
5.4 Deploying the Smart Contract . . . 62
5.4.1 Troubleshooting Deployment Issues . . . 63
5.5 Communicating with deployed chaincode . . . 63
5.5.1 The Vesper Communication Middleware . . . 64

6 Vesper-FPC **66**
6.1 The Vesper-FPC Development Process and Troubleshooting Encountered Issues . . . 67
6.2 The Digital Signing Replacement Function . . . 68
6.3 Communicating with FPC: A New Gateway . . . 70
6.4 Docker . . . 70

7 Results and Analyses **72**
7.1 Test Bed . . . 72
7.2 Dataset . . . 73
7.3 Performance Metrics . . . 73
7.4 Experimental Setup . . . 73
7.5 Standalone Results . . . 74
7.6 Digital Signing Results . . . 74
7.7 Full Pipeline Results with and without SGX . . . 74
7.7.1 performance under concurrent transactions . . . 75
7.8 Analysis . . . 76
7.8.1 Vesper . . . 76
7.8.2 Vesper-FPC Signing . . . 77
7.8.3 SGX Influence . . . 77

8 Discussion **78**

9 Conclusion **80**

10 Future Work **81**

11 Reflection **82**

Appendix A: LWE to R1CS Transformation Pseudocode 92
Appendix B: R1CS to QAP Transformation Pseudocode 93
Appendix C: Vesper Proof Payload Generation Pseudocode 95
Appendix D: LWE-hardness Estimator based on Parameter Input 96
Appendix E: Vesper Verification Pseudocode 97
Appendix F: Vesper Chaincode Pseudocode 98
Appendix G: Vesper Smart Contract Communication Middleware Pseudocode 99
Appendix H: Vesper-FPC Chaincode Pseudocode 101
Appendix I: Vesper-FPC Signing Pseudocode 104
Appendix J: Vesper-FPC Communication Middleware Pseudocode 105
Appendix K: Automatic Witness Generator Pseudocode 107

Chapter 1

Introduction

The emergence of quantum computing technology presents a formidable threat to many contemporary cryptographic methods. Public-key cryptography, which is crucial for securing digital data, faces significant risks. These algorithms are integral to various global communication, processing, and storage systems. With the eventual development of practical quantum computers, public-key algorithms and their associated protocols will become vulnerable to exploitation by malicious actors, competitors, and other threats. It is imperative to start planning now to replace the hardware, software, and services dependent on public-key algorithms to ensure the continued security of information against future quantum attacks.

In response to this imminent challenge, a new non-interactive zero-knowledge proof (ZKP) with SGX enclave integration [CD16], called Vesper, has been developed. Vesper is designed to be optimal for blockchain environments by featuring a small proof size and fast verification time. The goal of this project was to create an innovative approach for developing lattice-based zero-knowledge proofs, specifically tailored for blockchain environments, and to explore the feasibility of efficiently integrating SGX with these proofs. Vesper aims to improve existing lattice-based ZKPs by reducing proof size while maintaining low verification times and eliminating the need for a trusted setup. This ZKP, combined with SGX protection, will be utilized to implement a lattice-based smart contract deployed on Hyperledger Fabric [ABB+18].

Blockchain technology was chosen as the testing ground for this new post-quantum security measure due to its reliance on public-key cryptography to ensure security and integrity. In blockchain systems, each participant utilizes a public-private key pair to create digital signatures, validate transactions, and maintain the ledger's integrity. Additionally, blockchain transactions are irreversible; once added to the blockchain, they cannot be undone. This characteristic implies that if the cryptographic methods securing these transactions are compromised, unauthorized transactions could occur and be irreversible. Furthermore, blockchain data is designed to be immutable and secure for extended periods, often decades. Thus, even though quantum computers are not yet available, data being added now could be susceptible to future quantum attacks.

To thoroughly explore the potential of SGX integration, two projects have been developed. The Vesper Smart Contract project investigates how to efficiently integrate SGX with a zero-knowledge proof before deploying the ZKP's verifier as a smart contract in Hyperledger Fabric [ABB+18]. The Vesper-FPC project focuses on the feasibility of protecting Hyperledger Fabric chaincode with SGX and explores the requirements for running complex operations like Vesper in such a restricted environment and chaincode interaction. In Vesper-FPC, both proof generation and the deployed chaincode are protected by an SGX enclave.

Through this master thesis, it is the aim to demonstrate the viability, effectiveness, and potential of a post-quantum smart contract combined with SGX protection. This work presents the first post-quantum secure permissioned smart contract that combines a lattice-based ZKP with SGX, thereby paving the way for further improvements in cryptographic practices in the face of quantum advancements. The results of this research may contribute to ongoing efforts to future-proof cryptographic systems and ensure the continued security and integrity of digital data in a quantum computing era.

Chapter 2

Preliminaries

This section provides the foundational background necessary for understanding the advanced topics discussed in this thesis. The focus here lies on zero-knowledge proofs (ZKPs) [BGS+23], lattice-based cryptography [CCKK15], and blockchain technology. These components are integral to the development of the post-quantum smart contract system presented in this work. By exploring these preliminaries, the essential frameworks that underpin the innovations and analyses in the subsequent sections are established.

2.1 Zero-Knowledge Proof

Zero-knowledge proofs (ZKPs) are cryptographic protocols that enable one party (the prover) to convince another party (the verifier) that a statement is true without revealing any information beyond the validity of the statement itself. This concept, introduced by Goldwasser, Micali, and Rackoff in the 1980s [GMR85], has profound implications for privacy and security in various digital applications, including authentication systems, blockchain technologies, and secure multi-party computations.

A proof is a zero-knowledge proof if it satisfies the following properties:

- **Completeness:** If the statement is true, an honest verifier will be convinced by an honest prover. This ensures that the proof correctly validates the knowledge of the prover.
- **Soundness:** If the statement is false, no dishonest prover can convince an honest verifier that it is true, except with some negligible probability. This ensures the integrity of the proof process.
- **Zero-knowledge:** If the statement is true, the verifier learns nothing other than the fact that the statement is true. This means the verifier gains no additional knowledge about the information being proven [GMR85].

The zero-knowledge property is critical in maintaining the privacy of the prover's information, preventing any leakage of sensitive data. To rigorously define and prove this property, the Simulation Paradigm is employed [Dou10]. This approach involves demonstrating that whatever the verifier can compute after interacting with the prover, they could have computed independently with the help of a simulator.

2.1.1 The Simulation Paradigm

The simulation paradigm is a fundamental concept in cryptography used to demonstrate the security of zero-knowledge proofs (ZKPs). It shows that whatever an adversary can achieve by interacting with the actual protocol, it can also achieve by interacting with a simulator in an ideal setting. This means that the adversary gains no additional information from the real protocol compared to what it would learn from the ideal scenario, thereby ensuring the protocol's security [Dou10, Lin21].

The key elements of the simulation paradigm are:

- **View of the Verifier** ($\text{view}_{P,V^*}[x]$): The actual transcript of messages exchanged between the prover and the verifier during the proof.

- **Simulator** ($\text{Sim}(x, 1^\lambda)$): A polynomial-time algorithm that generates a transcript that is indistinguishable from a real interaction transcript, without knowing the prover's secret information.

The essence of the simulation paradigm is to show that the verifier's view of the interaction can be simulated without access to the prover's secret information. This implies that the verifier gains no knowledge beyond the truth of the statement being proven. Specifically, for a proof to be zero-knowledge, there must be indistinguishability between the transcript generated by the simulator and the transcript produced by an actual interaction with the prover. This ensures that the proof does not reveal any additional information.

Computational Indistinguishability: Two probability distributions D_1 and D_2 are computationally indistinguishable if for any probabilistic polynomial-time distinguisher $\mathcal{D}$, the probability that $\mathcal{D}$ can distinguish between D_1 and D_2 is at most negligibly better than guessing. Formally this can be written as:

$$\forall \mathcal{D}, |\Pr[\mathcal{D}(D_1) = 1] - \Pr[\mathcal{D}(D_2) = 1]| < \text{negl}(k)$$

where $\text{negl}(k)$ is a negligible function in the security parameter k. This concept is crucial for defining zero-knowledge proofs.

Zero-Knowledge Proof: An interactive protocol (P, V) for a language L is zero-knowledge if for every probabilistic polynomial-time verifier V^* (including potentially malicious verifiers), there exists an expected polynomial-time simulator algorithm Sim such that for every $x \in L$, the following two distributions are computationally indistinguishable:

$$\text{view}_{P,V^*}[x] \approx \text{Sim}(x, 1^\lambda)$$

where $\text{view}_{P,V^*}[x]$ is the real view of the verifier after interacting with the prover, and $\text{Sim}(x, 1^\lambda)$ is the output of the simulator given x and the security parameter λ [GMR85, BGS+23].

The core idea of zero-knowledge proofs is that the verifier's view of the interaction does not provide any additional information other than the validity of the statement. This is achieved by showing that a simulator can produce a view indistinguishable from the actual interaction, thereby ensuring that the proof does not leak any secret information.

There are two main classes of zero-knowledge proofs: Interactive Zero-Knowledge Proofs and Non-Interactive Zero-Knowledge Proofs.

2.1.2 Interactive Zero-Knowledge Proofs

Interactive Zero-Knowledge Proofs (IZKPs) are a powerful cryptographic technique that allows a prover to convince a verifier that a statement is true without revealing any information beyond the validity of the statement [GMR85]. This process involves both interaction and randomness, which are essential for ensuring the security and effectiveness of the proof.

In IZKPs, the verifier actively engages with the prover through a series of non-trivial interactions, as visualized in Figure 2.1. This interaction ensures that the verifier cannot be easily fooled by a dishonest prover.

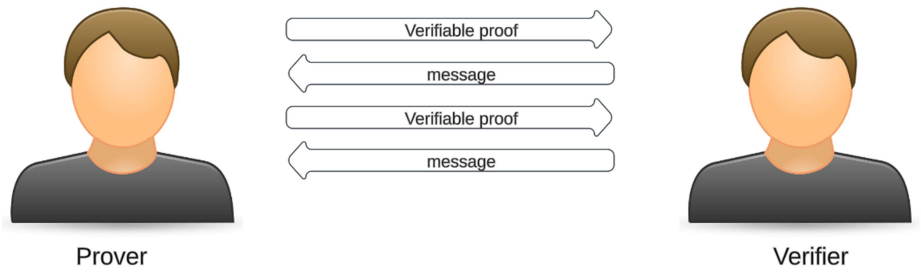

Figure 2.1: The back-and-forth communication in interactive zero-knowledge proofs.

To further enhance security, the verifier in an interactive zero-knowledge proof uses randomness. Instead of following a deterministic algorithm, the verifier performs random operations, such as coin

tosses, to decide the questions it will ask the prover. This randomness introduces a probability of error, but this probability can be made arbitrarily small through repetition.

Example

Consider a simple example to illustrate how IZKPs work. Suppose we have a page with each side being a different color. We want to prove to a blind verifier that the colors on each side of the page are different without revealing the actual colors.

The process would work as follows:

- **Random Decision**: The verifier performs a coin toss to decide whether to flip the page or not. This coin toss introduces randomness into the process.
- **Interaction**: The verifier sends the page to the prover. Since the prover can distinguish the colors, it can determine if the page was flipped or not.
- **Response**: The prover sends its guess (heads or tails) based on its observation of the colors.
- **Verification**: The verifier checks the prover's guess. If the guess is correct, it increases the verifier's confidence that the colors are indeed different.

If there were only one color, the prover would have a 50% chance of guessing correctly by random chance. However, if this process is repeated k times, the probability of the prover consistently guessing correctly by chance becomes $\frac{1}{2^k}$. As k increases, this probability becomes exceedingly small, making it highly unlikely for a dishonest prover to succeed.

After several rounds of interaction, if the verifier consistently receives correct responses from the prover, it becomes highly confident that the claim is true. The interaction between the prover and verifier produces a transcript, which includes the sequence of messages exchanged and the outcomes of the verifier's coin tosses. This transcript, combined with the randomness introduced by the verifier, ensures that the proof is sound, complete, and zero-knowledge. A visualization of this example can be found in Figure 2.2.

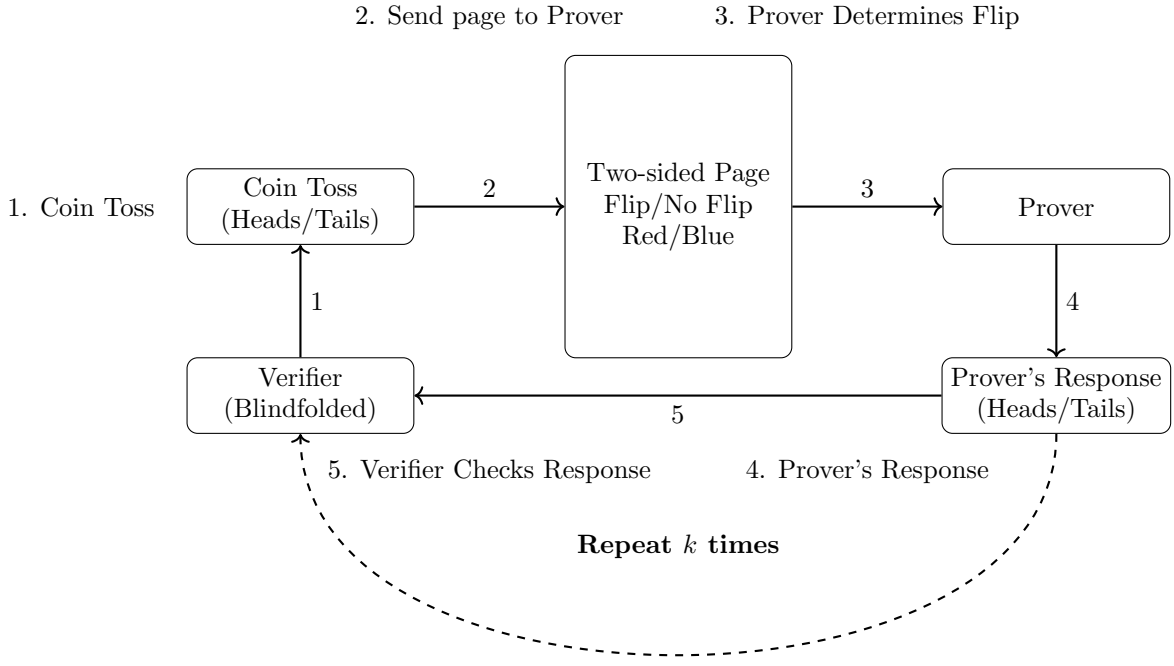

Figure 2.2: The color example illustrating interactive zero-knowledge proofs (IZKPs).

To demonstrate that an interactive proof is indeed zero-knowledge, we need to show that the verifier learns nothing beyond the validity of the statement being proved [GMR85]. This is typically done using the simulation paradigm [Dou10, Lin21]. A proof is zero-knowledge if, for every verifier, even a dishonest one, there exists a simulator that can produce a transcript indistinguishable from one generated during an actual interaction between the verifier and the prover. The simulator does this without access to the witness. The witness is the actual information that proves the statement. Such a simulator would take the following steps [Lin21]:

1. **Simulating the Verifier's View**: The simulator needs to generate a view that includes the messages exchanged between the prover and verifier and the outcomes of the verifier's coin tosses.
2. **Coin Toss Simulation**: The simulator can simulate the coin tosses made by the verifier. Since the coin tosses are random, the simulator can generate a sequence of coin tosses (heads or tails) that matches what the verifier would have done.
3. **Simulating Prover's Responses**: The simulator generates responses to the simulated coin tosses. Since the verifier cannot distinguish the actual colors, the simulator can generate responses (heads or tails) that appear as if they were given by the prover who knows the colors.
4. **Indistinguishability**: The key point is that the simulated transcript should be indistinguishable from an actual transcript. Since the verifier only sees the sequence of heads or tails guesses corresponding to coin flips, and since these can be simulated accurately without knowing the actual colors, the verifier cannot distinguish between the simulated interaction and a real interaction.

By constructing a simulator that can produce such indistinguishable transcripts, we can demonstrate that the verifier learns nothing more than the fact that the colors on the two sides of the page are different. This fulfills the zero-knowledge property.

The interaction between the prover (P) and verifier (V) can be represented as a probability distribution over the possible views of the interaction [Lin21]. This view, denoted as View(P, V), includes all the messages exchanged and the outcomes of the coin tosses. Because both the prover and verifier can toss coins, the view encompasses the entire probability space of their interactions and ensures that the proof remains secure and reliable.

2.1.3 Non-Interactive Zero-Knowledge Proofs

Non-Interactive Zero-Knowledge Proofs (NIZKPs) are a class of cryptographic proofs where a prover can convince a verifier of the truth of a statement without any interaction between them after the initial setup phase. These proofs are essential in scenarios where interaction is impractical or impossible, such as in blockchain and other distributed systems. In this section, the purpose, structure, and trade-offs of NIZKPs are discussed. Figure 2.3 depicts the typical communication in NIZKPs.

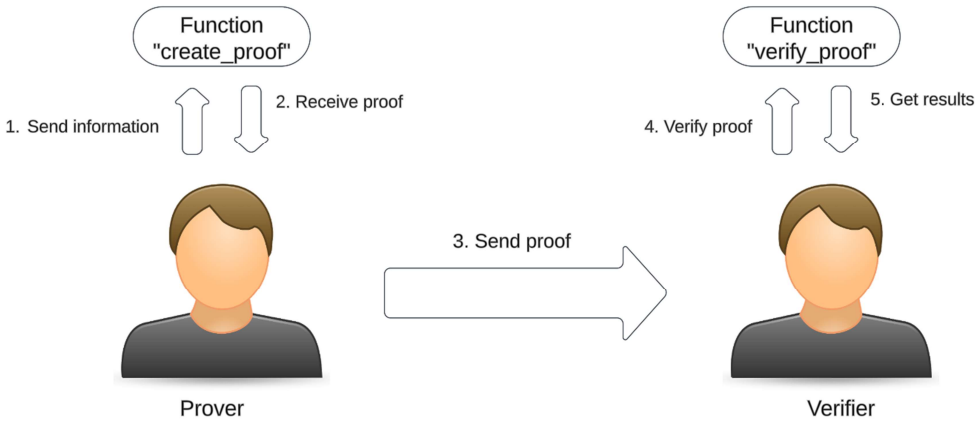

Figure 2.3: The communication in Non-Interactive Zero-Knowledge Proofs.

NIZKPs provide the same guarantees as interactive zero-knowledge proofs but without the need for multiple rounds of communication between the prover and the verifier. This makes them particularly

useful for applications where latency and communication overhead need to be minimized [BDSMP91]. Examples include:

- **Blockchain and Cryptocurrencies**: NIZKPs enable efficient and secure transactions without requiring parties to interact in real-time.
- **Privacy-Preserving Protocols**: NIZKPs allow users to prove they possess certain information without revealing the information itself, enhancing privacy and security.
- **Authentication Systems**: NIZKPs can be used to prove identity or other credentials without disclosing sensitive data.

NIZKPs typically involve the following three phases:

1. **Setup Phase**: In many NIZK systems, both the prover and verifier have access to a common reference string (CRS), which is a random string of data generated before the protocol begins. The CRS is assumed to be generated in a trusted manner [Gro16].
2. **Proof Generation**: The prover generates a proof using the CRS and the statement they wish to prove. This proof encapsulates all the information needed to convince the verifier.
3. **Verification**: The verifier uses the CRS to check the validity of the proof. If the proof is correct, the verifier is convinced of the statement's truth without learning any additional information.

A significant concern in NIZKPs is the handling of the CRS. If a malicious user gains access to the CRS, they could compromise the security of the entire system.For this reason, the CRS is often referred to as 'toxic waste'. Specifically, mishandling the CRS could allow a malicious actor to create fraudulent proofs that appear legitimate, thereby undermining the integrity of the proof system [WP22].

To mitigate the risks associated with the CRS, several approaches can be employed. One such method is Multi-Party Computation (MPC), like Groth16 [Gro16], where multiple parties contribute to the setup process. This makes it infeasible for any single participant to subvert the system without collusion. Another approach exploits homomorphic encryption [BAB+22], which allows setup values to be validated without being shared, thus enhancing security.

2.1.4 Fiat-Shamir Heuristic

The Fiat-Shamir heuristic, developed by Amos Fiat and Adi Shamir [FS86a], is a pivotal transformation in the realm of cryptographic protocols, providing a method to convert interactive zero-knowledge proof systems into non-interactive ones (see Figure 2.4).

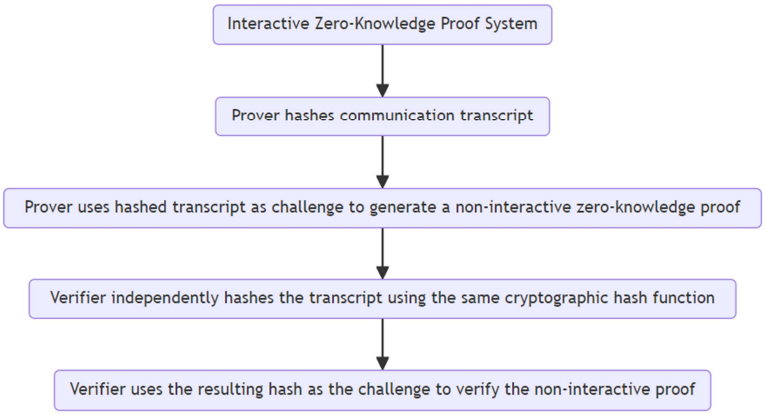

Figure 2.4: The Fiat Shamir heuristic.

This transformation greatly enhances the practicality and efficiency of zero-knowledge proofs by eliminating the need for multiple rounds of interaction between the prover and the verifier. The

heuristic relies on the use of cryptographic hash functions to simulate the randomness required in the interaction.

The process begins with an interactive zero-knowledge proof system [BDSMP91]. In the Fiat-Shamir heuristic, the verifier's random challenges are replaced with the output of a cryptographic hash function [FS86a]. Specifically, the hash function is applied to the transcript of the communication between the prover and the verifier. This transcript includes all the messages exchanged during the interactive proof. By hashing this transcript, a deterministic challenge is generated, which serves as a stand-in for the verifier's random challenge. This approach leverages the hash function's pseudo-randomness to simulate the interactive process in a non-interactive manner [FS86a].

The prover uses the hashed transcript as the challenge to generate a non-interactive zero-knowledge proof. This transformation is significant because it allows the prover to construct a proof independently, without requiring real-time interaction with the verifier. The prover essentially precomputes responses to the simulated challenges derived from the hash function, packaging them into a single proof.

For verification, the verifier independently hashes the transcript using the same cryptographic hash function. The verifier then uses the resulting hash as the challenge to verify the non-interactive proof. This verification process ensures that the proof is consistent with the responses the prover would have given during an interactive session. By matching the hash-derived challenge with the prover's responses, the verifier can be confident in the proof's validity while maintaining the non-interactive nature of the process.

The Fiat-Shamir heuristic has found widespread application in various cryptographic protocols, including digital signatures, identification schemes, and more. For example, the heuristic is utilized in the Goldwasser-Micali-Rivest (GMR) digital signature scheme, which is designed to be secure against adaptive chosen-message attacks. This scheme transforms interactive identification protocols into non-interactive ones using the Fiat-Shamir heuristic, thereby enhancing their practical applicability in digital signatures [GMR88].

Another notable application is in the Boneh-Franklin short signature scheme[BF01a], which leverages the Weil Pairing [Wei40]. This scheme uses the Fiat-Shamir transformation to produce short, efficient signatures that maintain strong security properties, making them suitable for a variety of cryptographic applications [BF01b].

These examples illustrate the versatility and importance of the Fiat-Shamir heuristic in modern cryptographic systems[FS86a], enhancing both security and efficiency in various protocols. By converting interactive proofs into non-interactive ones, the Fiat-Shamir heuristic enhances the scalability and deployability of cryptographic protocols, particularly in decentralized and distributed systems such as blockchain technologies. By leveraging cryptographic hash functions to simulate verifier challenges, this transformation maintains the security and zero-knowledge properties of the original interactive proofs while significantly enhancing their efficiency and applicability. The security of the resulting non-interactive proofs is contingent on the robustness of the underlying hash functions and their resistance to cryptographic attacks.

2.2 Lattice Cryptography

Lattice cryptography is an emerging field that leverages the mathematical structure of lattices to build secure cryptographic systems. A lattice is a structured arrangement of points in space, often with integer coordinates, exhibiting regularity and periodicity. Formally, a (complete) Euclidean lattice is a rank-n Z-module in the n-dimensional Euclidean space R^n spanned by a basis $B = (\mathbf{b}_1, \ldots, \mathbf{b}_n) \in R^{n \times n}$ with linearly independent vectors over R. Thus, the lattice consists of all integer linear combinations $a_1\mathbf{b}_1 + \cdots + a_n\mathbf{b}_n \in R^n$ for $a_1, \ldots, a_n \in Z$ [CCKK15].

The security of lattice-based cryptography is rooted in the hardness of certain lattice problems, notably the challenge of finding points in a Euclidean lattice that are close to zero or some other point. These problems, while tractable in low dimensions, become infeasible in higher dimensions[LNP22a]. In a cryptographic context, the private key can be represented as a set of lattice points, and the public key as another set of points that are further away. Deducing the private key from the public key would require a brute-force search of all possibilities, which remains computationally infeasible even with potential quantum computing speedups [ACL+22]. Consequently, lattice-based cryptography is considered resistant to quantum attacks.

Lattice Example

Consider $n = 2$ and two basis vectors $\mathbf{b}_1 = \begin{bmatrix} 4 \\ 2 \end{bmatrix}$ and $\mathbf{b}_2 = \begin{bmatrix} -3 \\ 4 \end{bmatrix}$. The lattice formed by these vectors consists of the set of points:

$$\{z_1\mathbf{b}_1 + z_2\mathbf{b}_2 \mid z_1, z_2 \in Z\}$$

Some points in this lattice include:

$$(z_1, z_2) = (0, 0) \rightarrow \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad (z_1, z_2) = (1, 0) \quad \rightarrow \begin{bmatrix} 4 \\ 2 \end{bmatrix}$$
$$(z_1, z_2) = (0, 1) \rightarrow \begin{bmatrix} -3 \\ 4 \end{bmatrix} \quad (z_1, z_2) = (1, 1) \quad \rightarrow \begin{bmatrix} 1 \\ 6 \end{bmatrix}$$

This can be visualized in Figure 2.5:

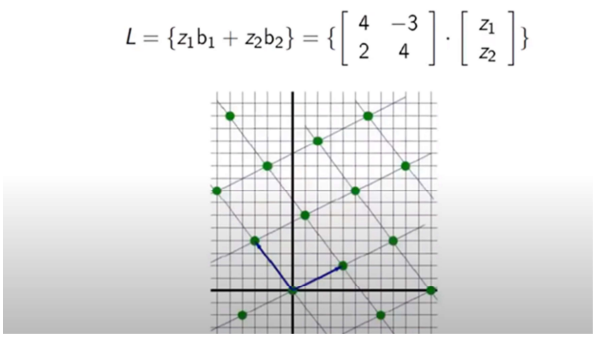

Figure 2.5: Simple lattice visualization in 2D space.

Understanding the basic construction of lattices is crucial for appreciating more complex lattice-based cryptographic systems [LNP22a]. Lattice cryptography is not only theoretically interesting but also practically relevant, offering strong security guarantees even in the presence of quantum computing advancements.

2.2.1 Shortest Vector Problem

The Shortest Vector Problem (SVP) is one of the fundamental problems underpinning the security of lattice-based cryptography. SVP involves finding the shortest non-zero vector in a given lattice [CFT18]. This problem is believed to be hard, especially in high-dimensional spaces, providing a strong security foundation for cryptographic systems.

Formally, let L be a lattice generated by the basis $B = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n\}$, where $\mathbf{b}_i \in R^n$. The lattice L consists of all integer linear combinations of the basis vectors:

$$L = \left\{ \sum_{i=1}^{n} z_i\mathbf{b}_i \mid z_i \in Z \right\}$$

The SVP is the problem of finding a non-zero vector $\mathbf{v} \in L$ such that:

$$\|\mathbf{v}\| = \min_{\mathbf{w} \in L \setminus \{\mathbf{0}\}} \|\mathbf{w}\|$$

where $\|\cdot\|$ denotes the Euclidean norm.

In a typical lattice-based cryptographic scheme, a user generates a public-private key pair. The public key is associated with a lattice, which is a set of points in a multi-dimensional space, while the private key is a "short vector" within this lattice. The public key serves as a lattice point (or set of points/vectors) linked to the lattice structure. The crux of the security lies in the challenge an attacker faces in finding the short vector (private key) within the lattice, given the public key [CFT18].

A brute-force search to solve SVP would require trying all possible vectors in the lattice to find the one matching the public key. However, due to the high dimensionality of the lattice and the exponential

growth in the number of lattice points, such a search becomes impractical. [CFT18] Specifically, for an n-dimensional lattice with a basis containing vectors of length at most B, the number of lattice points can be on the order of B^n. Therefore, the time complexity of a brute-force approach to find the shortest vector in a lattice can be expressed as $O(B^n)$, where B is related to the lattice density and n is the dimensionality of the lattice. Even with quantum computing capabilities, solving these problems within a reasonable timeframe is deemed infeasible, thus providing post-quantum security.

Example of SVP in 2-dimensional space

Consider the basis vectors:

$$\mathbf{b}_1 = \begin{bmatrix} 4 \\ 2 \end{bmatrix}, \quad \mathbf{b}_2 = \begin{bmatrix} -3 \\ 4 \end{bmatrix}$$

The lattice L generated by these basis vectors consists of all integer linear combinations of $\mathbf{b}_1$ and $\mathbf{b}_2$:

$$L = \left\{ z_1 \begin{bmatrix} 4 \\ 2 \end{bmatrix} + z_2 \begin{bmatrix} -3 \\ 4 \end{bmatrix} \mid z_1, z_2 \in Z \right\}$$

The goal is to find the shortest non-zero vector in this lattice. By examining the integer combinations of these vectors, we can find that the shortest vector in this example is likely to be a vector close to the origin, depending on the combinations used.

Understanding the Shortest Vector Problem and its complexity is crucial for appreciating the security guarantees provided by lattice-based cryptographic schemes. The hardness of SVP in high dimensions forms the backbone of the security in these systems, making them resilient against both classical and quantum attacks.

2.2.2 (Module) Short Integer Solution

The ModuleSIS commitment scheme [Ajt96b, BBC+18, Ngu22], first introduced in the seminal work by Ajtai [Ajt96b], is described here. To provide context, the Short Integer Solution (SIS) problem is first explained.

The Short Integer Solution (SIS) problem is defined over integer lattices [DEP23]. Given a uniformly random matrix $A \in Z_q^{n \times m}$, the goal is to find a non-zero vector $\mathbf{z} \in Z^m$ such that $A\mathbf{z} = \mathbf{0} \mod q$ and $\|\mathbf{z}\| \leq \beta$. The SIS problem is foundational in lattice-based cryptography and is used to construct various cryptographic primitives.

The Module Short Integer Solution (ModuleSIS) problem generalizes the SIS problem to module lattices. Instead of working over the integer ring Z, it operates over the ring of polynomials modulo some integer, typically $Z_q[x]/(f(x))$, where $f(x)$ is a polynomial of degree d. Given a uniformly random matrix $A \in (Z_q[x]/(f(x)))^{n \times m}$, the goal is to find a non-zero vector $\mathbf{z} \in (Z_q[x]/(f(x)))^m$ such that $A\mathbf{z} = \mathbf{0} \mod q$ and $\|\mathbf{z}\| \leq \beta$. The ModuleSIS problem, introduced by Ajtai [Ajt96b], benefits from the structure of module lattices, providing a balance between efficiency and security, and is particularly useful in practical lattice-based cryptographic schemes.

To implement the ModuleSIS commitment scheme, two matrices A_1 and A_2 are utilized, which are drawn uniformly at random from $\mathcal{R}_q^{\kappa_{MSIS} \times m_1}$ and $\mathcal{R}_q^{\kappa_{MSIS} \times (p\kappa_{MSIS} + \kappa_{MLWE})}$ respectively. Here, m_1 represents the number of elements to be committed [Ajt96b, Ngu22].

The commitment process involves taking a vector $\mathbf{s}_1$ and generating a random vector $\mathbf{s}_2$ with small coefficients. The commitment is then computed as:

$$\mathbf{t} = A_1\mathbf{s}_1 + A_2\mathbf{s}_2 \in \mathcal{R}_q^{\kappa_{MSIS}}.$$

The hiding property of this commitment is ensured because the vector $\mathbf{s}_2$ is much larger than the dimensions of A_2, making $A_2\mathbf{s}_2$ indistinguishable from a random vector under the Module-LWE assumption.

For the binding property, if an adversary can find two distinct pairs $(\mathbf{s}_1, \mathbf{s}_2)$ and $(\mathbf{s}'_1, \mathbf{s}'_2)$ that produce the same commitment $\mathbf{t}$, then a Module-SIS solution can be derived:

$$\begin{pmatrix} \mathbf{s}_1 - \mathbf{s}'_1 \\ \mathbf{s}_2 - \mathbf{s}'_2 \end{pmatrix}$$

for the matrix $[A_1\ A_2]$. Therefore, under the Module-SIS assumption, the commitment remains binding as long as the coefficients of the message vectors are small.

Additionally, the compact nature of the commitment scheme is evident as the size of the commitment does not directly depend on the length of the vector m_1.

2.2.3 Learning With Errors

The Learning With Errors (LWE) problem is a cornerstone of lattice-based cryptography. It involves generating a set of linear equations that include a secret vector and perturbing these equations with random errors [Reg10]. The objective is to recover the secret vector from the noisy equations, a task that is computationally challenging and forms the basis for secure cryptographic schemes [Reg10, Reg09].

In LWE-based cryptographic systems, a user generates a public-private key pair. The public key is derived by combining the secret vector with a set of random vectors and adding a noise term, resulting in a lattice point [Reg09, Bal21]. The security of these systems hinges on the difficulty of the LWE problem. An attacker would need to recover the secret vector from the noisy public key, which is computationally infeasible due to the high dimensionality and the added noise. A formal depiction of LWE can be found in Figure 2.6.

Given:

- Uniform $A \in Z_q^{k \times l}$
- "Noise distribution" χ
- Samples $A\mathbf{s} + \mathbf{e}$, with $\mathbf{e} \leftarrow \chi$

Search version: find $\mathbf{s}$

Decision version: distinguish LWE samples from uniform randoms

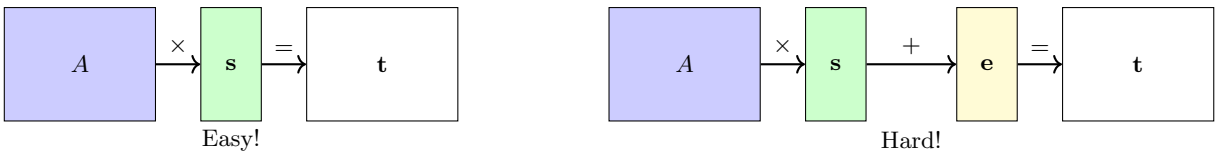

Figure 2.6: Illustrating the LWE problem.

Brute-forcing the LWE problem would entail trying all possible vectors to find the one that satisfies the system of noisy equations. The dimensionality of the lattice and the amount of noise determine the size of the search space, making it exponentially large and impractical to search exhaustively. The LWE problem, similar to SVP, involves finding a short vector in a lattice. However, in the case of LWE, the challenge is to recover the secret vector from noisy linear equations. LWE-based cryptography is also considered post-quantum, as quantum algorithms, like Shor's algorithm, do not offer significant advantages in solving LWE problems compared to classical algorithms.

2.2.4 Ring-LWE

Ring-Learning with Errors (Ring-LWE) is an optimization of the Learning with Errors (LWE) problem that improves efficiency by leveraging the algebraic structure of polynomial rings [LPR13a].

In Ring-LWE, the matrix A used in LWE is constructed such that all columns are based on the first (leftmost) column. The subsequent columns are created by cyclically shifting this first column. For example, column 2 is a 1-position shift of the first column, column 3 is a 2-position shift, and so on.

