Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Lars Sørensen
Vegard Nyeng

The Application and Use of Cryptographic Zero-Knowledge Protocols

Master's thesis in Communication Technology and Digital Security
Supervisor: Stig Frode Mjølsnes
June 2023

NTNU
Norwegian University of Science and Technology

Lars Sørensen
Vegard Nyeng

The Application and Use of Cryptographic Zero-Knowledge Protocols

Master's thesis in Communication Technology and Digital Security
Supervisor: Stig Frode Mjølsnes
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

Norwegian University of
Science and Technology

The Application and Use of Cryptographic Zero-Knowledge Protocols

Lars Sørensen and Vegard Nyeng

Submission date June 2023
Supervisor Professor Stig Frode Mjølsnes, NTNU

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Title: The Application and Use of Cryptographic Zero-Knowledge Protocols

Students: Lars Sørensen and Vegard Nyeng

Problem description:

Authentication is a process by which a prover seeks to convince a verifier of their identity. This is considered trivial in more common full-knowledge proofs in which the prover simply discloses some secret knowledge (such as a password) to the verifier. However, such authentication methods are vulnerable to privacy breaches if the secret knowledge is revealed to unauthorized parties. A zero-knowledge proof lets a prover convince a verifier that a given statement is true without revealing any additional information about the knowledge of the statement they possess. These proofs commonly rely on cryptography based on mathematically hard problems, such as integer factorization or discrete logarithms, to obfuscate the knowledge.

Given the advent of quantum computers and Shor's algorithm, it is imperative to develop quantum-resistant cryptography. Consequently, the objective of this thesis is to create a zero-knowledge proof that employs cryptography believed to be resistant to quantum attacks. Subsequently, the zero-knowledge proof will be implemented as a zero-knowledge protocol used by a client, functioning as the prover, to authenticate themselves to a server, functioning as the verifier. The resulting protocol will be evaluated for its performance, security, and practicality for authentication purposes. Overall, the objective of this thesis is to contribute to the development of secure and efficient authentication systems that preserve the privacy of users.

Approved on: 2023-02-21

Supervisor: Professor Stig Frode Mjølsnes, NTNU

Abstract

Cryptographic systems today rely mainly on three mathematical hard problems. With the rapid advancement of technology, these cryptosystems face an imminent threat from the development of powerful quantum computers. With the introduction of a sufficiently powerful quantum computer, many currently deployed cryptosystems may be broken, compromising the security of sensitive information. To address this challenge, post-quantum cryptography has emerged, aiming to develop cryptographic schemes resistant to attacks by classical and quantum computers. One promising branch of post-quantum cryptography is lattice cryptography, which utilizes hard problems within mathematical structures in multiple dimensions to provide strong security guarantees.

This master's thesis focuses on implementing and analyzing a lattice-based digital signature scheme. Digital signatures play a crucial role in ensuring data integrity, authentication, and non-repudiation in various applications. This research aims to investigate the practicality and performance of a lattice-based digital signature scheme within a passwordless authentication system. Integrating this scheme into the authentication process seeks to enhance security while providing a convenient and user-friendly experience for the end users. The efficiency of the lattice-based digital signature scheme will be reviewed through performance analysis.

The results of this thesis will contribute to the understanding of lattice-based cryptography and its practical application in passwordless authentication systems. Furthermore, the analysis of performance metrics will shed light on the strengths and limitations of the implemented digital signature.

Sammendrag

Kryptografiske systemer som benyttes i dag, er hovedsakelig basert på tre matematiske problemer. Rask utvikling innen teknologi gjør at disse systemene står i fare dersom kraftige nok kvantemaskiner blir utviklet. Slike maskiner har egenskaper som gjør at de kan løse de underliggende problemene som dagens kryptografiske systemer benytter. Kvantesikker kryptografi er et felt innen kryptografi som fokuserer på kryptografiske systemer som kan motstå angrep fra kvantemaskiner. Et lovende felt innen kvantesikker kryptografi er kryptografi basert på gitterstruktur, som bygger på vanskelige problemer innen matematiske strukturer i flere dimensjoner.

Denne oppgaven ser på implementasjonen av og analyserer en digital signatur som benytter kryptografi basert på gitterstruktur. Digitale signaturer spiller en viktig rolle for å sikre dataintegritet, autentisering og ikke-fornektelse i en rekke anvendelser. En slik digital signatur vil bli implementert i et passordløst autentiseringssystem, hvor ytelsen til signaturen vil bli vurdert. Målet med å integrere denne signaturen i en autentiseringsprosess er å forbedre sikkerheten samtidig som en sømløs og enkel brukeropplevelse tilbys. Gjennom en analyse av systemets ytelse vil signaturens effektivitet også bli vurdert.

Resultatene fra denne oppgaven vil bidra til en dypere forståelse av kryptografi basert på gitterstrukturer og dens praktiske implementasjon i autentiseringssystemer. Analysen av systemets ytelse vil gi innsikt i styrker og svakheter ved den implementerte digitale signaturen.

Preface

Through five years we persevered, side by side,
Exploring Communication Technology with pride.
At NTNU, we sought knowledge far and wide,
Equipped with our Master's, we embark on life's vibrant ride.

To Professor Stig Frode Mjølsnes, our guide so wise,
Your wisdom in cryptography, a beacon that flies.
Your guidance and feedback, invaluable and true,
Shaping our thesis, bringing it to breakthrough.

Our gratitude extends to friends and kin,
For the support and love that's always been,
Countless table tennis matches, oh, what delight,
In the lounge, bringing joy, making days so bright.

With gratitude in our hearts, we end this preface,
To acknowledge those who've contributed to our thesis.
The culmination of knowledge, hard work, and delight,
May this journey inspire others to reach new heights.

Lars Sørensen & Vegard Nyeng
Trondheim, Norway
June 2023

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

List of Acronyms

AES Advanced Encryption Standard.

API Application Programming Interface.

APN Apple Push Notification Service.

AVX2 Advanced Vector Extensions 2.

BSON Binary Encoded JavaScript Object Notation.

CORS Cross-Origin Resource Sharing.

COSE CBOR Object Signing and Encryption.

CPRNG Cryptographic Pseudo-Random Number Generator.

CPU Central Processing Unit.

CRT Chinese Remainder Theorem.

CSPRNG Cryptographically Secure Pseudo-Random Number Generator.

CTAP Client to Authenticator Protocol.

CVP Closest Vector Problem.

DSA Digital Signature Algorithm.

DVCS Distributed Version Control System.

ECC Elliptic Curve Cryptography.

ECDSA Elliptic Curve Digital Signature Algorithm.

EdDSA Edwards-curve Digital Signature Algorithm.

FFT Fast Fourier Transform.

FIDO2 Fast Identity Online.

HTTP Hypertext Transfer Protocol.

IDE Integrated Development Environment.

IETF Internet Engineering Task Force.

IP Internet Protocol.

IPSec Internet Protocol Security.

JSON JavaScript Object Notation.

LWE Learning With Errors.

MFA Multi-factor Authentication.

MSB Most Significant Bit.

NFC Near-Field Communication.

NIST National Institute of Standards and Technology.

NTT Number Theoretic Transform.

OS Operative System.

PIN Personal Identification Number.

PoC Proof of concept.

PoK Proof of Knowledge.

PQC Post-Quantum Cryptography.

QROM Quantum Random Oracle Model.

ROM Random Oracle Model.

RP Relying Party.

RPID Relying Party Identity.

RSA Rivest-Shamir-Adleman.

SDK Software Development Kit.

SIMD Single Instruction Multiple Data.

SIS Short Integer Solution.

SoC System on Chip.

SSH Secure Shell.

SVP Shortest Vector Problem.

TLS Transport Layer Security.

USB Universal Serial Bus.

UUID Universally Unique IDentifier.

VS Code Visual Studio Code.

W3C World Wide Web Consortium.

WebAuthn Web Authentication.

XML Extensible Markup Language.

XOF eXtendable-Output Function.

Chapter 1
Introduction

This introductory chapter begins by outlining the motivation behind the thesis. The problem area this thesis seeks to contribute to is presented before a research scope is defined. Next, possible challenges and limitations are described. The research questions this thesis seeks to address and the research objectives are listed. Finally, an outline of the entire thesis is presented.

1.1 Motivation

Proving identity is something many people face in their daily lives. Whether they are logging in on their work mail or simply shopping online, a proof model is needed for them, the prover, to prove their identity to a service, the verifier. This is traditionally done by revealing secret information to the service, which is used to verify their identity, e.g., a password. This is called *full-knowledge proofs*. Zero-knowledge proofs allow a prover to prove that a given statement is true without revealing any additional information about the statement beside the fact that it is true. Proving identity can therefore be done without revealing any sensitive information, thus preserving the user's privacy.

Zero-knowledge protocols are often constructed using public-key cryptography, which relies on the computational hardness of mathematical problems. Common problems are integer factorization, used in the Rivest-Shamir-Adleman (RSA) cryptosystem, discrete logarithms over finite fields, used in the Digital Signature Algorithm (DSA), and discrete logarithms over elliptic curves, used in the Elliptic Curve Digital Signature Algorithm (ECDSA). All these problems withstand the best-known algorithms on classical computers, but this is not the case with quantum computers due to Shor's algorithm [Sho97]. It is predicted that if the research and development of quantum computers progress, the building blocks of modern cryptosystems will break, thus also breaking common cryptographic protocols such as Transport Layer Security (TLS), Internet Protocol Security (IPSec), and Secure Shell (SSH). The

need to introduce new cryptosystems and standards that builds upon cryptography resistant to attacks from quantum computers, i.e., quantum-resistant cryptography, is thus present. Even though such a situation is not in the near future, one should be prepared as developing new cryptosystems and standards is tedious. Another reason for migrating to quantum-resistant cryptography is to protect ourselves from "store now, decrypt later" attacks, where cryptanalysis occurs after traditional cryptosystems are efficiently broken. New cryptographic algorithms need to be developed, validated, and standardized to ensure the cryptosystems of tomorrow can withstand quantum computers [NS22].

In 2017, the National Institute of Standards and Technology (NIST) initiated a process to "solicit, evaluate and standardize one or more quantum-resistant public-key cryptographic algorithms" [NIS17]. This process turned into the Post-Quantum Cryptography Competition, which in 2022 culminated in four algorithms selected to be standardized [NIS22].

This thesis is a combination of both theoretical and practical work. Implementing a solution that results in a product appealed to both authors and served as a motivation throughout the work.

1.2 Research Scope

Digital access systems today are mainly based on knowledge factors, where the user has to provide some knowledge, e.g., a password, to gain access to a service. Textual passwords are the most common authentication method, where the user and a service share a secret. Such authentication methods are weak, as anyone with the correct password can authenticate to the service. In addition, authentication methods based on personal information require safe storage of this information. All passwords need to be stored as hashes in a database for comparison when a user tries to authenticate. The fact is that not all services have sufficient protection of stored user data. One of the most known data breaches is the attack against LinkedIn in 2014, where approximately 100 million non-hashed passwords were leaked online [Sco16]. The leakage of passwords is the main contributor to data breaches. Verizon's data breach investigation report for 2022 shows that over 80% of all data breaches in the last 15 years was a result of stolen credentials [Ver22]. This demonstrates the weaknesses and risks of using passwords as an authentication method.

Alternative solutions to passwords are heavily researched. Multi-factor Authentication (MFA) is one solution to password leakages. Unfortunately, MFA is not mandatory, and not all services implement it. MFA does not solve the aforementioned shortcomings of passwords but merely adds a layer of security. This means that problems like the reuse of passwords and phishing attacks still pose a threat. Adding

layers of security to existing systems is a short-term solution, as future technology may introduce new attack vectors [NS22].

The title of this master's thesis is "The Application and Use of Cryptographic Zero-knowledge Protocols", and the problem description for this thesis states the following: "The objective of this thesis is to create a zero-knowledge proof that employs cryptography believed to be resistant to quantum attacks ... the zero-knowledge proof will be implemented as a zero-knowledge protocol used by a client, functioning as the prover, to authenticate themselves to a server, functioning as the verifier". Based on this, the chosen research scope for the thesis is authentication systems and the use of zero-knowledge protocols within such systems. An understanding of zero-knowledge protocols and relevant theory was required, which was done during the literature study. As the goal was to construct a quantum-resistant zero-knowledge protocol used for authentication, the authors focused on a quantum-resistant zero-knowledge digital signature scheme used within a self-developed authentication system. The title of this master's thesis is "The Application and Use of Cryptographic Zero-knowledge Protocols", and the problem description for this thesis states the following: "The objective of this thesis is to create a zero-knowledge proof that employs cryptography believed to be resistant to quantum attacks ... the zero-knowledge proof will be implemented as a zero-knowledge protocol used by a client, functioning as the prover, to authenticate themselves to a server, functioning as the verifier". Based on this, the chosen research scope for the thesis is authentication systems and the use of zero-knowledge protocols within such systems. An understanding of zero-knowledge protocols and relevant theory was required, which was done during the literature study. As the goal was to construct a quantum-resistant zero-knowledge protocol used for authentication, the authors focused on a quantum-resistant zero-knowledge digital signature scheme used within a self-developed authentication system. The digital signature scheme gains its quantum-resistant property from utilizing instances of hard problems within lattice cryptography. It was created by converting a quantum-resistant interactive zero-knowledge protocol into a digital signature via the "Fiat-Shamir with Aborts" technique [Lyu09]. The resulting scheme was then implemented in a passwordless authentication system. The architecture of the authentication system is inspired by the Fast Identity Online (FIDO2) standard [All23]. It was created by converting a quantum-resistant interactive zero-knowledge protocol into a digital signature via the "Fiat-Shamir with Aborts" technique [Lyu09]. The resulting scheme was then implemented in a passwordless authentication system. The architecture of the authentication system is inspired by the FIDO2 standard [All23].