This cyclic structure means that instead of storing all n columns of the matrix A, only the first column needs to be stored, reducing storage complexity to $O(n)$. Additionally, because the columns in Ring-LWE are not independent but related through cyclic shifts, the computational complexity of operations can be optimized. Specifically, while the computation in normal LWE is $O(n^2)$ due to the independence of columns, Ring-LWE allows certain operations to be performed in $O(n \log n)$ time

$$\begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{pmatrix}$$

Storage: $O(n^2)$

Computation: $O(n^2)$

$$\begin{pmatrix} a_0 & -a_{n-2} & -a_{n-3} & \cdots & -a_1 & -a_0 \\ a_1 & a_0 & -a_{n-2} & \cdots & -a_2 & -a_1 \\ a_2 & a_1 & a_0 & \cdots & -a_3 & -a_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n-2} & a_{n-3} & a_{n-4} & \cdots & a_0 & -a_{n-2} \\ a_{n-1} & a_{n-2} & a_{n-3} & \cdots & a_1 & a_0 \end{pmatrix}$$

Storage: $O(n)$

Computation: $O(n \log n)$

Figure 2.7: Comparison of Matrix Storage and Computation Complexity between LWE and Ring-LWE.

using techniques such as the Fast Fourier Transform (FFT) [LPR13a, Bal21]. This difference can be seen in Figure 2.7.

2.2.5 Modular-LWE

Modular-Learning with Errors (Modular-LWE) builds upon the concepts of Ring-LWE by adding an extra layer of complexity [WW19]. This enhancement is designed to increase security, potentially making it as secure as, if not more secure than, Ring-LWE. However, this added security comes with a trade-off in efficiency.

In Modular-LWE, the matrix A is repeated k times in both rows and columns, increasing the dimensions and the complexity of the problem. This repetition and modular structure add redundancy and complexity, making brute-force attacks more difficult while slightly reducing computational efficiency compared to Ring-LWE, as can be seen in Figure 2.8. Despite this trade-off, the modular version remains a robust and secure cryptographic method, benefiting from the foundational strengths of Ring-LWE while aiming to enhance security further.

$$\begin{pmatrix} A_{0,0}(X) & \cdots & A_{0,k-1}(X) \\ \vdots & \ddots & \vdots \\ A_{k-1,0}(X) & \cdots & A_{k-1,k-1}(X) \end{pmatrix} \cdot \begin{pmatrix} s_0(X) \\ \vdots \\ s_{k-1}(X) \end{pmatrix} + \begin{pmatrix} e_0(X) \\ \vdots \\ e_{k-1}(X) \end{pmatrix} = \begin{pmatrix} t_0(X) \\ \vdots \\ t_{k-1}(X) \end{pmatrix}$$

Storage: $O(k^2 \cdot n)$

Computation: $O(k^2 \cdot n \log n)$

Figure 2.8: Matrix equation with storage and computation complexities of Modular-LWE.

Both Ring-LWE and Modular-LWE are considered post-quantum cryptographic techniques, providing resistance against potential quantum attacks and ensuring the long-term security of cryptographic systems in the advent of quantum computing.

2.3 Fully Homomorphic Encryption (FHE)

Traditional encryption allows for the secure storage of data on remote servers by ensuring that the data remains inaccessible without the proper decryption key. However, this security measure falls short when the need arises for the server to perform computations on the encrypted data without revealing the plaintext. Fully Homomorphic Encryption (FHE) addresses this limitation by enabling remote servers to compute functions on encrypted data and return the encrypted result to the client, all while maintaining the confidentiality of the data [LW23].

Formal Definition: An encryption scheme (KeyGen, Enc, Dec, Eval) is C-homomorphic for a class of circuits $C : \{0,1\}^n \to \{0,1\}$ if for any function $f \in C$ and ciphertexts $c_1, \ldots, c_n$:

$$\text{Eval}(f, c_1, \ldots, c_n) = c^* \text{ such that if } \text{Dec}(c_i) = m_i \text{ for all } c_i, \text{ then } \text{Dec}(c^*) = f(m_1, \ldots, m_n)$$

The scheme is fully homomorphic if it supports all polynomial-sized circuits C.

2.3.1 Addition and Multiplication in FHE

Homomorphic addition and multiplication are the fundamental operations that enable FHE to support arbitrary computations on encrypted data [CKKS17].

Addition

Consider a private key scheme based on Learning With Errors (LWE). Let the secret key $s \in Z_q^n$ and the message $m \in \{0,1\}$. Encryption is defined as:

$$\text{Enc}(s, m) = (a, \langle a, s \rangle + e + \left\lceil \frac{q}{2} \right\rceil m)$$

where e is a small random error term.

For ciphertexts $c_1 = (a_1, b_1)$ and $c_2 = (a_2, b_2)$, addition is performed as follows:

$$c_1 + c_2 = (a_1 + a_2, b_1 + b_2) = (a_1 + a_2, \langle a_1 + a_2, s \rangle + (e_1 + e_2) + \left\lceil \frac{q}{2} \right\rceil (m_1 + m_2))$$

Decryption retrieves $m_1 + m_2$ if $|e_1 + e_2| \leq \frac{q}{4}$.

For the addition of $l = \text{poly}(n)$ ciphertexts, the total error is $\sum_{i=1}^{l} e_i$. Initial error $|e_i| \leq \frac{q}{4l}$ ensures correct decryption, assuming LWE hardness for weaker parameters.

Theorem 3 (Worst-Case to Average-Case Reduction for LWE)

$$\text{gapSVP}_{n, \frac{q}{\sigma}} \leq \text{LWE}_{n,m,q,\chi} = D_\sigma \text{LWE}_{n,m,q,\chi} = D_\sigma \text{LWE}_{n,m,q,\chi} = D_\sigma \text{LWE}_{n,m,q,\chi} = D_\sigma$$

where D_σ is a discrete Gaussian distribution and m affects the reduction's running time. The approximation factor depends on $\frac{q}{\sigma}$ [GINX16].

For efficient parameter selection, best known algorithms for $\text{gapSVP}_{n,2^{n^\epsilon}}$ take time $2^{\tilde{O}(n^{1+\epsilon})}$. Set $\frac{q}{\sigma} = 2^{n^{0.99}}$, with $q = n \log n$ and $\sigma = \text{poly}(n)$ for arbitrary additions.

Multiplication

The Approximate Eigenvector Encryption Scheme is used. Initially, it employs an approximation factor n^d (where d is the circuit's depth) as proposed by the work of Gentry, Sahai, and Waters [GSW13]. This was improved to n^3 by the work of Brakerski [Bra14].

Scheme Public Key $P \in Z_q^{n \times n}$ with $\text{nullity}(P) > 0$. Secret Key $s \in Z_q^n$ such that $sP \equiv 0 \mod q$.

Encryption is given by:

$$\text{Enc}(m) = PR + mI \text{ where } R \leftarrow \$Z_q^{n \times n}$$

Decryption is performed as:

$$\text{Dec}(C) = sC = s(PR + mI) = ms$$

This scheme is insecure due to the recoverability of s. It is modified to avoid s in $\text{null}(P)$, ensuring $sP \approx 0 \mod q$:

$$s = (s' \mid -1) \text{ with } s' \in Z_q^{n-1}$$
$$P = [P' \mid s'P' + e]$$

where e is an error term.

Encryption is performed as:

$$\text{Enc}(\mu) = PR + \mu I$$

Decryption is:

$$\text{Dec}(C) = sC = s(PR + \mu I) = -eR + \mu s$$

To further improve the scheme:
Increase LWE samples to $m = n(\log q + 1)$ and replace I with matrix G for error correction.

Approximate Eigenvector Encryption Scheme Secret Key $s = (s' \mid -1)$ with $s' \in Z_q^{n-1}$.
Public Key $P = [P' \mid s'P' + e]$ where e is the error term.
Encryption:

$$\text{Enc}(\mu) = PR + \mu G$$

Decryption:

$$\text{Dec}(C) = sC = s(PR + \mu G) = -eR + \mu sG$$

Matrix G is defined as:

$$G = \begin{bmatrix} 1 & 2 & \dots & 2^{\log q} & 0 & \dots & 0 & & & \\ 0 & \dots & 0 & 1 & 2 & \dots & 2^{\log q} & 0 & \dots & 0 \\ \vdots & & \vdots & & \vdots & & \vdots & & \vdots & \\ 0 & \dots & 0 & 1 & 2 & \dots & 2^{\log q} & & & \end{bmatrix}$$

Addition remains the same. For multiplication, use $C \cdot G^{-1}(C')$ or $C' \cdot G^{-1}(C)$:

$$s(C \cdot G^{-1}(C')) = eG^{-1}(C') + \mu sC' = eG^{-1}(C') + \mu e' + \mu\mu' sG$$

Error management is given by:

$$\|e_{mult}\| \leq m\|e\|_\infty + \mu\|e'\|_\infty$$

2.3.2 Bootstrapping

Bootstrapping is a crucial technique in FHE that allows for the computation of arbitrary-depth circuits by reducing accumulated error, a concept first proposed by Gentry in 2009 [Gen09]. The process involves decrypting encrypted inputs using an encrypted version of the secret key, thereby resetting the error growth and enabling further computations.

The bootstrapping process works as follows:

1. **Encrypted Decryption**: Decrypt the ciphertext using an encrypted version of the secret key $Es(s)$, which generates a new ciphertext with reduced error. 2. **Error Reduction**: Homomorphic decryption alters the error profile of the ciphertext, reducing it to a manageable level:

$$\text{final error} = n^{O(d_{DEC})}\text{poly}(n)$$

3. **Parameter Setting**: By setting $q > n^{O(d_{DEC})}\text{poly}(n)$, the scheme supports continued operations while keeping the error within acceptable bounds.

Repeated application of bootstrapping allows the evaluation of arbitrary circuits, achieving full homomorphism and making FHE practical for real-world applications.

2.3.3 An Regev's Encryption Security

Regev's encryption scheme [Reg09] is secure even when the encryption of the secret key is released:

$$(a, \langle a, s\rangle + e + s_i \left\lceil \frac{q}{2} \right\rfloor) = (a', \langle a', s\rangle + e)$$

where $a' = a + \left\lceil \frac{q}{2} \right\rfloor u_i$ and u_i is the i-th standard basis vector, ensuring indistinguishability from random.

2.3.4 Applications of FHE

Fully Homomorphic Encryption has a wide range of applications due to its ability to perform computations on encrypted data:

- **Secure Data Analysis**: Enables data analysis on sensitive information without exposing the underlying data, useful in healthcare, finance, and personal data privacy.
- **Cloud Computing**: Allows cloud servers to process encrypted data without needing access to the plaintext, enhancing security in cloud services.
- **Private Information Retrieval**: Supports retrieving information from databases without revealing the query to the database server.
- **Encrypted Search**: Facilitates searching within encrypted data, providing privacy-preserving search capabilities.

2.3.5 Challenges and Future Directions

While FHE provides powerful capabilities, there are still challenges and areas for improvement:

- **Performance Overhead**: FHE operations are computationally intensive and slower than traditional encryption methods. Optimizing these operations is an ongoing research area.
- **Key Management**: Managing and distributing keys securely remains a challenge, especially in distributed systems.
- **Scalability**: Ensuring that FHE schemes can scale to handle large datasets and complex computations efficiently is critical for practical deployment.
- **Standardization**: Developing standards for FHE to ensure compatibility and security across different implementations is necessary for widespread adoption.

Continued research and advancements in these areas will help to make Fully Homomorphic Encryption more practical and widely usable, enhancing security and privacy in various applications.

Blockchain technology is distinguished by several key characteristics that set it apart from traditional centralized systems. The foremost characteristic is `decentralization`. Rather than being controlled by a single entity, a blockchain is maintained by a distributed network of nodes, each of which holds a copy of the entire blockchain. This decentralization ensures that no single point of failure exists and enhances the system's robustness [Nak08a, Woo14a]. Another vital characteristic is `immutability`; once data is recorded in a block, altering it becomes nearly impossible without changing all subsequent blocks, a feat that would require consensus from the majority of the network. This property is underpinned by cryptographic hashing, where each block contains the hash of the previous block, a timestamp, and transaction data [NBF+16]. Additionally, blockchain ensures `transparency` as all transactions are visible to all participants in the network, fostering trust and accountability [TT16]. Security is also a fundamental aspect, with blockchain employing cryptographic techniques to secure data, ensuring the integrity and authenticity of transactions [Ant14].

The operational mechanics of a blockchain begin when a user requests a transaction. This request is broadcast to a peer-to-peer network of nodes. If the transaction involves a smart contract, it undergoes this process initially. The network of nodes then validates the transaction using consensus algorithms. Two prominent consensus mechanisms are Proof of Work (PoW) and Proof of Stake (PoS). In PoW, nodes, known as miners, compete to solve complex mathematical puzzles. The first to solve the puzzle earns the right to add the new block to the blockchain and is rewarded, as seen in Bitcoin. The mathematical puzzle typically involves finding a nonce such that the hash of the block's header is less than a specified target value:

$$H(\text{block header} + \text{nonce}) < \text{target}.$$

This process is computationally intensive, consuming significant energy [Nak08a, BMC+15]. Conversely, PoS selects validators based on the number of tokens they hold and are willing to "stake" as collateral. This approach reduces energy consumption and enhances scalability [KN12].

Once a transaction is validated, the smart contract, if applicable, is executed, and the transaction is added to a new block. This block is then appended to the blockchain, and the updated ledger is propagated across the network. Smart contracts are self-executing contracts with the terms of the agreement encoded directly into the blockchain. They automatically enforce and execute the contract when predefined conditions are met, ensuring reliability and eliminating the need for intermediaries [Woo14a].

Blockchain applications are diverse and impactful. In the realm of cryptocurrencies, blockchain technology powers Bitcoin and numerous other digital currencies, facilitating secure and transparent financial transactions. In supply chain management, blockchain enables the transparent and immutable tracking of the provenance and movement of goods, enhancing efficiency and reducing fraud. In finance, blockchain technology enables faster and cheaper cross-border payments, reduces fraud, and improves transaction security [CPVK16].

Despite its advantages, blockchain technology faces several challenges. Scalability remains a significant issue, as increasing transaction volumes can slow processing times and increase storage requirements [sca20]. Energy consumption, particularly for PoW blockchains like Bitcoin, is another concern due to the substantial energy required for mining activities [Fai17]. Furthermore, interoperability, the ability for different blockchain systems to communicate and work together, is crucial. Deploying secure smart contracts across various blockchain networks can enhance interoperability, but ensuring these contracts' security is paramount [HLP19].

Chapter 3

Previous Work

This section reviews existing research related to this thesis. Several key studies have served as sources of inspiration, shaping this work's conceptual framework and guiding the methodology. Additionally, comparative works are examined to benchmark our findings against established results.

3.1 zk-SNARKs

Developing a new Zero-Knowledge Proof (ZKP) for use in a blockchain environment necessitated a focus on small proof sizes and fast verification times. zk-SNARKs [Nit20] (Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge) were a primary source of inspiration due to their succinctness, which allows for short proofs and quick verification times. Being non-interactive, zk-SNARKs do not require back-and-forth communication between the prover and verifier.

Table 3.1 compares various zk-SNARKs, estimating bit security at 128 bits for most protocols, except zk-STARKs with 96-bit security and SuperSonic with 120-bit security. zk-STARKs also offer post-quantum security.

Groth16 stands out with the smallest proof size and fastest verification time, making it the primary inspiration for this work's ZKP, detailed in Chapter 3.2.

Protocol	**Proof Size**	**Verification Time**	**Prover Run-Time**	**Setup**
Sonic	$O(1)$ (constant, $\sim$ 1472 bytes)	$O(n \log n)$ (0.7 ms for small instances)	$O(n \log n)$	Trusted
Fractal	$O(\log n)$ (up to 250 kB)	$O(\log n)$	$O(n \log n)$	Trusted
Halo	$O(\log n)$ (around 3.5 KiB)	$O(\sqrt{n})$ (sublinear)	$O(n \log n)$	None
SuperSonic	$O(\log n)$ (around 10 kB)	$O(\log n)$ (under 100 ms)	$O(n \log n)$	None
Marlin	$O(\log n)$ (around 880 bytes)	$O(n)$	$O(n \log n)$	Trusted
Plonk	$O(\log^2 n)$	$O(\log^k n)$	$O(n \log n)$	Trusted (Universal, Updatable)
zk-STARKs	$O(\log^2 n)$ (varies)	$O(\log n)$	$O(n\text{polylog}(n))$	None
Groth16	$O(1)$ (constant, $\sim$ 192 bytes)	$O(1)$ (constant, $\sim$ 1-10 ms)	$O(n + m)$	Trusted

Table 3.1: Comparison of Zero-Knowledge Proof Protocols.

3.1.1 Sonic

Sonic [MBKM19] introduced a universal and updatable common reference string, allowing the same setup to be reused for multiple circuits. This reduces the complexity and risk associated with multiple trusted setups. Sonic achieves constant proof sizes and supports batch verification, which improves efficiency when verifying multiple proofs. However, individual verification is more resource-intensive compared to Groth16 [Gro16]. Sonic's flexibility and ability to update the reference string make it suitable for dynamic applications like smart contracts and evolving blockchain protocols.

3.1.2 Fractal

Fractal [COS19] employs a transparent setup, eliminating the need for a trusted setup. It allows recursion without pairing-friendly elliptic curves by preprocessing circuits, enabling succinct verification.

While current proof sizes are relatively large (up to 250 kB), ongoing optimizations aim to reduce these sizes. Fractal balances the need for transparency and scalability, making it ideal for applications requiring transparent setups and recursive proofs.

3.1.3 Halo

Halo [BGH19] leverages recursive proof composition and nested amortization to eliminate the need for a trusted setup. It supports transparent setups and recursive verifications over cycles of elliptic curves. Halo's verification time is linear, unlike other zk-SNARKs, which can limit its succinctness. However, its recursive nature allows for continuous improvements and optimizations, making it suited for scenarios requiring robust recursion and transparent setups.

3.1.4 SuperSonic

SuperSonic [BFS19] improves on Sonic with practical prover time and logarithmic proof size and verification time. It uses a transparent setup, making it versatile and secure. SuperSonic balances efficiency and proof size, achieving practical prover times while maintaining small proof sizes, typically around 10 KB. It is designed for real-world applications requiring both efficiency and security.

3.1.5 Marlin

Marlin [CHM+19] builds on Sonic [MBKM19], offering significant improvements in prover and verification times. It uses a universal setup that can be reused across different circuits, enhancing flexibility. With a tenfold improvement in prover time and a fourfold improvement in verification time, Marlin is highly efficient for generating and verifying proofs quickly. Marlin is well-suited for applications needing rapid proof generation and verification, making it ideal for dynamic and scalable blockchain solutions.

3.1.6 Plonk

Plonk [GWC19] refines Sonic's approach, focusing on reducing prover time significantly. It uses a universal setup and supports efficient, scalable proof generation. Plonk achieves a fivefold improvement in prover time, making it one of the fastest zk-SNARKs for generating proofs while maintaining practical verification times. Plonk is ideal for large-scale applications requiring fast and scalable proof generation.

3.1.7 zk-STARKs by StarkWare

StarkWare Industries is a pioneering company in the field of zero-knowledge proofs, particularly known for developing zk-STARKs (Zero-Knowledge Scalable Transparent Arguments of Knowledge) [BSBHR18, Sta23]. zk-STARKs use advanced cryptographic techniques to create zero-knowledge proofs that are both scalable and transparent. They avoid the need for a trusted setup by relying on cryptographic hash functions and polynomial commitments. zk-STARKs generate proofs that can be verified with minimal computational resources, making them highly efficient. While zk-STARK proofs are larger than some traditional zk-SNARK proofs, they remain practical for real-world applications. Proof sizes typically range from a few kilobytes to a few hundred kilobytes, depending on the complexity of the computation. StarkWare's solutions, such as StarkNet and StarkEx, leverage zk-STARK technology to provide scalable and private transactions on blockchain platforms, improving both throughput and security.

3.1.8 Understanding the Mathematical Basis of These Frameworks

To comprehend the complexities involved in proof size, verification time, and prover run-time for various zero-knowledge proof systems, we need to delve into the underlying cryptographic techniques and optimizations employed by each system.

Polynomial Commitments: A fundamental aspect of many zk-SNARKs, including Sonic [MBKM19], Marlin [CHM+19], and Plonk [GWC19], is the use of polynomial commitments. These commitments

allow a prover to commit to a polynomial and later reveal evaluations of this polynomial at specific points. The Fast Fourier Transform (FFT) is typically used to evaluate and interpolate these polynomials efficiently, resulting in a complexity of $O(n \log n)$.

Batch Verification: Techniques such as those used in Sonic [MBKM19] enable batch verification, where multiple proofs can be verified together efficiently. This reduces the per-proof verification complexity, especially when amortized over many proofs. This is particularly useful in applications where multiple proofs need to be validated simultaneously.

Recursive Proofs: Systems like Halo [BGH19] utilize recursive proofs, where a proof can attest to the correctness of another proof. This recursive nature, while adding to the proof size and verification time, allows for ongoing improvements through amortization. Recursive proofs enable the construction of complex proof systems that can verify large computations in a scalable manner.

Transparent Setups: zk-STARKs [Sta23] and Fractal [COS19] avoid the need for trusted setups by relying on cryptographic hash functions and public randomness. This transparency enhances security and simplifies the setup process, although it can increase the proof size. The trade-off here is between the ease of setup and the overall proof size, which remains practical for real-world applications.

Optimizations: Provers in many zero-knowledge proof systems perform polynomial evaluations, FFTs, and cryptographic operations efficiently, resulting in a prover run-time of $O(n \log n)$. Verifiers benefit from various optimizations that reduce verification time to $O(\log n)$ in many cases. These optimizations are crucial for making zero-knowledge proofs practical for real-time applications such as blockchain and smart contracts.

3.2 Groth16

Groth16 is a highly efficient zk-SNARK protocol, widely recognized for its practical applications in the realm of cryptography. Proposed by Jens Groth in 2016 [Gro16], Groth16 enables a prover to convince a verifier of the validity of a statement without revealing any underlying information about the statement itself.

The key features of Groth16 include its succinctness and efficiency. The proof size in Groth16 is constant, regardless of the complexity of the statement being proven, and the verification process is exceptionally fast. These characteristics make Groth16 particularly suitable for use in blockchain technologies and privacy-preserving applications, where it is essential to maintain the confidentiality of transactions while ensuring their validity.

The Groth16 zk-SNARK protocol is extensively used in various applications due to its efficiency and succinctness. It is integral to privacy-preserving transactions in cryptocurrencies like Zcash, ensuring transaction validity without revealing details. Additionally, it enhances privacy and scalability in blockchain platforms such as Ethereum by enabling efficient proof verification in smart contracts [Woo14a]. Several projects utilize Groth16 for confidential transactions in decentralized finance (DeFi) [Ano23]. Furthermore, the gnark library by ConsenSys employs Groth16 for creating and verifying zero-knowledge proofs in diverse applications, including secure voting systems and private data verification [Con24].

3.2.1 Constructing Groth16

In Figure 3.1, you can see the general zk-SNARK pipeline. This pipeline illustrates how to transform computational problems to apply a zk-SNARK. The formalization of this sequence and methodology is attributed to foundational works such as "Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture" by Ben-Sasson et al. (2013) [BSCG+14], and "Quadratic Span Programs and Succinct NIZKs without PCPs" by Gennaro et al. (2013) [GGP13]. These papers were critical in establishing the widely recognized zk-SNARK pipeline. Groth16 follows this pipeline as well [Gro16].

Before the pipeline is started, elliptic curve pairing values must be chosen. Let G_1, G_2, G_3 be groups with the same scalar field F, generators G_1, G_2, G_3, and a pairing $e : G_1 \times G_2 \to G_3$. For a multiset S of elements from F, define the zero polynomial $Z_S \in F[X]$ as:

$$Z_S(X) = \prod_{s \in S} (X - s)$$

The goal of the pipeline is to transform a starting computation into a zk-SNARK.

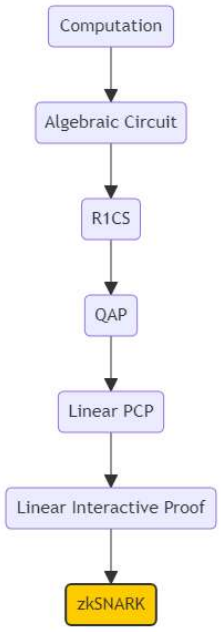

Figure 3.1: The general zk-SNARK pipeline.

Step 1: Computation

This first step consists of determining what to prove. This could be, for example, to prove you know an x and y for which

$$x^3 + 3x^2y = 100$$

Step 2: Algebraic Circuit

The next step is transforming your computation into circuits/logic gates. This can be done by flattening the computation [PHGR13]. Flattening is a procedure where the original code, which may contain arbitrarily complex statements and expressions, is converted into a sequence of statements that are of two forms: $x = y$, where y can be a variable or a number, and $x = y \,\mathrm{op}\, z$, where op can be $+$, $-$, $*$, $/$ and y and z can be variables, numbers or themselves sub-expressions. An example of such flattening can be found in Figure 3.2.

```
def qeval(x):
    y = x**3
    return x + y + 5
```

```
sym_1 = x * x
y = sym_1 * x
sym_2 = y + x
~out = sym_2 + 5
```

Figure 3.2: Flattening of a computation.

Step 3: Rank-1 Constraint System

Now that we have the logic gates, they can be transformed further into something called a rank-1 constraint system (R1CS). An R1CS is a sequence of groups of three vectors (a, b, c), and the solution to an R1CS is a vector s, where s must satisfy the equation $s \cdot a \cdot s \cdot b - s \cdot c = 0$ [PHGR13]. In the context of zero-knowledge proofs, s is called the witness.

Instead of having just one constraint, we are going to have many constraints, namely one for each logic gate. There is a standard way of converting a logic gate into a (a, b, c) triple depending on what the operation is ($+, -, *$ or $/$) and whether the arguments are variables or numbers. The length of each vector is equal to the total number of variables in the system, including a dummy variable "$\sim$ one" at

the first index representing the number 1, the input variables, a dummy variable "$\sim$ out" representing the output, and then all of the intermediate variables. An example of how to transform your flattened computation into R1CS matrices can be seen in Figure 3.3.

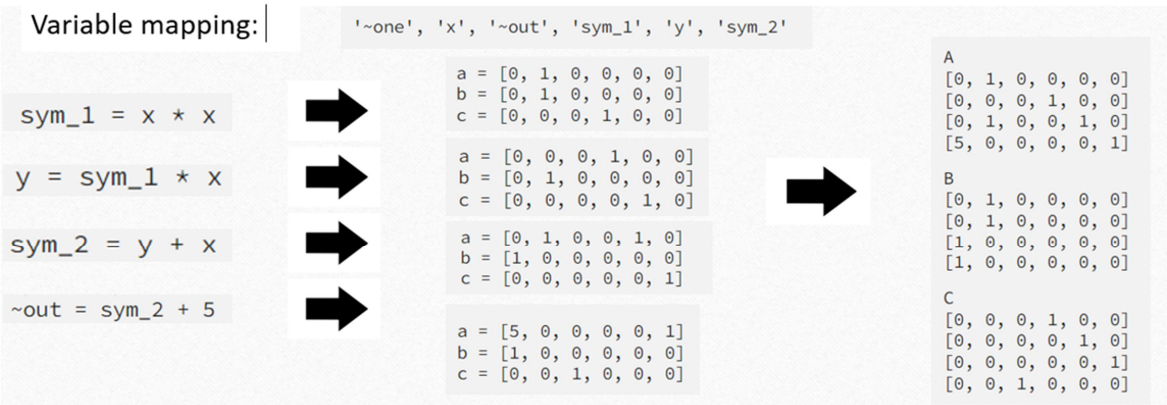

Figure 3.3: R1CS transform.

Step 4: Quadratic Arithmetic Program Transform by Lagrange Interpolation

The R1CS is not the final form of the constraint before generating the proof. It needs to be transformed into Quadratic Arithmetic Program (QAP) form first, which implements the exact same logic except using polynomials instead of dot products [SJTS13]. We transition from four groups of three vectors of length six to six groups of three degree-3 polynomials. Evaluating these polynomials at specific x coordinates represents individual constraints. For instance, evaluating the polynomials at $x = 1$ yields the first set of vectors, at $x = 2$ yields the second set, and so forth.

The QAP is defined as follows: Consider a basis $\mathbf{x} \in F^m$. For each constraint, we define polynomials such that $A_i(\mathbf{x}) = A_{ij}$, and similarly for B and C. Next, we define

$$A(X) = \sum_{i \in \{0,n\}} w_i \cdot A_i(X)$$

and similarly for B and C. The constraints can now be expressed as

$$A(x_i) \cdot B(x_i) = C(x_i).$$

This representation is known as a Quadratic Arithmetic Program (QAP) and its correctness can be verified by the existence of a low-degree polynomial $H(X)$ satisfying

$$A(X) \cdot B(X) - C(X) = H(X) \cdot Z_e(X).$$

To construct $H(X)$, we divide the left-hand side by $Z_e(X)$, resulting in a low-degree polynomial if the constraints are valid. This polynomial equation can be efficiently verified by evaluating it at a random value τ.

In order to transform the R1CS matrices into a QAP, we use a process called Lagrange interpolation. The problem that Lagrange interpolation solves is as follows: given a set of points (i.e., (x, y) coordinate pairs), Lagrange interpolation provides a polynomial that passes through all these points. This is achieved by decomposing the problem. For each x coordinate, we create a polynomial that has the desired y coordinate at that x coordinate and a y coordinate of 0 at all other x coordinates of interest. The final result is obtained by summing all these polynomials together.

Lagrange interpolation works as follows[But19]: suppose we want a polynomial that passes through the points $(1, 3)$, $(2, 2)$, and $(3, 4)$. We start by constructing a polynomial that passes through $(1, 3)$

and $(2, 0)$ and $(3, 0)$. To create a polynomial that "sticks out" at $x = 1$ and is zero at the other points, we use:

$$(x - 2) \cdot (x - 3)$$

Next, we rescale it so that its height at $x = 1$ is correct:

$$(x - 2) \cdot (x - 3) \cdot \frac{3}{(1 - 2) \cdot (1 - 3)}$$

This gives us:

$$1.5x^2 - 7.5x + 9$$

This polynomial is shown in the left image of Figure 3.4. We then repeat the process for the other two points to get similar polynomials that "stick out" at $x = 2$ and $x = 3$. Adding these polynomials together yields:

$$1.5x^2 - 5.5x + 7$$

This final result is shown in the right image of Figure 3.4.

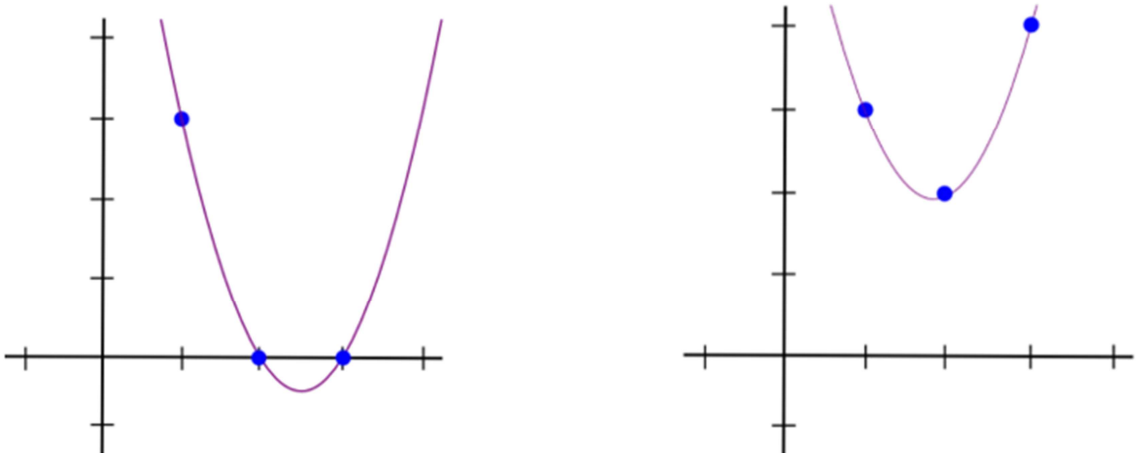

Figure 3.4: Lagrange interpolation visualization aid.

The algorithm described above takes $O(n^3)$ time, as there are n points and each point requires $O(n^2)$ time to multiply the polynomials together. With some optimization, this can be reduced to $O(n^2)$ time, and with further optimization using fast Fourier transform algorithms, it can be reduced even more. This optimization is crucial when dealing with functions used in zk-SNARKs, which often have many thousands of gates.

To create the actual QAP, take the first value out of every a vector, use Lagrange interpolation to make a polynomial out of that, where evaluating the polynomial at i gets you the first value of the i-th a vector. Repeat the process for the first value of every b and c vector, and then repeat that process for the second values, the third values, and so on. Be aware that the coefficients are in ascending order. Note that all of the work up until this point needs to be done only once for every function that you are trying to use zk-SNARKs to verify.

The main point of this QAP transformation is that instead of checking the constraints in the R1CS individually, we can now check all of the constraints at the same time by doing the dot product check on the polynomials.

Because in this case, the dot product check is a series of additions and multiplications of polynomials, the result is itself going to be a polynomial. If the resulting polynomial, evaluated at every x coordinate that we used above to represent a logic gate, is equal to zero, then that means that all of the checks pass. If the resulting polynomial evaluated at at least one of the x coordinates representing a logic gate gives a nonzero value, then that means that the values going into and out of that logic gate are inconsistent (i.e., the gate is $y = x \cdot \text{sym}_1$ but the provided values might be $x = 2$, $\text{sym}_1 = 2$, and $y = 5$).

To check correctness, we don't actually evaluate the polynomial $t = A \cdot s \cdot B \cdot s - C \cdot s$ at every point corresponding to a gate; instead, we divide t by another polynomial, Z, and check that Z evenly divides t - that is, the division t/Z leaves no remainder. Z is defined as $(x-1) \cdot (x-2) \cdot (x-3) \cdots$ - the simplest polynomial that is equal to zero at all points that correspond to logic gates.

Step 5: Trusted Setup

The trusted setup consists of two phases: the Power of Tau Ceremony and key generation.

Part 1: Power of Tau Ceremony

The first part of Groth16's trusted setup is called the Powers of Tau ceremony [NRBB22, Coi21]. Powers of Tau is a ceremony that is used to generate the initial parameters for zk-SNARKs. The ceremony is used to ensure the security of the initial parameters by using a multi-party computation (MPC) protocol to generate them [Fou18]. The main idea behind Powers of Tau is that by having multiple participants contribute randomness to the generation of the parameters, it becomes much more difficult for any individual machine/node, or group to tamper with them or compromise the entire security of the system. Note that the Powers of Tau ceremony is ONLY required in Non-Interactive Zero-Knowledge Proofs. The formulas below form the Powers of Tau ceremony.

$$
\begin{aligned}
&(\tau^0, \tau^1, \tau^2, \ldots, \tau^{2m-2}) \cdot G_1 \\
&(\tau^0, \tau^1, \tau^2, \ldots, \tau^{m-1}) \cdot G_2 \\
&\alpha \cdot (\tau^0, \tau^1, \tau^2, \ldots, \tau^{m-1}) \cdot G_1 \\
&\beta \cdot (\tau^0, \tau^1, \tau^2, \ldots, \tau^{m-1}) \cdot G_1 \\
&\beta \cdot G_2
\end{aligned}
$$

Part 2: Key Generation

Given circuit polynomials A_j, B_j, C_j, generate random values α, β, δ and define the polynomials L_j by:

$$L_j(X) = \beta \cdot A_j(X) + \alpha \cdot B_j(X) + C_j(X)$$

We cannot compute the $L_j(X)$ directly since we don't know α and β, but we can still construct $L_j(\tau) \cdot G_1$ using linear combinations of the values from the previous step. This way, we compute the proving key:

$$
\begin{aligned}
&(\alpha, \beta, \delta) \cdot G_1 \\
&(\tau^0, \tau^1, \tau^2, \tau^3, \ldots, \tau^{m-1}) \cdot G_1 \\
&\delta^{-1} \cdot (L_P(\tau), L_{P+1}(\tau), \ldots, L_{m-1}(\tau)) \cdot G_1 \\
&\delta^{-1} \cdot (\tau^0, \tau^1, \tau^2, \tau^3, \ldots, \tau^{m-2}) \cdot Z_{x(\tau)} \cdot G_1 \\
&(\beta, \delta) \cdot G_2 \\
&(\tau^0, \tau^1, \tau^2, \tau^3, \ldots, \tau^{m-1}) \cdot G_2
\end{aligned}
$$

and the verification key:

$$
\begin{aligned}
&\alpha \cdot G_1 \\
&\gamma^{-1} \cdot (L_0(\tau), L_1(\tau), L_2(\tau), \ldots, L_{P-1}(\tau)) \cdot G_1 \\
&(\beta, \gamma, \delta) \cdot G_2
\end{aligned}
$$

Instead of providing $\alpha \cdot G_1$ and $\beta \cdot G_2$, we could also provide $\alpha \cdot \beta \cdot G_3$ in the verifying key, but G_3 elements tend to be big and weird so we avoid them. The Phase 2 ceremony is critical for soundness. If δ is known, the prover can construct any $\delta^{-1} \cdot P(\tau) \cdot G_1$ of degree $2n-1$ where otherwise it is constrained to the form $\delta^{-1} \cdot H(\tau) \cdot Z_x(\tau) \cdot G_1$. The forced factorization into H and Z_x is critical for soundness.

Step 6: Multi-party Computation

It is crucial for the security of the system that the values $\tau, \alpha, \beta, \delta$ are kept secret. These values must be disposed of securely, hence they are referred to as toxic waste. The Zcash team has developed multi-party computation protocols to distribute the setup over many participants, so that each one only handles a fragment of the waste. This ensures that all fragments are needed to compromise the system. The system remains secure as long as at least one participant is trustworthy. The protocol is outlined in BGM17, with a historical context provided in a blog post by Vitalik [Fou18].

Here is a summary of how such protocols function. Suppose we need values $S_i = \alpha \cdot \beta^i \cdot A$ with secret α and β. We start with initial values $\alpha = \beta = 1$ and $S_i = A$, which are then sent to the first participant. The participant generates a, b randomly and computes:

$$S'_i = a \cdot b^i \cdot S_i = \alpha' \cdot \beta'^i \cdot A$$

with updated secrets $\alpha' = a \cdot \alpha$ and $\beta' = b \cdot \beta$. The S'_i is passed to the next participant, repeating the process. To ensure correctness, participants also publish $a \cdot G_2$, $b \cdot G_1$, and $b^i \cdot G_2$, and we verify:

$$\begin{aligned} e(S_0, a \cdot G_2) &= e(S'_0, G_2) \\ e(S'_{i-1}, b^i \cdot G_2) &= e(S'_i, G_2) \\ e(b \cdot G_1, b^{i-1} \cdot G_2) &= e(G_1, b^i \cdot G_2) \end{aligned}$$

The actual protocol in BGM17 includes additional safeguards such as proof-of-knowledge of a, b and a random beacon to ensure uniform distribution of results.

The setup for Groth16 has two phases, where the second phase is specific to the circuit you want to prove. This is unfortunate because it requires re-running the setup ceremony for each circuit. Later proof systems have simpler setups requiring only a secret value τ and values $(\tau^0, \tau^1, \tau^2, \tau^3, \ldots, \tau^{n-1}) \cdot (G_1, G_2)$. These can be reused for many proof systems and circuits. Well-constructed sets of values are available.

Step 7: Proof Generation

Given a witness vector $\mathbf{w} = (1, w_1, w_2, \ldots, w_n)$ that meets the constraints, we first compute the degree m polynomials A, B, C, H as described previously. Then, we compute the private witness polynomials:

$$L(X) = \sum_{i \in [0,m)} w_i \cdot L_i(X)$$

Next, we generate two random values r and s and compute:

$$\begin{aligned} A &= (\alpha + r \cdot \delta + A(\tau)) \cdot G_1 \\ B &= (\beta + s \cdot \delta + B(\tau)) \cdot G_2 \\ C &= \big(\delta^{-1} \cdot (L(\tau) + H(\tau) \cdot Z_x(\tau)) + s \cdot (\alpha + r \cdot \delta + A(\tau)) + r \cdot (\beta + s \cdot \delta + B(\tau)) - r \cdot s \cdot \delta\big) \cdot G_1 \end{aligned}$$

The proof consists of the triplet (A, B, C). To compute $A(\tau) \cdot G_1$ and $B(\tau) \cdot G_1$ efficiently, we can precompute a basis $A_i(X), B_i(X)$ from the proving key and use a multi-scalar multiplication (MSM). To compute $\delta^{-1} \cdot H(\tau) \cdot Z_x(\tau) \cdot G_1$, we use Jordi's trick: Notice that $H(X)$ is a degree-2 polynomial that evaluates zero on α. We pick a disjoint domain γ and prepare a $2n$ size basis over ω. Since evaluations on α are zero, we only need to MSM over the n values. To get evaluations on γ, we can interpolate $A(X), B(X), C(X)$ to γ and element-wise compute $A(X) \cdot B(X) - C(X)$. If x and y are chosen to be the odd/even 2-n roots of unity, the interpolation can be done using $6 \cdot \mathrm{NTT}_{F_p}$.

Q: Can we get rid of the NTTs on γ by MSming it directly and subtracting $C(\tau) \cdot Z_x(\tau)$? This will not work for A, B since we cannot multiply at τ. We also no longer have B evaluating to zero on α, so we end up with $4 \cdot \mathrm{NTT}_{2n}$ which cost about the same.

Note that the proof is malleable. Given a proof (A, B, C) and arbitrary values t, u, the triplet $(t^{-1} \cdot A, t \cdot B + t \cdot u \cdot G_2, C + u \cdot A)$ is also a valid proof and indistinguishable from a regularly produced one, see GM17 or BKSVD20.

Step 8: Proof Verification

Given a proof (A, B, C) and public inputs $\mathbf{w} = (1, w_1, \ldots, w_t)$, we aggregate the public inputs as follows:

$$\mathcal{L} = \sum_{i \in (0,p)} w_i \cdot (\gamma^{-1} \cdot L_i(\tau) \cdot G_1)$$

Next, we check the following equation:

$$e(A, B) = e(\alpha \cdot G_1, \beta \cdot G_2) + e(\mathcal{L}, \gamma \cdot G_2) + e(C, \delta \cdot G_2)$$

This process intuitively verifies that $A(\tau) \cdot B(\tau) = C(\tau) + H(\tau) \cdot Z_x(\tau)$ with additional terms to ensure consistency and cancel out any incorrect terms. Groth16 is notable for its complexity, where multiple aspects are validated simultaneously in one comprehensive expression. Unlike Groth16, later systems using polynomial commitments offer a more modular structure, though they tend to have larger proofs and require more steps in the verification process.

3.2.2 The Correctness

Completeness is derived from expanding the pairing check. **Soundness** comes from the fact that the values $\tau, \alpha, \beta, \gamma, \delta$ are unknown, so the prover's provided values must be linear combinations of the setup values. By considering their pairing check as a multivariate polynomial equation in these secret values, we can invoke Schwartz-Zippel and equate the coefficients. This tedious but straightforward process confirms the existence of a satisfying witness $\mathbf{w}$. **Zero knowledge** is ensured because A and B are uniformly random due to r and s, and C is fully determined through the claim.

3.2.3 Other Strengths and Drawbacks

A notable strength of Groth16 is its solid theoretical foundation. Groth16 is built upon well-established cryptographic assumptions, specifically the hardness of the Discrete Logarithm Problem (DLP) and the security of bilinear pairings. These assumptions are widely accepted in the cryptographic community, providing a high degree of confidence in the security of the protocol [BGM17].

Despite its many strengths, Groth16 is not without its drawbacks. One of the primary criticisms is the requirement for a trusted setup. The initial setup phase, known as the Common Reference String (CRS) generation, requires the generation of some cryptographic material that must remain secret. If the secrecy of this material is compromised, the security of the entire system is at risk. This requirement imposes significant trust on the setup phase and the entities involved, which can be a potential vulnerability. That is why the Common Reference String is commonly referred to as "Toxic Waste" [Fou18].

Another drawback is the complexity involved in the CRS setup. The setup process is circuit-specific, meaning a new CRS is required for each unique circuit. This limitation can be cumbersome in practice, especially in dynamic environments where circuits frequently change. The need for a trusted setup for each new circuit can lead to inefficiencies and increased operational costs.

The proof size remains constant irrespective of the complexity of the statement, which is crucial for minimizing data storage and transmission costs, particularly in blockchain technology. Verification time is also efficient, requiring a constant number of pairing evaluations and group operations.

While the verification time in Groth16 is minimal, the prover's computation can be intensive. The prover needs to perform a series of complex operations, including multi-scalar multiplications and pairings, which can be computationally demanding. This high computational requirement can be a bottleneck in applications where the prover's resources are limited or where fast proof generation is crucial. See Table 3.1 for how it compares to other zk-SNARKs.

3.3 Lattice-Based Zero-Knowledge Proofs: Where It Started

Lattice-based zero-knowledge proofs are an active area of research that can be broadly categorized into two groups. The first group focuses on proving statements tailored to practical applications, such as ring/group signatures, integer relations, and blockchain technologies. Key references in this area

include: [ALS20, BLS19, ENS20]. Specific applications include ring/group signatures [ESZ21, LNS21], proving integer relations [LNS20], and blockchain [ESZ21, LNS21]. The main drawback of using most of these protocols is that the proof size scales linearly with the length of the witness, making them unsuitable for proving larger statements.

The second group of research concentrates on developing protocols with asymptotically sub-linear (ideally poly-logarithmic) proof sizes [AL21, ACK21, BBC+18, BCS21, BLNS20]. These constructions can be transformed into efficient arguments of circuit satisfiability. However, in terms of concrete sizes, they lag behind the schemes in the first group for smaller statements due to parameters often overlooked in asymptotic analyses.

The practical foundations of lattice-based zero-knowledge proofs lie in an identification scheme by Lyubashevsky [Lyu22], which adapts the well-known Schnorr protocol [Sch89] to the lattice setting. This scheme aims to prove knowledge of a secret vector $s \in R_q^m$ of small norm that satisfies $As = t$ over R_q. The protocol for this proof can be seen in Figure 3.5.

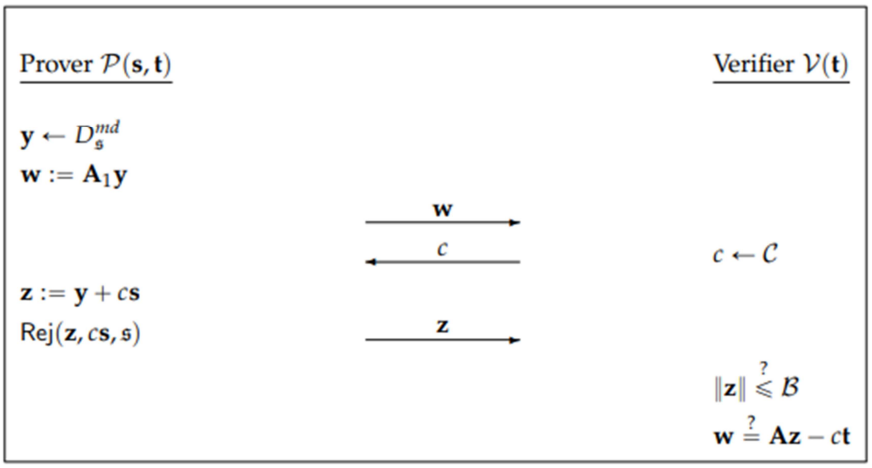

Figure 3.5: The identification scheme by Lyubashevsky [Lyu22] employs Rej, a rejection sampling algorithm, to maintain the zero-knowledge property of the protocol.

Consider the soundness property of the protocol. Using standard rewinding techniques, one can obtain two accepting transcripts (w, c, z) and (w, c', z') with the same first message w and distinct challenges $c, c' \in C$. From the verification equations, we deduce that $A(z - z') = (c - c')t$ and $\|z - z'\| \leq 2B$. If we were to adapt the strategy from the discrete logarithm setting, the next step would be to set $\bar{s} = \frac{z-z'}{c-c'} \in R_q^m$ and conclude that $A\bar{s} = t$.

However, this approach faces significant issues. Firstly, due to the verification condition $\|z\| \leq B$, it is crucial that the coefficients of the challenges in C are relatively small. Therefore, it is unclear whether, for distinct challenges $c, c' \in C$ with small coefficients, the difference $c - c'$ exists [Lyu22].

3.4 Lattice-Based Zero-Knowledge Proofs and Applications: Shorter, Simpler, and More General

The paper "Lattice-Based Zero-Knowledge Proofs and Applications: Shorter, Simpler, and More General" by Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Maxime Plançon [LNP22b] introduces a more efficient protocol for lattice-based zero-knowledge proofs. This protocol is grounded in the difficulty of the Module-SIS and Module-LWE problems and is designed to demonstrate knowledge of a short vector $\mathbf{s}$ that satisfies the equation

$$A\mathbf{s} = \mathbf{t} \mod q.$$

Earlier methods often depended on the ℓ_∞ norm and the Chinese Remainder Theorem (CRT) representation, which involved creating a commitment to a polynomial vector $\mathbf{m}$ derived from the coefficients of $\mathbf{s}$. Proofs then demonstrated polynomial product relations to indirectly show that the ℓ_∞

norm was small. This approach, seen in frameworks like those developed previously by Lyubashevsky et al. [LPR13b, Lyu22], led to larger and more complex proofs.

In contrast, this new protocol [LNP22b] offers a more direct and efficient way to prove that the coefficients of $\mathbf{s}$ have a small ℓ_2 norm without converting to the CRT representation. By leveraging the inner product between vectors and presenting them as coefficients of polynomial products, the system hides all but one coefficient, resulting in shorter and simpler proofs. This method is effective over various polynomial rings, especially those like $Z[X]/(X^n + 1)$.

The new approach significantly enhances efficiency and compactness. By bypassing the CRT representation and directly utilizing the ℓ_2 norm, proofs are reduced to around 14 KB, with about 8 KB dedicated to the "minimum" commitment. This demonstrates that the method is close to optimal for commitments to $\mathbf{s}$ using existing lattice-based schemes.

When comparing the group signature implementation in this work to the previously most efficient one by Lyubashevsky et al. [LNPS21], several improvements are noted. The signature generation and verification times in this scheme do not depend on the group size, making it more efficient for large groups. Although the signature length increases slightly with the group size, for groups larger than 2^{21} members, this protocol achieves a comparable signature size with a smaller public key and more efficient signing and verification times compared to tree-based group signatures like those from [ESZ21, BDK+21].

3.5 Lattice-Based Zero-Knowledge Proofs Under a Few Dozen Kilobytes

The paper "Lattice-Based Zero-Knowledge Proofs Under a Few Dozen Kilobytes" written by Ngoc Khanh Nguyen [Ngu22] builds upon the advancements made in "Lattice-Based Zero-Knowledge Proofs and Applications: Shorter, Simpler, and More General" [LNP22b]. It introduces an even more efficient protocol for lattice-based zero-knowledge proofs, leveraging the hardness of lattice problems to achieve proof sizes of just a few dozen kilobytes. The authors present further improvements in proof efficiency and compactness, making these protocols more suitable for practical applications in cryptography. The framework they propose here is called Lantern.

Lantern is a new lattice-based zero-knowledge protocol designed to provide efficient and compact proofs based on the hardness of the Module-SIS and Module-LWE problems. This framework aims to prove statements related to lattice-based cryptography, specifically focusing on proving knowledge of a short vector that satisfies certain conditions modulo a prime q. The framework is suitable for applications that require privacy-preserving primitives such as verifiable encryption schemes, ring signatures, and group signatures.

The Lantern framework is composed of several key components, each addressing different aspects of zero-knowledge proofs and lattice-based cryptographic constructions:

1. ABDLOP Commitment Scheme: The ABDLOP commitment (see Figure 3.6) scheme is an integral part of the Lantern framework, synthesizing the principles of the Ajtai and BDLOP commitment schemes to create a robust mechanism for secure commitments.

- **Ajtai Commitment Scheme:** The Ajtai commitment scheme [Ajt96a] is based on the hardness of lattice problems, particularly the Short Integer Solution (SIS) problem. This scheme leverages the difficulty of finding short vectors in a lattice without specific knowledge. Commitments are created by embedding the message into a lattice structure and adding a random noise vector. Formally, given a lattice basis $\mathbf{B}$ and a vector $\mathbf{v}$, the commitment $\mathbf{c}$ to a message vector $\mathbf{m}$ is generated as:

  $$\mathbf{c} = \mathbf{Bm} + \mathbf{r}$$

  where $\mathbf{r}$ is a random noise vector. The security of the Ajtai scheme relies on the difficulty of finding $\mathbf{m}$ and $\mathbf{r}$ given only $\mathbf{c}$. This ensures that the commitment is both hiding (the commitment does not reveal the message) and binding (the commitment uniquely defines the message, barring the noise).

- **BDLOP Commitment Scheme:** The BDLOP commitment scheme [BDL+16] extends the principles of hiding and binding commitments to the Learning With Errors (LWE) problem. The LWE problem involves distinguishing between random linear combinations of a secret vector and

- ABDLOP.KeyGen(1^κ): select parameters $q, d, \kappa_{\mathsf{MSIS}}, m_1, m_2, \ell, \ell_{\mathsf{ext}}, \nu, \omega, \alpha, B_1, B_2$. Let χ be the uniform distribution on S_ν. Sample uniformly random matrices $\mathbf{A}_1 \leftarrow \mathcal{R}_q^{\kappa_{\mathsf{MSIS}} \times m_1}, \mathbf{A}_2 \leftarrow \mathcal{R}_q^{\kappa_{\mathsf{MSIS}} \times m_2}, \mathbf{B} \leftarrow \mathcal{R}_q^{\ell \times m_2}, \mathbf{B}_{\mathsf{ext}} \leftarrow \mathcal{R}_q^{\ell_{\mathsf{ext}} \times m_2}$.
  Define

$$pp.\mathsf{dim} := (q, d, \kappa_{\mathsf{MSIS}}, m_1, m_2, \ell, \ell_{\mathsf{ext}})$$
$$pp.\mathsf{norms} := (\nu, \omega, \alpha, B_1, B_2)$$
$$pp.\mathsf{mat} := (\mathbf{A}_1, \mathbf{A}_2, \mathbf{B}, \mathbf{B}_{\mathsf{ext}}).$$

  Return $pp = (pp.\mathsf{dim}, pp.\mathsf{norms}, pp.\mathsf{mat})$.

  The public parameters define the following message, randomness, commitment spaces along with randomness distribution and a set of relaxation factors:

$$\mathcal{S}_M = \{\mathbf{s}_1 \in \mathcal{R}_q^{m_1} : \|\mathbf{s}_1\| \leqslant \alpha\} \times \mathcal{R}_q^{\ell}, \quad \mathcal{S}_C = \mathcal{C} \text{ as in Section 3.3.6,}$$
$$\mathcal{S}_T = \mathcal{R}_q^{\kappa_{\mathsf{MSIS}}+\ell}, \quad \mathcal{S}_R = \mathcal{R}_q^{m_2}, \quad \mathcal{D} = \chi^{m_2}.$$

- ABDLOP.Commit($\mathbf{s}_1, \mathbf{m}; \mathbf{s}_2$): Given message $(\mathbf{s}_1, \mathbf{m}) \in \mathcal{S}_M$ and randomness $\mathbf{s}_2 \in \mathcal{R}_q^{m_2}$, return $\mathbf{t} = (\mathbf{t}_A, \mathbf{t}_B)$ where

$$\begin{bmatrix} \mathbf{t}_A \\ \mathbf{t}_B \end{bmatrix} := \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{0} \end{bmatrix} \mathbf{s}_1 + \begin{bmatrix} \mathbf{A}_2 \\ \mathbf{B} \end{bmatrix} \mathbf{s}_2 + \begin{bmatrix} \mathbf{0} \\ \mathbf{m} \end{bmatrix} \in \mathcal{R}_q^{\kappa_{\mathsf{MSIS}}+\ell}.$$

- ABDLOP.Open($\mathbf{s}_1, \mathbf{m}, \mathbf{s}_2, c; \mathbf{t}$): Given $(\mathbf{s}_1, \mathbf{m}) \in \mathcal{S}_M$, $\mathbf{s}_2 \in \mathcal{R}_q^{m_2}$, relaxation factor $c \in \mathcal{R}_q$ and a commitment $\mathbf{t} \in \mathcal{R}_q^{\kappa_{\mathsf{MSIS}}+\ell}$, output 1 if all the conditions below hold:
  1. ABDLOP.Commit($\mathbf{s}_1, \mathbf{m}; \mathbf{s}_2$) $= \mathbf{t}$
  2. $c \in \mathcal{S}_C$
  3. $\|c\mathbf{s}_2\| \leqslant B_2$
  4. $\|c\mathbf{s}_1\| \leqslant B_1$.

Figure 3.6: The ABDLOP commitment scheme [Ngu22].

those with small errors added. In the BDLOP scheme, a commitment is formed using a matrix $\mathbf{A}$, a secret vector $\mathbf{s}$, and an error vector $\mathbf{e}$. The commitment $\mathbf{c}$ is given by:

$$\mathbf{c} = \mathbf{A}\mathbf{s} + \mathbf{e} + \mathbf{m}$$

where $\mathbf{m}$ is the message vector. The security of this scheme relies on the hardness of the LWE problem, ensuring that without the error vector $\mathbf{e}$, it is computationally infeasible to distinguish the commitment from a random vector, thus maintaining both the hiding and binding properties.

The ABDLOP scheme in the Lantern framework [Ngu22] generalizes these principles to the Module-SIS and Module-LWE problems. By combining message vectors $\mathbf{m}$ and randomness vectors $\mathbf{r}$, the scheme generates commitments that are secure under these harder problems, ensuring the integrity and confidentiality of the commitments.

2. Inner Product Proofs: The inner product proofs in the Lantern framework provide a method to verify that the inner product of two vectors $\mathbf{a}$ and $\mathbf{b}$ (or a vector with itself) is within specific bounds without revealing the vectors. This is achieved through polynomial product proof systems, which hide all coefficients except the constant term. Formally, given vectors $\mathbf{a}$ and $\mathbf{b}$, the proof ensures that:

$$\langle \mathbf{a}, \mathbf{b} \rangle \mod q$$

is within the specified bounds while maintaining the confidentiality of $\mathbf{a}$ and $\mathbf{b}$. This method is crucial for cryptographic protocols where data confidentiality must be maintained alongside verification [BCC+16, BCG+18].

3. Approximate Range Proofs: Approximate range proofs enable efficient verification that a value lies within a specified range with high probability. This capability is essential for the correctness and reliability of cryptographic protocols. If we need to prove that a value v lies within a range $[a, b]$, the proof system ensures that:

$$\Pr[a \leq v \leq b] \geq 1 - \epsilon$$

where ϵ is a small error probability. These proofs use techniques that reduce the overall proof size and complexity, making the verification process efficient while maintaining high accuracy and security [BSBHR19].

4. Proof of Exact Norm Bounds: The proof of exact norm bounds ensures that vectors used in cryptographic operations have norms within specific limits. This verification is critical for maintaining the security and efficiency of the protocol, particularly in lattice-based constructions where vector norms are integral to the system's security properties. Formally, for a vector $\mathbf{v}$, the framework ensures that:

$$\|\mathbf{v}\| \leq \beta$$

where β is a predefined bound. By ensuring these norms are within acceptable limits, the framework upholds the robustness of cryptographic operations [Lyu12].

5. Non-Interactive Commit-and-Prove System: The Lantern framework includes a non-interactive commit-and-prove system, significantly enhancing its practicality and ease of use. Utilizing the Fiat-Shamir transformation, this system converts interactive proofs into non-interactive ones, streamlining the proof process. Traditional interactive proofs require multiple rounds of communication between the prover and verifier, which can be cumbersome and inefficient. The non-interactive approach replaces the verifier's challenge with a hash function, allowing the prover to generate a single, non-interactive proof. This process ensures that the proof remains secure while eliminating the need for multiple rounds of communication [FS86b, CDP18].

The Lantern framework offers notable improvements over previous works in both proof size and efficiency, making it a practical solution for contemporary cryptographic applications.

The Lantern framework generates compact proofs, approximately 13 KB in size for basic statements [Ngu22]. These proofs are 2 to 3 times smaller than those produced by earlier protocols and 1 KB smaller than in "Lattice-Based Zero-Knowledge Proofs and Applications: Shorter, Simpler, and More General" [LNP22b]. If previous protocols produced proofs of size $\mathcal{O}(n \cdot k)$ kilobytes, where n is the security parameter and k is a constant factor, the Lantern framework reduces this to $\mathcal{O}(n)$ kilobytes. This reduction in proof size is beneficial in applications where bandwidth and storage are constrained, ensuring that cryptographic operations can be performed with lower resource consumption.

Just like the work it built on, Lantern does not rely on the Chinese Remainder Theorem (CRT) slots technique, which simplifies the proof process and reduces computational overhead. By avoiding the need to work with multiple CRT slots, the framework reduces the number of operations from $\mathcal{O}(n \log n)$ to $\mathcal{O}(n)$, where n is the dimension of the lattice. Additionally, Lantern employs efficient sampling techniques and avoids repeated commitments, which further reduces computational complexity. For instance, the commitment scheme used in Lantern requires fewer matrix multiplications and vector additions, leading to lower computational costs. These optimizations result in high performance and low computational cost, making the framework more efficient compared to traditional methods.

The Lantern framework optimizes prover runtime and verification time by utilizing an ABB-like group signature approach, as detailed in the works by del Pino et al. [dPLS18] and Lyubashevsky et al. [LNPS21]. This method ensures that the time required to generate a signature (prover runtime) and the time needed to verify a signature are both independent of the group size. In other words, the computational effort for these tasks remains constant, regardless of how large the group becomes.

Traditional group signature schemes often see increased signing and verification times as the group size grows, which can lead to scalability issues [dPLS18]. The ABB-like group signatures used in the Lantern framework maintain constant times for these operations, mitigating such concerns [Ngu22]. This property ensures that the performance of the signature generation and verification processes remains steady, making the framework suitable for scenarios where group size can vary significantly.

The Lantern framework's proof system can be integrated into various privacy-preserving applications, including verifiable encryption schemes, ring signatures, and group signatures. The combination of efficiency and compact proof sizes makes it suitable for real-world cryptographic protocols that require both security and performance. For example, in a ring signature scheme, the proof size directly impacts the overall size of the signature, and Lantern's reduced proof size translates to more compact

signatures. Similarly, the efficiency improvements mean that operations like signing and verification can be performed faster, which is crucial for applications requiring high throughput or operating in resource-constrained environments.

3.6 OpenFHE

Fully Homomorphic Encryption (FHE) has revolutionized the ability to perform computations on encrypted data, maintaining confidentiality without requiring decryption. This capability is pivotal for constructing zero-knowledge proofs (ZKPs) that ensure both privacy and security. OpenFHE, a comprehensive open-source library, has significantly advanced the practical implementation of FHE by integrating features from prior projects like PALISADE, HElib, and HEAAN, while introducing novel functionalities aimed at enhancing usability and performance [BAB+22, Mic20, H+20].

OpenFHE is designed to support a diverse array of FHE schemes, including BGV, BFV, CKKS, and DM-like schemes [GHS11b, GHS11a]. These schemes cater to different types of computations—integer, real-number, and Boolean arithmetic. OpenFHE assumes that all implemented schemes will eventually support bootstrapping and scheme switching, facilitating arbitrary-depth computations and seamless interoperability between different FHE schemes.

To optimize performance, OpenFHE includes support for multiple hardware acceleration backends via a standard Hardware Abstraction Layer (HAL). This modular design allows the library to leverage specific hardware capabilities, such as Intel® AVX512 instructions, for efficient polynomial arithmetic operations, which are central to FHE [A+18].

In the context of this work, OpenFHE's robust implementation of FHE serves as a foundational component for the commitment scheme. Utilizing FHE, we can perform necessary computations on committed data while preserving the zero-knowledge property, ensuring that the verifier can validate the proof without the need to decrypt the committed values [C+17a].

OpenFHE's support for various schemes and automated tools for parameter generation and noise estimation simplifies the development process. For instance, the CKKS scheme's approximate arithmetic is particularly useful for applications requiring efficient handling of real-number computations within the ZKP framework. Moreover, the ability to switch between schemes, such as from CKKS to a DM-like scheme for specific operations, enables optimized performance and functionality tailored to the ZKP's requirements [C+17b].

The integration of OpenFHE into our ZKP system offers several benefits:

- **Enhanced Security**: Leveraging the IND-CPA security guarantees provided by LWE-based FHE schemes implemented in OpenFHE, our ZKP ensures the commitment remains secure against chosen-plaintext attacks [Reg10].
- **Performance Optimization**: The hardware acceleration support in OpenFHE allows our ZKP to perform efficiently, even for complex and deep computations.
- **Flexibility and Usability**: The user-friendly modes and automated maintenance operations in OpenFHE reduce the complexity involved in managing modulus switching, key switching, and bootstrapping, thereby streamlining the development and deployment of our ZKP.

IND-CPA (Indistinguishability under Chosen-Plaintext Attack) security ensures that an adversary cannot distinguish between ciphertexts of any two chosen plaintexts, providing robust confidentiality. For commitment schemes within ZKPs, IND-CPA works very well in this work because:

- **Commitment Integrity**: The primary requirement is to keep the committed values confidential until they are revealed in the proof. IND-CPA ensures this by making it infeasible for an adversary to gain any information about the plaintext from the ciphertext.
- **No Decryption by Verifier**: In this work, the verifier does not need to decrypt the commitments before processing, relying instead on the encrypted computations with homomorphic nature. This reduces the attack surface, making chosen-ciphertext attacks less relevant.
- **Efficient and Practical**: IND-CPA provides a balance between strong security guarantees and computational efficiency. This is essential for practical implementations of FHE-based ZKPs.

3.7 Blockchain and Trusted Computing: Problems, Pitfalls, and a Solution for Hyperledger Fabric

A smart contract on a blockchain cannot maintain confidentiality because its data is duplicated across all nodes in the network. To solve this issue, it has been proposed by Marcus Brandenburger et al. [BCKS18] to combine blockchains with Trusted Execution Environments (TEEs), such as Intel SGX [CD16], for executing applications that require privacy.

Blockchain and Trusted Computing: Problems, Pitfalls, and a Solution for Hyperledger Fabric, written by Marcus Brandenburger et al. [BCKS18], explores the challenges and proposes a solution for integrating TEEs with Hyperledger Fabric [ABB+18], a prominent enterprise blockchain platform.

One pitfall of combining Hyperledger Fabric with Intel SGX is that TEE's, which are generally stateless, are vulnerable to rollback attacks, where previous states can be reinstated, thus compromising privacy [MAK+17]. This issue is exacerbated in blockchains with non-final consensus protocols, such as Ethereum's proof-of-work, where rollbacks are inherent to the design [Woo14b]. Therefore, TEEs cannot directly secure blockchains that do not have final consensus decisions, limiting their applicability to platforms with final consensus protocols.

The work of Marcus Brandenburger et al. [BCKS18] presents an architecture and prototype for executing smart contracts using Intel SGX technology within Hyperledger Fabric. Their system addresses the challenges of Fabric's execute-order-validate architecture, mitigates rollback attacks on TEE-based execution as effectively as possible, and reduces the trusted computing base. To enhance security, each application on the blockchain is encapsulated within its own enclave, protecting it from the host system.

3.7.1 Proposed Architecture and System Initialization

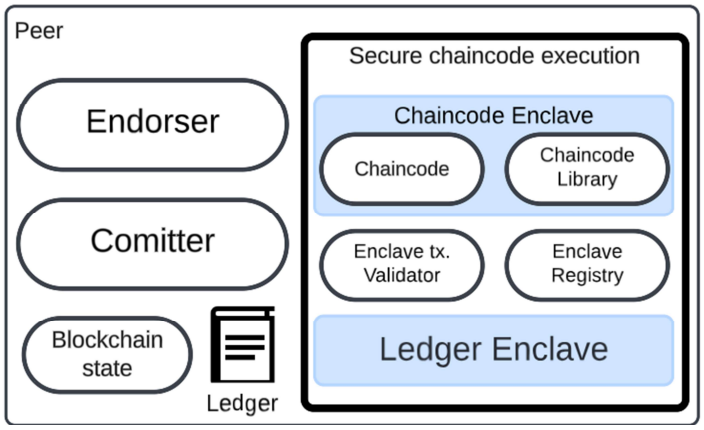

Figure 3.7: The System architecture discussed in the work of Brandenburger et al. [BCKS18]. Everything in the bold lined square has been added to the peer to enable chaincode execution with SGX enclave protection. The Ledger Enclave and the Chaincode Enclave (both marked in blue) run within SGX enclaves.

The approach proposed by Marcus Brandenburger et al. [BCKS18] extends a Fabric peer by integrating several components within SGX for secure execution (see Figure 3.7):

- **Chaincode Enclave**: Executes specific chaincode, isolating it from the peer and other chaincodes, using a chaincode library for interaction.
- **Ledger Enclave**: Maintains the ledger within an enclave, performing validation, generating cryptographic hashes, and exposing metadata.
- **Enclave Registry**: Runs outside SGX, maintaining identities of all chaincode enclaves and performing attestation.

- **Enclave Transaction Validator**: Validates transactions executed by chaincode enclaves before committing them to the ledger.

When a peer joins the network, the ledger enclave is initialized by the admin with the genesis block, verifying its integrity. The ledger enclave generates a key pair and maintains configuration values from the genesis block. It only accepts blocks from the ordering service, verifying each block's signature. The ledger enclave maintains information about the most recently processed transaction to ensure all blocks are in order. After the ledger enclave is set up, the peer admin installs and instantiates the enclave registry on each peer.

3.7.2 Chaincode Enclave Bootstrapping and Execution

To initialize a chaincode enclave, the peer admin follows these phases:

1. **Creating the Chaincode Enclave**: Install the chaincode enclave at the peer and send a setup transaction proposal, generating a key pair.
2. **Registering with the Enclave Registry**: Produce an attestation report, register with the enclave registry, which verifies the report with the IAS and stores the attestation result on the ledger.
3. **Provisioning of Secrets**: Optionally provision the chaincode enclave with secrets like encryption keys.
4. **Binding to the Ledger Enclave**: Bind to the ledger enclave through local attestation, verifying the ledger enclave's attestation report and storing its public key (PKLE).

This enclave registration process is depicted below in Figure 3.8.

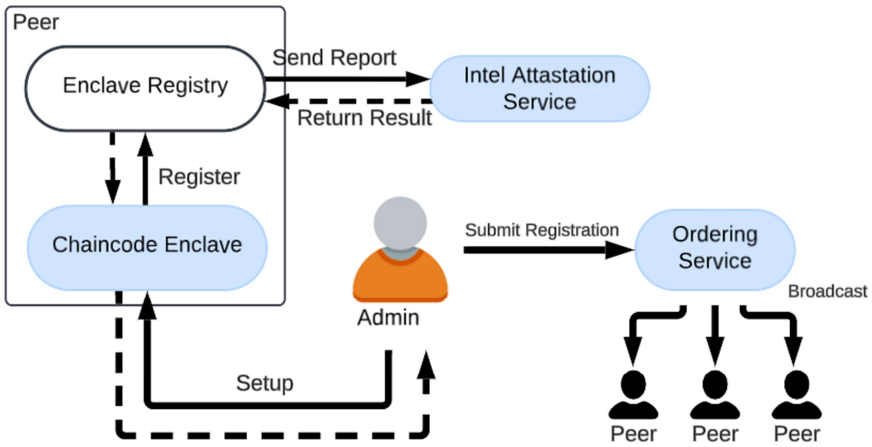

Figure 3.8: The depiction of the Enclave registration [BCKS18]. Also here, everything that is marked blue is under the protection of an SGX enclave.