1.3 Limitations and Challenges

Certain restrictions had to be put in place to ensure that the work with this thesis could be completed within six months. These limitations helped to establish a clear

scope for the thesis and define what should and should not be included. The following limitations were placed on the thesis:

- Possible side-channel attacks against the zero-knowledge protocol are disregarded.
- A secure communication channel in the form of TLS between the client and RP is assumed. This should provide encryption and integrity checks for messages being sent.
- The authors are only developing a prototype as a Proof of concept (PoC), not production grade software.

Before implementing the proposed authentication system, the authors envisioned some challenges that could have an impact on the results:

- As the zero-knowledge protocol is based on lattice cryptography, it will operate on vectors and matrices of polynomials with relatively high degrees. This results in large key sizes, which could have a negative impact on the time and resource consumption during key generation and storage of key pairs.
- To avoid leakage of the private key when producing signatures, a technique called *rejection sampling* must be used. This could result in a negative impact on time and resource consumption, especially on a mobile device.

1.4 Research Questions

The aforementioned motivation, problem area, and limitations and challenges result in the following main goal for the master's thesis:

How can we use lattice-based cryptography in a zero-knowledge protocol to implement an efficient and quantum-resistant authentication system?

Several sub-goals were defined to better answer the thesis's main goal. Both the main goal and the research questions were based on the project work conducted during the autumn of 2022 [NS22].

RQ1 How can instances of *hard* problems within lattice cryptography be used to construct a zero-knowledge protocol?

RQ2 How can such a protocol enable passwordless authentication?

RQ3 How does the performance of the implemented passwordless authentication system, incorporating the proposed zero-knowledge protocol, compare to similar state-of-the-art solutions?

The following objectives were pursued to help answer the research questions.

OBJ1 Construct a quantum-resistant zero-knowledge protocol.

OBJ2 Create a test environment that implements passwordless authentication.

OBJ3 Implement the constructed quantum-resistant zero-knowledge protocol in said test environment.

OBJ4 Test and validate the implemented solution in terms of performance.

1.5 Contribution

Existing implementations of FIDO2 make use of digital signature algorithms listed in CBOR Object Signing and Encryption (COSE), a set of cryptographic standards defined by the Internet Engineering Task Force (IETF). Except for the hash-based digital signature scheme HSS/LSM, defined in RFC 8708 [Hou20], none of the listed digital signature algorithms are designed to be quantum-resistant. This master's thesis offers the following contribution to the research fields of passwordless authentication and post-quantum cryptographic protocols:

Increased knowledge of the implementation of passwordless authentication through a lattice-based zero-knowledge digital signature.

1.6 Outline

The rest of this thesis is divided into the following five chapters:

Chapter 2: Background and Related Work introduces background knowledge, terminology, and a mathematical preliminary relevant to the proposed solution. Lastly, relevant work within the research field is presented.

Chapter 3: Methodology describes the methodology employed by the authors to create the final solution. This includes research study, design process, and methods for implementing the proposed solution.

Chapter 4: Proposed Solution presents the architecture, design choices, and implementation of the proposed solution. Code listings for important methods are shown and explained in detail. Lastly, the interaction between the different components in the proposed solution is presented.

Chapter 5: Performance and Discussion presents an evaluation of the proposed solution in terms of efficiency. Metrics such as key sizes and time usage for the different algorithms are presented. These results are discussed and compared to the state of the art within lattice-based digital signatures. An evaluation of the proposed solution in light of pre-defined functional and quality requirements is then carried out. Finally, the research questions are revisited and a discussion of how this thesis answers them is presented.

Chapter 6: Conclusion and Future Work summarizes the thesis and presents to which degree the proposed solution answers the problem description and research questions. Any unfinished work on the thesis is presented, and future work is proposed.

Chapter 2
Background and Related Work

This chapter will introduce relevant knowledge needed to understand the research area. As the main research area of this thesis deals with zero-knowledge protocols and their applications, a thorough review of zero-knowledge proofs and their properties, as well as examples of zero-knowledge protocols, is presented. An overview of Post-Quantum Cryptography (PQC) with a focus on instances of hard problems within lattice cryptography will then be given. An interactive lattice-based Σ-protocol with the zero-knowledge property is then presented. The protocol is transformed into a lattice-based zero-knowledge digital signature. The outline of the FIDO2 standard is followed by a discussion of related work within the field of post-quantum digital signatures and current FIDO2 implementations.

2.1 Zero-Knowledge Proofs

A zero-knowledge proof is a method for one party, the prover, to convince another party, the verifier, that a given statement is true without revealing any additional information about the statement. A zero-knowledge proof must satisfy three properties, the completeness property, the soundness property, and the zero-knowledge property.

2.1.1 Properties

Completeness A proof is *complete* if the statement is true and an honest verifier, i.e., one following the protocol, is convinced by a prover that the statement is true.

Soundness A proof is *sound* if no dishonest prover, i.e., a prover trying to cheat the verifier without having the required knowledge, can convince a verifier that a false statement is true, except with some small probability. This small probability makes all zero-knowledge proofs *probabilistic*.

Zero-knowledge Suppose the prover provides a true statement. The proof is *zero-knowledge* if the verifier learns nothing from the interaction with the prover except that the statement is true.

In order to prove soundness, i.e., that proof is a Proof of Knowledge (PoK), a proof demonstrating the existence of a special algorithm referred to as an "extractor" is required [VJ14]. This extractor functions as a verifier who interacts with the prover. If the prover successfully demonstrates knowledge of a secret, the extractor should be able to extract that secret with a probability $1 - \epsilon$ in expected polynomial time[1]. ϵ is called the soundness error. Even though the existence of an extractor contradicts the zero-knowledge property, it is important to note that the extractor is a *special* algorithm, and is not required to exist during a normal run of a zero-knowledge proof. The proof of the existence of an extractor is accomplished by granting it certain privileges in its interaction with the prover.

To prove zero-knowledge, it is required to present a proof that demonstrates the existence of a special algorithm with certain properties, known as a "simulator" [VJ14]. In contrast to an extractor, the simulator functions as a certain type of prover, but unlike a regular prover, the simulator has no knowledge of the secret. Nevertheless, the simulator must be able to convince any verifier that it has knowledge of the secret while producing transcripts that are indistinguishable from genuine zero-knowledge proofs executed with a real prover.

Since a simulator has no knowledge of the secret, it is clear that a verifier would not be able to extract some information about the secret based on their interaction. If a transcript from their interaction is distributed identically as a transcript from a real proof, and a verifier is able to differentiate between a real prover and a simulator, it would imply that the distribution is not identical. The sheer existence of a simulator contradicts the soundness property, but as for the extractor, the simulator is not bound to follow a normal run of the proof, hence being a *special* algorithm. To prove the existence of a simulator, the normal run of a proof is executed in "reverse". Rewinding the proof allows for a simulator to convince a verifier.

Completeness and *soundness* are found in almost every general proof system, but it is the *zero-knowledge* property that makes a proof system zero-knowledge.

2.1.2 Interactive and Non-Interactive Zero-knowledge Proofs

Zero-knowledge proofs can be categorized as either *interactive* or *non-interactive*. Interactive zero-knowledge proofs require online or present interaction between both

[1]For an algorithm to run in expected polynomial time, it requires the existence of a polynomial $p(|s|)$, which ensures that for any given secret s, the upper bound of the algorithm's expected run-time is $p(|s|)$. This expectation is calculated over the algorithm's random choices.

parties. In non-interactive zero-knowledge proofs, the information between the prover and the verifier can be authenticated by the prover itself. This increases performance as multiple verifications can be made offline.

2.1.3 Σ-Protocols

A Σ-protocol is a protocol that allows for a prover to prove knowledge of various statements without revealing any additional information by following these three steps [HLHL10]:

Step 1. Commitment The prover commits to a value while keeping the value itself hidden. A commitment has two properties that are essential in zero-knowledge proofs. The *hiding* property ensures that the verifier learns nothing about the committed value. The *binding* property ensures that the prover will not change its mind after committing.

Step 2. Challenge The verifier responds with a challenge chosen at random *after* receiving the commitment. It is essential that the commitment phase precedes the challenge phase, otherwise, a dishonest prover could cheat.

Step 3. Opening After receiving the challenge, the prover responds with the *opening*, a combination of the secret knowledge, the committed value, and the challenge. The verifier can then use the received commitment and generated challenge to verify the opening.

2.1.4 Fiat-Shamir Transform

Before explaining the Fiat-Shamir transform, a brief overview of the Random Oracle Model (ROM) is given. A random oracle is a theoretical "black box" that responds with a truly random response, chosen uniformly from the set of possible outputs to every unique query [BR93]. The same query produces the same response, making a random oracle *deterministic*. Amos Fiat and Adi Shamir demonstrated in [FS87] how a random oracle could be utilized to eliminate the need for interaction in protocols for the generation of signatures. This gave birth to the Fiat-Shamir transform. The Fiat-Shamir transform is a technique used to transform an interactive proof of knowledge into a non-interactive one. The verifier's random challenges, explained in Section 2.1.3, must be made public throughout the proof for the technique to work [FS87]. The technique replaces the interactive step of generating a random challenge with a random oracle. Often in cryptographic systems, a hash function is modeled as a random oracle, as proof using a hash function cannot be carried out using weaker assumptions on it. A proof relying on a random oracle is thus secure by showing that an adversary requires impossible behavior from the oracle, or can break some cryptographic one-way. If the interactive proof is used in identification schemes, the

resulting non-interactive proof could be used as a digital signature by including the message to be signed in the random oracle.

2.1.5 The Ali Baba Cave

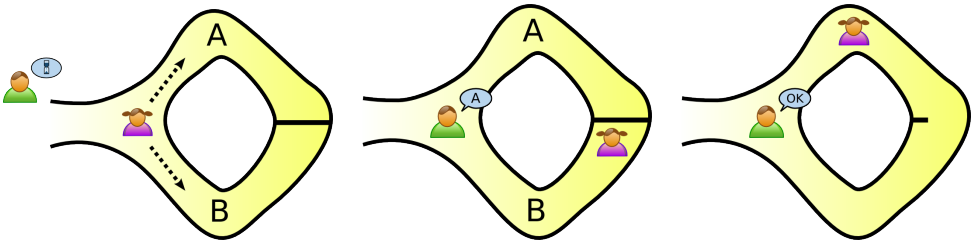
The Ali Baba Cave [QQQ+01] presents the fundamentals of zero-knowledge proofs in an intuitive way.

In the Ali Baba Cave, Peggy, the prover, wants to convince Victor, the verifier, that she knows the secret code to open a gate inside a cave, without revealing the code itself. To do so, Peggy and Victor will undergo a three-step Σ-protocol.

Interactive Zero-knowledge Proof

As illustrated in Figure 2.1, the cave has two entrances labeled "A" and "B". Peggy walks into the cave while Victor waits outside. While inside, Peggy randomly chooses one entrance out of the two, and *commits* to that entrance by walking in, as shown in Figure 2.1a. This commitment is hiding because once Peggy arrives at the gate and Victor walks in, there is no way for Victor to tell which entrance Peggy has committed to. The commitment is binding because once Victor has entered the cave, there is no way for Peggy to change her mind. Victor enters the cave and shouts the name of the entrance he wants Peggy to use to return, chosen at random as the *challenge*. This is shown in Figure 2.1b. Finally, Peggy returns out of the cave through the entrance challenged by Victor, as her *opening*. This is shown in Figure 2.1c. Depending on Peggy's commitment and Victor's challenge, there are two possibilities. The first possibility is that Peggy has to use her secret code to return out of the cave through the challenged entrance. The other possibility is that Peggy can return through the same entrance as she entered, thus not having to use her secret code.

The proof is complete because Peggy convinces Victor by returning through the challenged entrance. One will notice that a dishonest prover that could guess Victor's challenge would be able to convince Victor without knowing the secret code. The protocol could therefore be repeated several times to make it harder to guess the correct challenge for each round. A *soundness error* of $\frac{1}{2^m}$ is achieved by repeating the protocol m times. Finally, the proof is zero-knowledge as Victor learns nothing about Peggy's secret code.



(a) Peggy commits to an entrance.

(b) Victor challenges Peggy.

(c) Peggy returns through the challenged entrance.

Figure 2.1: Interactive zero-knowledge proof illustrated with the Ali Baba Cave.

Non-interactive Zero-knowledge Proof

The Ali Baba Cave can also be used to illustrate the fundamentals of non-interactive zero-knowledge proofs. By applying the Fiat-Shamir transform, a hash function $\mathcal{H}$ modeled as a random oracle is introduced at both sides of the gate inside the cave. If Peggy knows the secret code to the gate inside cave C, she can use $\mathcal{H}$ to output a random entrance $E = \mathcal{H}(C)$ for her to return through. By filming the whole sequence, a transcript of the protocol run can be made, which can be verified offline by Victor. Peggy can also increase the soundness of her proof by repeating the protocol several times and including the round i in the hash function for each round, i.e., $E_i = \mathcal{H}(C||i)$.

2.1.6 Interactive Schnorr Protocol

In practice, when constructing zero-knowledge protocols, cryptographic *one-way functions* are used to achieve the aforementioned properties. One protocol that is zero-knowledge is the interactive proof of knowledge of a discrete logarithm, also called the interactive Schnorr protocol [Dam02]. The protocol is defined over a cyclic group G_q of order q with generator g. The prover wants to prove knowledge of some secret value $s \in \mathbb{Z}_q$. Prover and verifier first agree on a large prime q and a generator g of the multiplicative group $\mathbb{Z}_q$. $t = g^s \bmod q$ is public information. The Schnorr protocol is an instance of Σ-protocols. The prover starts the protocol by generating a random value y and sending $w = g^y \bmod q$ as the *commitment*. Upon receipt, the verifier generates a random challenge $c \in \log_2 q$ and sends it to the prover. One should note that for the Schnorr protocol to be resilient against a malicious verifier, the challenge space should not exceed $\log_2 q$, as explained in [Mao03]. A challenge space of $\mathbb{Z}_q$ will reduce the zero-knowledge property to *honest-verifier* zero-knowledge, i.e., the verifier acts according to the protocol by choosing a challenge c uniformly at random, not adaptively dependent on any input. The reduced zero-knowledge property will not remain if the Fiat-Shamir transform is applied to transform the proof into a non-interactive one.

Upon receipt of the challenge, the prover responds with $z = c \cdot s + y \bmod (q-1)$ as the *opening*. The verifier verifies the proof by checking if $g^z = w \cdot t^c \bmod q$. Figure 2.2 shows each step of the protocol.

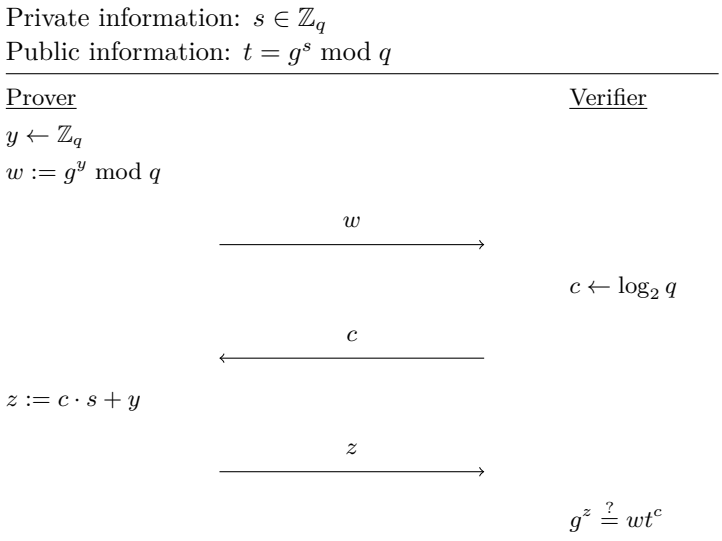


Figure 2.2: Sequence diagram of the interactive Schnorr protocol.

The Schnorr protocol is complete since

$$g^z = g^{cs+y} = g^y \cdot g^{sc} = w \cdot (g^s)^c = w \cdot t^c \quad \square$$

The protocol has a soundness error of $\epsilon = \frac{1}{\log_2 q}$, as a dishonest prover has to guess the correct challenge to prove knowledge without knowing s. If a dishonest prover can predict the challenge value c, they can generate a random opening $z \in \{0, q-2\}$ and send $w' = g^z t^{-c}$ as the commitment. The verifier will then respond with the already predicted challenge c before the prover responds with the pre-generated opening z. The verifier will be convinced by this as $g^z = w' \cdot t^c = (g^z \cdot t^{-c}) \cdot t^c = g^z$. A sequence diagram of the protocol with a dishonest prover can be viewed in Figure 2.5.

In order to prove that the interactive Schnorr protocol is a PoK, the existence of an extractor algorithm must be demonstrated. The extractor is able to retrieve the secret s with probability $1 - \epsilon$ by rewinding the prover's execution of the protocol. The extractor algorithm can be viewed in Figure 2.3. The protocol follows a normal protocol run, but after receiving the opening z, the extractor *rewinds* the protocol

and sends another randomly sampled challenge $c' \in \log_2 q$. After receiving another opening z', the extractor can retrieve the secret s as it has received two distinct openings to the same commitment.

$$\begin{aligned} \frac{g^z}{t^c} &= \frac{g^{z'}}{t^{c'}} \\ \frac{g^z}{g^{z'}} &= \frac{t^c}{t^{c'}} \\ g^{z-z'} &= g^{s(c-c')} \\ \implies z - z' &= s(c - c') \\ s &= \frac{z - z'}{c - c'} \end{aligned}$$

The extractor will not output the secret s if c' is sampled such that $c = c'$, hence a soundness error $\epsilon = \frac{1}{\log_2 q}$ is achieved.

For the Schnorr protocol to be (honest-verifier) zero-knowledge, the existence of a simulator must be demonstrated. The existence of a simulator is only possible with a challenge space of $\mathbb{Z}_q$, i.e., a honest-verifier must be assumed. One should notice that a challenge space of $\mathbb{Z}_q$ results in the simulator algorithm not running in the expected polynomial time. The simulator must be able to produce transcripts of a protocol run with a distribution indistinguishable from a normal protocol run, without having knowledge of the secret s. Such an algorithm is achieved by running the protocol "in reverse", as shown in Figure 2.4. As the opening z is chosen at random, the resulting commitment w is random, thus resulting in a distribution identical to that from a real protocol run.

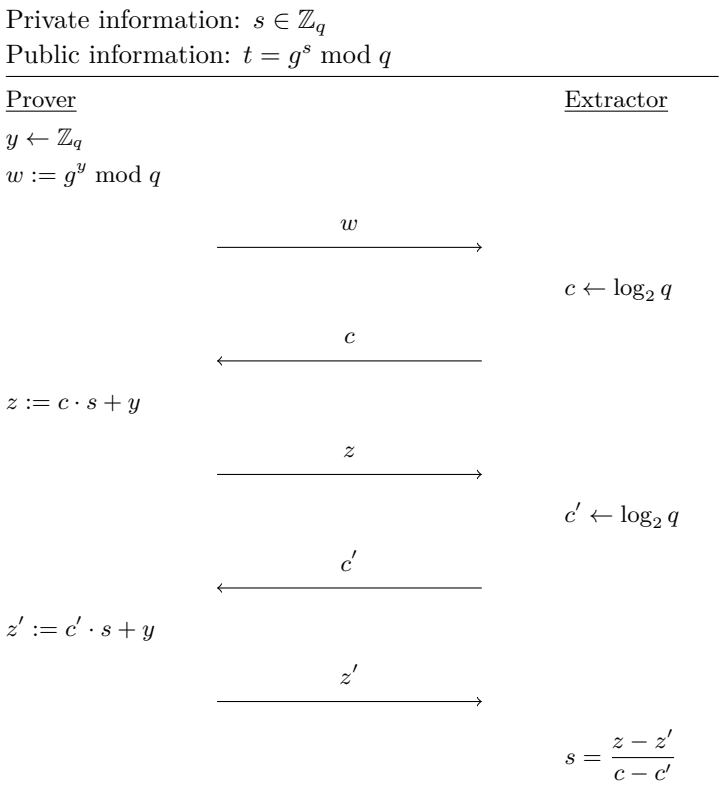


Figure 2.3: Extractor algorithm for interactive Schnorr.

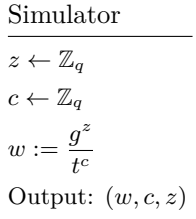


Figure 2.4: Simulator algorithm for interactive Schnorr.

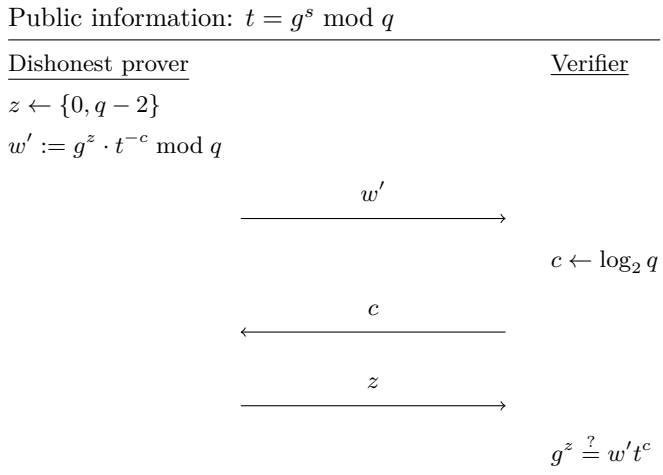


Figure 2.5: Sequence diagram of dishonest-prover interactive Schnorr protocol.

2.1.7 Schnorr Signatures

The interactive Schnorr protocol presented in Section 2.1.6 can be transformed into a non-interactive zero-knowledge proof by utilizing the Fiat-Shamir transform. By applying the Fiat-Shamir transform, the prover instead "simulates" the interactive step of receiving a challenge and replaces it with a random oracle. This is done by introducing a hash function $\mathcal{H}$ modeled as a random oracle. The prover can query the oracle with public parameters and commitment w, and receive a challenge c. This way, the prover can produce proofs that can be verified by any verifier offline. If the protocol is utilized for identification purposes, it can also function as a digital signature if the proof is bound to a specific message m. The signature procedure $\mathsf{Sig}(s, g, q, m, t)$ for the Schnorr digital signature scheme can be viewed in Figure 2.6a. The corresponding verification procedure $\mathsf{Vf}(g, q, t, \sigma, m)$ can be viewed in Figure 2.6b.

$\mathsf{Sig}(s, g, q, m, t)$

$y \leftarrow \mathbb{Z}_q$
$w := g^y \bmod q$
$c \leftarrow \mathcal{H}(g, t, w, m)$
$z := c \cdot s + y \bmod (q-1)$
Output $\sigma = (z, w, c)$

(a) Signature procedure in Schnorr digital signature scheme.

$\mathsf{Vf}(g, q, t, \sigma, m)$

$c \stackrel{?}{=} \mathcal{H}(g, t, w, m)$
$g^z \stackrel{?}{=} wt^c$

(b) Verification procedure in Schnorr digital signature scheme.

Figure 2.6

2.2 Post-Quantum Cryptography

One of the most active fields within cryptography is PQC. PQC is the branch of cryptography investigating cryptographic algorithms designed to withstand attacks from quantum computers. Quantum computers leverage the information encoded in systems that exhibit unique quantum properties [LJL+10]. The basic unit of information in a quantum computer is called qubits. Qubits can either be 0, 1, or a *superposition* of both 0 and 1 simultaneously, representing all possible configurations of the qubit. Another fundamental part of quantum computing is *entanglement*, which is when changes to one qubit directly impact another. By representing information through qubits and leveraging the entanglement of them, complex problems not yet solvable on classical computers can be solved on quantum computers [IBM]. Both superposition and entanglement enable the deployment of quantum-specific algorithms, such as Shor's algorithm [Sho97]. Shor's algorithm has proven to effectively solve mathematical problems fundamental to modern cryptosystems, such as *integer factorization* [Mer06], *discrete logarithm over finite fields*, and *discrete logarithms over elliptic curves* [PZ04], which systems like TLS, SSH, IPSec etc. builds upon. To successfully break these problems, a sufficiently powerful quantum computer must be in place, which is not the case as of now. Nevertheless, PQC should be in place *before* such an event, as the development of new cryptosystems and standards is tedious work. So-called "store now, decrypt later" attacks are also one of many reasons to migrate to PQC.

2.2.1 Lattice Cryptography

Lattice cryptography is one of the most prominent fields within PQC. This is reflected by the ongoing process led by NIST to standardize quantum-resistant public-key algorithms, where three out of four algorithms ready for standardization rely on lattice cryptography.

A lattice is a geometric structure consisting of repeating points in multiple dimensions. Figure 2.7 shows lattice structures in two and three dimensions. In general, higher dimensions are used for increased security. However, increased dimensions also result in increased computational complexity.

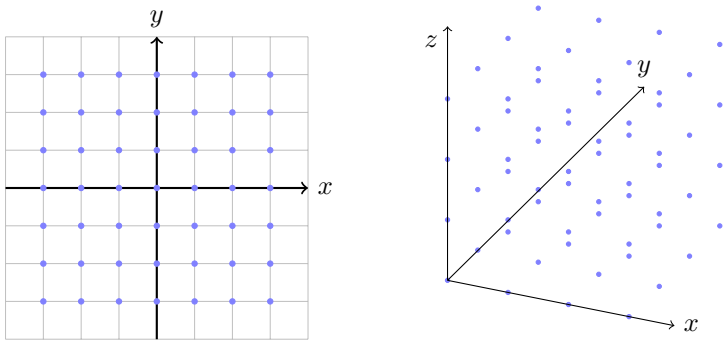


Figure 2.7: Lattice structures in two and three dimensions.