Endorsement: A client sends an invoke transaction proposal to the peer, which forwards it to the chaincode enclave. The chaincode enclave processes the operation and returns a signed response, which the peer sends back to the client. The client verifies the enclave's authenticity and encrypts the operation using PKCC before sending it.

Validation and State Update: The ordering service accepts transactions submitted by the clients, assigns them to a block, and broadcasts the block to all peers. Every peer validates the transaction and updates its ledger copy. For validating transactions produced by a chaincode enclave, the enclave transaction validator performs similar steps as the validation system chaincode (VSCC), checking for conflicts and evaluating the endorsement policy. Additionally, it verifies that the transaction was produced by the correct chaincode enclave by retrieving the attestation result and public key for the enclave indicated by the transaction from the enclave registry, verifying these, and the enclave's signature on the transaction. If successful, the transaction is marked as valid, the peer commits it to its local ledger, and the blockchain state is updated accordingly.

3.7.3 Security

The proposed system [BCKS18] ensures that any state update originates from authenticated enclaves, preventing rollback attacks and maintaining confidentiality. Each chaincode runs within its own SGX enclave, isolating sensitive data. The ledger enclave maintains the blockchain state, and the enclave registry manages the identities of chaincode enclaves.

The paper [BCKS18] presents a formal model for the security guarantees of the proposed system. The blockchain is modeled as a stateful function $F : S \times T \to S$, where S represents the state and T represents the transactions. The system ensures that any malicious peer can only learn subsequent states resulting from valid transactions, maintaining confidentiality and integrity.

The modular architecture separates chaincode execution from the peer and moves it into an enclave, using a trusted ordering service to produce a signed sequence of transactions. This ensures transactions are in order and untampered. However, Fabric's execute-order-validate paradigm means execution is speculative, potentially leaking information. To solve this issue, applications are adapted to follow the speculative execution nature of Hyperledger Fabric by using barriers to ensure speculative execution does not leak secrets. A barrier is stored on the blockchain, ensuring that the chaincode enclave evaluates the auction only if the barrier is present.

The SGX secure chaincode execution system preserves security even in the presence of malicious peers and potential the potential scenario where a malicious peer might reset the state of the ledger enclave to a previous state. The system ensures that such resets do not compromise the security and integrity of the chaincode execution due to the following reasons:

1. **State Update and Attestation**: Only state updates produced by a verified chaincode enclave with a specific public key (PK_{CC^*}) are accepted into the ledger state. The attestation report, which verifies the authenticity of the chaincode enclave, is stored on the ledger. This ensures that any state update accepted by the ledger enclave originates from a genuine chaincode enclave. Clients and peers can trust the validity of these state updates due to the attestation report.

2. **Ledger State Integrity**: The state entries accessed by the chaincode enclave accurately represent the blockchain state after executing a valid sequence of updates. Even if a malicious peer resets the ledger enclave, the sequence of state updates remains valid due to checks performed by the VSCC and the expected block number sequence. This ensures that the chaincode enclave always works with the correct blockchain state, as validated by the trusted ordering service (O).

3. **Confidentiality of State**: The state held within a chaincode enclave remains confidential, except for what is revealed through transactions. The execution logic and data within the enclave are protected by the TEE. The contents of the ledger enclave are sealed and stored securely, preventing undetected tampering. The chaincode library manages state encryption and decryption, ensuring data remains confidential. Only registered and verified chaincode enclaves can interact with the ledger, maintaining data integrity and confidentiality.

Together, these points prove that the system maintains security by ensuring:

- Only genuine state updates are accepted.
- The ledger state remains accurate and valid despite potential resets.
- The state within the enclaves remains confidential and protected.

3.7.4 Chaincode verification and data confidentiality

In Hyperledger Fabric, chaincode must exclusively access data stored on the blockchain to ensure security and integrity. This is achieved through a collaboration between the chaincode library and the ledger enclave, which work together to protect the data from tampering by the local peer (see Figure 3.9).

When chaincode requires data for a specific key, it uses the function `getState(k)` to retrieve the value *val* from the blockchain state within the enclave. The subsequent steps ensure the integrity and consistency of this data:

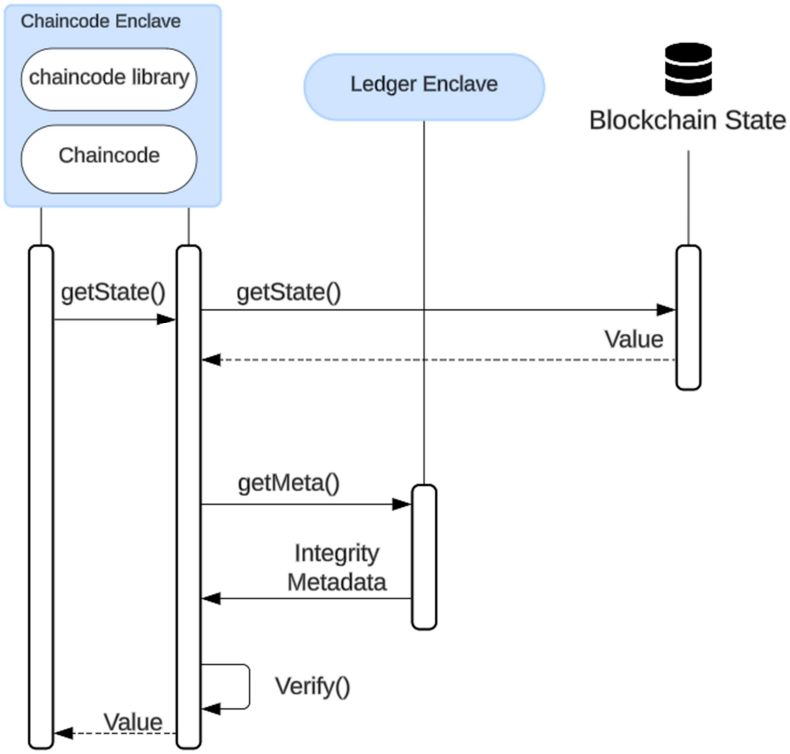

Figure 3.9: Verification of blockchain state utilizing the ledger enclave [BCKS18].

1. **Metadata Request**: The chaincode library requests integrity metadata from the ledger enclave using `getMeta(k, z)`, where z is a unique nonce.
2. **Enclave Response**: The ledger enclave responds with the expected hash $hval$ of the value and a signature ϕ.
3. **Signature Verification**: The chaincode library, equipped with the ledger enclave's public key, verifies the signature ϕ. A successful verification confirms that the value val is correct and untampered.
4. **Nonce Usage**: The nonce ensures that the response is fresh and not a replay of an old response, enhancing security against replay attacks.

The native method of sealing data in Intel SGX (Software Guard Extensions) is not suitable for scenarios where data needs to be shared across different enclaves on various peers. This limitation arises because sealed data can only be accessed by the same enclave that initially sealed it.

To overcome this challenge, the chaincode library in Hyperledger Fabric supports two modes of encryption:

1. **Client-Based Encryption**: In this mode, the client manages the encryption keys and provides a key with each operation performed by the chaincode.
2. **Chaincode-Based Encryption**: Here, an administrator provides a specific encryption key for the chaincode during the setup process. This key is then shared with all chaincode enclaves.

Both encryption methods operate transparently, meaning encryption and decryption occur automatically during `putState` and `getState` operations.

The encryption mechanism offers several advantages over native SGX methods:

- **Flexibility**: Client-based or chaincode-based key management provides additional flexibility.

- **Accessibility**: Data on the blockchain can be retrieved later without the need for enclave support, making the system more versatile and easier to manage.

These enhancements ensure that Hyperledger Fabric can maintain state confidentiality while addressing the limitations of SGX data sealing.

3.7.5 performance

The performance tests were conducted on a test bed with the following hardware specifications:

- **Servers**: Supermicro 5019-MR
- **CPU**: 3.4GHz four-core E3-1230 V5 Intel CPU with SGX support
- **Memory**: 32 GB
- **Network**: 1 Gbps connection
- **Storage**: SATA SSD drive
- **Operating System**: Ubuntu Linux 16.04 LTS Server, kernel version 4.13.0-32

Utilizing the Fabric Client SDK for Go (https://github.com/hyperledger/fabric-sdk-go), clients were built to measure transaction throughput. Concurrent transactions were executed over a period of at least 30 seconds, and the average transaction throughput was recorded.

To provide a meaningful comparison, the results were measured against a baseline environment running an unprotected, unmodified Fabric peer. These results can be found in Figure 3.10, which shows the endorsement throughput and latency with up to 128 clients concurrently invoking transactions at a single endorsement peer (the left two images) and the end-to-end throughput and latency with different numbers of clients (the right two images).

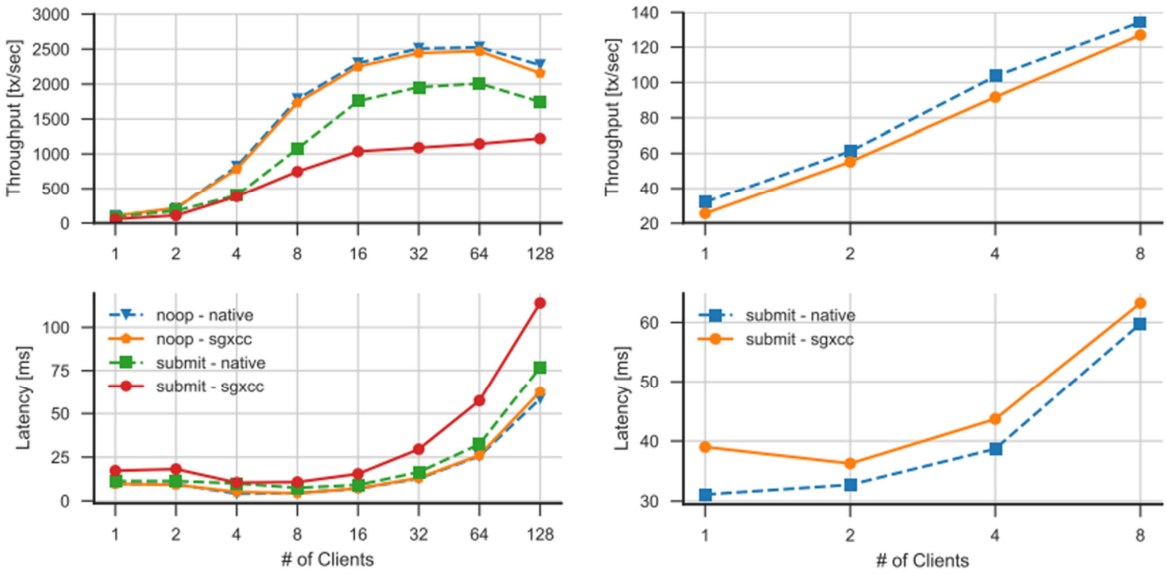

Figure 3.10: The left 2 images show the comparison between a native unaltered execution and an execution with SGX integration in terms of throughput and latency for both noop and submit transactions for up to 128 concurrent clients. The right 2 images show the end-to-end throughput and latency comparison between a native unaltered execution and an execution with SGX protection [BCKS18].

In Figure 3.10, it can be seen that noop transactions in SGX show nearly identical throughput and latency compared to native execution, reaching approximately 0.93–0.99 times the throughput. Submit transactions in SGX achieve 0.55–0.95 times the throughput of native execution, with an increased latency of about 6ms due to data retrieval and integrity metadata fetching. It can also be

seen that, with the default block size of 10 transactions per block, SGX execution achieves 0.80x–0.95x the throughput of native execution. This work measured the overhead of moving execution into SGX to be within 10%–20% and SGX adds a small constant overhead of about 100B for a 256-bit ECDSA signature formatted with JSON.

Chapter 4

Vesper: A New zk-SNARK

This work proposes Vesper, a Modular-LWE [Reg10]-based zk-SNARK [Nit20, BCG+18]. It involves a Regev commitment scheme [Reg09], combining zk-SNARK mechanics with fully homomorphic encryption [LW23]. The Vesper protocol structure is depicted in Figure 4.1.

Unlike state-of-the-art lattice-based ZKP, this work explores a novel approach by integrating Modular-LWE into the zk-SNARK construction, inspired by Groth16, but eliminating the need for a trusted setup. The core computation is based on LWE constraints, and a Regev Encryption [Reg09]-based commitment scheme replaces the elliptic curve-based elements of Groth16. Proof verification is handled differently to accommodate the absence of a trusted setup.

The ultimate goal is to develop a smart contract based on this ZKP. In blockchain applications, small proof sizes, fast verification, and non-interactive proofs are crucial for performance [BSBHR18]. Therefore, Groth16 serves as an inspiration for Vesper.

To ensure proof integrity, an LWE-based digital signing framework is used [Lyu12, DKL+18]. The detailed steps of this zk-SNARK construction are illustrated in Figure 4.2.

4.1 Flattening the Base Computation and R1CS Transform

Instead of using polynomials as in Groth16 [Gro16], we use the LWE problem constraints: $\langle A, s\rangle + e = t$. In Modular-LWE, we deal with matrices and vectors, leading to more logic gates due to the nature of matrix calculations. A function has been developed to generate the right R1CS matrices based on a specific Modular-LWE dimension. The pseudocode for this function can be found in Appendix A.

The total number of gates required is calculated as $2 \times n^2$. This results in $2 \times n^2$ gates in total. For every gate operation, the involvement of variables is logged in the R1CS matrices. The number of variables needed to represent the entire problem is:

$$\text{num\_variables} = 3n^2 + 2n + 1$$

To build the R1CS representation, initialize three matrices **A_matrix**, **B_matrix**, and **C_matrix** with zeros. These matrices will hold the constraints defining our R1CS.

The R1CS constraints for a valid witness $\mathbf{w}$ are:

$$\langle \mathbf{A}, \mathbf{w} \rangle \circ \langle \mathbf{B}, \mathbf{w} \rangle = \langle \mathbf{C}, \mathbf{w} \rangle$$

4.2 QAP Transformation

The computation to be proven is converted into a Quadratic Arithmetic Program (QAP). This transformation encodes the computation into polynomials for efficient proof generation and verification [Gro16, Nit20].

Polynomials succinctly represent computation constraints. For an arithmetic circuit with t gates, we construct a polynomial $P(x)$ of degree t whose roots correspond to the gates. The prover needs to show:

$$P(x) \cdot H(x) = \sum_i A_i(x) \cdot B_i(x) - \sum_i C_i(x)$$

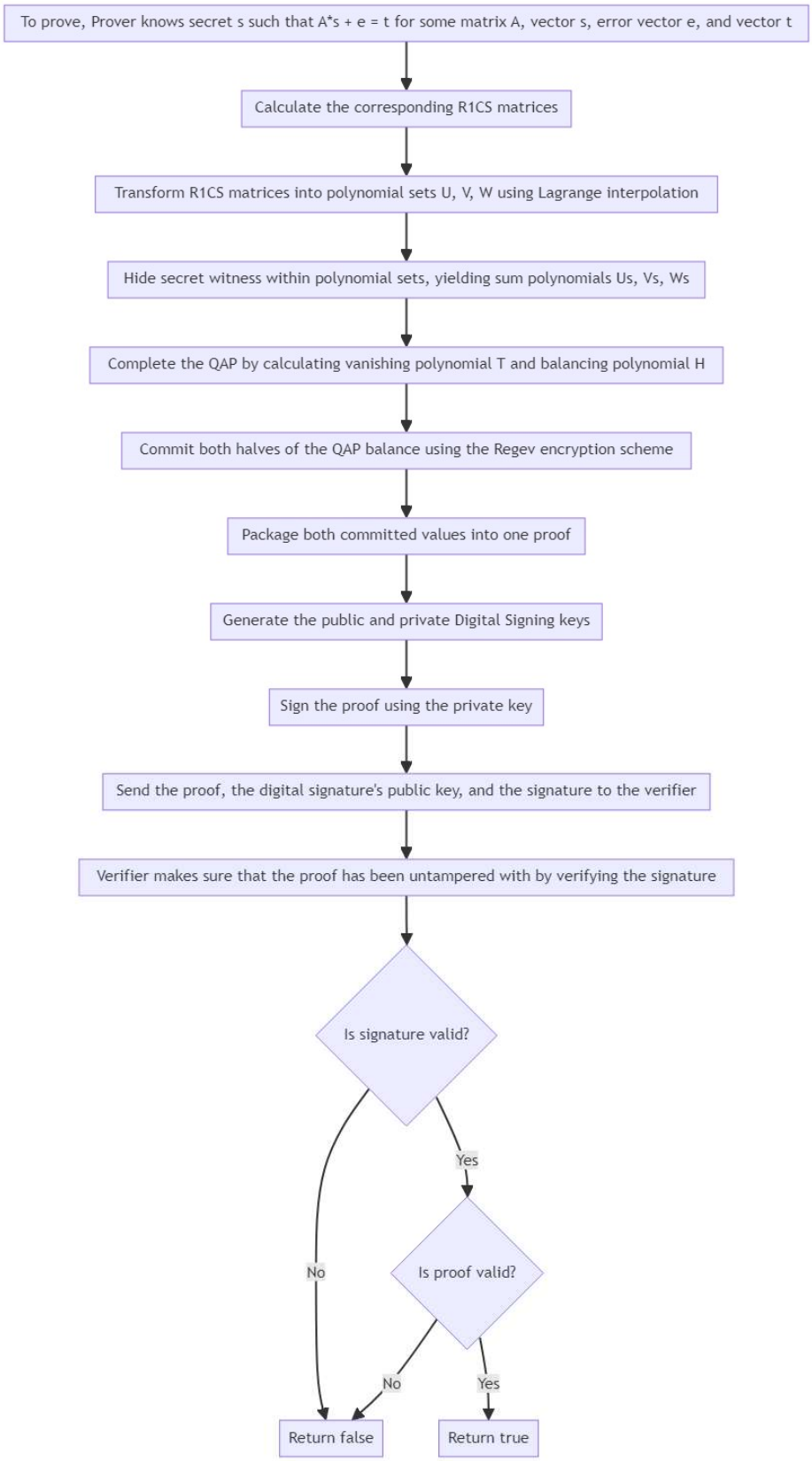

Figure 4.1: The logic construction of Vesper.

Secret info: $(s, e) \in \mathbb{Z}_q^n \times \mathbb{Z}_q^n$ such that $\|s\| \leq \alpha$, $\|e\| \leq \beta$.
$A \in \mathbb{Z}_q^{m \times n}$, $t = \langle A, s \rangle + e \in \mathbb{Z}_q^m$, where s is secret witness.

Security variables: order $\in \mathbb{Z}$, $\|\text{col}(A)\| = \|\text{row}(A)\| = \|\text{witness } s\| = \|\text{e}\| = \|\text{t}\| = n \in \mathbb{Z}$, $q \in \mathbb{Z}$, $k \in \mathbb{Z}$, $d \in \mathbb{Z}$, $\delta = q//k$. (the ring) $p_q = \text{np.poly1d}([1] + ([0] * (d-1)) + [1])$

Setup()

$\alpha \in \mathbb{Z}_t$ (randomly chosen integer from $[0, t]$), $sk(x) = \sum_{i=0}^{d-1} b_i x^i$ where $b_i \in \{0, 1\}$, $a_2(x) = \sum_{i=0}^{d-1} a_i x^i$ where $a_i \in \mathbb{Z}_q$ and a_i is randomly chosen, $e(x) = \sum_{i=0}^{d-1} c_i x^i$ where $c_i \sim \text{Normal}(0, 2) \mod q$, $pk_0(x) = -a_2(x) \cdot sk(x) + e(x)$, $pk_1(x) = a_2(x)$, $pk = (pk_0(x), pk_1(x))$, $u(x) = \sum_{i=0}^{d-1} u_i x^i$ where $u_i \in \{0, 1\}$, $e_1(x) = \sum_{i=0}^{d-1} e_{1,i} x^i$ where $e_{1,i} \sim \text{Normal}(0, 2) \mod q$

Prover(α, pk, e_1, u, sk):

(R1CS) A, B, C = LWEToR1CS_transform()
Transform R1CS to polynomials:
U = for each column i in A lagrange_interpolation($A(i)$)
V = for each column i in B lagrange_interpolation($B(i)$)
W = for each column i in C lagrange_interpolation($C(i)$)
Hide witness s in polynomial sets U, V and W:
$U_a = \sum_{i \in U} U(i) \cdot s(i)$
$V_a = \sum_{i \in V} V(i) \cdot s(i)$
$W_a = \sum_{i \in W} W(i) \cdot s(i)$
$T = \text{calculate\_vanishing\_polynomial}(U_a, V_a, W_a)$
$H = \frac{U_a \cdot V_a - W_a}{T}$
$L = U_a(\alpha) \cdot V_a(\alpha) - W_a(\alpha)$
$R = T(\alpha) \cdot H(\alpha)$
$c_1 = (pk[0] \cdot u + e_1) + \delta \cdot L$
$c_2 = (pk[0] \cdot u + e_1) + \delta \cdot R$
$pkey = Dilithium2Signer.generate\_keypair()$
$sig = Dilithium2Signer.sign(str((c_1, c_2)).encode())$

$pkey$
sig, c_1, c_2
$\longrightarrow$

Verifier(C_1, C_2, $pkey$, sig):

$is\_valid = verifier.verify($
$str((c_1, c_2)).encode(), sig, pkey)$
Return $C_1 \stackrel{?}{=} C_2$ && is_valid

Figure 4.2: The protocol for this work's zk-SNARK.

Where $A_i(x)$, $B_i(x)$, $C_i(x)$ are polynomials representing input wires, and $H(x)$ is the quotient polynomial proving computation correctness.

The QAP transformation for R1CS matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ into polynomials U, V, and W is achieved using Lagrange interpolation on the columns of the matrices.

To prove knowledge of a witness $\mathbf{w}$ such that $\langle \mathbf{A}, \mathbf{w} \rangle \circ \langle \mathbf{B}, \mathbf{w} \rangle = \langle \mathbf{C}, \mathbf{w} \rangle$ without revealing the witness or any other value used for committing the witness, the following protocol is used:

The witness $\mathbf{w}$ is hidden in the polynomials U, V, and W by multiplying each polynomial of U, V, and W by the corresponding witness value and summing the results. Let the number of polynomials in U, V, and W be n, which is also the length of $\mathbf{w}$. The hidden polynomials are defined as follows:

$$U_s = \sum_i U(i) \cdot s(i) \quad \text{for } i < n,$$

$$V_s = \sum_i V(i) \cdot s(i) \quad \text{for } i < n,$$

$$W_s = \sum_i W(i) \cdot w(i) \quad \text{for } i < n.$$

Hereby, the witness $\mathbf{s}$ is hidden inside the polynomials.

Next, the vanishing polynomial T is defined, which is based on the Rank-1 Constraint System. For a valid witness, the polynomial $(U_s \cdot V_s - W_s)$ should be divisible by T. In other words, the remainder when $(U_s \cdot V_s - W_s)$ is divided by T should be zero:

$$(U_s \cdot V_s - W_s) \mod T = 0.$$

Then, the polynomial H is defined as follows:

$$H = \frac{U_s \cdot V_s - W_s}{T}.$$

The QAP relation can now be stated. Due to the involvement of the witness $\mathbf{w}$ in the polynomials U_s, V_s, and W_s, the following equation should hold when a valid witness is used, based on the constraints:

$$U_s(\alpha) \cdot V_s(\alpha) - W_s(\alpha) = T \cdot H.$$

For this equation to hold, $(U_s \cdot V_s - W_s) \mod T$ must be zero, and as mentioned before, for this to be the case, the witness $\mathbf{w}$ needs to be valid [Gro16]. Appendix B contains the pseudocode for the transformation from R1CS matrices to a QAP.

4.3 Proof Generation

In order to generate a zero-knowledge proof, the QAP balance is exploited. Both halves of the QAP balancing equation are encrypted by committing them to an LWE-based homomorphic encryption scheme called Regev's Encryption [Reg09]. Given the homomorphic nature [BAB+22, LW23] of this encryption, the verifier can verify the proof without decrypting it first. This eliminates the need for a trusted setup, as only the prover requires a setup to generate the proof. Consequently, the scheme ensures both security and efficiency, leveraging homomorphic encryption and a lattice-based digital signing framework called Dilithium to protect against tampering while maintaining simplicity in setup requirements.

The security variables consist of the following:

- **Order of the Galois Field**: $order \in Z$
- **Norms**: $\|\mathbf{A}\|_\infty = \|\mathbf{A}\|_{\text{row}} = \|\mathbf{A}\| = \text{witness } \|\mathbf{s}\| = \|\mathbf{e}\|$
- **Dimensions**: $n \in Z, q \in Z, k \in Z, d \in Z, \delta = q/t$
- **Ring**: $Z_q = \text{np.poly1d}([1] + ([0] * (d-1)) + [1])$

Besides the security variables, there are other variables needed for committing. These values consist mainly of randomly generated polynomials and a value alpha that is used for evaluating polynomials at a specific point.

- Generate $\alpha \in Z_t$ (randomly chosen integer from $[0, t]$).
- Define $a(x) = \sum_{i=0}^{l-1} a_i x^i$ where $a_i \in Z_q$ and a_i is randomly chosen.
- Define $e(x) = \sum_{i=0}^{d} c_i x^i$ where $c_i \sim \text{Normal}(0, 2) \mod q$.
- Define $pk = (pk_0(x), pk_1(x))$, where

$$pk_0(x) = a(x)$$

$$pk_1(x) = (pk_0(x) \cdot sk(x)) + \sum_{i=0}^{d} e_{1,i} x^i \quad \text{where } e_{1,i} \sim \text{Normal}(0, 2) \mod q$$

- Generate $u(x) = \sum_{i=0}^{d} u_i x^i$ where $u_i \in \{0, 1\}$.
- Generate $e_1(x) = \sum_{i=0}^{d} e_{1,i} x^i$ where $e_{1,i} \sim \text{Normal}(0, 2) \mod q$.

With the setup complete, the next step is to construct the proof. These commitments are based on fully homomorphic encryption (FHE) using the Learning With Errors (LWE) problem [Reg10, Reg09, LW23, BAB+22], which ensures that the commitments are secure and can be evaluated without revealing the underlying plaintext. The pseudocode of the proof generation can be found in Appendix C.

The QAP balance consists of the following:

- **Left Polynomial**: Left $= U_s(\alpha) \cdot V_s(\alpha) - W_s(\alpha)$.
- **Right Polynomial**: Right $= T \cdot H$.

These two polynomials will be committed as follows [BAB+22]:

- **Commitment** $c1$:

$$c1 = \text{add}(\text{add}(\text{multiply}(pk[0], u), e1), \text{multiply}(\delta, \text{Left}))$$

- **Commitment** $c2$:

$$c2 = \text{add}(\text{add}(\text{multiply}(pk[0], u), e1), \text{multiply}(\delta, \text{Right}))$$

This protocol is non-interactive [BDSMP91, BSCG+14], meaning that it does not require a challenge-response phase. The commitments $c1$ and $c2$ are generated independently and can be verified without further interaction between parties.

Optimizations: In order to further reduce the proof size, the proof is converted to a hex string before sending to the digital signer and the verifier

The rest of this chapter will explain more about the commitment scheme and parameter selection used for optimal security.

4.3.1 Regev's Encryption Scheme

Regev's encryption scheme is a foundational lattice-based cryptographic system, introduced by Oded Regev [Reg09], which leverages the hardness of the Learning With Errors (LWE) problem for security. The scheme offers strong security guarantees based on worst-case lattice problems, making it resistant to quantum attacks. It operates over a finite field [Neu11] and uses a secret key to encode messages into ciphertexts that are computationally infeasible to decode without the key. Regev's scheme is notable for its simplicity, efficiency, and provable security under standard lattice assumptions [Reg09, ACL+22, Reg10, Lyu22]. In this section, we will discuss its key generation, encryption and decryption protocols, and its security.

Key Generation

- Sample a secret key $s \in Z_q^n$ uniformly at random.
- The public key consists of a matrix $A \in Z_q^{m \times n}$ and a vector $u = As + e \mod q$, where e is a small error vector sampled from a discrete Gaussian distribution.

Encryption

- Given a message $m \in \{0, 1\}$.
- Sample $r \in \{0, 1\}^m$ uniformly at random.
- The ciphertext is $c = (A^T r \mod q, u^T r + m \lceil \frac{q}{2} \rceil \mod q)$.

Decryption

Given a ciphertext $c = (c_1, c_2)$:

$$\begin{aligned} c_2 - c_1^T s &= \left(u^T r + m \left\lceil \frac{q}{2} \right\rceil - (A^T r)^T s \right) \mod q \\ &= \left(e^T r + m \left\lceil \frac{q}{2} \right\rceil \right) \mod q. \end{aligned}$$

If $\left| e^T r \right| < \frac{q}{4}$, then the term $e^T r$ is small enough that the result can be correctly interpreted to recover m.

Definition IND-CPA: An encryption scheme is IND-CPA secure if no polynomial-time adversary can distinguish between the encryptions of two chosen plaintexts with more than negligible advantage [BAB+22].

The security of Regev's encryption scheme reduces to the hardness of the Learning With Errors (LWE) problem [Reg10, Reg09]. If an adversary can break the IND-CPA (Indistinguishability under Chosen Plaintext Attack) security of the scheme, then this adversary can be used to solve the LWE problem. This reduction implies that breaking Regev's encryption scheme is as hard as solving the LWE problem, providing strong security guarantees based on the assumed difficulty of LWE.

Consider a series of hybrid experiments that transition from a real encryption to a random encryption:

- **Hybrid 0:** Real encryption with the actual public key (A, u).
- **Hybrid 1:** Replace u with a random vector v in the public key.

The security proof shows that the difference between Hybrid 0 and Hybrid 1 is indistinguishable due to the hardness of the LWE problem. In Hybrid 1, the ciphertext becomes uniformly random and independent of the plaintext, ensuring that the encryption remains indistinguishable from a random string.

Circular security involves encrypting the secret key under the same encryption scheme and analyzing if the scheme remains secure.

Challenge: If an adversary gains access to the encrypted secret key components $\text{Enc}(s_i)$ for each component s_i of the secret key s, can the scheme still be considered secure?

Proof of Security for Circular Encryption:

- **Encryption of the Secret Key**: Consider the encryption of each component of the secret key s:

$$\text{Enc}(s_i) = (a, \langle a, s \rangle + e + s_i \left\lceil \frac{q}{2} \right\rceil) = (a, b),$$

  where a is a random vector and $b = \langle a, s \rangle + e + s_i \lceil \frac{q}{2} \rceil$.

- **Reduction to IND-CPA Security**: The security argument shows that transforming $\text{Enc}(s_i)$ into an indistinguishable form:

$$\text{Enc}(s_i) = (a, \langle a + u_i \left\lceil \frac{q}{2} \right\rceil, s \rangle + e),$$

  where u_i is the i-th standard basis vector. This transformation indicates that $\text{Enc}(s_i)$ is indistinguishable from a random encryption due to the LWE assumption, ensuring that the encryption of the secret key does not reveal additional information.

Regev's encryption scheme is secure under the LWE assumption, providing IND-CPA security [BAB+22, Reg09]. The circular security analysis shows that even when the secret key components are encrypted, the scheme remains secure, further solidifying its robustness for practical cryptographic applications.

IND-CPA security is particularly advantageous for zero-knowledge proofs using Regev's encryption scheme for several reasons:

- **Non-leakage of Information:** IND-CPA security ensures that ciphertexts reveal no information about the plaintexts, preventing any leakage of the prover's secret information. This is crucial in zero-knowledge proofs where the verifier should learn nothing beyond the validity of the statement.
- **Simulator Construction:** The indistinguishability property of IND-CPA allows the construction of simulators that can generate fake proofs indistinguishable from real ones without knowing the secret. This helps in demonstrating the zero-knowledge property.
- **Robustness Against Adaptive Attacks:** IND-CPA security guarantees that even if an adversary can choose plaintexts adaptively, the encryption remains secure. This robustness is essential for zero-knowledge proofs to ensure that adaptive strategies by a malicious verifier do not compromise the proof's security.

The IND-CPA security of Regev's encryption scheme provides a strong foundation for zero-knowledge proofs. It ensures that no information is leaked about the prover's secret inputs, supports the construction of effective simulators, and offers robustness against adaptive attacks, making it an ideal choice for practical and secure zero-knowledge proof systems.

4.4 Modular-LWE Hardness Assumption

The security of this work's cryptographic constructions relies heavily on the hardness of the Module Learning With Errors (MLWE) problem. MLWE generalizes the Learning With Errors (LWE) problem to module structures over polynomial rings [WW19, LPR13a]. Specifically, the MLWE problem involves distinguishing between a sample $(A, As + e \mod q)$ and a uniformly random sample, where A is a matrix in $R_q^{n \times (n+m)}$, s is a secret vector, and e is an error vector sampled from a specified distribution χ.

To evaluate the difficulty of breaking MLWE-based cryptographic schemes, an estimator is often used [Ngu22] to provide an estimated root Hermite factor δ. Lower Hermite factor values of δ correspond to more difficult lattice problems, implying stronger security.

The Block-Korkine-Zolotarev (BKZ) [AKS01, CN11] algorithm is a powerful lattice reduction algorithm that utilizes the root Hermite factor to find short non-trivial vectors in a lattice Λ. BKZ operates by solving the Shortest Vector Problem (SVP) in sublattices of dimension b, referred to as the block size. When applying the best known SVP algorithm without memory constraints by Becker et al., the time required for BKZ to run on an md-dimensional lattice Λ with block size b is given by:

$$T = 8^{md} \cdot 2^{0.292b+16.4}$$

This shows that the time complexity grows exponentially with both the dimension md of the lattice and the block size b. The BKZ algorithm outputs a vector whose norm is approximately:

$$\delta^{md} \cdot (\det(\Lambda))^{1/md}$$

where δ is the root Hermite factor. This norm indicates how close the found vector is to the true shortest vector in the lattice. Within the BKZ context, the root Hermite factor δ is given by:

$$\delta = \left(\frac{b \cdot \pi b}{2\pi e} \right)^{1/(2(b-1))}$$

This formula relates δ to the block size b and involves fundamental constants π (pi) and e (Euler's number). The exponential time complexity and the quality of the output vector's norm in BKZ are directly influenced by the block size b and the root Hermite factor δ, underscoring their importance in the algorithm's design and application.

Table 4.1 shows levels of bit security that correspond to specific Hermite factors based on BKZ's equation. Since these factors are used in previous works, such as in the study by Ngoc Khanh [Ngu22], they are also utilized in this work to select parameters. The next section will elaborate further on how these Hermite factors are used in combination with the hardness estimator to select the parameters for this project.

level of bit security	80	128	256
root Hermite factor δ	1.0066	1.0044	1.0025

Table 4.1: Root Hermite factors for different levels of bit security

Ensuring that the matrix A is of full rank is crucial because it makes the MLWE problem robust. If A is not of full rank, it means that the rows of A are linearly dependent, implying redundant or predictable relationships between them [Ngu22]. This redundancy can be exploited by an attacker to solve the problem more easily, reducing the overall security. In other words, a full-rank matrix A ensures that the problem remains complex and hard to solve, as there are no simple linear dependencies that can be used to break the encryption scheme.

In order to make further hardness assumptions, let's take a look at the knapsack MLWE. This is a variant of MLWE that retains the hardness properties of the MLWE problem while being conceptually simpler to understand and analyze, especially when the matrix A is full rank. The hardness is up to an additive factor, which is the probability that a uniformly random matrix $A \in R_q^{n \times (n+m)}$ is singular.

The knapsack MLWE problem involves an adversary distinguishing between two scenarios:

- **Structured Scenario**: The adversary is given a matrix A and a product $A \cdot s \mod q$, where s is a secret vector.
- **Random Scenario**: The adversary is given the same matrix A and a random vector b.