Mathematically speaking, an integer lattice is the set of all possible integer linear combinations of *basis vectors* with coefficients in $\mathbb{Z}$. An n-dimensional integer lattice Λ is a subgroup of the group $(\mathbb{Z}^n,+)$. Lattices are defined by a set of bases, where a basis is defined as $\mathbf{B} \in \mathbb{Z}^{n\times m}$.

Definition 1. An **integer lattice** Λ generated by $\mathbf{B}$ is defined as

$$\Lambda_{\mathbf{B}} = \mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_m) = \{\sum_{i=1}^{m} z_i \mathbf{b}_i \ : \ z_i \in \mathbb{Z}, \mathbf{b}_i \in \mathbf{B}\}$$

Definition 1 defines all possible integer lattices, but some lattices are more interesting than others. Two groups of interesting lattices are the *ideal* lattice and the *q-ary* lattice.

Definition 2. Given the quotient polynomial ring $\mathbb{Z}[x]/(f)$, where (f) is the ideal I created by a monic[2] polynomial $f \in \mathbb{Z}[x]$ of degree d. A n-dimensional **ideal lattice** $\Lambda_I \subset \mathbb{Z}^n$, corresponding to the ideal I, is defined as

$$\Lambda_I = \left\{ (a_0, \ldots, a_{n-1}) \ : \ \sum_{i=0}^{n-1} a_i x_i \in I \right\}$$

[2]A monic polynomial is a non-zero single-variable polynomial whose leading coefficient is 1.

In cryptography, ideal lattices are particularly relevant as the number of parameters required to define them can be reduced by a square root [Mic07], making them more efficient. Examples of ideal lattices are $\mathbb{Z}^n$ and any lattice corresponding to the ideal $\mathbb{R}[x]/\langle X^d + 1\rangle$. The lattices corresponding to the ideal $\mathbb{Z}_q/\langle X^d + 1\rangle$ where q is prime are especially well suited for use in cryptography. The ideal $\mathbb{Z}_q/\langle X^d + 1\rangle$ will therefore be denoted by $\mathcal{R}_{q,f}$ for the remainder of this thesis.

The last group of interesting lattices is the q-ary lattice, which is a lattice of all vectors $\mathbf{z} \in \mathbb{Z}^m$ where $\mathbf{Az} \equiv \mathbf{0}$. They are of particular interest as they can be represented by a uniformly random matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$.

Definition 3. Given a uniformly random matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$, a **q-ary lattice** is defined as

$$\Lambda_q^{\perp}(\mathbf{A}) = \{\mathbf{z} \in \mathbb{Z}^m \ : \ \mathbf{Az} \equiv \mathbf{0}\}$$

Lattice Problems

Cryptographic one-way functions relying on the hardness of a set of lattice problems are conjectured to be quantum-resistant [KP20; Pei+16]. The following presents such lattice problems, but first, some notation is required.

Equation 2.1 shows the sampling of vector values for a vector $\mathbf{v}$ of length m, where the values are drawn from the set $[\beta]$. The set $[\beta]$ is defined for any positive integer β and denotes the set $[\beta] = \{-\beta, \ldots, -1, 0, 1, \ldots, \beta\}$.

$$y \leftarrow [\beta]^m \tag{2.1}$$

A way to check if all elements of a vector $\mathbf{v}$ lie in the interval denoted by $[\beta]$ is to take the L-infinity norm of $\mathbf{v}$. The L-infinity norm is the element with the largest absolute value in a vector. One can therefore indicate that a vector $\mathbf{v}$ is in $[\beta]$ by writing $||\mathbf{v}||_\infty \leq \beta$. The L-infinity norm may also be used to check whether a vector $\mathbf{v}$ is *short*. To do so, one must define a maximum limit for what is considered short; let's call this limit L. By checking if $||\mathbf{v}||_\infty \leq L$, one has determined whether or not $\mathbf{v}$ is short. The shortest vector in a lattice Λ is denoted by $\lambda(\Lambda)$.

Shortest Vector Problem

Definition 4. Given the vector space V defined by a lattice Λ, the **Shortest Vector Problem (SVP)** is to output a short non-zero vector $\mathbf{v} \in V$ in Λ such that $||\mathbf{v}|| = \lambda(\Lambda)$.

One should note SVP asks for *a* short vector, not *the* short. This implies that several vectors $\mathbf{v}_i \in V$ fulfill the equation in Definition 4.

To guarantee a larger set of possible short vectors, one can use the γ-approximate SVP, denoted by SVP_γ. In SVP_γ, one is asked to output a vector $\mathbf{v} \in V$ where $||\mathbf{v}|| \leq \gamma \cdot \lambda(\Lambda)$, where $\gamma \geq 1$.

Closest Vector Problem

Definition 5. Given the vector space V defined by a lattice Λ as well as a vector $\mathbf{v}$ in V, but not necessarily in Λ, the **Closest Vector Problem (CVP)** is to output the vector closest to $\mathbf{v}$ in Λ.

The γ-approximate CVP, denoted by CVP_γ, asks for a vector $\mathbf{x}$ in Λ where $||\mathbf{x} - \mathbf{v}|| \leq \gamma$. As CVP is a generalization of SVP, it follows that the hardness of SVP implies the hardness of CVP, as shown in [GMSS99].

The two most prominent lattice problems used in lattice-based cryptographic schemes are the *Short Integer Solution (SIS) problem* and the *Learning With Errors (LWE) problem*.

Short Integer Solution Problem The SIS problem was introduced in [Ajt96] and is a hard problem within the field of lattice cryptography that involves finding a short, non-zero vector in a defined lattice.

Definition 6. Given a matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ consisting of m uniformly random column-vectors $\mathbf{a}_i$. The **SIS problem** is to find a vector $\mathbf{z} \in \Lambda_q^\perp(\mathbf{A})$ such that

$$||z|| \leq \beta \text{ and}$$
$$\mathbf{Az} \equiv \mathbf{0}$$

Finding a vector $\mathbf{z}$ that satisfies $\mathbf{Az} \equiv \mathbf{0}$ is trivial through *Gaussian elimination*. The constraint on the norm of $\mathbf{z}$ makes the SIS problem hard.

In the breakthrough paper of Miklós Ajtai [Ajt96], a family of one-way functions based on SIS was presented, the most prominent one being *Ajtai's one-way function*, which relies on the hardness of SIS.

Definition 7. Given a matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ and a vector $\mathbf{x} \in \{0,1\}^m$, **Ajtai's one-way function** is defined as

$$f_\mathbf{A}(\mathbf{x}) = \mathbf{Ax} \mod q$$

Ajtai also showed in [Ajt96] that the SIS problem is secure in an average-case scenario if SVP_γ is hard in a worst-case scenario.

Figure 2.8 shows a q-ary lattice formed by the basis $\mathbf{B} = (\mathbf{b}_1, \mathbf{b}_2)$. As with the general SIS problem, the hardness lies in finding a short, non-zero vector $\mathbf{z}$, which satisfies the equation $\mathbf{Az} \equiv \mathbf{0}$. All lattice points satisfy the equation, but only the points within the blue circle are considered to be short.

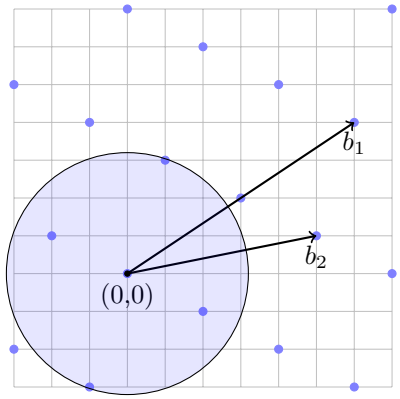


Figure 2.8: Lattice defined by basis $\mathbf{b_1}$ and $\mathbf{b_2}$. The short solution lies within the blue circle, defined by β.

Learning with Errors Problem The LWE problem was first introduced by Oded Regev in 2005 [AR05], and is a way to hide a secret vector by adding noise to it. There are two main versions of the LWE problem, *search* and *decision* LWE. These two versions were shown to be equivalent in [Reg09].

Definition 8. Let $\mathcal{X}$ be a probability distribution over the additive group on reals modulo one, denoted by $\mathbb{T} = \mathbb{R}/\mathbb{Z}$, and let $\mathbf{s} \in \mathbb{Z}_q^n$ be a fixed vector. The **search LWE problem** is to output the correct fixed vector $\mathbf{s}$ given access to polynomially many samples of $\mathbf{A}_{\mathbf{s},\mathcal{X}}$, where a sample is a pair $(\mathbf{a}, t)$ such that

$$\begin{aligned} &\mathbf{a} \xleftarrow{\$} \mathbb{Z}_q^n \text{ and} \\ &t = \langle \mathbf{a}, \mathbf{s} \rangle + e \mod q, \text{ where} \\ &e \xleftarrow{\mathcal{X}} \mathbb{T} \end{aligned}$$

Definition 9. The **decision LWE problem** asks to distinguish between samples from $\mathbf{A}_{\mathbf{s},\mathcal{X}}$, defined in Definition 8, and uniformly random samples from the vector space $\mathbb{Z}_q^n \times \mathbb{T}$.

2.2.2 Reduction of Lattice Problems

All lattice problems presented in this thesis can be reduced to the CVP in polynomial time [TJB12]. Problem A can be reduced to problem B if any of the methods used to solve problem B can be used to solve problem A. This means problem B is at

least as hard as problem A. This reduction between lattice problems is illustrated in Figure 2.9, where an arrow from problem B to problem A reflects the reduction from problem B to A.

CVP = CVP_1 is known to be NP-hard [TJB12], and as seen in Figure 2.9 all problems can be reduced to CVP. This implies that solving LWE and SIS is as hard as solving CVP.

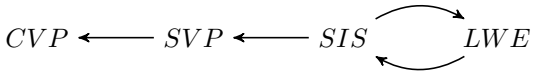


Figure 2.9: Reduction of lattice problems. Inspired by [TJB12]

2.2.3 Interactive Lattice-Based Zero-knowledge Protocol

The following is a presentation of how a lattice-based digital signature can be constructed by the use of *module* LWE and *module* SIS. Module refers to vectors of polynomials. This is preferred as polynomials are able to represent a large amount of information more concisely. For example, one polynomial of degree $n-1$ is able to represent a list of n integers, as shown in Equation 2.2.

$$c_{n-1}x^{n-1} + \dots c_2x^2 + c_1x + c_0 \mapsto \{c_{n-1}, \dots, c_2, c_1, c_0\} \tag{2.2}$$

The use of polynomials results in more efficient protocols. The soundness of protocols working with polynomials leverages the fact that two distinct polynomials disagree on almost all possible points in a field $\mathbb{F}$ [LEMT22].

The following protocols will therefore work with polynomials in the ring $(\mathcal{R}_{q,f}, +, \times)$, where $\mathcal{R}_{q,f} = \mathbb{Z}_q/\langle X^d + 1\rangle$. $f \in \mathbb{Z}[X]$ is a monic irreducible[3] polynomial of degree d. Sampling a random vector $\mathbf{v} \leftarrow [\beta]^m$ where $\mathbf{v} \in \mathcal{R}_{q,f}^m$ means to sample a vector of m polynomials of degree d with coefficients in $[\beta]$.

The following interactive zero-knowledge protocol and digital signature scheme are from Vadim Lyubashevsky's survey [Lyu20]. First, a Σ-protocol that is honest-verifier zero-knowledge, similar to the interactive Schnorr protocol from Section 2.1.6, will be constructed. The Fiat-Shamir transform will then be applied to transform the interactive lattice-based zero-knowledge protocol into a lattice-based digital signature scheme.

Equation 2.3 is the one-way function to be used in the lattice-based zero-knowledge protocol. The matrix $\mathbf{A} \in \mathcal{R}_{q,f}^{n\times m}$ defines the q-ary lattice $\Lambda_q^{\perp}$, while $\mathbf{s_1} \in [\beta]^m, \mathbf{s_2} \in$

[3]An irreducible polynomial is a polynomial that cannot be factorized into the product of two non-constant polynomials.

$[\beta]^n$ are vectors of short norm polynomials[4] in $\Lambda_q^{\perp}$. The public key is $(\mathbf{A}, \mathbf{t})$, while the private key is $(\mathbf{s_1}, \mathbf{s_2})$. Recovering the private key given the public key is as hard as solving the Module LWE problem.

$$\mathbf{As_1} + \mathbf{s_2} = \mathbf{t} \in \mathcal{R}_{q,f}^n \tag{2.3}$$

The goal of the lattice-based zero-knowledge protocol is to prove knowledge of two vectors $(\mathbf{s_1}, \mathbf{s_2})$ of short norm polynomials, but this turns out to be considerably less efficient than for Schnorr and the discrete logarithm setting. The reason for this is that not only does one need to prove knowledge of $(\mathbf{s_1}, \mathbf{s_2})$ satisfying Equation 2.3, but also that the coefficients of $(\mathbf{s_1}, \mathbf{s_2})$ lies in a certain range. This range would ideally be $[\beta]$, but a slightly larger $[\bar{\beta}]$ is also fine [Lyu20].

The prover and verifier first agree on a dimension (n, m), as well as a degree d and a large prime q which together define the polynomial ring $\mathcal{R}_{q,f}$. They also agree on the integer values $\bar{\beta}$ and γ. Finally, they agree on a uniformly random matrix $\mathbf{A} \in \mathcal{R}_{q,f}^{n \times m}$. $\mathbf{t} = \mathbf{As_1} + \mathbf{s_2} \in \mathcal{R}_{q,f}^n$ is public information.

The protocol starts with the prover sampling two vectors of polynomials $\mathbf{y_1} \in [\gamma + \bar{\beta}]^m, \mathbf{y_2} \in [\gamma + \bar{\beta}]^n$ uniformly, before sending $\omega = \mathcal{H}(\mathbf{Ay_1} + \mathbf{y_2})$ to the verifier as the commitment. The verifier responds with a challenge $c \leftarrow \mathcal{C} \subset \mathcal{R}_{q,f}$ before prover responds with $\mathbf{z_1} = c \cdot \mathbf{s_1} + \mathbf{y_1}, \mathbf{z_2} = c \cdot \mathbf{s_2} + \mathbf{y_2}$ as the opening. The verifier can then verify by checking if $\mathcal{H}(\mathbf{Az_1} + \mathbf{z_2} - c \cdot \mathbf{t}) = \omega$ and that $||\mathbf{z_1}||_\infty \leq \bar{\beta}$ and $||\mathbf{z_2}||_\infty \leq \bar{\beta}$. A hash of $\mathbf{Ay_1} + \mathbf{y_2}$ is sent as the commitment instead of the actual vector to keep the size of the output small.

One difference between the interactive lattice-based zero-knowledge protocol and the interactive Schnorr protocol explained in Section 2.1.6 is that $(\mathbf{z_1}, \mathbf{z_2})$ need to consist of short norm polynomials. Without this constraint, forging valid $(\mathbf{z_1}, \mathbf{z_2})$ would be trivial through Gaussian elimination. To countermeasure this, both the polynomials in $(\mathbf{y_1}, \mathbf{y_2})$ and the challenge polynomial c must be of short norm.

Since a short-norm challenge polynomial c is needed to output $(\mathbf{z_1}, \mathbf{z_2})$ consisting of short-norm polynomials, the challenge space $\mathcal{C}$ needs to consist of only short-norm polynomials. According to [Lyu20], the challenge space is constructed such that exactly η coefficients are from the set $\{-1, 1\}$, and the rest are 0. The challenge space $\mathcal{C}$ is thus defined as

$$\mathcal{C} = \{c \in [1] : ||c||_1 = \eta\} \tag{2.4}$$

[4] In a short norm polynomial, all coefficients are short, i.e., $[c_0, c_1, \ldots, c_{n-1}] \in [\beta]^n$.

where $||c||_1$ is the L1-norm[5], or *Manhattan Distance*, of c. η is defined as the smallest integer such that $2^{\eta}\binom{d}{\eta} > 2^{256}$. This way, the challenge space is larger than the output domain of e.g. `SHA-256`.

Since the polynomials in $(\mathbf{y_1}, \mathbf{y_2})$ and the challenge polynomial c are of short norm, the resulting $(\mathbf{z_1}, \mathbf{z_2})$ would "leak" information about the secret $(\mathbf{s_1}, \mathbf{s_2})$, as the distribution of $(\mathbf{z_1}, \mathbf{z_2})$ is not uniform. To prevent this, a technique called *rejection sampling* is used to ensure that the distribution of $(\mathbf{z_1}, \mathbf{z_2})$ are independent of $(\mathbf{s_1}, \mathbf{s_2})$.

Rejection Sampling is a technique used to generate samples within a target probability distribution, by drawing samples from another known probability distribution. Generated values that fall outside the target probability distribution are rejected, while those within are kept. The technique ensures the distribution contains randomness, while still being true to the given probability distribution. A practical example would be to generate samples within the normal distribution $\mathcal{N}(\sigma = 1, \mu = 0)$ by drawing samples from the uniform distribution $\mathcal{U}_{[a=-5,b=5]}$. An illustration of how this would look like can be viewed in Figure 2.10.

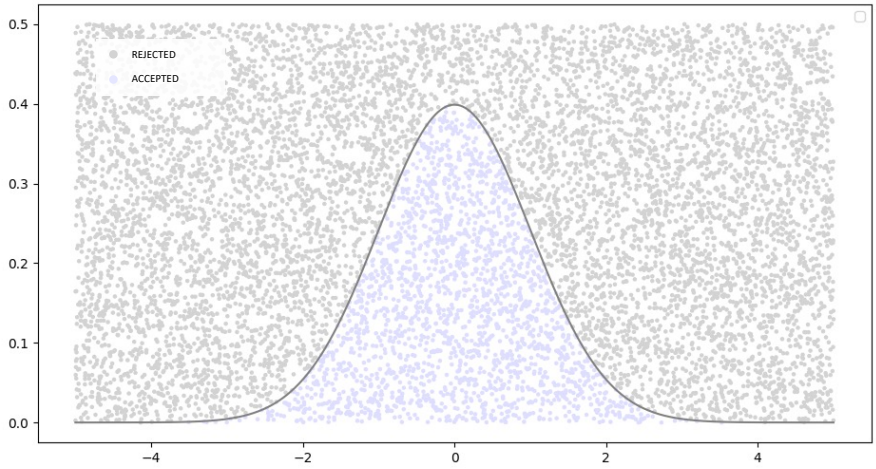


Figure 2.10: The figure shows rejection sampling when generating random values following a normal distribution. The majority of sampled values fall outside the target distribution and are thus rejected.

Rejection sampling in light of the interactive lattice-based zero-knowledge protocol will be done a little differently as the goal is not to draw samples from a given

[5]The L1-norm of a vector v is the sum of the magnitudes of a vector in a space.

probability distribution, but to remove any dependency between the distribution of the secret $(\mathbf{s_1}, \mathbf{s_2})$ and the opening $(\mathbf{z_1}, \mathbf{z_2})$. The goal is to sample $(\mathbf{y_1}, \mathbf{y_2})$ from a distribution, and together with c, compute an opening $(\mathbf{z_1}, \mathbf{z_2})$ which has an output distribution which is indistinguishable from the output distribution of $(\mathbf{y_1}, \mathbf{y_2})$. A simple and effective way to do this is to restrain the L-infinity norm of the polynomials that comprise $\mathbf{z_1}$ and $\mathbf{z_2}$ to $[\bar{\beta}]$. This way, the output distribution of e.g. $\mathbf{z_1}$ is independent of the output distribution of $\mathbf{s_1}$, due to $\mathbf{y_1}$ acting as a masking vector [Lyu20].

The resulting interactive lattice-based zero-knowledge protocol after applying rejection sampling can be viewed in Figure 2.14. Private information is $\mathbf{s_1} \in [\beta]^m, \mathbf{s_2} \in [\beta]^n$, while public information is $\mathbf{A} \in \mathcal{R}_{q,f}^{n \times m}, \mathbf{t} = \mathbf{A}\mathbf{s_1} + \mathbf{s_2} \in \mathcal{R}_{q,f}^{n}$. Unlike the interactive Schnorr protocol, rejection sampling is added. If the rejection sampling algorithm rejects $\mathbf{z_1}$ or $\mathbf{z_2}$, the protocol is terminated and has to restart, thus decreasing the overall performance of the protocol. It is important to mention that the probability for rejection and consequently restarting the protocol is independent of the secret $(\mathbf{s_1}, \mathbf{s_2})$. This is important because any dependency between run-time and the secret would lead to the possibility of side-channel attacks where information about the secret could be deduced by observing the run-time of the protocol [Lyu20].

The protocol is complete since

$$\begin{aligned}
\mathcal{H}(\mathbf{A}\mathbf{z_1} + \mathbf{z_2} - c \cdot \mathbf{t}) &= \mathcal{H}(\mathbf{A}(c\mathbf{s_1} + \mathbf{y_1}) + (c\mathbf{s_2} + \mathbf{y_2}) - c(\mathbf{A}\mathbf{s_1} + \mathbf{s_2})) \\
&= \mathcal{H}(\mathbf{A}c\mathbf{s_1} + \mathbf{A}\mathbf{y_1} + c\mathbf{s_2} + \mathbf{y_2} - \mathbf{A}c\mathbf{s_1} - c\mathbf{s_2}) \\
&= \mathcal{H}(\mathbf{A}c\mathbf{s_1} - \mathbf{A}c\mathbf{s_1} + c\mathbf{s_2} - c\mathbf{s_2} + \mathbf{A}\mathbf{y_1} + \mathbf{y_2}) \\
&= \mathcal{H}(\mathbf{A}\mathbf{y_1} + \mathbf{y_2}) \\
&= \omega \quad \square
\end{aligned}$$

The protocol has a soundness error of $\epsilon = \frac{1}{|\mathcal{C}|}$, where $|\mathcal{C}|$ is the cardinality[6] of the challenge space $\mathcal{C}$. A dishonest prover who is able to guess the challenge c will be able to prove knowledge of the secret $(\mathbf{s_1}, \mathbf{s_2})$ without having knowledge of it. Figure 2.11 shows how a dishonest prover who is able to guess challenge c would convince the verifier. The prover would here sample random $\mathbf{z_1} \leftarrow [\bar{\beta}]^m$ and $\mathbf{z_2} \leftarrow [\bar{\beta}]^n$ and send $\omega = \mathcal{H}(\mathbf{A}\mathbf{z_1} + \mathbf{z_2} - c\mathbf{t})$ as commitment. The verifier will respond with challenge c and prover responds with $(\mathbf{z_1}, \mathbf{z_2})$. The verifier is convinced as $||\mathbf{z_1}||_\infty \leq \bar{\beta}, ||\mathbf{z_2}||_\infty \leq \bar{\beta}$ and $\mathcal{H}(\mathbf{A}\mathbf{z_1} + \mathbf{z_2} - c \cdot \mathbf{t}) = \omega$.

In order to prove that the protocol is a PoK, the existence of an extractor algorithm must be demonstrated. The extractor must be able to retrieve the secret

[6]The cardinality of a set is the number of elements in that set.

$(\mathbf{s_1}, \mathbf{s_2})$ with probability $1 - \epsilon$. It does so in a similar manner as for the interactive Schnorr protocol, presented in Section 2.1.6, by rewinding the protocol run. Figure 2.12 shows a sequence diagram of the extractor. If the extractor is able to receive two different openings $(\mathbf{z_1}, \mathbf{z_2})$ and $(\mathbf{z'_1}, \mathbf{z'_2})$ with two different challenges c, c' from the same commitment ω, the following equation shows that the secret $(\mathbf{s_1}, \mathbf{s_2})$ can be retrieved, unless one can find a collision in $\mathcal{H}$ [Lyu20].

$$
\begin{aligned}
&\mathbf{A}\mathbf{z_1} + \mathbf{z_2} - c\mathbf{t} = \mathbf{A}\mathbf{z'_1} + \mathbf{z'_2} - c'\mathbf{t} \\
&\mathbf{A}(\mathbf{z_1} - \mathbf{z_2}) + (\mathbf{z'_1} - \mathbf{z'_2}) = (c - c')\mathbf{t} \\
&\frac{\mathbf{A}(\mathbf{z_1} - \mathbf{z_2}) + (\mathbf{z'_1} - \mathbf{z'_2})}{(c - c')} = \mathbf{t} \\
&\mathbf{A}\frac{\mathbf{z_1} - \mathbf{z_2})}{c - c'} + \frac{\mathbf{z'_1} - \mathbf{z'_2}}{c - c'} = \mathbf{t} \\
\implies &\begin{cases} \mathbf{s_1} = \frac{\mathbf{z_1} - \mathbf{z_2}}{c - c'} \\ \mathbf{s_2} = \frac{\mathbf{z'_1} - \mathbf{z'_2}}{c - c'} \end{cases}
\end{aligned}
$$

The above equation holds if $c = c'$, hence resulting in the protocol achieving $\epsilon = \frac{1}{|C|}$ as soundness error.

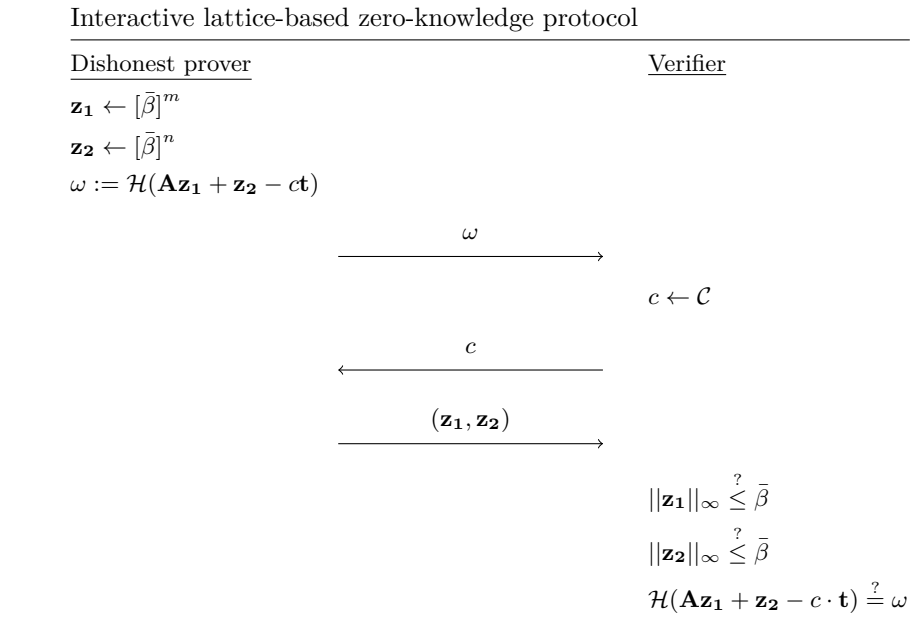


Figure 2.11: Sequence diagram of dishonest-prover lattice-based zero-knowledge protocol.