Formally, the knapsack MLWE problem with parameters n, $m > 0$, and an error distribution χ over R asks the adversary A to distinguish between:

$$(A, As \mod q) \quad \text{for} \quad A \in R_q^{n \times (n+m)}, \quad s \in \chi^{n+m}$$

and

$$(A, b) \quad \text{for} \quad A \in R_q^{n \times (n+m)}, \quad b \in R_q^n$$

The adversary A is said to have an advantage ϵ in solving $\text{MLWE}_{m,n,\chi}$ if:

$$\left| \Pr \left[b = 1 \mid A \in R_q^{n \times (n+m)}, s \in \chi^{n+m}, b \in A(A, As \mod q) \right] - \Pr \left[b = 1 \mid A \in R_q^{n \times (n+m)}, b \in R_q^n, b \in A(A, b) \right] \right| \geq \epsilon$$

We say that $\text{MLWE}_{m,n,\chi}$ is hard if for all probabilistic polynomial-time (PPT) adversaries A, the advantage ϵ in solving $\text{MLWE}_{m,n,\chi}$ is negligible. This variant is considered to be as hard as the original MLWE problem, except for an additive factor related to the probability that the matrix A is singular.

4.5 Parameter Selection

Now, let's delve into how the parameters are selected. The parameters used in the proof generation of this work are selected using the same method as in the previous work by Ngoc Khanh Nguyen in *Lattice-Based Zero-Knowledge Proofs Under a Few Dozen Kilobytes.*

As mentioned in the previous section, an estimator is used to determine the modular-LWE hardness. A popular choice for such an estimator, and also the one used in this work, is the LWE estimator developed by Albrecht et al. [Alb17]. Appendix D contains the pseudocode for this estimator. This tool runs simulations to see how well-known attack methods, like sieving and enumeration methods, work against MLWE. The estimator helps set a value called the root Hermite factor δ, which reflects the problem's hardness. For high security (e.g., 128-bit security or higher), for example, δ would need to be set to 1.0044 as can be seen in Table 4.1.

This tool takes the following inputs:

- **nu**: Defines the range of the secret coefficients, which are uniformly chosen between $-nu$ and nu.
- **n**: Represents the dimension of the secret vector s.
- **d**: Denotes the dimension of the polynomial ring R.
- **logq**: The base-2 logarithm of the modulus q.

By incrementally increasing the dimension n and analyzing the corresponding security analyses outputs of each method used in the estimator, this tool determines the specific dimension at which the MLWE problem provides at least the specific bit security corresponding to the hermite factor used (see Table 4.1) given the provided input parameters. This process ensures that the MLWE instance remains hard under the best-known attacks, achieving the desired security level.

First, the function begins by loading the LWE Estimator from an external source. This can be found here: "https://bitbucket.org/malb/lwe-estimator/raw/HEAD/estimator.py". This estimator is a robust tool that simulates different attack methodologies to evaluate the hardness of LWE problems, which includes the MLWE problem as a special case.

Next, the function adjusts the problem dimensions to align with the module structure. Specifically, the dimension n is scaled by the ring dimension d, and the modulus q is computed as $2^{\log q}$. These adjustments ensure that the parameters are correctly configured for the MLWE context.

$$n = n \times d$$

$$q = 2^{\log q}$$

The function then calculates the standard deviation of the error distribution. This standard deviation is derived from the uniform distribution of the secret coefficients and is a critical factor in understanding the noise level inherent in the MLWE problem, which directly influences its hardness.

The standard deviation σ of the error distribution is calculated as:

$$\sigma = \sqrt{\frac{(2\nu + 1)^2 - 1}{12}}$$

Following this, the α parameter, representing the noise rate, is computed using the standard deviation and the modulus q. This parameter is essential as it quantifies the amount of noise introduced into the system, affecting the difficulty of distinguishing between different scenarios in the MLWE problem.

$$\alpha = \frac{\sigma}{q}$$

Now all variables are set, they are passed as input to the loaded estimator. The function then proceeds to estimate the hardness of the MLWE problem using the enumeration attack method. It simulates this attack and retrieves various estimates of the root Hermite factor δ_0 for different attack strategies, including the unique shortest vector problem (usvp), decoding (dec), and dual attacks. Each strategy offers a distinct perspective on the problem's hardness.

Subsequently, the function repeats the estimation process using the sieving attack method, which is often a more efficient attack strategy. It retrieves the root Hermite factor δ_0 for the same set of strategies (usvp, dec, and dual), thereby ensuring a comprehensive evaluation of the problem's hardness.

Finally, the function returns the maximum value of δ_0 from all simulated attacks. This value represents the most challenging scenario for an attacker, providing a conservative and reliable estimate of the MLWE problem's hardness.

The security and complexity of cryptographic schemes often hinge on several other key parameters as well, especially the ones that influence the LWE-ring [LNS21]. One of the most crucial is the modulus q. It is important to make modulus q large as it ensures a vast field size. This large field size is fundamental as it makes the underlying mathematical problems significantly more challenging to solve, thereby enhancing the security of the scheme [Alb17].

Equally important are the controlled noise parameters, represented by α and β. These parameters must strike a delicate balance: they need to add enough noise to obscure the underlying lattice structure, thereby enhancing security, while also ensuring that this noise does not introduce errors in operations, which could compromise the correctness of the scheme [Alb17].

Lastly, the structured polynomial p, typically considered in relation to q, ensures the necessary algebraic properties for the scheme's security. For instance, this can be represented as:

$$p_q = \text{np.poly1d}([1] + ([0] \times (d-1)) + [1])$$

This structure is pivotal in maintaining the robustness of the cryptographic system, as it underpins the fundamental operations and their resistance to attacks.

Together, these parameters form a cohesive framework that balances security, complexity, and correctness, ensuring the robustness of cryptographic schemes in protecting sensitive information.

4.6 Digital Signing

To ensure that the generated proof reaching the verifier has not been tampered with, digital signing is used [GMR88]. Digital key signing involves cryptographic algorithms to generate a digital signature, which can be verified by anyone with access to the corresponding public key. This process relies on asymmetric cryptography, using a pair of keys (private and public). The private key, known only to the signer, creates the signature, while the public key allows others to verify it.

A signing system contains a cryptographic system that uses pairs of keys. Each pair consists of a public key, which can be shared with everyone, and a private key, which is kept secret. It also includes a mathematical scheme for demonstrating the authenticity of digital messages or documents. A valid digital signature gives a recipient reason to believe that the message was created by a known sender (authentication) and that the message was not altered in transit (integrity).

The security of digital key signing schemes is based on complex mathematical problems that are computationally infeasible to solve without the private key. The most widely used digital signature schemes include RSA, DSA (Digital Signature Algorithm), and ECDSA (Elliptic Curve Digital Signature Algorithm). However, these schemes will be rendered useless in a post-quantum world. This work, therefore, uses an LWE lattice-based signature scheme called Crystal-Dilithium [DKL+18].

4.6.1 Crystal-Dilithium

Crystal-Dilithium digital signing is a cryptographic protocol designed to secure digital signatures against the threat of quantum computing [DKL+18]. As part of the NIST post-quantum cryptography project, it uses lattice-based cryptography to create complex mathematical problems that are resistant to quantum attacks. This ensures that digital signatures remain secure, even in an era where quantum computers could potentially break traditional encryption methods. Crystal-Dilithium is crucial for maintaining the integrity and authenticity of digital transactions and communications in the future.

The Dilithium scheme employs the "Fiat-Shamir with Aborts" methodology [Lyu09] and shares similarities with other established designs. We will now break down its components to provide an understanding of how it works.

The key generation algorithm of Dilithium constructs a $k \times \ell$ matrix A, where each entry is a polynomial in the ring $R_q = Z_q[X]/(X^n + 1)$. Here, q is set to $2^{23} - 2^{13} + 1$ and $n = 256$. The algorithm then samples random secret key vectors s_1 and s_2, with each coefficient being a small element in R_q, limited by η. The public key is computed as $t = As_1 + s_2$. All operations are performed within the polynomial ring R_q.

The signing algorithm creates a masking vector y composed of polynomials with coefficients smaller than γ_1. The parameter γ_1 is chosen to balance security (preventing the secret key from being revealed) and resistance to forgery. The signer computes Ay and determines w_1 as the "high-order" bits of the

resulting coefficients. Each coefficient w in Ay is expressed as $w = w_1 \cdot 2\gamma_2 + w_0$ where $|w_0| \leq \gamma_2$; w_1 represents all such high-order coefficients.

The challenge c is then formed as the hash of the message and w_1. The output c is a polynomial in R_q with exactly 60 $\pm$1s, the rest being zeros, ensuring a small norm and a domain size greater than 2^{256}. The candidate signature is computed as $z = y + cs_1$.

To prevent the secret key from being exposed, rejection sampling is used. The parameter β represents the maximum possible coefficient of cs_i. Given that c contains 60 $\pm$1s and the largest coefficient in s_i is η, $\beta \leq 60\eta$. If any z coefficient exceeds $\gamma_1 - \beta$ or if any low-order coefficient of $Az - ct$ exceeds $\gamma_2 - \beta$, the signing process restarts. This ensures security and correctness. The expected number of retries ranges from 4 to 7.

The basic Dilithium protocol without its optimizations can be found in Figure 4.3.

Gen

$\mathbf{A} \leftarrow R_q^{k \times \ell}$

$(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^\ell \times S_\eta^k$

$\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$

return $(pk = (\mathbf{A}, \mathbf{t}), sk = (\mathbf{A}, \mathbf{t}, \mathbf{s}_1, \mathbf{s}_2))$

Sign(sk, M)

$\mathbf{z} := \perp$

while $\mathbf{z} = \perp$ **do**

  $\mathbf{y} \leftarrow S_{\gamma_1 - 1}^\ell$

  $\mathbf{w}_1 := \mathsf{HighBits}(\mathbf{A}\mathbf{y}, 2\gamma_2)$

  $c \in B_{60} := \mathsf{H}(M \parallel \mathbf{w}_1)$

  $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$

  if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathsf{LowBits}(\mathbf{A}\mathbf{y} - c\mathbf{s}_2, 2\gamma_2)\|_\infty \geq \gamma_2 - \beta$, then $\mathbf{z} := \perp$

return $\sigma = (\mathbf{z}, c)$

Verify$(pk, M, \sigma = (\mathbf{z}, c))$

$\mathbf{w}_1' := \mathsf{HighBits}(\mathbf{A}\mathbf{z} - c\mathbf{t}, 2\gamma_2)$

if return $[\![\|\mathbf{z}\|_\infty < \gamma_1 - \beta]\!]$ **and** $[\![c = \mathsf{H}(M \parallel \mathbf{w}_1')]\!]$

Figure 4.3: The basic Dilithium protocol [DKL+18].

Dilithium was designed to be secure, practical, and adaptable. To ensure simplicity and security, it uses uniform sampling instead of discrete Gaussian distribution, which is challenging to secure against side-channel attacks. All operations, including polynomial multiplication and rounding, are executed in constant time to prevent timing attacks.

The scheme adopts a conservative approach to parameter selection, prioritizing long-term security. It accounts for potential quantum algorithms that, though currently unrealistic due to high space and time demands, might become feasible in the future. This foresight ensures Dilithium's resilience against evolving computational threats.

Dilithium also aims to minimize the size of the public key and signature, which is crucial for applications like certificate chains. It achieves the smallest known combination of these sizes among post-quantum signature schemes that avoid discrete Gaussian sampling.

Finally, Dilithium is designed to be modular and easy to adjust for different security levels. The primary operations are the expansion of an extendable output function (XOF) using SHAKE-128 and SHAKE-256, and polynomial ring multiplication in $Z_q[X]/(X^n+1)$. These operations are optimized for constant-time execution, and the scheme uses consistent ring parameters across all security levels. This makes it straightforward to scale security by adjusting the number of operations and XOF expansions, allowing easy adaptation of optimized implementations to various security requirements.

4.7 Verification

After a proof has been generated, it needs to be verified to ensure its integrity and validity. The verifier receives a proof package containing the encrypted QAP balancing elements c_1 (left side) and c_2 (right side), a public key for digital signing, and the digital signature itself. The verification process involves two critical checks.

First, the verifier performs Dilithium verification [DKL+18] to ensure the proof has not been tampered with. This involves checking the digital signature against the provided public key. If the Dilithium verification is successful, it confirms the authenticity and integrity of the proof package. This step is crucial because it ensures that the proof package received by the verifier is exactly what was sent by the prover, maintaining the trustworthiness of the proof.

Next, the verifier checks whether $c_1 - c_2 = 0$ by leveraging the homomorphic properties of the encryption scheme [Reg09]. This check is performed without decrypting the data, thanks to the homomorphic nature of the encryption. If $c_1 - c_2 = 0$ holds true, it indicates that the prover possesses a secret witness. This witness is a secret value s that satisfies the equations $(Us \cdot Vs - Ws) \mod T = 0$ and $Us(\alpha) \cdot Vs(\alpha) - Ws(\alpha) = T \cdot H$. This condition confirms that the prover knows the solution to the problem, thus validating the proof.

Only if both conditions, $c_1 - c_2 = 0$ and successful Dilithium signature verification, are met can the verifier confirm the validity of the witness. This dual verification process ensures that the prover's knowledge is validated while maintaining both security and confidentiality. The pseudocode for vesper's signing method can be found in Appendix E.

4.7.1 Dilithium Verification

The proof is only valid if the verifier can ensure that it has not been tampered with by checking the validity of the digital signature.

To verify the Dilithium digital signature, the verifier computes w_0^1 as the high-order bits of $Az - ct$ and accepts if all z coefficients are below $\gamma_1 - \beta$ and if c matches the hash of the message and w_0^1 [DKL+18]. Verification is effective because $Az - ct = Ay - cs_2$. This means we need to verify:

$$\text{HighBits}(Ay, 2\gamma_2) = \text{HighBits}(Ay - cs_2, 2\gamma_2).$$

This holds because a valid signature ensures that the infinity norm of the low-order bits of $Ay - cs_2$ is less than $\gamma_2 - \beta$. Since the coefficients of cs_2 are less than β, adding cs_2 will not increase any low-order coefficient beyond γ_2. Therefore, this verification process ensures that the signature is correct and has not been tampered with.

Chapter 5

The Vesper Smart Contract and its Components

During this project, two versions of the Vesper smart contract have been made. One version (Vesper Smart Contract) uses Crystal-Dilithium [DKL+18] for the Digital signing and is only SGX protected on the proof generation side. The other version (Vesper-FPC) uses a self-made LWE based digital signature scheme and both the prover and deployed chaincode (the verifier) are SGX protected. The security of these smart contract rests upon 3 things: The ZKP from chapter 4, intel SGX [CD16] protection and the security features and benefits of a permissioned blockchain like the Hyperledger Fabric [ABB+18]. The the general shared project structure and is components can be found in Figure. The rest of this chapter will focus on the Vesper Smart Contract and its components: SGX, deploying Vesper in the Hyperledger Fabric, and interacting with the deployed smart contract. Vesper-FPC is introduced in the next chapter.

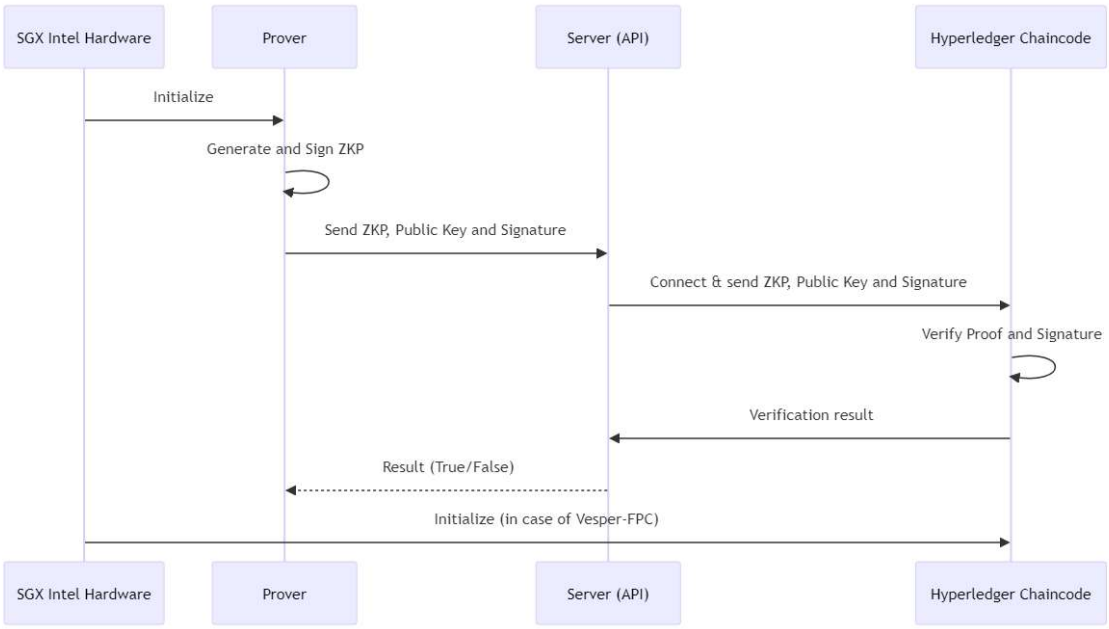

Figure 5.1: The general project structure for both Vesper and Vesper-FPC.

5.1 Intel SGX

Intel Software Guard Extensions (SGX) [CD16] is a set of security-related instruction codes built into some modern Intel CPUs. SGX enables the creation of secure enclaves, which are protected areas

of execution in memory that ensure the integrity and confidentiality of data and code, even in the presence of malicious software. As cyber threats become increasingly sophisticated, the importance of hardware-based security solutions like SGX cannot be overstated.

At its core, Intel SGX allows developers to execute their application code within a trusted execution environment (TEE) called an enclave. These enclaves are isolated from the rest of the system, including the operating system and hypervisors, significantly reducing the attack surface. The enclave's memory is encrypted and cannot be accessed directly by any other code running on the system, providing robust defense against various attack vectors.

The functioning of SGX involves several key steps [CD16]:

1. **Enclave Creation**: Developers define parts of their application that need to run within an enclave. These sections of code are compiled into a separate binary called the enclave binary.

2. **Enclave Loading**: When the application runs, the enclave binary is loaded into a secure memory region. This loading process includes verification checks to ensure the code has not been tampered with.

3. **Secure Execution**: The CPU ensures that the code within the enclave executes securely and in isolation. Any attempt to access the enclave's memory from outside is denied.

4. **Attestation**: This crucial feature allows remote parties to verify that an enclave has been correctly instantiated on a genuine Intel processor with SGX capabilities.

Applications of Intel SGX span various domains, including cloud computing and secure data processing. Notable applications include:

1. **Cloud Security**: SGX is valuable in cloud environments where sensitive data is processed. By using SGX, cloud service providers can assure clients that their data remains confidential, even if the underlying infrastructure is compromised. For example, SGX can protect data during processing in a multi-tenant cloud environment, ensuring that one tenant's data cannot be accessed by another.

2. **Blockchain and Cryptocurrencies**: SGX can enhance the security of blockchain operations by protecting cryptographic keys and algorithms. This reduces the risk of key theft and other attacks, thereby increasing overall trust in the blockchain infrastructure.

3. **Secure Web Browsing**: Web browsers can use SGX to isolate sensitive operations, such as password management and secure transactions, from the rest of the system. This protects users from browser-based attacks like man-in-the-browser.

4. **Confidential Computing**: Organizations that handle highly sensitive information, such as financial institutions or healthcare providers, can use SGX to process data confidentially. This is particularly important for complying with regulations that mandate strict data privacy and security standards.

Intel SGX offers several significant advantages:

1. **Enhanced Security**: SGX provides a high level of security through hardware-based isolation. This ensures that even if the OS or hypervisor is compromised, the data and code within the enclave remain secure.

2. **Performance**: Unlike software-based security solutions that can introduce significant performance overhead, SGX is designed to minimize such impacts. The CPU efficiently handles the encryption and decryption of enclave memory, ensuring that applications can run securely without a substantial performance penalty.

3. **Scalability**: SGX can be utilized in various scenarios, from personal devices to large-scale cloud infrastructures, making it a versatile solution for different security needs.

4. **Attestation**: The ability to remotely verify the integrity of enclaves adds an additional layer of trust. This is particularly beneficial for applications involving multiple parties who need assurance that their computations are being performed securely.

Despite its numerous advantages, Intel SGX has some limitations:

1. **Memory Constraints**: Enclaves are limited in size due to the secure memory area being relatively small. This can pose integration challenges for applications that require large amounts of memory.

2. **Complexity**: Developing applications that leverage SGX can be complex. Developers need to carefully partition their code and ensure that the enclave code is minimal and contains only the most sensitive operations.

3. **Vulnerabilities**: While SGX provides strong isolation, it is not invulnerable to all forms of attacks. Rollback attacks, where an adversary compromises the integrity of a protected application state by replaying old persistently stored data or initiating multiple application instances, represent a potential threat to SGX [MAK+17]. Additionally, side-channel attacks, such as those exploiting timing or power consumption patterns, can potentially leak information from enclaves [OTK+18].

4. **Adoption Barriers**: The need for compatible hardware and software support can hinder the widespread adoption of SGX. Organizations may need to invest in new hardware and update their software to take full advantage of SGX capabilities.

As cyber threats continue to evolve, the role of technologies like Intel SGX in safeguarding digital assets will become increasingly critical. By providing a secure foundation for application execution, SGX helps build a more secure and trustworthy computing environment, paving the way for innovations that rely on the confidentiality and integrity of data.

5.1.1 SGX Usage and Experience

To protect your code with SGX, you need the SGX Software Development Kit (SDK) and the Platform Software (PSW). The SGX SDK and PSW are fundamental tools for creating and running SGX applications [Cor24a]. These components provide the necessary libraries, tools, and runtime support for enclave development and execution.

The SDK includes a variety of development tools and libraries to facilitate the creation of SGX enclaves. It provides header files, sample codes, and comprehensive documentation to help developers understand and utilize SGX features effectively. The SDK also includes tools for signing enclaves, which is a critical step in the enclave lifecycle.

The PSW is required for the runtime execution of SGX applications. It includes the SGX runtime libraries and the SGX driver, which are necessary for managing enclaves and interacting with the CPU's SGX features. The PSW ensures that the enclaves are properly initialized, maintained, and terminated.

These tools can be downloaded from the official Intel SGX website [Cor24b]. It is important to choose the correct version of the SDK and PSW that matches your development platform and operating system.

Enabling SGX in the BIOS is a one-time setup step but is crucial for ensuring that the development environment can fully utilize SGX capabilities. If SGX is not an option in the BIOS, installing the Intel Software Guard Extensions Activation App can resolve this issue. After installation, SGX can be activated from the BIOS, allowing the SDK and PSW to be installed.

A compiler that supports SGX extensions is required to build SGX applications. The choice of compiler depends on the operating system used for development:

- **Linux:** The GNU Compiler Collection (GCC) is commonly used for SGX development on Linux platforms. GCC provides the necessary support for SGX instructions and can be configured to work with the SGX SDK.

- **Windows:** Microsoft Visual Studio is the preferred compiler for SGX development on Windows platforms. Visual Studio integrates well with the SGX SDK, offering tools for compiling, debugging, and signing SGX enclaves.

Most SGX SDK examples and documentation are provided in C/C++. This language is preferred for low-level system programming and performance-critical applications. Ensure proper use of pointers and memory management when dealing with enclave boundaries to avoid security vulnerabilities.

While Python is not typically used for core SGX enclave development due to performance considerations, it can be used for scripting and orchestration around SGX applications. Tools like `py-sgx` can help interface Python with SGX for specific use cases. Additionally, frameworks like `gramine` can make it more convenient to run Python and JavaScript applications within SGX enclaves.

Because Hyperledger Fabric only accepts Java, JavaScript, or Go chaincode, Vesper Smart Contract was written in JavaScript. Protecting JavaScript code with SGX can be complex since SGX is C/C++ based, so an SGX wrapper called Gramine was used in this work to reduce complexity.

5.2 Gramine

Gramine (formerly known as Graphene) is an open-source library operating system designed to facilitate the secure execution of applications [TPV17, Sch21]. Leveraging hardware security features such as Intel SGX (Software Guard Extensions), Gramine provides a lightweight and flexible environment for running applications securely. It addresses the challenges of protecting sensitive data and code in untrusted environments, making it an essential tool for developers aiming to enhance the security of their applications.

Gramine operates by providing a minimalistic library operating system that runs applications in an isolated environment. This isolation ensures that the application's execution is protected from external interference, including attacks from the host operating system and other applications. Gramine can run unmodified applications designed for Linux, which makes it highly compatible and easy to integrate. In this project, Gramine is mainly used as an SGX wrapper to reduce its complexity and to support SGX usage with languages other than C/C++ [TPV17].

The key components of Gramine include:

- **Library OS**: Gramine provides a library OS that includes essential components such as system call handlers, file system support, and network stack. This library OS acts as an intermediary between the application and the underlying hardware.
- **Shielded Execution**: Gramine leverages hardware features like Intel SGX to create shielded execution environments. These environments, or enclaves, ensure that the application's code and data are protected from unauthorized access and tampering.
- **Runtime Loader**: Gramine includes a runtime loader that initializes the library OS and the application within the enclave. The loader ensures that the application runs securely by verifying the integrity of the code before execution.
- **Remote Attestation**: Similar to SGX, Gramine supports remote attestation, allowing remote parties to verify the integrity and authenticity of the enclave before sharing sensitive information.

Gramine offers several key advantages for secure application execution:

- **Compatibility**: One of the major advantages of Gramine is its ability to run unmodified Linux applications. This compatibility reduces the development effort required to secure existing applications.
- **Security**: By leveraging hardware security features like Intel SGX, Gramine provides strong isolation and protection against a wide range of attacks, including those originating from the host OS.
- **Flexibility**: Gramine's library OS approach allows developers to customize the environment according to their application's needs. This flexibility is beneficial for applications with specific security and performance requirements.
- **Remote Attestation**: The support for remote attestation enhances trust between parties by allowing them to verify the integrity and authenticity of the enclave before engaging in sensitive operations.

Despite its advantages, Gramine has some limitations [LnLnLn22]:

- **Performance Overhead**: The additional layer of the library OS and the encryption/decryption operations can introduce performance overhead. While Gramine is designed to minimize this, some performance impact is inevitable.
- **Complexity**: Setting up and configuring Gramine for specific applications can be complex, especially for developers unfamiliar with enclave technology and secure computing practices.
- **Hardware Dependency**: Gramine's security benefits are heavily reliant on the underlying hardware features, such as Intel SGX. This dependency means that its effectiveness is limited to systems with compatible hardware.
- **Memory Constraints**: Similar to other enclave-based solutions, the secure memory available for enclaves is limited. This can be a constraint for applications requiring large amounts of memory.

5.2.1 Gramine Development

To protect this project's zero-knowledge proof, Gramine SGX protection is used [Sch21, CD16]. In Gramine, a manifest template is used to describe the configuration and security policies of the application that will run inside the enclave. The template is a text file with a specific syntax that outlines various settings such as file paths, environment variables, and SGX-specific parameters.

An example of a manifest template:

```
loader.preload = "file:/libsysdb.so"
loader.env.LD_LIBRARY_PATH = "/lib:/usr/lib"

sgx.enclave_size = "256M"
sgx.thread_num = 8
sgx.heap_size = "128M"

fs.mount.lib.type = "chroot"
fs.mount.lib.path = "/lib"
fs.mount.lib.uri = "file:/path/to/lib"

fs.mount.libusr.type = "chroot"
fs.mount.libusr.path = "/usr/lib"
fs.mount.libusr.uri = "file:/path/to/usr/lib"

sgx.trusted_files = ["/bin/app", "/lib/libc.so.6", "/lib/libsysdb.so"]
```

The manifest template is processed by Gramine to produce a final manifest file. This file contains all the configuration details required to run the application inside the enclave. The manifest file includes both the settings from the template and additional metadata generated by Gramine.

The process of converting a manifest template to a manifest file involves the use of the `gramine-manifest` tool:

```
gramine-manifest -Dentrypoint=/bin/app -Darch_libdir=/lib \
    -Darch_libdir_usr=/usr/lib manifest.template
```

This command will produce a file named `manifest.sgx` which includes all the necessary details for running the application inside an SGX enclave.

Before signing the manifest, you need to generate a personal key. This key is used to create a cryptographic signature that ensures the integrity and authenticity of the enclave configuration.

To generate a personal key, use the `gramine-sgx-gen-private-key` tool. This command activates the Gramine SGX key generator and saves the key. This key will be used for signing the manifest file.

Once you have generated your personal key, the next step is to sign the manifest file. The signing process involves creating a cryptographic signature of the manifest file using the SGX signing tool and your personal key.

The `gramine-sgx-sign` tool is used for this purpose:

```
gramine-sgx-sign --key signing.key --manifest manifest.sgx --output manifest.sgx.sig
```

The signing process generates a signature file (`manifest.sgx.sig`) that is used to verify the manifest when the enclave is created.

Once the manifest is signed, the application can be launched inside an SGX enclave. The Gramine loader initializes the enclave, verifies the manifest signature, and sets up the runtime environment according to the configurations specified in the manifest.

To run the application, use the Gramine loader:

```
gramine-sgx ./app
```

In case it is a Python file, specify this as follows:

```
gramine-sgx ./python app
```

The overall process of Gramine development simplifies the creation and management of SGX enclaves by automating many of the complex steps involved in enclave configuration and execution. By using manifest templates, developers can easily specify application requirements and security policies, while the Gramine tools handle the details of enclave creation and management.

5.2.2 Gramine: User Experience and Troubleshooting Challenges

After deciding to try out Gramine for this project, the steps outlined above were followed. The template manifest was created containing all paths to libraries and environments used by the ZKP. This template was then used to generate the final manifest file and was signed using my personal key. By executing the ZKP with the `gramine-sgx` command, the proof would be generated under SGX protection. However, an issue arose after the Dilithium digital signature was added to the ZKP. The Open Quantum Safe (OQS) framework [Ope] that contains Crystal-Dilithium [DKL+18] was not compatible with SGX. Several causes contributed to this issue.

Firstly, integrating computationally intensive post-quantum algorithms like Crystal Dilithium into SGX enclaves presents several challenges. SGX enclaves have limited memory (128 MB in SGX2), restricting the space available for such complex algorithms. Additionally, the instruction set limitations of SGX enclaves necessitate software-based implementations of certain CPU instructions, further increasing computational burden and complexity.

Secondly, the development environment for SGX enclaves is inherently complex and difficult to debug due to the need for maintaining secure boundaries and limited debugging tools. Adding a sophisticated post-quantum algorithm exacerbates these difficulties, requiring advanced debugging and testing methods to ensure correctness and security.

Furthermore, post-quantum algorithms typically have larger codebases. The constrained size of SGX enclaves may struggle to accommodate these large codebases without significant optimization, potentially compromising performance or security.

To overcome this compatibility issue, a specific Dilithium version was used that removed the need for an additional Open Quantum Safe (OQS) wrapper [Sar24]. To use this SGX compatible Dilithium library, the lattice-based ZKP was translated into JavaScript, and `@asanrom/dilithium` was installed and imported. This solution worked well with `gramine-sgx`, and Crystal-Dilithium signing [DKL+18] could now be integrated into the project.

5.3 Hyperledger Fabric

In digital technology, blockchain is a significant innovation, providing secure, transparent, and decentralized data management. Hyperledger Fabric, a platform developed under the Linux Foundation's Hyperledger project, is a notable example. This platform features a modular architecture that ensures confidentiality, resilience, flexibility, and scalability, essential for enterprise applications.

Blockchain technology originated as the underlying infrastructure for Bitcoin in 2008 [Nak08b]. While initial blockchains were predominantly public and permissionless, designed for cryptocurrency transactions, they lacked features essential for enterprise use, such as enhanced security, privacy, and controlled access.

To address these limitations, the Linux Foundation initiated the Hyperledger project in 2015 [ABB+18]. The objective was to develop cross-industry blockchain technologies, providing a suite of robust frameworks, tools, and libraries. Hyperledger Fabric emerged as a key framework within this initiative, tailored specifically for enterprise requirements.

Hyperledger Fabric is distinguished by its modular and configurable architecture, allowing businesses to tailor the blockchain network to their specific needs. The principal architectural components of Hyperledger Fabric include:

- **Membership Service Provider (MSP)**: The MSP defines the identities of participants within the network, providing authentication and authorization. This ensures that only approved entities can participate in the blockchain.
- **Ordering Service**: This component maintains the chronological order of transactions across the network. It collects endorsed transactions from different peers, orders them into blocks, and ensures their correct sequence.
- **Peers**: Peers are nodes in the network that host ledgers and smart contracts (also known as chaincode). They are responsible for endorsing transactions and maintaining the ledger's state.
- **Chaincode**: Chaincode refers to smart contracts in Hyperledger Fabric. These are programs written in languages such as Go, Java, or Node.js, and they define the business logic that governs the interactions and state changes in the ledger.
- **Channels**: Channels enable the partitioning of the network, allowing a subset of participants to create a separate ledger of transactions. This ensures data privacy and confidentiality, as only members of a channel can access its ledger.
- **Ledger**: The ledger in Hyperledger Fabric consists of two parts: the world state and the transaction log. The world state represents the current state of the ledger, while the transaction log records all state changes.

Hyperledger Fabric employs a unique approach to consensus, differing from traditional blockchain platforms. It decouples the transaction endorsement and ordering processes, which enhances scalability and performance [ABB+18]. The consensus process in Hyperledger Fabric involves three stages:

- **Endorsement**: Transactions are proposed by clients and endorsed by a subset of peers according to endorsement policies defined by the network. Endorsement policies specify the required number and type of endorsements needed for a transaction to be considered valid.
- **Ordering**: Endorsed transactions are submitted to the ordering service, which organizes them into blocks. The ordering service can be implemented using various consensus algorithms, such as SOLO (a single-node ordering service for testing), Kafka (a crash fault-tolerant service), and Raft (a leader-based consensus algorithm).
- **Validation and Commit**: Once ordered, blocks of transactions are distributed to all peers in the network. Peers validate the transactions against endorsement policies and the current world state. Valid transactions are then committed to the ledger, and the world state is updated accordingly.

A high level overview of the transaction flow of Hyperledger fabric can be seen below in Figure 5.2.

Hyperledger Fabric supports various methods of communication between client applications and the blockchain network, enhancing its flexibility and usability [ABB+18]:

- **SDK**: Software Development Kits (SDKs) facilitate communication between client applications and the blockchain network, providing necessary libraries and tools.
- **Fabric Command Line Interface (CLI)**: Communication can be established simply using command line commands, allowing for straightforward interaction with the network.
- **REST API with Hyperledger Fabric Gateway**: This method provides better mechanisms for managing user identities and access control. It simplifies communication with the deployed chaincode and is often preferred due to its comprehensive management capabilities, despite the complexity associated with setting up SDK communication.

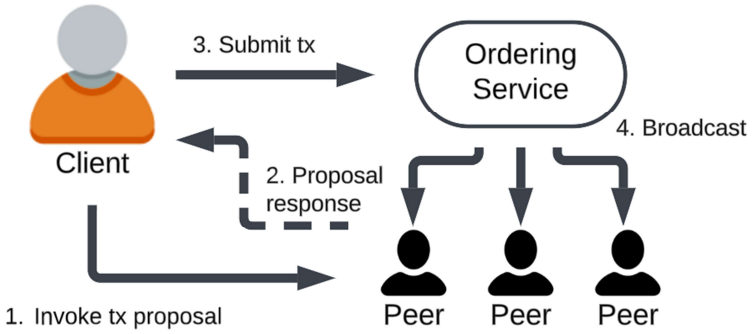

Figure 5.2: A high level depiction of a Fabric Network and its transaction flow [BCKS18].

Hyperledger Fabric offers several advantages that make it a preferred choice for enterprise blockchain solutions:

- **Modularity**: The modular architecture allows businesses to select and configure components according to their specific requirements, enhancing flexibility and customization.
- **Permissioned Network**: Hyperledger Fabric operates as a permissioned network, ensuring that only authorized participants can access the blockchain. This enhances security and control, which is crucial for enterprise applications.
- **Scalability**: By decoupling the transaction endorsement and ordering processes, Hyperledger Fabric achieves high scalability, accommodating a large number of transactions without compromising performance.
- **Data Privacy**: The use of channels and private data collections enables Hyperledger Fabric to provide data privacy and confidentiality, ensuring that sensitive information is accessible only to authorized parties.
- **Support for Complex Business Logic**: Hyperledger Fabric's chaincode can be written in widely-used programming languages, allowing businesses to implement complex business logic and integrate existing systems seamlessly.