Interactive lattice-based zero-knowledge protocol

Prover		Extractor
$\mathbf{y_1} \leftarrow [\gamma + \bar{\beta}]^m$		
$\mathbf{y_2} \leftarrow [\gamma + \bar{\beta}]^n$		
$\omega := \mathcal{H}(\mathbf{A}\mathbf{y_1} + \mathbf{y_2})$		
	$\xrightarrow{\omega}$	
		$c \leftarrow \mathcal{C}$
	$\xleftarrow{c}$	
$\mathbf{z_1} := c\mathbf{s_1} + \mathbf{y_1}$		
$\mathbf{z_2} := c\mathbf{s_2} + \mathbf{y_2}$		
$\texttt{if } \mathbf{z_1} \notin [\bar{\beta}]^m \texttt{or } \mathbf{z_2} \notin [\bar{\beta}]^n :$		
$(\mathbf{z_1}, \mathbf{z_2}) := \perp$		
	$\xrightarrow{(\mathbf{z_1}, \mathbf{z_2})}$	
		$c' \leftarrow \mathcal{C}$
	$\xleftarrow{c'}$	
$\mathbf{z'_1} := c'\mathbf{s_1} + \mathbf{y_1}$		
$\mathbf{z'_2} := c'\mathbf{s_2} + \mathbf{y_2}$		
$\texttt{if } \mathbf{z'_1} \notin [\bar{\beta}]^m \texttt{or } \mathbf{z'_2} \notin [\bar{\beta}]^n :$		
$(\mathbf{z'_1}, \mathbf{z'_2}) := \perp$		
	$\xrightarrow{(\mathbf{z'_1}, \mathbf{z'_2})}$	
		$\mathbf{s_1} = \frac{\mathbf{z_1} - \mathbf{z'_1}}{c - c'}$
		$\mathbf{s_2} = \frac{\mathbf{z_2} - \mathbf{z'_2}}{c - c'}$

Figure 2.12: Extractor algorithm for the interactive lattice-based zero-knowledge protocol.

In order to prove that the protocol is (honest-verifier) zero-knowledge, a proof showing that the probability of $\perp$ is the same for all $(\mathbf{s_1}, \mathbf{s_2})$ where $||\mathbf{s_i}||_\infty \leq \beta$ and that the distribution of $(\mathbf{z_1}, \mathbf{z_2})$ are independent of $(\mathbf{s_1}, \mathbf{s_2})$ must be provided. Lemma 4 in [Lyu20] provides both, where the following probability for not rejecting $(\mathbf{z_1}, \mathbf{z_2})$ is also given:

$$\Pr_{\mathbf{y_1},\mathbf{y_2}}[(\mathbf{z_1},\mathbf{z_2}) \neq \perp] = \left(\frac{2\bar{\beta}+1}{2(\bar{\beta}+\gamma)+1}\right)^{d(m+n)} \tag{2.5}$$

The protocol is (honest-verifier) zero-knowledge since a simulator is able to produce transcripts with output distribution indistinguishable from real transcripts. Equation 2.5 must be used for a simulator to produce a valid transcript. A random oracle query on $\mathbf{w}$, $\mathcal{H}(\mathbf{w})$, is simulated by checking whether or not a value for $\mathbf{w}$ is assigned, and if not, samples a random value $\omega \leftarrow \{0,1\}^{256}$ and sets $\mathcal{H}(\mathbf{w}) = \omega$ as the output from the random oracle. A valid transcript is simulated by the simulator outputting $(\omega \leftarrow \{0,1\}^{256}, c \leftarrow \mathcal{C}, \perp)$ with probability 1 - $\Pr_{\mathbf{y_1},\mathbf{y_2}}[(\mathbf{z_1},\mathbf{z_2}) \neq \perp]$. Otherwise, with probability $\Pr_{\mathbf{y_1},\mathbf{y_2}}[(\mathbf{z_1},\mathbf{z_2}) \neq \perp]$, the simulator does the same as the simulator in Figure 2.4 and runs the protocol in "reverse". The simulator samples $\mathbf{z_1} \leftarrow [\bar{\beta}]^m, \mathbf{z_2} \leftarrow [\bar{\beta}]^n, c \leftarrow \mathcal{C}, \omega \leftarrow \{0,1\}^{256}$, programs $\omega = \mathcal{H}(\mathbf{A}\mathbf{z_1} + \mathbf{z_2} - c\mathbf{t})$ and outputs $(\omega, c, \mathbf{z_1}, \mathbf{z_2})$. Figure 2.13 shows the steps performed by the simulator.

Simulator

With prob. $1 - \Pr_{\mathbf{y_1},\mathbf{y_2}}[(\mathbf{z_1},\mathbf{z_2}) \neq \perp]$:

- $\omega \leftarrow \{0,1\}^{256}$
- $c \leftarrow \mathcal{C}$
- Output: $(\omega, c, \perp)$

With prob. $\Pr_{\mathbf{y_1},\mathbf{y_2}}[(\mathbf{z_1},\mathbf{z_2}) \neq \perp]$:

- $\mathbf{z_1} \leftarrow [\bar{\beta}]^m$
- $\mathbf{z_2} \leftarrow [\bar{\beta}]^n$
- $c \leftarrow \mathcal{C}$
- $\omega := \mathcal{H}(\mathbf{A}\mathbf{z_1} + \mathbf{z_2} - c\mathbf{t})$
- Output: $(\omega, c, \mathbf{z_1}, \mathbf{z_2})$

Figure 2.13: Simulator in the interactive lattice-based zero-knowledge protocol.

Interactive lattice-based zero-knowledge protocol

Prover		Verifier
$\mathbf{y_1} \leftarrow [\gamma + \bar{\beta}]^m$		
$\mathbf{y_2} \leftarrow [\gamma + \bar{\beta}]^n$		
$\omega := \mathcal{H}(\mathbf{A}\mathbf{y_1} + \mathbf{y_2})$		
	$\omega \longrightarrow$	
		$c \leftarrow \mathcal{C}$
	$\longleftarrow c$	
$\mathbf{z_1} := c\mathbf{s_1} + \mathbf{y_1}$		
$\mathbf{z_2} := c\mathbf{s_2} + \mathbf{y_2}$		
$\texttt{if } \mathbf{z_1} \notin [\bar{\beta}]^m \texttt{ or } \mathbf{z_2} \notin [\bar{\beta}]^n :$		
$(\mathbf{z_1}, \mathbf{z_2}) := \perp$		
	$(\mathbf{z_1}, \mathbf{z_2}) \longrightarrow$	
		$\|\|\mathbf{z_1}\|\|_\infty \overset{?}{\leq} \bar{\beta}$
		$\|\|\mathbf{z_2}\|\|_\infty \overset{?}{\leq} \bar{\beta}$
		$\mathcal{H}(\mathbf{A}\mathbf{z_1} + \mathbf{z_2} - c \cdot \mathbf{t}) \overset{?}{=} \omega$

Figure 2.14: Sequence diagram of the lattice-based (honest-verifier) zero-knowledge protocol.

2.2.4 Lattice-Based Digital Signature

The following presents a lattice-based digital signature scheme. The combination of rejection sampling and the Fiat-Shamir transform forms a paradigm called "Fiat-Shamir with Aborts", which is used to "transform an interactive identification protocol that has a non-negligible probability of aborting into a signature by repeating executions until a loop iteration does not trigger an abort" [DFPS23]. The paradigm was first introduced by Vadim Lyubashevsky in [Lyu09]. The variant of "Fiat-Shamir with Aborts" used to transform the interactive lattice-based zero-knowledge protocol presented in Section 2.2.3 into a digital signature scheme is simplified in comparison to the version explained in [Lyu09].

The step where the prover sends a commitment ω to the verifier and receives a challenge c is replaced by a *quantum random oracle*. In the case of classical computers, i.e., ROM, an adversary can make queries to the random oracle on any combination of inputs. However, in the case of quantum computers and Quantum

Random Oracle Model (QROM), an adversary can also make quantum queries on the oracle, i.e., querying the oracle on a superposition of the inputs, as shown in [BDF+11]. Traditionally, random oracles are modeled with the use of cryptographic hash functions, but the security proofs for ROM do not automatically apply to QROM. However, the recent works of [DFG13; Unr17; DFMS19; LZ19] show that as long as hash functions with the recommended quantum security are used, no successful attacks on the ROM exist when the adversary is given the added quantum capability, i.e., querying on a superposition of the inputs. The recommended hash functions for 128-bit quantum security are 256-bit outputs for 2^{nd}-preimage resistance and 384-bit outputs for collision resistance [Lyu20].

The quantum random oracle used in the digital signature scheme is modeled by a hash function $\mathcal{H}$, which outputs a polynomial $c \in \mathcal{C}$. The protocol acts as a digital signature if the proof is bound to a specific message μ. Figure 2.15a and 2.15b present the signature and verification procedure respectively for the lattice-based digital signature. While the interactive protocol restarts if $(\mathbf{z_1}, \mathbf{z_2}) = \perp$, the signing procedure in the digital signature scheme can run in a loop until the rejection sampling algorithm "accepts" the opening $(\mathbf{z_1}, \mathbf{z_2})$ and outputs the signature σ.

$\underline{\mathsf{Sig}(q, n, m, \gamma, \bar{\beta}, f, \mathbf{s_1}, \mathbf{s_2}, \mathbf{A}, \mathbf{t}, \mu)}$

while `true` :

$\mathbf{y_1} \leftarrow [\gamma + \bar{\beta}]^m$

$\mathbf{y_2} \leftarrow [\gamma + \bar{\beta}]^n$

$\omega := \mathcal{H}(\mathbf{A}\mathbf{y_1} + \mathbf{y_2})$

$c \leftarrow \mathcal{H}(\mathbf{A}, \mathbf{t}, \omega, \mu)$

$\mathbf{z_1} := c\mathbf{s_1} + \mathbf{y_1}$

$\mathbf{z_2} := c\mathbf{s_2} + \mathbf{y_2}$

`if` $\mathbf{z_1} \in [\bar{\beta}]^m$ `and` $\mathbf{z_2} \in [\bar{\beta}]^n$:

Output $\sigma = (\mathbf{z_1}, \mathbf{z_2}, \omega, c)$

(a) Signature procedure in the lattice-based digital signature scheme.

$\underline{\mathsf{Vf}(q, n, m, \bar{\beta}, f, \mathbf{A}, \mathbf{t}, \sigma, \mu)}$

$c \stackrel{?}{=} \mathcal{H}(\mathbf{A}, \mathbf{t}, \omega, \mu)$

$||\mathbf{z_1}||_\infty \stackrel{?}{\leq} \bar{\beta}$

$||\mathbf{z_2}||_\infty \stackrel{?}{\leq} \bar{\beta}$

$\mathcal{H}(\mathbf{A}\mathbf{z_1} + \mathbf{z_2} - c \cdot \mathbf{t}) \stackrel{?}{=} \omega$

(b) Verification procedure in the lattice-based digital signature scheme.

Figure 2.15

The signature procedure in Figure 2.15a resembles "Bernoulli trials", i.e., repeated independent trials with exactly two possible outcomes. The probability of both $\mathbf{z_1} \in [\bar{\beta}]^m$ and $\mathbf{z_2} \in [\bar{\beta}]^n$ can be written as $\mathrm{P}(\mathbf{z_1} \in [\bar{\beta}]^m \cap \mathbf{z_2} \in [\bar{\beta}]^n)$. The probability of $\mathrm{P}(\mathbf{z_1} \in [\bar{\beta}]^m \cap \mathbf{z_2} \in [\bar{\beta}]^n)$ is equivalent to the probability in Equation 2.5. $(\mathbf{z_1}, \mathbf{z_2}) \neq \perp$ is defined as "success", while $(\mathbf{z_1}, \mathbf{z_2}) = \perp$ is defined as "failure". X is defined as the number of signature attempts until a valid signature is produced, including the

attempt where the valid signature is produced. According to [PU02], the expected number of attempts $\mathbb{E}[X]$ until a valid signature is produced, i.e., $(\mathbf{z_1}, \mathbf{z_2}) \neq \perp$, is given by

$$\mathbb{E}[X] = \frac{1}{\mathrm{P}_{\mathrm{success}}}$$

By substituting $\mathrm{P}_{\mathrm{success}}$ with Equation 2.5, the following expression gives the expected number of signature attempts until a valid signature is produced

$$\mathbb{E}[X] = \frac{1}{\left(\frac{2\bar{\beta}+1}{2(\bar{\beta}+\gamma)+1}\right)^{d(m+n)}} \tag{2.6}$$

As pointed out by the authors of [Lyu20], the presented lattice-based digital signature scheme is fairly similar to the digital signature scheme CRYSTALS-Dilithium, which is to be presented in Section 2.4.2. The schemes differ in Dilithium's focus on reducing the public key and signature size, as well as the performance of the key generation, signature, and verification algorithms.