Despite its numerous advantages, Hyperledger Fabric faces certain challenges:

- **Complexity**: The modular architecture, while offering flexibility, can also introduce complexity in deployment and management. Organizations may require specialized expertise to effectively implement and maintain Hyperledger Fabric networks.
- **Interoperability**: Interoperability with other blockchain platforms and legacy systems remains a challenge. Efforts are ongoing to develop standards and protocols that facilitate seamless integration across different blockchain ecosystems.
- **Performance Overheads**: Although Hyperledger Fabric is designed for scalability, certain use cases with extremely high transaction volumes may still encounter performance bottlenecks. Continuous optimization is necessary to address these issues.

The future of Hyperledger Fabric appears promising, with ongoing developments aimed at enhancing its performance, security, and interoperability. The integration of advanced technologies such as artificial intelligence and the Internet of Things (IoT) with Hyperledger Fabric holds the potential to unlock new possibilities, further solidifying its role in the digital transformation of businesses.

Hyperledger Fabric represents a significant advancement in blockchain technology, providing a robust, flexible, and scalable platform tailored for enterprise use. Its modular architecture, permissioned network, and support for complex business logic make it an ideal choice for a diverse range of applications across various industries. While challenges persist, the continuous evolution of Hyperledger Fabric promises to address these issues and expand its capabilities. As businesses increasingly seek to leverage blockchain technology, Hyperledger Fabric is poised to play a pivotal role in shaping the future of digital commerce and data management.

5.3.1 Hyperledger Fabric Performance Analyses

The work of Murat Kuzlu et al. [KPGR19] thoroughly analyses the performance of the Hyperledger Fabric blockchain framework. They use a benchmark tool called Hyperledger Caliper [Hyp24d], which can represent multiple client threads, meaning it can simulate many clients injecting workloads into the blockchain network, as visualized in Figure 5.3.

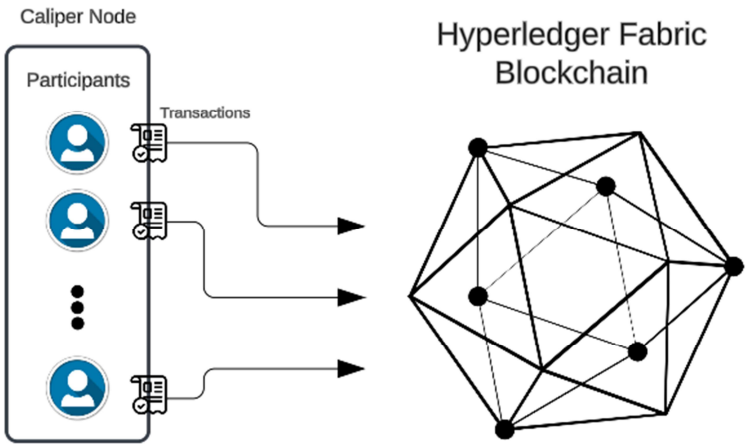

Figure 5.3: Hyperledger Caliper.

To set up the Caliper benchmark tool, a configuration file was used to vary transaction rates, numbers, and types. Transaction number (*txNumber*) defines the number of transactions per round. Transaction rate (*rateControl*) specifies the fixed interval of transactions per second (TPS). Transaction types include "open" (read and write) and "query" (read). The network configuration file defines parameters such as organizations and peers. The benchmark experiments were executed on the following execution environment: An AWS EC2 instance with 16 vCPUs, Intel Xeon processors, and 32GB RAM ran the benchmark. Ubuntu 16.04 LTS was used along with Fabric release v1.4.

The performance test involved varying transaction rates, numbers, and simultaneous transactions across three cases (see Table 5.1).

- **Case I**: Varying transaction rates (100-300 TPS) with 1000 total transactions.
- **Case II**: Varying transaction numbers (1000, 10000, 100000) at a fixed 200 TPS.
- **Case III**: Varying simultaneous transactions (100, 200, 300).

Key parameters	Transaction per second (TPS)	Number of transactions	Simultaneous transactions
Case I: Impact of test environment and transaction rates	100, 150, 200, 250, 300 TPS	1,000	N/A
Case II: Impact of number of transactions	200 TPS	1,000, 10,000, 100,000	N/A
Case III: Impact of simultaneous transactions	Equivalent to number of simultaneous transactions	N/A	100, 200, 300

Table 5.1: Parameters for performance evaluation [KPGR19].

In all cases, both open and query transactions were tested with two organizations and the performance was evaluated based on throughput, latency, and scalability.

5.3.2 Case I Results: The Impact of Transaction Rates

The performance analysis shows that for *open* transactions (one read, one write), the network could handle up to 200 TPS without significant latency. Beyond this rate, throughput decreases and latency increases. For *query* transactions (one read), the network managed 300 TPS with minimal latency, indicating it could support even higher transaction rates without delay.

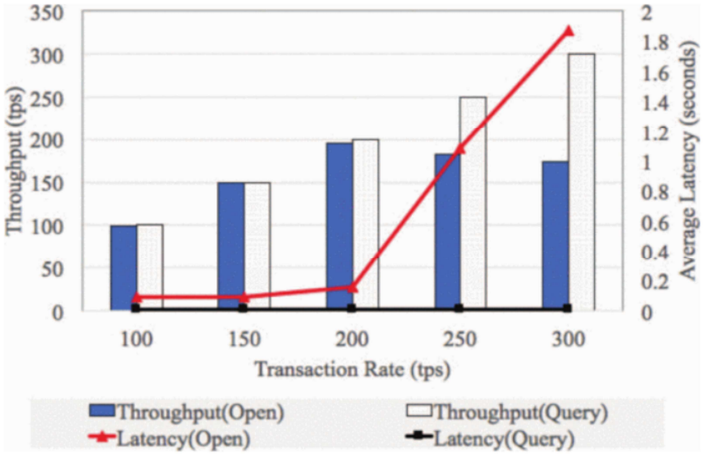

Figure 5.4: Case I Results [KPGR19].

5.3.3 Case II Results: The Impact of the Number of Transactions

Results in Figure 5.5 show that the number of transactions has no significant impact on throughput, which remains constant at 200 TPS. Latency for "query" transactions stays at 0.01 seconds, while

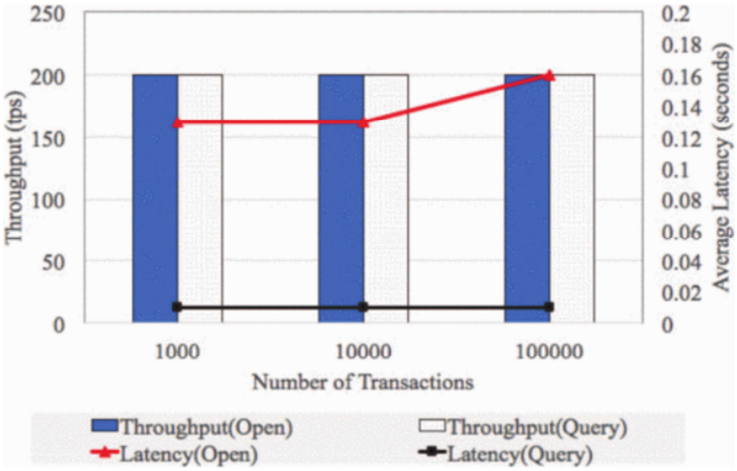

Figure 5.5: Case II Results [KPGR19].

for "open" transactions it slightly increases to 0.13-0.16 seconds. The findings suggest that a high number of transactions can be supported with minimal latency, especially for read-only operations. The scalability of the Hyperledger platform depends on hardware and network configurations, with the potential to support 100,000 participants or even more with adequate resources.

5.3.4 Case III Results: The Impact of Simultaneous Transactions

Figure 5.6 shows that with 100, 200, and 300 participants, both throughput and latency increase for "open" and "query" transactions. For "open" transactions, throughput increases to 34.6, 40.3, and 44.8 TPS respectively, but latency also rises, indicating the system's maximum capacity. For "query" transactions, higher throughput and lower latency can be observed, handling up to 200 simultaneous transactions without queuing.

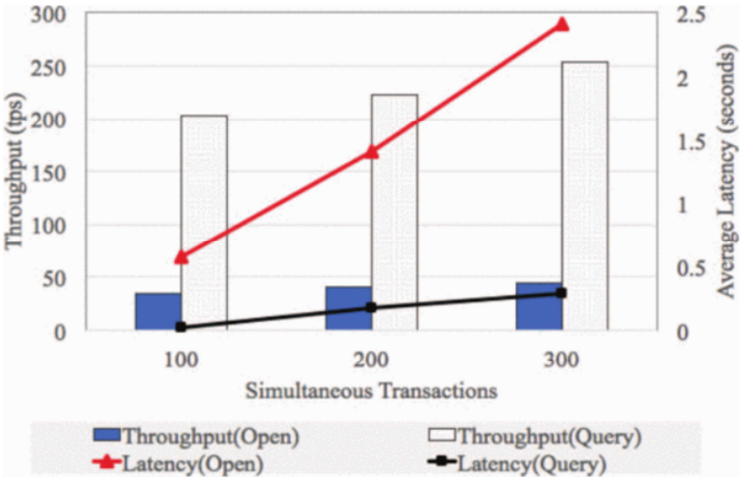

Figure 5.6: Case III Results [KPGR19].

These results highlight that transaction type and the number of simultaneous transactions significantly impact blockchain performance, particularly latency, with throughput remaining flat at the system's limit and latency increasing as the system reaches its maximum capacity.

5.3.5 Improving Performance Through Peers

Adding more peers can improve throughput by distributing the transaction validation and endorsement workload across more nodes. This allows more transactions to be processed concurrently [SWTR18].

Increasing the number of peers also enhances fault tolerance and network reliability. If some peers fail or are temporarily unavailable, the network can continue to operate smoothly with the remaining peers, ensuring higher uptime and resilience [SWTR18].

However, the benefit plateaus after a certain number of peers due to increased communication overhead and complexity in consensus mechanisms. More peers can increase network latency because transactions must be propagated and confirmed across a larger number of nodes. Each peer must validate transactions and maintain consensus, which takes more time as the number of peers grows. As the number of peers increases, the consensus process also becomes more complex [SWTR18]. This can lead to longer times for reaching consensus on transactions, especially in environments with stringent endorsement policies or high network latency.

Performance also improves when more than one channel is being used, as a peer can achieve more throughput if more than one channel is utilized [Fou23]. When a single channel is used, all transactions must pass through this one channel. This creates a bottleneck, limiting the number of transactions that can be processed concurrently. The throughput is thus constrained by the channel's capacity to handle transactions sequentially. Peers in Hyperledger Fabric are designed to handle multiple channels, each representing a separate logical ledger [TNV19]. When only one channel is in use, the peer's computational and storage resources are not fully utilized because they are dedicated to managing just one set of transactions.

5.4 Deploying the Smart Contract

The goal is to deploy the verification side of Vesper as a smart contract into the Hyperledger Fabric. However, before we can test the developed chaincode (i.e., the smart contract), a Hyperledger Fabric network needs to be brought up [ABB+18].

The key components of a Hyperledger Fabric network are:

- **Orderer**: The orderer serves as the backbone of the network, ensuring the consistency and ordering of transactions. It collects transactions from various peers and arranges them into blocks, which are subsequently added to the blockchain. The orderer can be configured to utilize different consensus mechanisms, such as Raft, Kafka, or a simple solo orderer for testing purposes.
- **Peers**: Peers are nodes that maintain a copy of the ledger and validate transactions. There are two principal types of peers:
  - **Endorsing Peers**: These simulate and sign transactions.
  - **Committing Peers**: These commit transactions to the ledger.

  Peers ensure the integrity and authenticity of transactions by validating them in accordance with the endorsement policy.
- **Certificate Authority (CA)**: The CA issues digital certificates to network participants, enabling secure identity management. Each participant must possess a certificate to interact with the network, ensuring that only authorized entities can perform actions.
- **Channels**: Channels are private communication pathways that allow specific network participants to conduct transactions confidentially. Each channel maintains its own ledger, providing data isolation and privacy.

The smart contract that this network deploys contains the verification side Vesper. The pseudocode for this can be found in Appendix F. It consists of zero-knowledge proof verification and digital signature verification [Sar24]. This is beneficial because, besides all the security measures already in place, the smart contract itself does not contain sensitive data. The data that it later receives from the prover side is already encrypted, and the verifier does not need to decrypt it due to the homomorphic nature of the proof generation. Thus, we aim to deploy an efficient verifier in Hyperledger Fabric.

As mentioned, to deploy the smart contract as chaincode on Hyperledger Fabric, a network needs to be brought up first. This can be easily done by cloning the Fabric-Samples repository [Hyp24a], navigating to their test-network directory, and bringing up their test network with the following command: `./network.sh up createChannel`. This will start the Docker containers for the peers and other network components like the orderer. Additionally, it will create a channel and join a peer from each organization to that channel.

Channels provide a private communication layer for specific network members. Only organizations invited to the channel can use it, and it remains invisible to other network members. Each channel has its own blockchain ledger. Invited organizations "join" their peers on the channel to store the channel ledger and validate transactions.

With the network up, a channel created, and the peers joined, the chaincode can now be deployed. To deploy chaincode in Hyperledger Fabric, follow these steps:

1. **Package the Chaincode**: Package the chaincode into a `.tar.gz` file using the following command:

   ```
   peer lifecycle chaincode package mycc.tar.gz --path github.com/chaincode_path
   --label mycc_1
   ```

2. **Install the Chaincode**: Install the packaged chaincode on each peer using the following command:

   ```
   peer lifecycle chaincode install mycc.tar.gz
   ```

3. **Extract Package ID**: Query the installed chaincode to extract the package ID using the following command:

   ```
   peer lifecycle chaincode queryinstalled
   ```

4. **Approve Chaincode**: Approve the chaincode definition for your organization using the following command:

   ```
   peer lifecycle chaincode approveformyorg --channelID mychannel --name mycc
   --version 1.0 --package-id <PACKAGE_ID> --sequence 1 --tls --cafile
   $ORDERER_CA
   ```

5. **Check Commit Readiness**: Verify that the chaincode is ready to be committed with the following command:

   ```
   peer lifecycle chaincode checkcommitreadiness --channelID mychannel--name mycc
   --version 1.0 --sequence 1 --output json
   ```

6. **Commit Chaincode**: Commit the chaincode definition to the channel using the following command:

   ```
   peer lifecycle chaincode commit --channelID mychannel --name mycc --version 1.0
   --sequence 1 --tls --cafile $ORDERER_CA --peerAddresses peer0.org1.example.com:7051
   --tlsRootCertFiles $PEER0_ORG1_CA --peerAddresses peer0.org2.example.com:9051
   --tlsRootCertFiles $PEER0_ORG2_CA
   ```

7. **Invoke Chaincode**: Initialize the chaincode by invoking an initial transaction with the following command:

   ```
   peer chaincode invoke -o orderer.example.com:7050 --tls true -cafile
   $ORDERER_CA -C mychannel -n mycc -c '{"Args":["initLedger"]}'
   ```

These commands automate the process of deploying chaincode, ensuring it is properly packaged, installed, approved, committed, and invoked on the network [Hyp24a].

5.4.1 Troubleshooting Deployment Issues

During the deployment process, an issue was encountered related to peer authentication. The paths in the test network that point to the TLS and MSP information were not configured correctly, causing the peers not to join the channel. After checking and correcting all paths in `network.sh`, the problem was resolved.

5.5 Communicating with deployed chaincode

There are several ways to establish communication with a Hyperledger Fabric network [ABB+18, Hyp24a]. One common approach is to use SDKs (Software Development Kits), which facilitate interaction between client applications and the blockchain network [Hyp24b]. These SDKs, available for languages such as Node.js, Java, Go, and Python, provide APIs that enable developers to submit transactions and query the ledger seamlessly.

Another way to interact with Hyperledger Fabric is through the Fabric Command Line Interface (CLI) [Hyp24a]. The CLI offers a set of commands that can be used to perform various administrative tasks, such as installing and instantiating chaincode. Additionally, it allows users to invoke transactions and query the ledger directly from the command line.

An alternative method, found to be particularly effective and used in this project, involves utilizing the REST API with the Hyperledger Fabric Gateway. The Gateway not only provides a straightforward way to communicate with the deployed chaincode but also offers enhanced mechanisms for

managing user identities and access control. This approach simplifies the process significantly, especially compared to setting up SDK communication, which can be quite complex.

As shown in Figure 5.1, after the proof generation and digital signature generation are completed, a JSON object is created consisting of the generated encrypted proof, the digital signature, and the public key used for signing. This JSON package is then sent via middleware to the deployed chaincode where the proof and the signature are verified. If both are valid, the response yields true; otherwise, it yields false. This middleware will now be introduced. This section presents how the middleware used to communicate between the prover and the deployed verifier works exactly.

5.5.1 The Vesper Communication Middleware

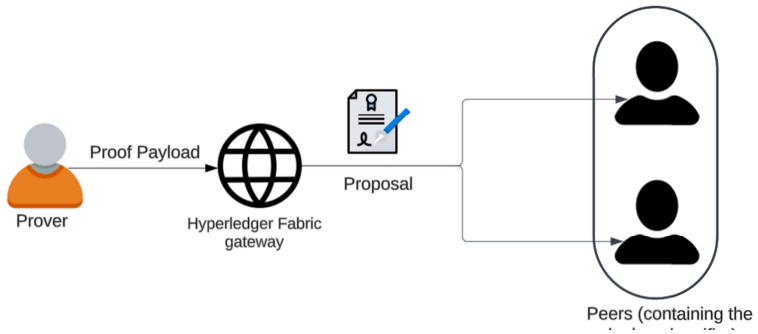

Figure 5.7: The high-level communication overview between the prover and the deployed verifier.

Figure 5.7 shows the basic communication between the prover and deployed verifier of Vesper via the Hyperledger Fabric gateway. Before the connection between prover and the gateway can be made and to ensure the gateway knows how to locate and connect with the verifier, some preparations need to be made first.

The communication middleware used in Vesper Smart Contract starts by retrieving necessary configuration details such as channel name, chaincode name, MSP ID, and file paths for cryptographic materials from environment variables or default values [Hyp24b]. These details are crucial for establishing a secure and authenticated connection to the blockchain network. Since these values are set during the process of creating a channel and deploying the chaincode, they should be known.

Secondly, it creates a gRPC connection to the peer node. This connection is secured using TLS certificates obtained from the filesystem. The gRPC client is configured with the peer endpoint and the TLS root certificate, ensuring a secure communication channel.

The middleware then reads the user's certificate and private key from the specified directories. The certificate is used to create a new identity object representing the user, while the private key is used to create a signer object [ABB+18, Hyp24b]. These are essential for authenticating the user and signing transactions.

Using the gRPC client, identity, and signer, the middleware establishes a connection to the Hyperledger Fabric gateway. This gateway acts as an intermediary, facilitating interactions with the blockchain network as shown in Figure 5.7. The gateway connection is configured with default timeouts for various operations such as evaluating, endorsing, submitting transactions, and checking commit status.

Once connected to the gateway, the middleware accesses the network (specified by the channel name) and retrieves the smart contract instance (specified by the chaincode name). This contract instance represents the deployed chaincode that the middleware intends to interact with.

Next, the middleware calls a specific function on the smart contract, `VerifyProof`, passing the necessary arguments. The arguments in this case are the encrypted proof, the signature, and the public key used to create the digital signature. This function call is made using the contract instance and is typically performed by evaluating a transaction, which reads data from the ledger without making any modifications.

Finally, the result of the transaction evaluation is processed, decoded, and parsed. The verifier can now check if the proof has been untampered with by making a call to the `VerifySignature` function

and, if so, can check if the proof itself is valid or not. The result is then returned to the client through the API endpoint.

By following these steps, the middleware securely connects to the Hyperledger Fabric network, accesses the deployed chaincode, and performs the required operations, such as verifying proofs, while ensuring secure and authenticated interactions. The pseudocode for the fabric gateway middleware used in Vesper Smart Contract can be found in Appendix G.

Chapter 6

Vesper-FPC

Blockchain protocols provide transparency and resilience, ensuring the integrity of blockchain applications. However, this transparency can conflict with the need to keep application state confidential and maintain user privacy. To enhance the Vesper Smart Contract even further, a new technology has been developed to protect the deployed verifier (the smart contract) with Intel Software Guard Extensions (SGX). This new technology that serves as an extension project to the Vesper Smart Contract is called Vesper-FPC.

Figure 6.1 below illustrates the structural overview of Vesper-FPC. [Hyp24c].

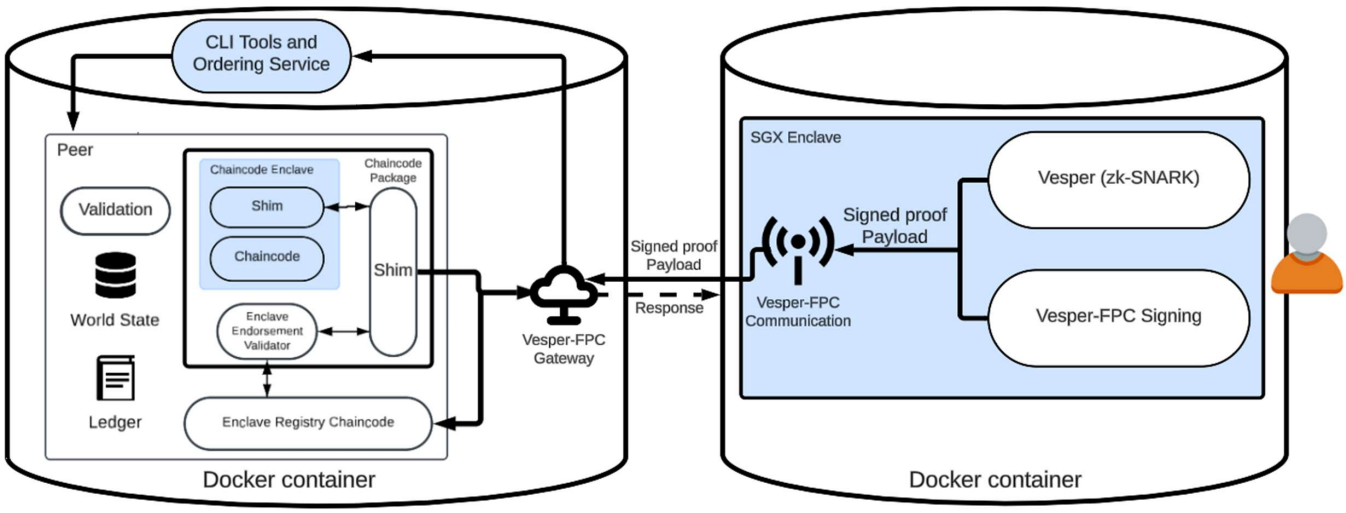

Figure 6.1: The high-level communication overview between the prover and the deployed verifier.

To create Vesper-FPC, the Fabric-Private-Chaincode framework (FPC) [Hyp24c] was utilized. FPC leverages Trusted Execution Environments (TEEs), specifically Intel SGX [CD16], to protect the privacy of chaincode data and computations, even in the presence of potentially untrusted peers. FPC is a prototype developed based on the paper by Brandenburger et al.: "Blockchain and Trusted Computing: Problems, Pitfalls, and a Solution for Hyperledger Fabric" [BCKS18]. How FPC integrates SGX into Hyperledger Fabric, what components are added to a Fabric peer, the benefits of this SGX integration, and its performance can be found in Section 3.7, where the work of Brandenburger et al. [BCKS18] is introduced.

This project extends the regular Hyperledger Fabric [ABB+18], specifically the fabric-samples framework, by integrating several key components. First, a chaincode enclave is introduced, which executes specific chaincode within an SGX environment for added security. In the peer's untrusted section, there is an enclave registry that tracks the identities of all chaincode enclaves. Additionally, an enclave transaction validator is implemented to validate transactions processed by a chaincode enclave before they are committed to the ledger.

Integrating FPC with the Vesper Smart Contract required significant modifications. The Dilithium Signature scheme [DKL+18] was replaced by a simplified LWE-based digital signature scheme. SGX SDK development was used instead of Gramine, and communication through the middleware was handled differently. Additionally, the deployed chaincode now required an SGX protection layer. These differences warranted maintaining them as two separate projects to better understand the trade-offs. Since the proof generation, except for the digital signing, remains the same in Vesper-FPC as in Vesper Smart Contract, the rest of this chapter elaborates on the development process of Vesper-FPC, primarily focusing on blockchain verification and SGX integration.

6.1 The Vesper-FPC Development Process and Troubleshooting Encountered Issues

To integrate FPC with the project, the FPC helloworld tutorial was initially followed. It was first noticed that the chaincode needed to be in C/C++. The Dilithium [DKL+18] version compatible with SGX was JavaScript-specific. Thus, the chaincode was translated, temporarily leaving out Dilithium signing, and the process continued. Besides the chaincode, a CMake file, a Makefile, a test application, and a test bash script to deploy the chaincode and run the test application had to be created. The CMake file included another CMake file that contained centralized information on how the enclave should be constructed and what it should contain.

The first issue encountered was that the test script would not succeed, not even with the helloworld example. After investigating, it was found that the test bash script did not bring up a network before trying to deploy the chaincode. FPC is an extension of fabric samples, but fabric samples were not included yet after the pull. In another tutorial, instructions were found on how to bring up a network in FPC by pulling the fabric samples repository into FPC and using that to start the network. However, another issue arose when trying to create a channel. There seemed to be something wrong with the TLS and MSP data of the peers, but it did not seem to be a configuration issue.

After contacting the Hyperledger developers [BCKS18] via their Discord server, it was discovered that they were encountering the same issue. Significant changes in Hyperledger Fabric had caused many functionalities in FPC to break.

After further research, the cause of the channel issue was identified. It was due to the binaries FPC used. Using the latest Hyperledger binaries resolved the issue.

With the network up and running, the helloworld test script worked, but the Vesper chaincode still caused an error. The issue was that json/json.h was being imported for processing the prover's info. It turned out to be extremely difficult to use libraries not already in the FPC environment due to the way the enclave is formed in this project. The only usable libraries were the ones in the shim.

The local CMake file was altered to include the library path and link it explicitly, but it still would not link with the chaincode. The way the enclave was formed prevented it from linking. The enclave formation in FPC is distributed throughout the entire project and linked with many CMake files. Including the needed path in every CMake file related to the enclave was attempted, but this resulted in different errors related to linking. Finally, the chaincode was rewritten to avoid using external libraries by replacing JSON with I/O streams. The test script for the chaincode then succeeded.

Next, digital signing needed to be added. Attempting to add the C++ liboqs [Ope] wrapper quickly made it clear that it would not be compatible with SGX. Instead, a custom LWE-based digital signing function compatible with SGX was implemented using only the functions provided by the environment to avoid any clashes with the enclave. This self-made digital signing scheme is explained further in Section 6.2 and the Vesper-FPC chaincode, including signature verification, can be found in pseudocode format in Appendix H.

With the SGX-protected smart contract now deployable, the last step was establishing communication. Due to the addition of the enclave around the chaincode, the original middleware would not work anymore. FPC's instructions on how to communicate with the SGX-protected chaincode after deployment were followed.

In Fabric, the FPC chaincode is installed using the peer lifecycle commands and then transactions are invoked. To install the FPC chaincode, `$FPC_PATH/fabric/bin/peer.sh`, a custom FPC wrapper designed to replace the standard peer CLI command from Fabric [Hyp24c], is used. The `${PEER_CMD}` variable is defined in `$FPC_PATH/fabric/bin/lib/common_ledger.sh` and conveniently points to the

necessary script file. With the variables set and `common_ledger.sh` executed, the usage of `peer.sh` is as follows:

First, package the FPC chaincode:

```
${PEER_CMD} lifecycle chaincode package --lang fpc-c --label ${CC_ID} --path ${CC_PATH} ${PKG}
```

Next, install the packaged chaincode:

```
${PEER_CMD} lifecycle chaincode install ${PKG}
```

Approve the chaincode for the organization:

```
${PEER_CMD} lifecycle chaincode approveformyorg -o ${ORDERER_ADDR} -C ${CHAN_ID}
--package-id ${PKG_ID} --name ${CC_ID} --version ${CC_VER} --sequence ${CC_SEQ}
--signature-policy ${CC_EP}
```

Check the commit readiness of the chaincode:

```
${PEER_CMD} lifecycle chaincode checkcommitreadiness -C ${CHAN_ID} --name ${CC_ID}
--version ${CC_VER} --sequence ${CC_SEQ} --signature-policy ${CC_EP}
```

Commit the chaincode:

```
${PEER_CMD} lifecycle chaincode commit -o ${ORDERER_ADDR} -C ${CHAN_ID} --name ${CC_ID}
--version ${CC_VER} --sequence ${CC_SEQ} --signature-policy ${CC_EP}
```

Create an enclave that runs the FPC chaincode:

```
# create an FPC chaincode enclave
${PEER_CMD} lifecycle chaincode initEnclave -o ${ORDERER_ADDR} --peerAddresses "localhost:7051" \
--name ${CC_ID}
```

The FPC library already contained a script called installFPC.sh to correctly configure the environment and network, allowing convenient over-network chaincode interaction similar to the old middleware [Hyp24c]. However, when executing these environment preparation scripts, the final and hardest hurdle was encountered. Changes in Hyperledger Fabric caused FPC's network environment configuration scripts to break, preventing communication with the FPC chaincode over the network.

To overcome this, a temporary gateway was created. This new communication method proves that it is possible to protect deployed chaincode with SGX in Hyperledger Fabric. A fix for the current FPC issues is expected soon. More details on this temporary communication method are in Chapter 6.3 and further discussion in Chapter 10.

6.2 The Digital Signing Replacement Function

To implement a digital signing scheme in FPC without using external libraries, a simplified version of the lantern's ABDLOP [Ngu22] was created. This protocol is illustrated in Figure 6.2.

This protocol as it is, is interactive. To make it non-interactive, Fiat-Shamir heuristic [GMR85, Reg09] is used in combination with incorporating the message into the transformation process.

The signing function follows these steps:

1. **Sampling**: Sample two vectors, $\mathbf{y1}$ and $\mathbf{y2}$, from the discrete Gaussian distribution as ephemeral keys.

2. **Computation of w**: Compute $\mathbf{w}$ as the sum of the matrix-vector products of $\mathbf{A1}$ with $\mathbf{y1}$ and $\mathbf{A2}$ with $\mathbf{y2}$.
$$\mathbf{w} = \mathbf{A1} \cdot \mathbf{y1} + \mathbf{A2} \cdot \mathbf{y2}$$

3. **Challenge Generation**: Generate a challenge c using the Fiat-Shamir heuristic by hashing $\mathbf{w}$ and the flattened message $\mathbf{m}$.
$$c = \text{FiatShamirChallenge}(\mathbf{w}, \mathbf{m})$$

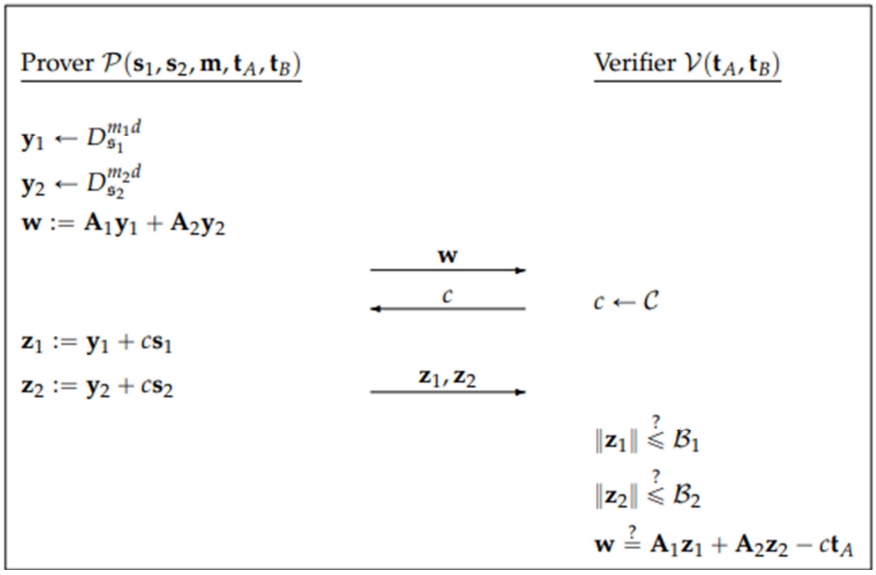

Figure 6.2: A simplified version of ABDLOP, forming the basis of the digital signing scheme of Vesper-FPC [Ngu22].

4. **Response to Challenge**: Compute responses **z1** and **z2** by modifying **y1** and **y2** with the secret vectors **s1** and **s2** scaled by the challenge c.

$$\mathbf{z1} = (\mathbf{y1} + c \cdot \mathbf{s1}) \mod q$$

$$\mathbf{z2} = (\mathbf{y2} + c \cdot \mathbf{s2}) \mod q$$

5. **Construction of $\mathbf{t_A}$ and $\mathbf{t_B}$**: Precompute $\mathbf{t_A}$ and $\mathbf{t_B}$ for the verification process, combining the effects of the secret vectors and the public matrices.

$$\mathbf{t_A} = \mathbf{A1} \cdot \mathbf{s1} + \mathbf{A2} \cdot \mathbf{s2}$$

$$\mathbf{t_B} = \mathbf{Bext} \cdot \mathbf{s2} + \sum_{i=1}^{d} \mathbf{m}[i]$$

6. **Signature Formation**: Form the signature as the tuple $(\mathbf{w}, \mathbf{z1}, \mathbf{z2})$.

$$\text{Signature} = (\mathbf{w}, \mathbf{z1}, \mathbf{z2})$$

The verification function checks the validity of the signature as follows:

1. **Norm Checks**: Verify the norms of **z1** and **z2** against predefined bounds to ensure they are within acceptable limits.

$$\|\mathbf{z1}\| \leq 1.5 \cdot \text{boundValue} \quad \text{and} \quad \|\mathbf{z1}\| \geq \text{boundValue}$$

$$\|\mathbf{z2}\| \leq 1.5 \cdot \text{boundValue} \quad \text{and} \quad \|\mathbf{z2}\| > \text{boundValue}$$

2. **Recomputation of $\mathbf{w}'$**: Recompute $\mathbf{w}'$ using the received signature components and public matrices.

$$\mathbf{w}' = (\mathbf{A1} \cdot \mathbf{z1} + \mathbf{A2} \cdot \mathbf{z2} - c \cdot \mathbf{t_A}) \mod q$$

3. **Final Check**: If $\mathbf{w}$ matches $\mathbf{w}'$ and the norms of **z1** and **z2** are within bounds, the signature is valid.

$$\text{Valid Signature if } \mathbf{w} = \mathbf{w}'$$

Just in case, the pseudocode has been made available as well in Appendix I.

6.3 Communicating with FPC: A New Gateway

Due to broken functionalities in FPC, the original Vesper middleware would not function in an FPC environment, as communication over the server was not possible.

To achieve communication with the deployed chaincode over the server, FPC's custom peer CLI commands were used directly.

The problem was now reduced to passing the proof arguments to a script in an FPC Docker container in a way that could be understood and handled by both the script and the chaincode.

To accomplish this, Golang's ability to pass arguments to scripts was utilized. A Golang server starts from within the FPC Docker container and when the prover sends the arguments to this server, it passes them along as input to the CLI query and invoke commands which provides interaction with the SGX protected deployed chaincode. This commands constructed as follows:

```
# Invoke VerifyProof transaction
say "- invoke VerifyProof transaction"
${PEER_CMD} chaincode invoke -o ${ORDERER_ADDR} -C ${CHAN_ID} -n \\
${CC_ID} -c "$(jq -nc --argjson input \\
"${PROOF_STRING}" '{"Args":["VerifyProof", $input]}')" --waitForEvent

# Query with VerifyProof transaction
say "- query with VerifyProof transaction"
${PEER_CMD} chaincode query -o \\
${ORDERER_ADDR} -C ${CHAN_ID} -n ${CC_ID} -c \\
"$(jq -nc --argjson input "${PROOF_STRING}" '{"Args":["VerifyProof", $input]}')"
```

Here, the proof string is the input provided by the Golang server. The JSON input string needs to be escaped to avoid errors due to certain characters being read differently than intended. An escaped JSON parser was implemented inside the chaincode for it to understand this format. Communication from an SGX-protected prover to an SGX-protected verifier was now established. The pseudocode for this middleware can be seen in Appendix J.