2.3 FIDO2

The path towards a passwordless future is being paved, with FIDO2 being the standard opted for by industry leaders. FIDO2 is an umbrella term that covers multiple specifications. Two of these are Client to Authenticator Protocol (CTAP) and Web Authentication (WebAuthn), which are two specifications defining the communication between the components in the FIDO2 standard. CTAP is a protocol for secure communication between authenticators and clients, while WebAuthn defines a protocol specification for communication between clients and RPs. Figure 2.16 illustrates the relationship between the components and the different communication protocols defined in the FIDO2 standard.

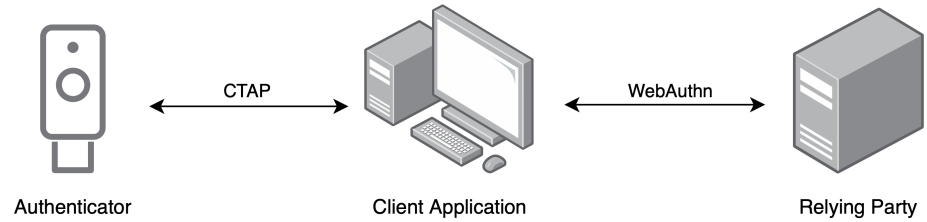


Figure 2.16: The components of the FIDO2 standard and communication protocols between them.

2.3.1 Terminology

Some protocol-specific terminology is needed before diving into specifics of FIDO2.

- **Client Application** An intermediary entity typically implemented in the user agent. Responsible for communication with the RP server and authenticator [Con19].
- **Relying Party** The server that processes register and authentication requests. It also handles user data such as public keys and is responsible for creating challenges for users authenticating towards the service [Con19].
- **Authenticator** A cryptographic entity used by a FIDO2 client to generate a public key credential and register it with an RP, as well as authenticating by verifying the user. The authenticator handles the cryptographic signing of the challenge received from RP [Con19].
- **Test of User Presence** A test of user presence is a simple form of authorization gesture and technical process where a user interacts with an authenticator, yielding a boolean result. Note that this does not constitute user verification as a user presence test, by definition, is not capable of biometric recognition, nor does it involve the presentation of a shared secret such as a password or Personal Identification Number (PIN) [Con19].
- **User Verification** The technical process by which an authenticator verifies the user registering or authenticating towards the RP server. User verification *may* be instigated through various authorization gesture modalities, e.g., through a touch plus PIN code, password entry, or biometric recognition, e.g., presenting a fingerprint. The intent is to distinguish individual users [Con19].
- **Relying Party Identity (RPID)** A unique ID used for identifying a specific RP. The ID is used in various verification checks throughout the protocol [Con19].

The WebAuthn specification defines two ceremonies, registration and authentication. Both of these are passwordless, a user only needs to enter their unique username and have an authentication method (authenticator app, physical security key, or biometrical identification) ready. There are three main components in the specification, RP, a browser + RP JavaScript Application, hereinafter referred to as client application, and an authenticator. The following two sections will explain in detail the key steps in each of the two ceremonies.

2.3.2 Registration Ceremony

The WebAuthn documentation by World Wide Web Consortium (W3C) [Con19] defines the registration ceremony as follows: "The ceremony where a user, a Relying Party, and the user's client (containing at least one authenticator) work in concert to create a public key credential and associate it with the user's Relying Party account. Note that this includes employing a test of user presence or user verification." Most of this process is hidden from the user's perspective, the user only needs to enter a unique username and use an authenticator to register an account. Figure 2.17 shows the registration ceremony as described by the WebAuthn documentation. A user registering for a service will use a browser to visit the webpage. This webpage will be running the JavaScript Application, which is responsible for communicating with the RP server.

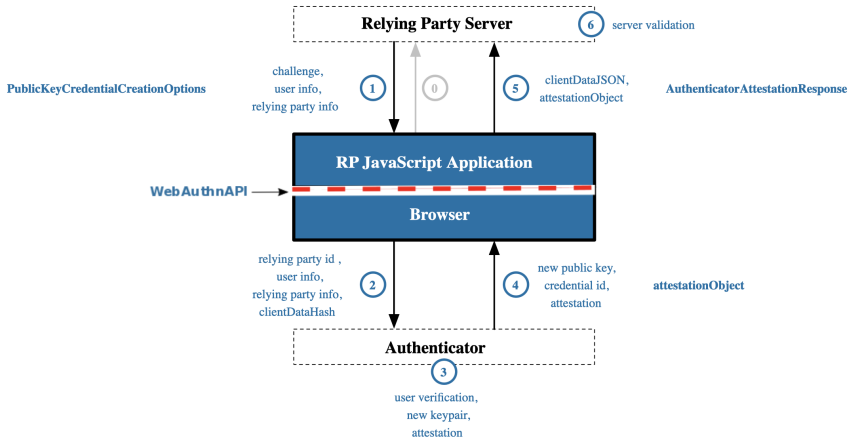


Figure 2.17: WebAuthn Registration Ceremony [Con19].

There are seven main steps in the WebAuthn registration ceremony. Figure 2.17 shows the steps as well as the message flow between the three main components in the protocol.

0. A user enters their username. A request for registration is sent from the client application to the RP server

1. When the RP server receives the username, a unique challenge, user info, and RP information is sent to the client application.

2. The client application forwards this information to the authenticator, together with `clientDataHash`. The `clientDataHash` is a hash of the challenge concatenated with the RPID, i.e., $\texttt{clientDataHash} = \mathcal{H}(\text{challenge} + \text{RPID})$.

3. On reception, the authenticator requires to verify user presence or user verification. After this is completed, a new key pair and credential ID are generated, together with attestation data[7]. The private key is stored with a mapping to the credential ID on the authenticator.

4. From the key pair, the public key is retrieved and sent to the client application together with credential ID and attestation data.

5. The public key and credential data are concatenated into one object called `clientDataJSON`. This object and the attestation data are sent to the RP server

6. RP server runs a series of verification checks on the received data before the public key is stored together with the user ID for use in the authentication ceremony.

2.3.3 Authentication Ceremony

The WebAuthn documentation by W3C [Con19] defines the authentication ceremony as follows: "The ceremony where a user, and the user's client (containing at least one authenticator) work in concert to cryptographically prove to a Relying Party that the user controls the credential private key associated with a previously-registered public key credential. Note that this includes a test of user presence or user verification." As stated in the documentation, the overall goal of the authentication ceremony is to cryptographically prove to the RP server that the user is in possession of the correct private key corresponding with the public key stored by the RP server. The only actions needed from the user are providing their username and a user verification or test of user presence, which are both described in Section 2.3.1.

[7]Attestation data is the generated public key signed with the attestation certificate, which is built into the authenticator when manufacturing it. The attestation certificate is specific to a model [Con23].

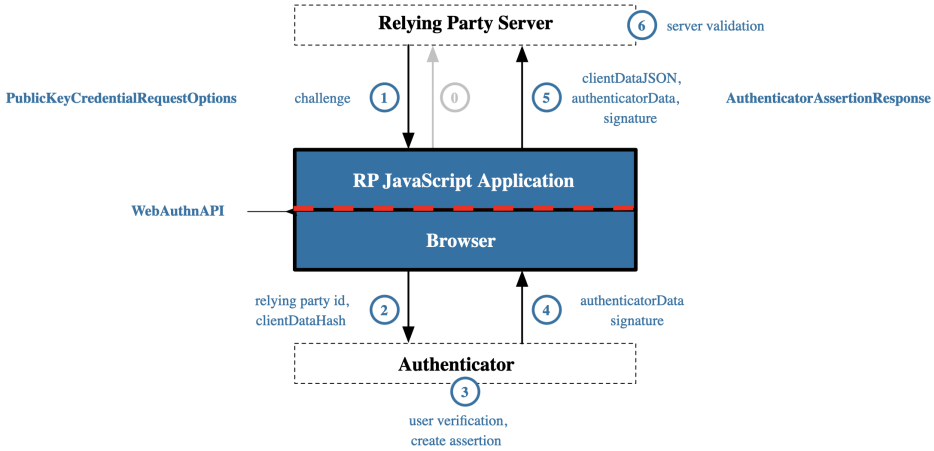


Figure 2.18: WebAuthn Authentication Ceremony [Con19].

Similar to the registration ceremony, the authentication ceremony consists of seven main steps. These are all presented in Figure 2.18.

0. A user enters their username. A request for authentication is sent from the client application to the RP server.

1. If the RP server has an entry for the received username, the corresponding credential ID, as well as a challenge, are sent as a response.

2. The client application sends the RPID and `clientDataHash` to the authenticator.

3. After receiving the request from the client application, the authenticator does a lookup on the received RPID and credential ID. If it finds a private key stored with that specific mapping, it asks the user to consent to the authentication attempt before it signs the received `clientDataHash` with the retrieved private key. It generates the authenticator data which is the hash of the RPID, i.e., `authenticatorData` $= \mathcal{H}(\text{RPID})$.

4. The authenticator returns the signature and authenticator data to the client application.

5. Client application creates a new data object called `clientDataJSON`, which is sent to the RP server together with the signature and authenticator data.

6. RP server conducts a series of verification checks, including verification of the received signature by the use of the corresponding public key received during

registration. If all verification checks are passed, the client is successfully authenticated.

2.3.4 WebAuthn Authenticators

During the registration and authentication ceremony in WebAuthn, an authenticator needs to be used. There are two groups of authenticators, *roaming* and *platform* authenticators. Roaming authenticators are devices separate from the client, such as security keys and smartphones. A platform authenticator is located on the same device as the client, such as on-device biometrical authentication. Common for these authenticators is that they are responsible for generating cryptographic key pairs during the registration ceremony, and responsible for producing legitimate signatures during the authentication ceremonies. Per the WebAuthn documentation, there are four digital signature algorithms that must be implemented by an RP server: the ECDSA on the NIST curve P-256, the RSA signature algorithm with SHA-256, the RSASSA-PSS[8] with SHA-256 and Edwards-curve Digital Signature Algorithm (EdDSA) with 256-bit keys [Con15]. Note, all these signature algorithms are based on underlying problems in cryptography that are considered unsafe with the introduction of a sufficiently powerful quantum computer.

2.4 Related Work

Exploring existing literature relevant to the master's thesis is essential to identify any gaps our thesis may fill. The following presents the current state of research within the field of post-quantum cryptography, as well as the inner workings of existing authenticators that function as key components in WebAuthn.

2.4.1 NIST Post-Quantum Competition

The foreseen realization of a sufficiently efficient and powerful quantum computer, that is able to run Shor's algorithm, is heavily debated. Nevertheless, it is crucial to have cryptosystems and standards that can withstand quantum computers in place *prior* to such an occurrence. Developing and implementing new cryptographic standards is tedious work. The ongoing process to "solicit, evaluate, and standardize one or more quantum-resistant public-key cryptographic algorithms" is led by NIST, and has culminated in four candidates chosen for standardization, CRYSTALS-Kyber, CRYSTALS-Dilithium, Falcon, and SPHINCS$^+$ [NIS17]. CRYSTALS-Kyber is the only public-key encryption key-establishment algorithm, while the other three are digital signature algorithms. The security of both Dilithium and Falcon relies on the hardness of lattice problems, explained in Section 2.2.1, while SPHINCS$^+$ is a stateless hash-based signature algorithm [BHH+15]. Dilithium and Falcon are

[8] RSASSA-PSS is a probabilistic version of RSA.

the most efficient algorithms, and NIST have chosen Dilithium as the primary one. Falcon is to be used by applications in need of smaller signatures than those offered by Dilithium. SPHINCS$^+$ was included as a backup option as its security relies on another mathematical hard problem than the other three algorithms. This section will focus on Dilithium, as it is considered by NIST as the primary digital signature, as well as being the digital signature most similar to the digital signature presented in Section 2.2.4.

2.4.2 CRYSTALS-Dilithium

CRYSTALS-Dilithium [DKL+18] is a digital signature scheme that relies on the hardness of LWE and SIS over module lattices. The scheme is constructed by transforming an interactive zero-knowledge proof into a non-interactive proof, with the use of "Fiat-Shamir with Aborts".

The authors of Dilithium claim that the scheme is easy to implement. A direct cause of this is that sampling of vectors is done following a uniform distribution instead of a Gaussian distribution, as is the case for the digital signature scheme Falcon [FHK+18]. The implementation of sampling from a Gaussian distribution is much more complex and error-prone, and can easily lead to implementation errors, especially if the scheme is deployed in a larger system containing several nuances. Secure implementation of such sampling protected against side-channel attacks is also highly non-trivial, as pointed out by [GHLY16].

The public key size of Dilithium is claimed to be reduced by a factor of 2.5 compared to the previously most efficient lattice-based signatures schemes that also sample from a uniform distribution. This shrinkage is done at the expense of signature size, which increases by roughly 150 bytes for the recommended security level.