6.4 Docker

Both the project with FPC usage and without have been put in a Docker container. Docker is an open-source platform designed to automate the deployment, scaling, and management of applications using containerization [BRBA17]. Containers are lightweight, portable, and self-sufficient environments that include everything needed to run a piece of software, including the code, runtime, system tools, libraries, and settings. A visualization of docker containers can be found below in Figure 6.3.

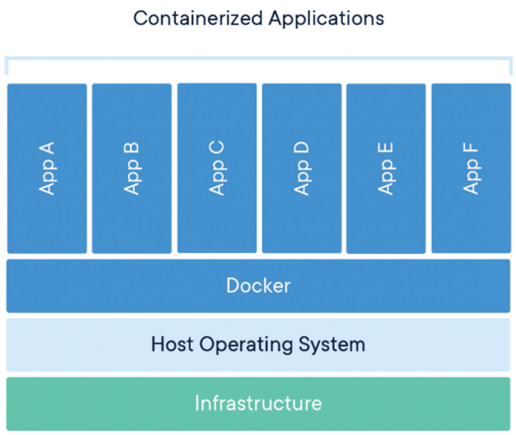

Figure 6.3: A high-level visualization of docker containers [Inc24].

Using a Docker container comes with several benefits. Containers share the host system's OS kernel, reducing overhead compared to VMs and allowing for more efficient use of resources. Containers run in isolated environments, enhancing security and reducing conflicts between applications. Most importantly, Docker containers can run on any machine with Docker installed, ensuring that applications behave the same regardless of the environment. This makes the project portable and significantly reduces setup time.

Chapter 7

Results and Analyses

In this chapter, the results of this work are shared and compared to previous studies. We split the results into the following categories: standalone, zk-SNARK with SGX protection, and the sample transaction where the ZKP communicates with the deployed chaincode. We discuss both the work where regular Hyperledger Fabric was used for chaincode deployment with Dilithium digital signing, and the Fabric-private-chaincode (FPC) version with a custom digital signing scheme.

7.1 Test Bed

Hardware Specifications: The experiments were conducted on a machine with the following specifications:

- **CPU**: Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz, 6 Cores, 12 Logical Processors
- **Memory**: 16GB RAM
- **Graphics**: Intel(R) UHD Graphics 630, 1GB NVIDIA Quadro P1000
- **Operating System**: Ubuntu 23.10

Software Environment Vesper Smart Contract: The Vesper zero-knowledge proof is written in JavaScript, utilizing the following software and libraries:

- **NPM**: Version 9.10.1
- **Mathjs**: For mathematical operations
- **@guildofweavers/galois**: For Galois field computations
- **Node-fetch and Express**: For communication with middleware and chaincode
- **@asanrom/dilithium**: For digital signature creation and verification
- **Hyperledger Fabric**: The smart contract is registered using the fabric-contract-api
- **Gramine**: Used to protect proof generation with SGX, running inside a Docker container

Software Environment Vesper-FPC Smart Contract: The Vesper-FPC environment has similarities to the Vesper smart contract environment:

- The prover side is executed from a Docker container with Gramine SGX protection
- **Prover Implementation**: In JavaScript, using Mathjs and @guildofweavers/galois
- **Verifier Implementation**: Added to the FPC environment and implemented in C++

The Vesper-FPC environment requires the following packages:

- **CMake**

- **Go**
- **Docker and docker-compose**
- **yq**
- **Protocol Buffers and Nanopb**
- **Hyperledger Fabric**
- **Clang-format**
- **jq, hex, and PlantUML**

A Golang server facilitates communication between the prover and the FPC-protected chaincode.

7.2 Dataset

In these experiments, synthetic data representing valid witnesses based on the MLWE dimension is used. These witnesses serve as authentication keys that require verification before performing operations. The datasets are generated using a function that produces random input witnesses based on specific security parameters. This generation process is aligned with the construction of R1CS matrices, ensuring that the witness generation function scales effectively with the LWE dimension. The pseudocode for this automatic witness generation function is provided in Appendix K.

7.3 Performance Metrics

The primary metrics for zero-knowledge proofs and zk-SNARKs are:

- **Verification Time**: The elapsed time for the verifier to verify the proof, including the time to verify any digital signature.
- **Proof Construction Time**: The elapsed time for the prover to generate a proof, including optimizations to reduce proof size.
- **Proof Size**: The size of the proof generated by the prover, measured in bytes.

Additionally, **scalability** is analyzed by measuring how these metrics increase with the MLWE dimension.

7.4 Experimental Setup

First, the parameters are configured. A combination of a ring dimension of 64, a proof modulus of 2^8, and a randomness vector ranging between 1 and -1 were used to determine what Modular-LWE dimesnion was needed to achieve the desired security level using the LWE hardness estimator of Albrecht et al. [Alb17].

The experiments begin with a zk-SNARK standalone evaluation. Vesper is run 100 times, each time with a newly generated input witness that fits the specified dimensions. This repetition allows for averaging the results to obtain accurate standalone time measurements. Following this, the same experiment is conducted, but this time with SGX protection enabled. Both sets of experiments are performed within a Docker container [BRBA17].

Subsequently, the smart contracts are analyzed and compared. The metrics analyzed include speed, size, scalability differences between Dilithium and the simplified digital signing scheme, and the execution speed of the full pipeline.

7.5 Standalone Results

The LWE hardness estimator [Alb17] showed that with these parameters, an LWE-Dimension of 4 is needed to achieve at least 80-bit security, a dimension of 6 is needed to achieve at least 128-bit security and a dimension of 12 is needed to achieve at least a 256-bit security.

In Table 7.1 below you can find the standalone results for different security level configurations.

Security Level	Proof Generation	Verification Time	Proof Size
80-bit	0.5189 s	0.1752 ms	128 bytes
128-bit	2.4833 s	0.1478 ms	128 bytes
256-bit	401.9758 s	0.2591 ms	128 bytes

Table 7.1: Standalone results of Vesper with proof size optimizations at Different Security Levels

To also get and idea of what kind of effects the proof size optimizations had on the results, the results without optimizations where measured as well. These results are presented in Table 7.2.

Security Level	Proof Generation	Verification Time	Proof Size
128-bit	2.2762 s	0.0078 ms	512 bytes

Table 7.2: Standalone results without proof size optimizations for comparison purposes

7.6 Digital Signing Results

Table 7.3 shows the average signing and verification times obtained from both Vesper-FPC's signing scheme and Dilithium2 [Sar24].

Security Level	Vesper-FPC		Dilithium2	
	Signing Time	Verification Time	Signing Time	Verification Time
80-bit	0.412 ms	1.273 ms	75.924 ms	13.152 ms
128-bit	0.494 ms	1.461 ms	76.524 ms	13.248 ms
256-bit	0.885 ms	2.413 ms	132.674 ms	23.641 ms

Table 7.3: Comparison of Signing and Verification Times for Vesper-FPC's signing function and Dilithium2 at Different Security Levels

Besides speed, another important metric is the size of the keys. Table 7.4 shows the size comparison between Vesper-FPC digital signing and Dilithium.

Method	**Public Key Size (bytes)**	**Private Key Size (bytes)**	**Signature Size (bytes)**	**Security Level**
Crystals Dilithium 2	1,312	2,528	2,420	1 (128-bit) Lattice
Vesper-FPC Signing	48,000	512	768	1 (128-bit) Lattice

Table 7.4: Comparison of Crystals Dilithium2 and Vesper-FPC's Signing scheme in terms of key and signature sizes.

7.7 Full Pipeline Results with and without SGX

To determine the execution time of the Vesper smart contract with SGX protection for payload generation (prover side), the docker container running the prover side under Gramine SGX [Sch21] protection is examined. The difference between the "StartedAt" and "FinishedAt" timestamps in the Docker container logs reveals the execution time.

The execution time of the Vesper-FPC smart contract, with SGX protection for both payload creation on the prover side and the deployed chaincode, was determined similarly. After multiple rounds of execution and examinations of the Docker containers, the average runtime including chaincode

Security Level	Without SGX	With SGX	
		Proverside Only	Fully (Vesper-FPC)
80-bit	0.317 s	15.36 s	41.361 s
128-bit	2.087 s	26.339 s	51.229 s
256-bit	67.482 s	6 min and 58 s	8 min and 12 s

Table 7.5: Comparison of Execution Times Without SGX and With SGX at Different Security Levels

deployment, proof generation and verification could be determined. Further analysis of the output.log file indicated that initializing FPC's SGX protection provided more details on how long each specific operation took as output.log contains ledger data with timestamps.

7.7.1 performance under concurrent transactions

Figures 7.1 and 7.2 illustrate the effect of multiple concurrent transactions on the chaincode developed in this work, comparing both the Vesper Smart Contract and Vesper-FPC in terms of throughput and latency, respectively.

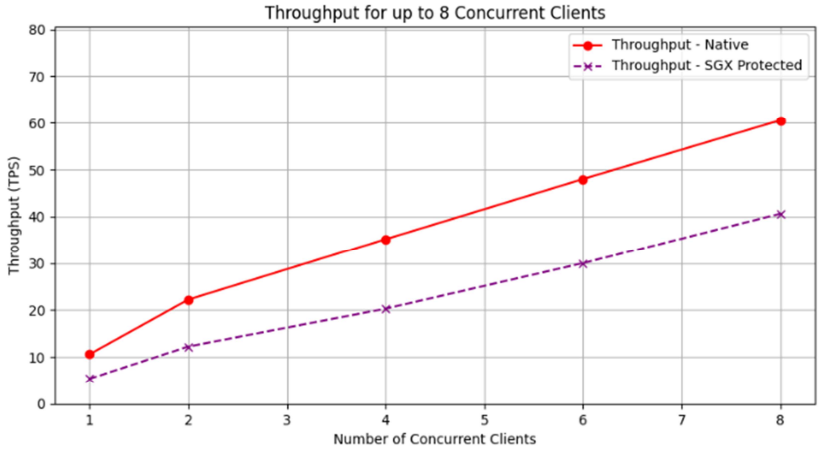

Figure 7.1: Performance of Vesper Smart Contract and Vesper-FPC in terms of throughput under concurrent transactions.

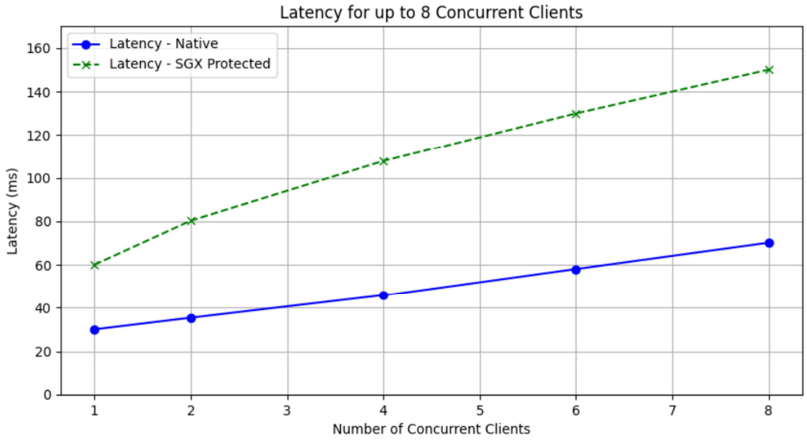

Figure 7.2: Performance of Vesper Smart Contract and Vesper-FPC in terms of latency under concurrent transactions.

To measure these results for the native Hyperledger Fabric used in the Vesper Smart Contract,

Hyperledger Caliper was utilized [Hyp24d]. Hyperledger Caliper is a performance benchmarking tool for blockchain networks that allows users to evaluate the performance of various blockchain implementations by running predefined test cases.

Due to the structure of Fabric Private Chaincode and the setup of communication in Vesper-FPC, a different tool was selected to measure the performance under concurrent transactions for Vesper-FPC. For this purpose, an increasing number of clients built with the Hyperledger Fabric SDK for Go was used [Hyp24e].

7.8 Analysis

7.8.1 Vesper

When comparing the Vesper zk-SNARK with the zk-SNARKs from Table 3.1, we can see that Vesper has a faster verification time and a proof size small enough to rival Groth16 while achieving at least 128-bit and even 256-bit security. Another significant feature of Vesper is that it does not require a trusted setup and the associated toxic waste. Additionally, Vesper benefits from post-quantum security. Table 7.6 shows the comparison between Groth16 and Vesper.

Protocol	Proof Size	Verification Time	Prover Run-Time	Setup
Groth16	$O(1)$ ~192 bytes	$O(1)$ ~1-10 ms	$O(n+m)$	Trusted
Vesper	$O(d)$ 128 bytes	$O(n)$ ~0.14-0.2 ms	$O(n^2)$	None

Table 7.6: Comparison of Protocols at 128-bit security

Vesper achieves short proof sizes and fast verification times at high-security levels. However, its bottleneck is the proof generation process, which lacks scalability. Due to how the R1CS is formed based on the inherent LWE constraints, the R1CS matrices contain $3n^2 + 2n + 1$ columns. For each of these columns, Lagrange Interpolation needs to be executed. Since the R1CS matrix columns grow quadratically and Lagrange interpolation is also $O(n^2)$, proof generation time increases drastically as the LWE dimension grows.

The effect of the LWE dimension on proof size, verification time, and proof generation can be clearly seen in Table 7.1. It is evident that as the security level increases, proof generation time rises more drastically, especially as the dimension and security level increase. The verification time remains roughly constant, and the proof size stays constant as well.

This is because verification time and proof size grow linearly with the ring dimension d, since the generated proof takes the form of a 2D array containing two arrays with the same length as the ring dimension. Besides signature validation, the verifier only needs to compare these arrays. It is important to note that the proof size does not grow linearly with the size of the witness, meaning the proof size and verification time remain constant under larger statements, which is very beneficial in blockchain environments. This gives the effect of a constant proof size. Therefore, improving the scalability of Vesper would come down to optimizing the proof generation process.

When comparing Vesper to previous lattice-based zero-knowledge proofs like those by V. Lyubashevsky et al. and Ngoc Khanh Nguyen [LNP22b, Ngu22], the first thing that stands out is the difference in proof size. State-of-the-art previous work has proof sizes of 13KB and 14KB at 128-bit security, while Vesper only has a proof size of 128 bytes with 128-bit security. This proof size comparison with previous work can also be seen in Table 7.7.

	Proof Size
[LNPS21]	33KB
[LNP22b]	14KB
[Ngu22]	13KB
Vesper	128 bytes

Table 7.7: Comparison of Proof Sizes

7.8.2 Vesper-FPC Signing

Due to the restrictions in the FPC environment, a simplified LWE-based digital signing scheme was developed as a replacement. The results show that the signing and verification times of this scheme are much faster than those of Dilithium [DKL+18] at high-security levels. As illustrated in Table 7.3, even at higher security levels such as 256-bit security, the signing and verification times remain small, indicating good scalability.

Additionally, the private key and signature sizes are smaller than those of Dilithium. Since the same parameters used in Vesper were applied in Vesper-FPC signing, the key generation remains constant, as it is based on the ring dimension rather than the LWE dimension.

However, the public key size is significantly larger, necessitating a much larger payload to be sent to the verifier. This increased payload decreases scalability, as large payloads reduce the number of transactions per second that the blockchain can handle. Consequently, this would also increase latency and transaction fees compared to Dilithium. Further discussion on this topic will be provided in Chapter 8.

7.8.3 SGX Influence

The addition of SGX protection significantly increases the execution time of the Vesper smart contract, as shown in Table 7.5. When both the prover and verifier sides are protected with SGX using the FPC framework, the execution time increases by more than the 10%-20% increase anticipated by Brandenburger et al. [BCKS18]. This was expected, given that the new gateway implemented in Vesper-FPC introduces additional overhead. However, this overhead is expected to decrease once the intended communication methods are fully implemented.

The results in Table 7.5, along with the concurrent performance results shown in Figures 7.1 and 7.2, indicate that the addition of SGX protection for the chaincode introduces an increase in latency of approximately 94.5%-135.3% and a decrease in throughput of about 34.8%-49.5

Chapter 8

Discussion

This work's core consists of three main aspects: zero-knowledge proof, SGX protection, and Blockchain. The results show that Vesper offers significant improvements in proof size and verification time compared to previous works, making it highly suitable for blockchain applications. Vesper demonstrates that a lattice-based zero-knowledge proof improves upon previous work by providing a smaller proof size while maintaining at least 128-bit security.

However, scalability in proof generation and the impact of SGX protection on execution time need further optimization. Future work should focus on enhancing scalability and reducing the overhead introduced by SGX protection.

A significant focus of this project is SGX protection. The results indicate that adding SGX protection to proof generation significantly increases execution time. The question becomes: is adding SGX protection to proof generation worth the trade-offs? The primary goal of this project was to create a solution secure in a post-quantum world.

Given the need to protect against threats as powerful as quantum computers, defense in depth is essential, especially since our protection measures are mostly theoretical at this point. Quantum computers have the potential to break many cryptographic algorithms currently in use, rendering traditional security mechanisms ineffective. Therefore, ensuring the highest level of security for the system is paramount, even if it means accepting certain performance trade-offs. The integrity and confidentiality of proof generation are critical. SGX ensures that sensitive computations are performed in a secure environment, protecting against both software-based and physical attacks. This is particularly important in a post-quantum context, where new vulnerabilities may emerge. While post-quantum algorithms protect against quantum attacks, SGX protects the implementation and execution of these algorithms from other forms of compromise.

However, the increased execution time due to SGX protection is a significant consideration. The enhanced security it provides might not justify this performance cost, especially for applications where security is not the highest priority.

This consideration is particularly relevant for Vesper-FPC. In Vesper-FPC, both proof generation and deployed verification receive SGX protection and this comes with increased overhead. Here, we must ask: are the benefits worth the overhead, and can we improve it?

It can be argued that adding SGX protection to the deployed verifier might not significantly enhance security enough to allow these costs for the following reasons:

1. The consensus mechanism in Hyperledger Fabric ensures that no single peer can unilaterally make decisions or manipulate the ledger. Multiple peers must agree on the state of the ledger, making it very difficult for a malicious peer to cause harm. This is particularly effective since it is unlikely for the majority of peers to be malicious, especially with known and authenticated participants.

2. Once a chaincode (smart contract) is deployed on a Hyperledger Fabric network and peers have joined the channel, the chaincode cannot be altered directly. Any updates require consensus.

3. Hyperledger Fabric is designed for enterprise use, where all participants are typically known and authenticated, reducing the risk of unauthorized entities joining the network.

4. The smart contract does not contain confidential data to protect. It only verifies the lattice encrypted data and the post-quantum signature it receives without needing to decrypt it first.

However, besides providing extra protection against potentially malicious peers, the main reason for adding an SGX protection layer around the smart contract is privacy. The transparency and resilience offered by blockchain protocols guarantee the integrity of blockchain applications. However, this transparency can conflict with the goal of keeping the application state confidential and maintaining user privacy. An SGX enclave can address this issue. Yet, the privacy and application state confidentiality benefits might not justify the performance cost, depending on the application. Fortunately, improvements are possible.

One cause for the increased execution time is the improvised communication gateway used due to some broken functionalities in FPC. This is a temporary solution, and as detailed in Chapter 10, a solution is already in development. In the meantime, it should be considered that performance costs will likely be reduced when the intended communication method becomes available.

Another reason for the increased execution time is the simplified replacement digital signing scheme. While it signs and verifies quickly, it has a large public key, which is typically not well-suited for SGX enclaves with limited memory and computational resources. Further improvements to Vesper-FPC's signing could be challenging due to the environment restrictions of FPC.

Overall, weighing security against performance is crucial. This consideration led to the creation of two projects: one with optional SGX protection only on the prover side and NIST-approved Dilithium signing (Vesper Smart Contract), and one with SGX protection for both proof generation and deployed chaincode.

Chapter 9

Conclusion

This work presents a novel lattice-based zero-knowledge proof with a smaller proof size than previous approaches, achieving verification times of about a tenth of a millisecond while maintaining at least 128-bit security. Vesper combines an LWE lattice-based zk-SNARK with the Regev commitment scheme, ensuring both quantum resistance and the efficiency benefits of zk-SNARKs. Vesper Smart Contract demonstrates how this efficient zk-SNARK can be utilized in a permissioned blockchain environment, such as Hyperledger Fabric, by integrating it as a smart contract within an SGX enclave. This integration makes it the first smart contract based on a lattice-based zero-knowledge proof deployed inside an SGX enclave within Hyperledger Fabric.

Vesper-FPC proves that it is possible to protect a smart contract in an SGX enclave when deploying it in Hyperledger Fabric and interacting with it, even when dealing with complex constructs like lattice-based zero-knowledge proofs.

As outlined in Chapter 11, further improvements are already underway, and the full potential of Vesper-FPC has yet to be realized. Vesper Smart Contract and Vesper-FPC represent new technologies that pave the way for a new kind of blockchain protection mechanism.

The implications of this work extend beyond the immediate scope of blockchain technology. By integrating quantum-resistant zk-SNARKs with SGX protection in a practical blockchain environment, this research contributes to the broader field of cryptographic security. It provides a potential pathway for developing robust post-quantum cryptographic systems that can withstand future threats posed by quantum computing.

Chapter 10

Future Work

While this research has successfully demonstrated the feasibility of deploying LWE-based zk-SNARKs with SGX integration in permissioned blockchains like Hyperledger Fabric, several areas warrant further exploration to enhance and extend the current work.

Firstly, optimizing Vesper further to improve its scalability is crucial. One option is to develop a custom Galois library with an optimized Lagrange interpolation function, as this is the bottleneck in the proof generation. This can be achieved using Fast Fourier Transform (FFT).

Secondly, FPC would become significantly more user-friendly if it were easier to add and use external libraries. Facilitating this would have many benefits, simplifying the deployment of SGX-protected chaincode. This improvement would also allow for the use of verified post-quantum digital signing schemes like Dilithium instead of a simplified scheme.

Lastly, after sharing my findings with the Hyperledger developers and collaborating with them, I received confirmation that a fix for FPC's communication issues is being worked on and will be merged soon. This means that, very soon, chaincode communication and interaction can be integrated into Vesper-FPC as initially intended, instead of the current sub-optimal communication method. This will likely improve overall execution time, further solidifying the roles of Vesper and Vesper-FPC in secure and efficient blockchain applications.

Chapter 11

Reflection

Throughout this master's thesis project, I have learned more than I could have ever imagined. Beyond gaining knowledge about zero-knowledge proofs, enclaves, and blockchain, I have also gained insights into my strengths and weaknesses. I discovered that my greatest strengths during this project were patience and determination.

This work required numerous setups and installations. For example, I had to set up SGX, install related software, install Gramine, set up Hyperledger Fabric, understand it, and correctly configure FPC. None of these installations went smoothly. Finding the right instructions specific to my operating system and software environment was a struggle, as there was no single tutorial with all the correct steps. The necessary commands and instructions were scattered across multiple sources. However, I patiently kept searching, trying commands from different tutorials, and even modifying commands (especially for Gramine) to fit my operating system version (Ubuntu 23.10) until everything was fully installed and ready to use. Installing the components used in this work rarely comes without hiccups. This is one of the reasons I decided to use Docker containers for my work, simplifying the setup for those who want to use Vesper and Vesper-FPC.

I encountered multiple hurdles that some might have considered dead ends, but I persevered. When looking for a way to SGX protect Vesper, the only suitable wrapper I found was Gramine. For Gramine to work, SGX needed to be installed and connected to Gramine. However, after installing the SGX SDK and PWS, Gramine still indicated that SGX was not enabled, even though it was activated in the BIOS. After some research, I discovered that Gramine only works on Linux, while I only had a Windows PC. Instead of giving up on SGX integration, I dual-booted my PC and continued.

Shortly after, I ran into a memory issue with my PC. There was not enough memory for Gramine to put my code into an enclave within the Docker container. Rather than quitting and looking for another path without SGX, I bought an external hard drive and continued.

I had never worked with Hyperledger Fabric before, and because it is a large project, it took a long time to understand how it works. When the time came to deploy chaincode, it would not work due to issues with the TLS and MSP configuration. I checked every path in the fabric-samples and ensured they were correct and absolute paths. This task required significant patience, but eventually, the problem was resolved.

There was one point where I thought it might have been a lost cause. Initially, it seemed like integrating FPC would not succeed due to broken communication and network functionalities. However, while developing Vesper Smart Contract and Vesper-FPC, I learned a lot about SGX and Hyperledger Fabric and became intrigued by the idea of deploying SGX-protected chaincode. Even though FPC seemed to be a dead end, I could not let it go and decided to try to find solutions for the FPC issue. After extensive research and experimentation, I managed to create something that worked. I solved every single problem and error I encountered.

First, I discovered that it was not possible to create a channel after bringing up the network in FPC. I investigated whether I was doing something wrong or if there was an issue with the environment configuration. When I could not find any solution, I turned to the Hyperledger Discord server referenced in FPC. There, I learned that this was indeed a known issue, and more people were experiencing it. Nevertheless, I kept trying every possible solution and eventually discovered that outdated Hyperledger binaries were being used. Replacing them with newer binaries solved the problem.

This was not the last hurdle in FPC. FPC has a script called installFPC.sh, which should configure

the network to allow regular CLI communication over the network, even with chaincode in an enclave. However, this functionality was also broken, making me think there might be no way forward. Determined to make it work, I kept researching FPC and its components. Eventually, I had an idea: if I could pass arguments to a bash script, I could interact with the chaincode directly using custom FPC Peer CLI commands. This led to the development of the temporary gateway discussed previously.

One of my biggest weaknesses I discovered is that my determination sometimes turns into an inability to let go. Until I find a solution, it becomes all I can think about, and sometimes I work all night to find it. However, I realized that taking breaks or getting some sleep is often crucial for solving complex problems. Staring too long at the same issue is usually not beneficial, and difficult problems can be solved more easily with fresh eyes.

Sharing my findings with the Hyperledger developers, I discovered a blockchain community that covers a vast array of topics and helps each other with many different projects. Learning about the existence of such a community has been another valuable lesson during this project. Now, I know there are people who could potentially aid me in future cybersecurity endeavors, and I also have a way to help others using the knowledge and experiences I acquired during my studies, projects, and work.

Ultimately, this experience has not only expanded my technical expertise but also underscored the importance of perseverance and collaboration in research. I am grateful for the opportunity to have worked on this project and am eager to continue facing future challenges and making new discoveries in the field of cybersecurity.

Bibliography

[A+18] Martin Albrecht et al. Efficient polynomial arithmetic for lattice-based cryptography. *IACR Cryptol. ePrint Arch.*, page 491, 2018.

[ABB+18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15. ACM, 2018.

[ACK21] Thomas Attema, Ronald Cramer, and Lisa Kohl. A compressed ς-protocol theory for lattices. In *CRYPTO (2)*, volume 12826 of *Lecture Notes in Computer Science*, pages 549–579. Springer, 2021.

[ACL+22] Martin R. Albrecht, Valerio Cini, Russell W. F. Lai, Giulio Malavolta, and Sri AravindaKrishnan Thyagarajan. Lattice-based SNARKs: Publicly verifiable, preprocessing, and recursively composable. Cryptology ePrint Archive, Paper 2022/941, 2022. https://eprint.iacr.org/2022/941.

[Ajt96a] 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.

[Ajt96b] Miklós Ajtai. Generating hard instances of lattice problems (extended abstract). In *Proceedings of the twenty-eighth annual ACM symposium on Theory of Computing (STOC)*, pages 99–108. ACM, 1996.

[AKS01] Miklós Ajtai, Ravi Kumar, and Dandapani Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing, STOC 2001, Heraklion, Crete, Greece, July 6-8, 2001*, pages 601–610, New York, NY, USA, 2001. ACM.

[AL21] Martin R. Albrecht and Russell W. F. Lai. Subtractive sets over cyclotomic rings - limits of schnorr-like arguments over lattices. In *CRYPTO (2)*, volume 12826 of *Lecture Notes in Computer Science*, page 519. Springer, 2021.

[Alb17] Martin R. Albrecht. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 11(3-4):169–203, 2017.

[ALS20] Thomas Attema, Vadim Lyubashevsky, and Gregor Seiler. Practical product proofs for lattice commitments. In *CRYPTO (2)*, volume 12171 of *Lecture Notes in Computer Science*, pages 470–499. Springer, 2020.

[Ano23] Anonymous. zkdfl: An efficient and privacy-preserving decentralized federated learning with zero-knowledge proof. *arXiv preprint arXiv:2312.04579*, 2023.

[Ant14] Andreas M Antonopoulos. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. O'Reilly Media, Inc., 2014.

[BAB+22] Ahmad Al Badawi, Andreea Alexandru, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Carlo Pascoe, Yuriy Polyakov, Ian Quah, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. OpenFHE: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, Paper 2022/915, 2022. https://eprint.iacr.org/2022/915.

[Bal21] David Balbás. The hardness of LWE and ring-LWE: A survey. Cryptology ePrint Archive, Paper 2021/1358, 2021. https://eprint.iacr.org/2021/1358.

[BBC+18] Carsten Baum, Jonathan Bootle, Andrea Cerulli, Rafaël del Pino, Jens Groth, and Vadim Lyubashevsky. Sub-linear lattice-based zero-knowledge arguments for arithmetic circuits. In *CRYPTO*, page 669, 2018.

[BCC+16] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 327–357. Springer, 2016.

[BCG+18] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zk-snarks for c: Verifying program executions succinctly and in zero knowledge. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 566–583. IEEE, 2018.

[BCKS18] 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. Accessed: 2024-07-26.

[BCS21] Jonathan Bootle, Alessandro Chiesa, and Katerina Sotiraki. Sumcheck arguments and their applications. In *CRYPTO (1)*, volume 12825 of *Lecture Notes in Computer Science*, page 742. Springer, 2021.

[BDK+21] Ward Beullens, Samuel Dobson, Shuichi Katsumata, Yi-Fu Lai, and Federico Pintore. Group signatures and more from isogenies and lattices: Generic, simple, and efficient. Cryptology ePrint Archive, Paper 2021/1366, 2021. https://eprint.iacr.org/2021/1366.

[BDL+16] Carsten Baum, Ivan Damgård, Vadim Lyubashevsky, Sabine Oechsner, and Chris Peikert. More efficient commitments from structured lattice assumptions. Cryptology ePrint Archive, Paper 2016/997, 2016. https://eprint.iacr.org/2016/997.

[BDSMP91] Manuel Blum, Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. Noninteractive zero-knowledge. *SIAM Journal on Computing*, 20(6):1084–1118, 1991.

[BF01a] Dan Boneh and Matthew K. Franklin. Short signatures from the weil pairing. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology (CRYPTO 2001)*, volume 2139 of *Lecture Notes in Computer Science*, pages 514–532, London, UK, UK, 2001. Springer-Verlag.

[BF01b] Dan Boneh and Matthew K. Franklin. Short signatures from the weil pairing. In *Advances in Cryptology — ASIACRYPT 2001*, pages 514–532. Springer, 2001.

[BFS19] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK compilers. Cryptology ePrint Archive, Paper 2019/1229, 2019. https://eprint.iacr.org/2019/1229.

[BGH19] Sean Bowe, Jack Grigg, and Daira Hopwood. Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Paper 2019/1021, 2019. https://eprint.iacr.org/2019/1021.

[BGM17] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model. *IACR Cryptology ePrint Archive*, 2017:1050, 2017.

[BGS+23] Dan Boneh, Shafi Goldwasser, Dawn Song, Justin Thaler, and Yupeng Zhang. Zero-knowledge proofs mooc lectures, 2023. Accessed: 2024-07-26.

[BLNS20] Jonathan Bootle, Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Gregor Seiler. A non-pcp approach to succinct quantum-safe zero-knowledge. In *CRYPTO (2)*, volume 12171 of *Lecture Notes in Computer Science*, page 441. Springer, 2020.

[BLS19] Jonathan Bootle, Vadim Lyubashevsky, and Gregor Seiler. Algebraic techniques for short(er) exact lattice-based zero-knowledge proofs. In *CRYPTO (1)*, volume 11692 of *Lecture Notes in Computer Science*, pages 176–202. Springer, 2019.

[BMC+15] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *2015 IEEE Symposium on Security and Privacy*, pages 104–121. IEEE, 2015.

[Bra14] Zvika Brakerski. Efficient fully homomorphic encryption from (standard) lwe. In *CRYPTO*, pages 1–19. Springer, 2014.

[BRBA17] Babak Bashari Rad, Harrison Bhatti, and Mohammad Ahmadi. An introduction to docker and analysis of its performance. *IJCSNS International Journal of Computer Science and Network Security*, 173:8, 03 2017.

[BSBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptology ePrint Archive*, 2018:46, 2018.

[BSBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 701–732. Springer, 2019.

[BSCG+14] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA, 2014.

[But19] Vitalik Buterin. Quadratic arithmetic programs: From zero to hero. https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649, 2019. Accessed: 2024-08-30.

[C+17a] Melissa Chase et al. Zero-knowledge proofs with homomorphic encryption. *IACR Cryptol. ePrint Arch.*, page 139, 2017.

[C+17b] Jung Hee Cheon et al. Approximate homomorphic encryption with reduced approximation error. *IACR Cryptol. ePrint Arch.*, page 139, 2017.

[CCKK15] Dong Pyo Chi, Jeong Woon Choi, Jeong San Kim, and Taewan Kim. Lattice based cryptography for beginners. Cryptology ePrint Archive, Paper 2015/938, 2015. https://eprint.iacr.org/2015/938.

[CD16] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[CDP18] Ronald Cramer, Ivan Damgård, and Charalampos Papamanthou. Batched zero-knowledge proofs and applications to verification of data structures. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 44–57. ACM, 2018.

[CFT18] Yu-Lun Chuang, Chun-I Fan, and Yi-Fan Tseng. An efficient algorithm for the shortest vector problem. *IEEE Access*, 6:61478–61487, 2018.

[CHM+19] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. Cryptology ePrint Archive, Paper 2019/1047, 2019. https://eprint.iacr.org/2019/1047.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT (1)*, pages 409–437. Springer, 2017.

[CN11] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2011.

[Coi21] Coinmonks. Simplifying powers of tau and the trusted setup ceremony, 2021. Accessed: 2024-07-26.

[Con24] ConsenSys. gnark: Your guide to write zksnarks in go. https://docs.gnark.consensys.io, 2024. Accessed: 2024-07-29.

[Cor24a] Intel Corporation. Intel sgx for linux. https://github.com/intel/linux-sgx, 2024. Accessed: 2024-07-30.

[Cor24b] Intel Corporation. Intel software guard extensions (sgx) overview. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html, 2024. Accessed: 2024-07-30.

[COS19] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. Cryptology ePrint Archive, Paper 2019/1076, 2019. https://eprint.iacr.org/2019/1076.

[CPVK16] Michael Crosby, Pradan Pattanayak, Sanjeev Verma, and Vignesh Kalyanaraman. Blockchain technology: Beyond bitcoin. *Applied Innovation*, 2:6–10, 2016.

[DEP23] Léo Ducas, Thomas Espitau, and Eamonn W. Postlethwaite. Finding short integer solutions when the modulus is small. Cryptology ePrint Archive, Paper 2023/1125, 2023. https://eprint.iacr.org/2023/1125.

[DKL+18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-dilithium: A lattice-based digital signature scheme. In *Advances in Cryptology – CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 468–499. Springer, 2018.

[Dou10] Mohammad Sadeq Dousti. Black-box computational zero-knowledge proofs, revisited: The simulation-extraction paradigm. Cryptology ePrint Archive, Paper 2010/150, 2010. https://eprint.iacr.org/2010/150.

[dPLS18] Rafaël del Pino, Vadim Lyubashevsky, and Gregor Seiler. Lattice-based group signatures and zero-knowledge proofs of automorphism stability. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 574–591. ACM, 2018.

[ENS20] Muhammed F. Esgin, Ngoc Khanh Nguyen, and Gregor Seiler. Practical exact proofs from lattices: New techniques to exploit fully-splitting rings. In *ASIACRYPT (2)*, pages 259–288. Springer, 2020.

[ESZ21] Muhammed F. Esgin, Ron Steinfeld, and Raymond K. Zhao. Matrict+: More efficient post-quantum private blockchain payments. *IACR Cryptology ePrint Archive*, 2021:545, 2021.

[Fai17] Peter Fairley. Blockchain world - feeding the blockchain beast if bitcoin ever does go mainstream, the electricity needed to sustain it will be enormous. *IEEE Spectrum*, 54(10):36–59, 2017.

[Fou18] Zcash Foundation. Powers of tau: Trusted setup ceremony. *Zcash Foundation Blog*, 2018.

[Fou23] Hyperledger Foundation. Benchmarking hyperledger fabric 2.5 performance, 2023. Accessed: 2024-07-30.

[FS86a] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, volume 86, pages 186–194. Springer, 1986.

[FS86b] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. *Advances in Cryptology - CRYPTO'86*, pages 186–194, 1986.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC)*, pages 169–178, 2009.

[GGP13] Rosario Gennaro, Craig Gentry, and Bryan Parno. Quadratic span programs and succinct nizks without pcps. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 626–645. Springer, 2013.

[GHS11a] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. *IACR Cryptol. ePrint Arch.*, page 133, 2011.

[GHS11b] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the aes circuit. *IACR Cryptol. ePrint Arch.*, page 682, 2011.

[GINX16] Nicolas Gama, Mohamed Izabachène, Phong Q. Nguyen, and Xiaoyun Xie. Structural lattice reduction: Generalized worst-case to average-case reductions and homomorphic cryptosystems. In *EUROCRYPT 2016, volume 9666 of Lecture Notes in Computer Science*, pages 528–558. Springer, 2016.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing (STOC)*, pages 291–304, New York, NY, USA, 1985. ACM.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. Cryptology ePrint Archive, Paper 2016/260, 2016. https://eprint.iacr.org/2016/260.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *CRYPTO*, pages 75–92. Springer, 2013.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019. https://eprint.iacr.org/2019/953.

[H+20] Shai Halevi et al. Helib: An implementation of homomorphic encryption. *GitHub Repository*, 2020.

[HLP19] Thomas Hardjono, Alex Lipton, and Alex Pentland. Towards a design philosophy for interoperable blockchain systems. *arXiv preprint arXiv:1805.05934*, 2019.

[Hyp24a] Hyperledger. Hyperledger fabric samples, 2024. Accessed: 2024-07-30.

[Hyp24b] Hyperledger. Writing your first application. https://hyperledger-fabric.readthedocs.io/en/release-2.2/write_first_app.html, 2024. Accessed: 2024-07-30.

[Hyp24c] Hyperledger Fabric Private Chaincode Contributors. Hyperledger fabric private chaincode. https://github.com/hyperledger/fabric-private-chaincode, 2024. Accessed: 2024-07-29.

[Hyp24d] Hyperledger Foundation. Hyperledger caliper, 2024. Accessed: 2024-07-30.

[Hyp24e] Hyperledger Project. Hyperledger fabric sdk go. https://github.com/hyperledger/fabric-sdk-go, 2024. Accessed: 2024-08-08.

[Inc24] Docker Inc. What is a container? https://www.docker.com/resources/what-container/, 2024. Accessed: 2024-08-30.

[KN12] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. 2012.

[KPGR19] Murat Kuzlu, Manisa Pipattanasomporn, Levent Gurses, and Saifur Rahman. Performance analysis of a hyperledger fabric blockchain framework: Throughput, latency and scalability. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 536–540, 2019.

[Lin21] Yehuda Lindell. How to simulate it – a tutorial on the simulation proof technique. *Cryptology ePrint Archive*, April 2021. Paper 2021/508.

[LnLnLn22] First name Last name, First name Last name, and First name Last name. Analyzing the performance impact of hpc workloads with gramine+sgx on 3rd generation xeon scalable processors. *Journal/Conference Name*, X(Y):Z–ZZ, 2022.

[LNP22a] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Maxime Plancon. Efficient lattice-based blind signatures via gaussian one-time signatures. In *Public-Key Cryptography – PKC 2022: 25th IACR International Conference on Practice and Theory of Public-Key Cryptography, Virtual Event, March 8–11, 2022, Proceedings, Part II*, page 498–527, Berlin, Heidelberg, 2022. Springer-Verlag.

[LNP22b] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Maxime Plançon. Lattice-based zero-knowledge proofs and applications: Shorter, simpler, and more general. In *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part II*, pages 67–97. Springer, 2022. Accessed: 2024-07-26.

[LNPS21] Vadim Lyubashevsky, Ngoc Khanh Nguyen, Maxime Plançon, and Gregor Seiler. Shorter lattice-based group signatures via "almost free" encryption and other optimizations. In *ASIACRYPT (4)*, pages 218–248. Springer, 2021.

[LNS20] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Gregor Seiler. Practical lattice-based zero-knowledge proofs for integer relations. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1051–1070. ACM, 2020.

[LNS21] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Gregor Seiler. Smile: set membership from ideal lattices with applications to ring signatures and confidential transactions. In *CRYPTO (2)*, volume 12826 of *Lecture Notes in Computer Science*, pages 611–640. Springer, 2021.

[LPR13a] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *J. ACM*, 60(6), nov 2013.

[LPR13b] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *J. ACM*, 60(6), nov 2013.

[LW23] Zeyu Liu and Yunhao Wang. Amortized functional bootstrapping in less than 7ms, with $\tilde{O}(1)$ polynomial multiplications. Cryptology ePrint Archive, Paper 2023/910, 2023. https://eprint.iacr.org/2023/910.

[Lyu09] Vadim Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, pages 598–616, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[Lyu12] Vadim Lyubashevsky. Lattice signatures without trapdoors. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 738–755. Springer, 2012.

[Lyu22] Vadim Lyubashevsky. Lattice-based zero-knowledge proofs, 2022. Presented at the Quantum and Lattices Joint Reunion Workshop, Simons Institute for the Theory of Computing, June 17, 2022. Accessed: 2024-07-26.

[MAK+17] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. Rote: rollback protection for trusted execution. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 1289–1306, USA, 2017. USENIX Association.

[MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updateable structured reference strings. Cryptology ePrint Archive, Paper 2019/099, 2019. https://eprint.iacr.org/2019/099.

[Mic20] Microsoft. Microsoft seal (release 3.6). *GitHub Repository*, 2020.

[Nak08a] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Consulted*, 1, 2008.

[Nak08b] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008. Accessed: 2024-07-30.

[NBF+16] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016.

[Neu11] Peter M. Neumann. *The Mathematical Writings of Évariste Galois*. European Mathematical Society, Zürich, Switzerland, 2011.

[Ngu22] Ngoc Khanh Nguyen. *Lattice-Based Zero-Knowledge Proofs Under a Few Dozen Kilobytes*. PhD thesis, ETH Zurich, 2022. Doctoral dissertation.

[Nit20] Anca Nitulescu. zk-snarks: A gentle introduction. 2020.

[NRBB22] Valeria Nikolaenko, Sam Ragsdale, Joseph Bonneau, and Dan Boneh. Powers-of-tau to the people: Decentralizing setup ceremonies. Cryptology ePrint Archive, Paper 2022/1592, 2022. https://eprint.iacr.org/2022/1592.

[Ope] Open Quantum Safe Project. Open quantum safe. https://openquantumsafe.org/. Accessed: 2024-07-30.

[OTK+18] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical Side-Channel attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 227–240, Boston, MA, July 2018. USENIX Association.

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 238–252. IEEE, 2013.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6), sep 2009.

[Reg10] Oded Regev. The learning with errors problem. In *Proceedings of the 25th Annual IEEE Conference on Computational Complexity (CCC)*, pages 191–204, 2010.

[Sar24] Agustín Sarmiento. Crystals-dilithium (javascript implementation). https://github.com/AgustinSRG/crystals-dilithium-js, 2024. Accessed: 2024-07-30.

[sca20] Scalable computing: Challenges and opportunities. *IEEE Access*, 8:145222–145234, 2020.

[Sch89] Claus-Peter Schnorr. Efficient signature generation by smart cards. In *Journal of Cryptology*, volume 4, pages 161–174, 1989.

[Sch21] Felix Schuster. Introduction to gramine, 2021. YouTube video, Accessed: 2024-07-26.

[SJTS13] Srinath Setty, Abhishek Jain, Abhradeep Thakurta, and Elaine Shi. Proving the correctness of a cloud-hosted program. In *ACM Symposium on Operating Systems Principles*, pages 237–252. ACM, 2013.

[Sta23] StarkWare Team. *ethSTARK Documentation – Version 1.2*, 2023. https://starkware.co.

[SWTR18] Harish Sukhwani, Nan Wang, Kishor S. Trivedi, and Andy Rindos. Performance modeling of hyperledger fabric (permissioned blockchain network). In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–8, 2018.

[TNV19] Parth Thakkar, Senthil Nathan, and Balaji Viswanathan. Performance analysis of hyperledger fabric platform: A hierarchical model approach. *Peer-to-Peer Networking and Applications*, 13(2):528–544, 2019.

[TPV17] Chia-Che Tsai, Donald E. Porter, and Mihir Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658. USENIX Association, 2017.

[TT16] Don Tapscott and Alex Tapscott. *Blockchain Revolution: How the Technology Behind Bitcoin Is Changing Money, Business, and the World*. Penguin, 2016.

[Wei40] André Weil. Sur les fonctions algébriques à corps de constantes finis. *C. R. Acad. Sci. Paris*, 210:592–594, 1940. = Oeuvres Scientifiques, Volume I, pp. 257–259.

[Woo14a] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014.

[Woo14b] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014. https://ethereum.github.io/yellowpaper/paper.pdf.

[WP22] Yuyu Wang and Jiaxin Pan. Non-interactive zero-knowledge proofs with fine-grained security. Cryptology ePrint Archive, Paper 2022/548, 2022. https://eprint.iacr.org/2022/548.

[WW19] Yang Wang and Mingqiang Wang. Module-lwe versus ring-lwe, revisited. Cryptology ePrint Archive, Paper 2019/930, 2019. https://eprint.iacr.org/2019/930.

Appendix A: LWE to R1CS Transformation Pseudocode

```
FUNCTION LWEToR1CS_transform()
    n <- initialize dimension

    num_gates <- n * n * 2
    num_variables <- n * n + 2 * n + num_gates + 1

    A_matrix <- zeros(num_gates, num_variables)
    B_matrix <- zeros(num_gates, num_variables)
    C_matrix <- zeros(num_gates, num_variables)

    // Processing gates a0 - an^2
    FOR i FROM 0 TO n * n - 1 DO
        A_matrix[i][i + n + 1] <- 1
        B_matrix[i][1 + n + n * n + (i % n)] <- 1
        C_matrix[i][1 + 3 * n + n * n + i] <- 1
    END FOR

    // Initialize counters
    c <- 0
    d <- 0
    z <- 0
    p <- 0

    // Processing gates c0 - cn^2 and outputs/t
    FOR i FROM 0 TO n * n - 1 DO
        B_matrix[n * n + i][0] <- 1

        IF (i % n) = 0 THEN
            A_matrix[n * n + i][1 + 3 * n + i + n * n] <- 1
            A_matrix[n * n + i][1 + 3 * n + n * n + i + 1] <- 1
            C_matrix[n * n + i][1 + 3 * n + z + 2 * n * n] <- 1
            z <- z + 1
        ELSE IF (i + 1) % n  0 THEN
            A_matrix[n * n + i][1 + 3 * n + i + n * n + 1] <- 1
            A_matrix[n * n + i][1 + 3 * n + c + 2 * n * n] <- 1
            C_matrix[n * n + i][1 + 3 * n + z + 2 * n * n] <- 1
            c <- c + 1
            z <- z + 1
        ELSE
            IF (1 + 3 * n + c + 2 * n * n) < num_variables THEN
                A_matrix[n * n + i][1 + 3 * n + c + 2 * n * n] <- 1
                A_matrix[n * n + i][1 + 2 * n + n * n + d] <- 1
                C_matrix[n * n + i][1 + p] <- 1
                c <- c + 1
                d <- d + 1
                p <- p + 1
            END IF
        END IF
    END FOR

    RETURN A_matrix, B_matrix, C_matrix
END FUNCTION
```

Appendix B: R1CS to QAP Transformation Pseudocode

```
FUNCTION interpolateColumn(col, nb)
    xs <- GF.newVectorFrom(Array from 1 to nb)
    ys <- GF.newVectorFrom(col)
    RETURN GF.interpolate(xs, ys)

FUNCTION getPolysOfMatrix(matrix)
    polys <- []
    nbOfRows <- number of rows in matrix
    nbOfColumns <- number of columns in matrix

    FOR colId FROM 0 TO nbOfColumns - 1 DO
        column <- []
        FOR row FROM 0 TO nbOfRows - 1 DO
            column.append(BigInt(matrix[row][colId]))
        END FOR
        polys.append(interpolateColumn(column, nbOfRows))
    END FOR

    RETURN polys

FUNCTION polySum(s)
    {A_matrix: A, B_matrix: B, C_matrix: C} <- LWEToR1CS_transform()

    sBigInt <- s mapped to BigInt
    sVector <- math.matrix(sBigInt)

    A_mat <- math.matrix(A)
    B_mat <- math.matrix(B)
    C_mat <- math.matrix(C)

    C_result <- math.multiply(C_mat, sVector)
    AB_result <- math.dotMultiply(math.multiply(A_mat, sVector), math.multiply(B_mat, sVector))

    U_polys <- getPolysOfMatrix(A)
    V_polys <- getPolysOfMatrix(B)
    W_polys <- getPolysOfMatrix(C)

    Ua <- GF.newVectorFrom([0n])
    FOR EACH si IN sBigInt DO
        Ua <- GF.addPolys(Ua, GF.mulPolyByConstant(U_polys[i], si))
    END FOR

    Va <- GF.newVectorFrom([0n])
    FOR EACH si IN sBigInt DO
        Va <- GF.addPolys(Va, GF.mulPolyByConstant(V_polys[i], si))
    END FOR

    Wa <- GF.newVectorFrom([0n])
    FOR EACH si IN sBigInt DO
        Wa <- GF.addPolys(Wa, GF.mulPolyByConstant(W_polys[i], si))
    END FOR

    T <- GF.newVectorFrom([1n, order - 1n])
```

```
FOR i FROM 2 TO length of A DO
    poly <- GF.newVectorFrom([1n, order - BigInt(i)])
    T <- GF.mulPolys(T, poly)
END FOR

RETURN { Ua, Va, Wa, T }
```

Appendix C: Vesper Proof Payload Generation Pseudocode

```
INITIALIZE order
INITIALIZE cypher modulus q
INITIALIZE plaintext modulus t
INITIALIZE ring dimension d
delta <- BigInt(q) / BigInt(t)

polynomial_modulus_p_q <- [1n, followed by d-1 zeros, 1n]

FUNCTION setup()
    alpha <- RANDOM BigInt in range [0, q)
    sk <- FF.newVectorFrom(Array of length d with RANDOM BigInt values in range [0, 2))
    a2 <- FF.newVectorFrom(Array of length d with RANDOM BigInt values in range [0, q))
    e <- FF.newVectorFrom(Array of length d with RANDOM BigInt values in range [0, q))
    neg_a2_sk <- FF.negVectorElements(FF.mulPolys(a2, sk))
    pk_0 <- FF.addPolys(neg_a2_sk, e)
    pk_1 <- a2
    pk <- [pk_0, pk_1]
    u <- FF.newVectorFrom(Array of length d with RANDOM BigInt values in range [0, 2))
    e1 <- FF.newVectorFrom(Array of length d with RANDOM BigInt values in range [0, q))
    e2 <- FF.newVectorFrom(Array of length d with RANDOM BigInt values in range [0, q))

    RETURN { alpha, sk, a2, e, pk, u, e1, e2 }
END FUNCTION

FUNCTION prover(pk, u, e1, alpha, s)
    { Ua, Va, Wa, T } <- polySum(s)

    UaFiltered <- FILTER Ua.toValues() WHERE value != 0n
    VaFiltered <- FILTER Va.toValues() WHERE value != 0n
    WaFiltered <- FILTER Wa.toValues() WHERE value != 0n
    T_rev <- T.toValues()

    H <- FF.divPolys(FF.subPolys(FF.mulPolys(Ua, Va), Wa), T)
    HFiltered <- FILTER H.toValues() WHERE value != 0

    Left <- (FF.evalPolyAt(Ua, alpha) * FF.evalPolyAt(Va, alpha) - FF.evalPolyAt(Wa, alpha)) % order
    Right <- (FF.evalPolyAt(T, alpha) * FF.evalPolyAt(H, alpha)) % order

    c1 <- FF.addPolys(FF.addPolys(FF.mulPolys(pk[0], u), e1), FF.mulPolyByConstant(FF.newVectorFrom([
    c2 <- FF.addPolys(FF.addPolys(FF.mulPolys(pk[0], u), e1), FF.mulPolyByConstant(FF.newVectorFrom([

    proof <- [c1, c2]
    proofHex <- proofToHex(proof)
    { public_key, signature } <- signProof(proofHex)
    RETURN { proofHex, signature, public_key }
END FUNCTION
```

Appendix D: LWE-hardness Estimator based on Parameter Input

```
INITIALIZE nu TO 1
INITIALIZE kmlwe TO 0
INITIALIZE kmlwe_hardness TO 2
INITIALIZE d TO 64
INITIALIZE logq TO 8

FUNCTION findMLWEdelta(nu, n, d, logq)
    LOAD "https://bitbucket.org/malb/lwe-estimator/raw/HEAD/estimator.py"

    n <- n * d
    q <- 2^logq
    stddev <- sqrt(((2 * nu + 1)^2 - 1) / 12)
    alpha <- alphaf(sigmaf(stddev), q)

    SET verbose TO 1
    L <- estimate_lwe(n, alpha, q, reduction_cost_model=BKZ.enum)

    delta_enum1 <- L['usvp']['delta_0']
    delta_enum2 <- L['dec']['delta_0']
    delta_enum3 <- L['dual']['delta_0']

    L <- estimate_lwe(n, alpha, q, reduction_cost_model=BKZ.sieve)

    delta_sieve1 <- L['usvp']['delta_0']
    delta_sieve2 <- L['dec']['delta_0']
    delta_sieve3 <- L['dual']['delta_0']

    RETURN max(delta_enum1, delta_enum2, delta_enum3, delta_sieve1, delta_sieve2, delta_sieve3)
END FUNCTION

WHILE kmlwe_hardness > 1.0045 DO
    kmlwe <- kmlwe + 1
    kmlwe_hardness <- findMLWEdelta(nu, kmlwe, d, logq)
END WHILE
```

Appendix E: Vesper Verification Pseudocode

```
FUNCTION verifier(proofHex, public_key, signature)
    isSigValid <- verifySignature(public_key, proofHex.hex_string, signature) == 1

    proof <- hexToProof(proofHex.hex_string, proofHex.shape)

    IF arrayEquals(proof[0], proof[1]) AND isSigValid THEN
        RETURN true
    ELSE
        RETURN false
    END IF
END FUNCTION
```

Appendix F: Vesper Chaincode Pseudocode

```
CLASS SmartContract EXTENDS Contract

    FUNCTION verifySignature(publicKeyStr, proof, signatureStr)
        TRY
            level <- GET_SECURITY_LEVEL(2)
            publicKey <- CREATE_PUBLIC_KEY_FROM_BASE64(publicKeyStr, level)
            message <- CONVERT_PROOF_TO_BYTE_ARRAY(proof)
            signature <- CREATE_SIGNATURE_FROM_BASE64(signatureStr, level)
            RETURN VERIFY_SIGNATURE(publicKey, message, signature)
        CATCH error
            RETURN false
        END TRY
    END FUNCTION

    FUNCTION VerifyProof(ctx, proofString)
        TRY
            payloadParsed <- PARSE_JSON(proofString)
            proofHex <- payloadParsed.proof
            publicKey <- payloadParsed.public_key
            signature <- payloadParsed.signature

            proof <- CONVERT_HEX_TO_PROOF(proofHex.hex_string, proofHex.shape)

            IF NOT IS_ARRAY(proof) OR NOT ARRAY_EQUALS(proof[0], proof[1])
                THROW 'Invalid proof format'

            isSignatureValid <- verifySignature(publicKey, proofHex.hex_string, signature)
            RETURN isSignatureValid
        CATCH error
            RETURN false
        END TRY
    END FUNCTION

END CLASS

EXPORT SmartContract
```

Appendix G: Vesper Smart Contract Communication Middleware Pseudocode

```
INITIALIZE express
INITIALIZE app
port <- 2800

// Import necessary modules
IMPORT necessary modules (e.g., grpc, fabric-gateway, crypto, fs, path, util)

// Define utility functions for binding and importing modules
DEFINE __createBinding
DEFINE __setModuleDefault
DEFINE __importStar

// Define configurations with default values
channelName <- environment variable 'CHANNEL_NAME' OR 'default_channel'
chaincodeName <- environment variable 'CHAINCODE_NAME' OR 'default_chaincode'
mspId <- environment variable 'MSP_ID' OR 'default_msp'

// Define paths to various directories and files
cryptoPath <- environment variable 'CRYPTO_PATH' OR 'default_crypto_path'
keyDirectoryPath <- environment variable 'KEY_DIRECTORY_PATH' OR 'default_key_directory_path'
certDirectoryPath <- environment variable 'CERT_DIRECTORY_PATH' OR 'default_cert_directory_path'
tlsCertPath <- environment variable 'TLS_CERT_PATH' OR 'default_tls_cert_path'
peerEndpoint <- environment variable 'PEER_ENDPOINT' OR 'default_peer_endpoint'
peerHostAlias <- environment variable 'PEER_HOST_ALIAS' OR 'default_peer_host_alias'

// Define helper functions
FUNCTION newGrpcConnection()
    tlsCert <- READ tlsCertPath
    tlsCredentials <- CREATE tlsCredentials USING tlsCert
    RETURN new grpc.Client(peerEndpoint, tlsCredentials, options)
END FUNCTION

FUNCTION newIdentity()
    certFileName <- getFirstDirFileName(certDirectoryPath)
    certificate <- READ certFileName
    RETURN { mspId, credentials: certificate }
END FUNCTION

FUNCTION getFirstDirFileName(dirPath)
    files <- READ files in dirPath
    RETURN first file in files
END FUNCTION

FUNCTION newSigner()
    keyFileName <- getFirstDirFileName(keyDirectoryPath)
    privateKeyPem <- READ keyFileName
    privateKey <- CREATE privateKey USING privateKeyPem
    RETURN fabric_gateway_1.signers.newPrivateKeySigner(privateKey)
END FUNCTION

FUNCTION envOrDefault(key, defaultValue)
    RETURN environment variable key OR defaultValue
END FUNCTION
```

```
FUNCTION displayInputParameters()
    PRINT 'Channel Name:', channelName
    PRINT 'Chaincode Name:', chaincodeName
    PRINT 'MSP ID:', mspId
    PRINT 'Crypto Path:', cryptoPath
    PRINT 'Key Directory Path:', keyDirectoryPath
    PRINT 'Cert Directory Path:', certDirectoryPath
    PRINT 'TLS Cert Path:', tlsCertPath
    PRINT 'Peer Endpoint:', peerEndpoint
    PRINT 'Peer Host Alias:', peerHostAlias
END FUNCTION

// Define VerifyProof function
FUNCTION VerifyProof(contract, proof)
    PRINT 'Evaluating Transaction: VerifyProof'
    resultBytes <- EVALUATE transaction 'VerifyProof' on contract WITH proof
    resultJson <- DECODE resultBytes
    result <- PARSE resultJson
    RETURN result
END FUNCTION

// Define main function
FUNCTION main(inputProof)
    CALL displayInputParameters()
    client <- CREATE gRPC client connection
    gateway <- CONNECT to fabric gateway USING client, identity, signer, and timeout options
    TRY
        network <- GET network instance representing the channel
        contract <- GET smart contract from the network
        response <- VerifyProof(contract, inputProof)
        CLOSE gateway
        CLOSE client
        RETURN response
    CATCH error
        PRINT error
    END TRY
END FUNCTION

// Middleware to parse JSON bodies
USE express.json()

// Define the /checkProof endpoint
DEFINE endpoint '/checkProof' TO handle POST requests:
    LOG 'API called'
    data <- GET data from request body
    result <- main(data)
    RETURN JSON response WITH result
END DEFINE

// Start the server
START server ON port 2800
PRINT 'Server is running on http://localhost:2800'
```

Appendix H: Vesper-FPC Chaincode Pseudocode

```
CONSTANT boundValue TO 900 (for q = 2**8 and d = 64)

FUNCTION unescapeJsonString(escapedJson)
    result <- empty string
    i <- 0
    WHILE i < LENGTH(escapedJson)
        IF escapedJson[i] = '\' AND i + 1 < LENGTH(escapedJson)
            i <- i + 1
            SWITCH escapedJson[i]
                CASE '\"': ADD '\"' TO result
                CASE '\\': ADD '\\' TO result
                CASE '/': ADD '/' TO result
                CASE 'b': ADD BACKSPACE TO result
                CASE 'f': ADD FORM FEED TO result
                CASE 'n': ADD NEWLINE TO result
                CASE 'r': ADD CARRIAGE RETURN TO result
                CASE 't': ADD TAB TO result
                DEFAULT: ADD escapedJson[i] TO result
        ELSE
            ADD escapedJson[i] TO result
        i <- i + 1
    RETURN result

FUNCTION extractJsonValue(json, key)
    SEARCH json FOR key
    IF key IS FOUND
        value <- EXTRACT value FOR key FROM json
        IF value IS A STRING
            RETURN string value
        ELSE IF value IS AN ARRAY
            RETURN array value
        ELSE
            RETURN number OR boolean value
    ELSE
        RETURN empty string

FUNCTION parseJsonArray(jsonArrayStr)
    REMOVE BRACKETS FROM jsonArrayStr
    SPLIT jsonArrayStr BY COMMAS
    result <- empty list
    FOR each value IN SPLIT jsonArrayStr
        ADD CONVERTED integer value TO result
    RETURN result

FUNCTION parseJsonArrayOfArrays(jsonArrayStr)
    result <- empty list
    REMOVE BRACKETS FROM jsonArrayStr
    SPLIT jsonArrayStr BY '],['
    FOR each array string IN SPLIT jsonArrayStr
        REMOVE BRACKETS FROM array string
        SPLIT array string BY COMMAS
        arrayList <- empty list
        FOR each value IN SPLIT array string
            ADD CONVERTED integer value TO arrayList
```

```
        ADD arrayList TO result
    RETURN result

FUNCTION fiatShamirChallenge(data, c)
    hashInt <- SUM(data)
    hashInt <- hashInt MOD LENGTH(c)
    RETURN c[hashInt]

FUNCTION verifySignature(signature, m, t_A, c, a1, a2, Bext, q)
    w, z1, z2 <- signature
    mFlat <- FLATTEN(m) USING flatten2DArray
    wData <- CONCATENATE(w, mFlat)
    challenge <- fiatShamirChallenge(wData, c)
    IF norms OF z1 AND z2 ARE WITHIN bounds USING norm
        a1z1 <- matrixVectorMultiply(a1, z1)
        a2z2 <- matrixVectorMultiply(a2, z2)
        a1z1_plus_a2z2 <- vectorAdd(a1z1, a2z2)
        c_times_tA <- mod(vectorMultiply(t_A, challenge), q)
        w_prime <- mod(vectorSubtract(a1z1_plus_a2z2, c_times_tA), q)
        RETURN TRUE IF arrayEquals(w, w_prime), ELSE FALSE
    ELSE
        RETURN FALSE

FUNCTION VerifyProof(escapedJson, ctx)
    TRY
        json <- unescapeJsonString(escapedJson)
        hexString <- extractJsonValue(json, "hex_string")
        shapeStr <- extractJsonValue(json, "shape")
        signatureStr <- extractJsonValue(json, "signature")
        tAStr <- extractJsonValue(json, "t_A")
        cStr <- extractJsonValue(json, "C")
        a1Str <- extractJsonValue(json, "A1")
        a2Str <- extractJsonValue(json, "A2")
        bextStr <- extractJsonValue(json, "Bext")
        qStr <- extractJsonValue(json, "q")

        shape <- parseJsonArray(shapeStr)
        signature <- parseJsonArrayOfArrays(signatureStr)
        tA <- parseJsonArray(tAStr)
        c <- parseJsonArray(cStr)
        a1 <- parseJsonArrayOfArrays(a1Str)
        a2 <- parseJsonArrayOfArrays(a2Str)
        bext <- parseJsonArrayOfArrays(bextStr)
        q <- CONVERT qStr TO INTEGER

        proofArray <- CONVERT hexString TO proof array BASED ON shape

        IF proofArray IS EMPTY OR NOT arrayEquals(shape, proofArray)
            THROW EXCEPTION "invalid proof format"

        isValidSig <- verifySignature(signature, proofArray, tA, c, a1, a2, bext, q)
        RETURN "true" IF isValidSig, ELSE "false"
    CATCH EXCEPTION
        RETURN "false"

FUNCTION invoke(response, max_response_len, actual_response_len, ctx)
```

```
LOG "Executing chaincode invocation"
function_name, params <- ctx
result <- ""
IF function_name = "VerifyProof"
    IF LENGTH(params)  1
        result <- "Invalid number of parameters"
    ELSE
        result <- VerifyProof(params[0], ctx)
ELSE
    LOG "Unknown transaction"
    RETURN -1
IF LENGTH(result) > max_response_len
    LOG "Response buffer too small"
    actual_response_len <- 0
    RETURN -1
COPY result TO response
actual_response_len <- LENGTH(result)
LOG "Response: " + result
RETURN 0
```

Appendix I: Vesper-FPC Signing Pseudocode

```
INITIALIZE cypher modulus q
INITIALIZE boundValue TO 900 (for q = 2**8 and d = 64)
INITIALIZE ring dimension d

FUNCTION sign(m, A1, A2, Bext, s1, s2, C, q)
    n <- length of s1

    y1 <- discreteGaussianSample(0, 1, n, q)
    y2 <- discreteGaussianSample(0, 1, n, q)

    w <- vectorAdd(matrixVectorMultiply(A1, y1), matrixVectorMultiply(A2, y2))

    mFlat <- flatten2DArray(m)
    wData <- concatenate(w, mFlat)

    c <- fiatShamirChallenge(wData, C)

    z1 <- (y1 + c * s1) MOD q
    z2 <- (y2 + c * s2) MOD q

    t_A <- vectorAdd(matrixVectorMultiply(A1, s1), matrixVectorMultiply(A2, s2))
    t_B <- vectorAdd(matrixVectorMultiply(Bext, s2), sum of rows in m MOD q)

    signature <- [w, z1, z2]
    RETURN [signature, t_A, t_B]
END FUNCTION

FUNCTION verifySignature(signature, m, t_A, C, A1, A2, Bext, q)
    [w, z1, z2] <- signature

    mFlat <- flatten2DArray(m)
    wData <- concatenate(w, mFlat)

    c <- fiatShamirChallenge(wData, C)

    IF norm(z1) <= 1.5 * boundValue AND norm(z1) >= boundValue AND norm(z2) <= 1.5 * boundValue AND n
        w_prime <- vectorSubtract(vectorAdd(matrixVectorMultiply(A1, z1), matrixVectorMultiply(A2, z2
        RETURN w EQUALS w_prime
    ELSE
        RETURN false
    END IF
END FUNCTION
```

Appendix J: Vesper-FPC Communication Middleware Pseudocode

```
PACKAGE main

IMPORT necessary packages (encoding/json, fmt, ioutil, log, net/http, os, os/exec, strconv)

DEFINE STRUCT Input
    FIELD Proof
        FIELD HexString AS string
        FIELD Shape AS array of integers
    FIELD Signature AS 2D array of integers
    FIELD TA AS array of integers
    FIELD C AS array of integers
    FIELD A1 AS 2D array of integers
    FIELD A2 AS 2D array of integers
    FIELD Bext AS 2D array of integers
    FIELD Q AS integer
END STRUCT

FUNCTION handler(w, r)
    IF request method IS NOT POST
        RETURN method not allowed error

    body <- READ all bytes from request body
    IF error reading body
        RETURN bad request error

    input <- new Input
    error <- UNMARSHAL JSON body INTO input
    IF error unmarshaling JSON
        RETURN bad request error

    PRINT received input

    inputJson <- MARSHAL input TO JSON
    IF error marshaling JSON
        RETURN internal server error

    escapedInputJson <- QUOTE string representation of inputJson
    LOG escaped JSON input

    cmd <- CREATE new command to execute script with escapedInputJson as argument
    output <- RUN command and get combined output
    IF error executing command
        LOG error
        RETURN internal server error

    logFile <- CREATE new log file
    IF error creating log file
        LOG error
        RETURN internal server error
    DEFER CLOSE log file

    WRITE output TO log file
    IF error writing to log file
```

```
        LOG error
        RETURN internal server error

    SET response header content type to application/json
    SET response status to OK
    response <- CREATE map with message and output
    ENCODE response to JSON AND write to response writer
END FUNCTION

FUNCTION main()
    REGISTER handler function for /verifyProof endpoint
    PRINT server listening message
    START server on port 8080 AND log fatal errors if any
END FUNCTION
```

Appendix K: Automatic Witness Generator Pseudocode

```
FUNCTION witnessGenerator()
Initialize dimension n
SET max_value TO 12 (adjustable)

// Function to generate random integers between min and max (inclusive)
FUNCTION getRandomInt(min, max)
    RETURN FLOOR(RANDOM() * (max - min + 1)) + min
END FUNCTION

// Generate matrix A with values between 0 and max_value
INITIALIZE A AS empty list
FOR i FROM 0 TO n - 1
    INITIALIZE A[i] AS empty list
    FOR j FROM 0 TO n - 1
        A[i][j] <- getRandomInt(0, max_value)
    END FOR
END FOR

// Generate vector s with values between 0 and max_value
INITIALIZE s AS empty list
FOR i FROM 0 TO n - 1
    s[i] <- getRandomInt(0, max_value)
END FOR

// Generate vector e with values between 0 and max_value
INITIALIZE e AS empty list
FOR i FROM 0 TO n - 1
    e[i] <- getRandomInt(0, max_value)
END FOR

INITIALIZE a AS list of n lists each containing n zeros
INITIALIZE c AS list of n lists each containing (n-1) zeros
INITIALIZE out AS list of n zeros

// Compute a values
FOR i FROM 0 TO n - 1
    FOR j FROM 0 TO n - 1
        a[i][j] <- A[i][j] * s[j]
    END FOR
END FOR

// Compute c values and final output values
FOR i FROM 0 TO n - 1
    c[i][0] <- a[i][0] + a[i][1]
    FOR j FROM 1 TO n - 2
        c[i][j] <- c[i][j - 1] + a[i][j + 1]
    END FOR
    out[i] <- c[i][n - 2] + e[i]
END FOR

A_flat <- FLATTEN A
a_flat <- FLATTEN a
c_flat <- FLATTEN c
```

```
    // Initialize s_example with the constant 1
    INITIALIZE s_example AS list containing 1

    // Add output values
    FOR EACH value IN out
        ADD FLOOR(value) TO s_example
    END FOR

    // Add matrix A elements
    FOR EACH value IN A_flat
        ADD FLOOR(value) TO s_example
    END FOR

    // Add vector s elements
    FOR EACH value IN s
        ADD FLOOR(value) TO s_example
    END FOR

    // Add vector e elements
    FOR EACH value IN e
        ADD FLOOR(value) TO s_example
    END FOR

    // Add intermediate a values
    FOR EACH value IN a_flat
        ADD FLOOR(value) TO s_example
    END FOR

    // Add intermediate c values
    FOR EACH value IN c_flat
        ADD FLOOR(value) TO s_example
    END FOR

    // Display the first 100 values to check
    PRINT first 100 values of s_example
    PRINT LENGTH of s_example

    RETURN s_example
END FUNCTION
```