Another improvement by the authors of Dilithium is the improved implementation of the Number Theoretic Transform (NTT), which is the main component in lattice-based schemes. NTT is the most efficient method for multiplying two polynomials of high degree with integer coefficients [LZ22], and is a key component in modern cryptosystems that operates on polynomials. For the Advanced Vector Extensions 2 (AVX2) implementation of Dilithium, an improvement by a factor of 2 is achieved compared with similar solutions. AVX2 is an extension for vectorization in the Intel x86 instruction set, allowing for the execution of Single Instruction Multiple Data (SIMD) instructions on 256-bit vectors [AS20].

With the assumption that LWE and SIS are resistant to quantum algorithms, Dilithium is designed to achieve long-term security against such threats. The creators of DIlithium have therefore assumed a very favorable viewpoint for the adversaries which have access to quantum algorithms that require virtually as much space as time.

At the moment, the space required for these algorithms is not feasible, and there are significant obstacles to overcome. However, Dilithium is designed to overcome the challenge of possible future enhancements.

A popular digital signature scheme in use is ECDSA on NIST curve P-256. ECDSA rely on the hardness of finding a discrete logarithm over a group defined by points on an elliptic curve. ECDSA is widely used mainly due to its efficiency, offering shorter keys and signatures, and faster signature generation than most schemes. For example, it allows for the public key to be extracted by the signature itself. ECDSA is a fast, compact, and secure scheme, but this is not the case if quantum computers get sufficiently efficient and powerful. Shor's algorithm is capable of finding solutions to hard problems within Elliptic Curve Cryptography (ECC) in polynomial time [Sho97], effectively breaking all cryptosystems that rely on ECC. This is not the case with post-quantum schemes like Dilithium.

Table 2.1 shows the different key and signature sizes for ECDSA on NIST curve P-256 and Dilithium2. This specific ECDSA instance provides 128-bit security against classical computers, while Dilithium2 offers the same security level against the same adversary given quantum capabilities, i.e., 128-bit quantum security. The data is from [PKLN22]. It is clear that ECDSA offer key and signature sizes far smaller than Dilithium for the same security level on classical computers.

Table 2.2 shows the performance for ECDSA on P-256 and Dilithium2, i.e., the time spent in milliseconds to perform key generation, signing, and verification. The data is from [PKLN22][9]. It can be observed that ECDSA outperforms Dilithium on all procedures except verification. Signing with Dilithium is notably slower than with ECDSA, which is a direct consequence of the signature procedure running in a loop until a valid signature is produced. It is worth mentioning that ECDSA on curve P-256 is the product of years of research and that the performance of post-quantum schemes like Dilithium likely will improve over time.

Table 2.1: Key and signature sizes in bytes for ECDSA on NIST P-256 and Dilithium2. Data is from [PKLN22].

Algorithm	**Public key**	**Private key**	**Signature**
ECDSA P-256	65	32	73
Dilithium2	1312	2544	2420

[9]The tests were carried out on embedded systems with resource constraints, i.e., a Raspberry Pi.

Table 2.2: Time usage in milliseconds for ECDSA on P-256 and Dilithium2 to generate key pairs, signatures, and verification. Data is from [PKLN22].

Algorithm	Key generation	Signing	Verification
ECDSA P-256	1.52	1.94	4.85
Dilithium2	2.04	11.9	2.21

Dilithium's authors use "Fiat-Shamir with Aborts" to transform an interactive zero-knowledge protocol into a non-interactive zero-knowledge protocol and bind it to a message m for it to become a digital signature scheme. As a consequence, rejection sampling is performed before outputting a valid signature. If the rejection sampling algorithm rejects a signature, the signature procedure has to restart. The expected number of attempts before outputting a valid signature can be viewed in Table 2.3. The data is from [BNG22] and shows the expected number of attempts for three different Dilithium implementations corresponding to different security levels.

Table 2.3: The expected number of signature attempts before outputting a valid signature. Covers three different implementations of Dilithium corresponding to three different security levels. Data is from [BNG22].

Algorithm	Attempts
Dilithium2	4.25
Dilithium3	5.1
Dilithium5	3.85

2.4.3 FIDO2 Implementations

FIDO2 is already widely adopted by several services ranging from small start-ups to large-scale enterprises. Microsoft is one of the largest companies marking its spot as a leader in fully adopting the FIDO2 standards for passwordless authentication. Microsoft introduced browser support for fully passwordless authentication with FIDO2 in 2018 [Din]. Users can now authenticate to Azure Active Directory[10] using FIDO2 authenticators, explained in Section 2.3.4. Some other large-scale enterprises that have applied the FIDO2 standard are Amazon Web Services [Ama] and eBay [20a]. eBay has made their RP server open-source to promote open collaboration among the technological community to specify the requirements for secure online service authentication.

[10]Azure Active Directory is Microsoft's identity management system within their cloud service Microsoft Azure.

Authenticators

Authenticators within FIDO2 are categorized based on security level, with Universal Serial Bus (USB) tokens offering the highest level of security. USB tokens are small devices with a USB port that essentially works like a keychain [11]. USB tokens are hardware-based keychains, thus not vulnerable to malicious applications designed to exploit software implementations. The private key is never exposed in USB tokens, as key generation, signing, and storage of private keys are hardware implemented. Some of the commercially available USB tokens are *YubiKey* developed by Yubico, *Solo* developed by SoloKeys, and *SafeNet eToken* developed by the Thales Group. SoloKeys differs from the rest by being open-source [20c], thus appearing more trustworthy than competitors that are not open-source. SoloKeys even offers USB tokens for people to hack. These tokens are flashable, meaning that any firmware update can be installed, and they offer the possibility of installing any unsigned application and enabling debugging on the device.

In early 2023, Apple, Google, and Microsoft announced that they were working on integrating *passkeys* in their solutions [All]. Passkeys are here referred to as the private key used to create legitimate signatures. They aim to offer passwordless authentication with increased usability compared to e.g., USB tokens. They are using copyable passkeys rather than hardware-based ones to provide the possibility of synchronizing passkeys across devices. This broadens the attack surface, potentially reducing security while enhancing usability.

Common for both passkeys and USB tokens is that they only support the digital signature algorithms listed in COSE. Except for HSS/LSM [Hou20], none of the digital signature algorithms listed in COSE are believed to be quantum-resistant, thus compromising passwordless authentication through the FIDO2 standards with the arrival of quantum-capable adversaries.

[11] A keychain is a collection that securely stores and manages a user's digital credentials, such as passwords, private keys, and encryption keys, for convenient and secure access.

Chapter 3
Methodology

The following chapter presents the methodology used to answer the research questions presented in Section 1.4. A design cycle is conducted in order to achieve the main goal of the master's thesis. Each step of the conducted design cycle will be explained and the chosen research methods will be further explained and justified. Chosen research methods will also be linked to the research objectives presented in Section 1.4. A summary of the resources and tools used to achieve the master's thesis' main goal will also be provided. The function and purpose of each resource within the context of the thesis will also be described.

3.1 The Design Cycle

A design cycle, as described by Wieringa [Wie14], is the process of developing an *artifact* and the desired interaction between the artifact and the problem context, i.e., the *treatment*. An artifact "is something created by people for some practical purpose" [Wie14]. The developed artifact, in the context of this master's thesis, is the authentication system that enables passwordless authentication through the use of a quantum-resistant digital signature. The design cycle is a subset of the *engineering cycle*. "The engineering cycle is a rational problem-solving process" consisting of five steps [Wie14]:

Step 1. Problem investigation The investigation of a problem and possible improvement *before* an artifact is designed. The goal is to improve a problematic situation, where problem investigation helps to identify, describe, explain, and evaluate the problem to be treated.

Step 2. Treatment design Specification of the requirements of the artifact and how the defined requirements contribute to the goals. The design process either consists of utilizing available treatments or designing new ones.

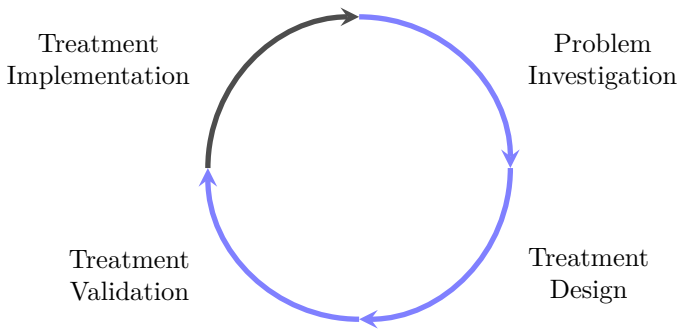


Figure 3.1: The four steps in the engineering cycle defined by Wieringa [Wie14]

Step 3. Treatment validation This involves showing how a designed treatment would help achieve the goals of the stakeholders if applied to a situation where an actual problem exists. This requires the aforementioned requirements to be specified and justified in order to validate that a treatment meets the defined requirements. Due to the lack of an available real-world problem context, one conducts treatment validation. It aims to *predict* the outcomes of a treatment implementation.

Step 4. Treatment implementation The application of the treatment in the original problem context. This involves deploying the developed artifact to a real-world problem context.

Step 5. Implementation evaluation: The investigation of how the deployed artifact interacts with the real-world problem context. The outcome of this evaluation can often result in a new engineering cycle.

The engineering cycle is iterative when the outcome of the implementation evaluation results in a new problem investigation. This iterative process is the reason why it is called a "cycle". Figure 3.1 shows how the engineering cycle can be an iterative process.

The design cycle consists of the three first steps in the engineering cycle, i.e. problem investigation, treatment design, and treatment validation. The result of a design cycle is the treatment, while the engineering cycle on the other hand encompasses the transfer of the treatment to a real-world problem context. Only a design cycle is conducted in this master's thesis. That is because the master's thesis does not indulge in the deployment of the developed artifact and the application of the designed treatment in a real-world problem context, i.e. treatment implementation, and consequently the evaluation of such an implementation, i.e. implementation evaluation. A design cycle, like the engineering cycle, may also be iterative. In a

design science project, researchers often iterate over the three steps in a design cycle many times [Wie14].

A design cycle is conducted in this master's thesis in order to achieve the main goal for the master's thesis, "How can we use lattice-based cryptography in a zero-knowledge protocol to implement an efficient and quantum-resistant authentication system?". The following sections present how each step of the design cycle is conducted.

3.1.1 Problem Investigation

The goal of problem investigation is to improve a problematic situation, where it helps to identify, describe, explain, and evaluate the problem to be treated. Therefore, insight into the existing solutions and their characteristics need to be obtained. A study of the problems within the research scope was conducted, which resulted in knowledge about authentication systems in use today. Common for most authentication systems is the use of textual passwords, which affects both the security and usability of authentication systems. The study of existing solutions revealed the use of cryptosystems vulnerable to quantum adversaries. Furthermore, there is a lack of zero-knowledge protocols used in authentication systems.

After having researched the problem area, a literature study was conducted to identify key aspects of a potential artifact. One goal during the literature study is to identify stakeholders for the artifact, while another is to specify the goals of the system. Before the authors could create the artifact, further knowledge of its composition had to be obtained. This included a literature review of theoretical concepts such as zero-knowledge protocols, lattice cryptography, and FIDO2. These are all concepts that could be utilized to construct a solution to the problems identified during problem investigation. One goal of the chosen methodology is to answer the research questions defined in Section 1.4. Literature study of relevant research helps answer RQ1, as it answers how instances of hard problems within lattice cryptography could be utilized in a zero-knowledge protocol.

3.1.2 Treatment Design

A treatment is a desired interaction between the artifact and the problem context [Wie14]. The design of this treatment involves specifying requirements and further explaining how these requirements contribute to the overall goal of the design cycle. This thesis aims to develop a passwordless authentication system that is both quantum-resistant and zero-knowledge. As a result, a new treatment design is created in lieu of utilizing available treatments. The desired treatment is an interaction where the developed authentication system makes authentication user-friendly through passwordless authentication. The interaction should also be secure against quantum

adversaries as well as leak no information about the private key used to authenticate, i.e., the protocol used to authenticate must be zero-knowledge.

In order to design the desired treatment, requirements that encapsulate the desired properties of the treatment must be specified. These are presented in Table 3.1. The requirements aim to motivate the research questions that were defined in Section 1.4. Requirement 1 describes that the interaction with the artifact, i.e., the authentication system, is pleasant and easy for the end-user while delivering a seamless experience. Pursuing research objectives 2 and 4, from Section 1.4, will have an impact on whether or not requirement 1 is fulfilled. Requirement 2 encapsulates that the system must be fast and efficient when authenticating, thus implying the same requirements on the underlying primitives at work, i.e., the digital signature scheme. Achieving research objective 4 will answer whether or not requirement 2 is reached. Requirement 3 captures how the artifact removes the real-world problem of authentication through passwords. This requires that both research objective 2 and 3 is achieved, as either one alone is not enough in order to fulfill requirement 3. Requirement 4 conveys how the underlying digital signature scheme in the artifact provides end-users with a way to identify themselves towards a service. This requires the pursuit of research objective 3 but also requires that research objective 1 is achieved. Requirement 5 encapsulates the zero-knowledge property of the artifact. Research objective 1 needs to be achieved in order to fulfill this requirement. Requirement 6, covers that the underlying protocol must make use of PQC. This solves the potential future problem of a quantum adversary. Similar to requirement 5, this requires the pursuit of research objective 1, as the resulting protocol needs to be quantum-resistant.

Table 3.1: Treatment requirements.

ID	Description
R1	The system must be usable, i.e., easy to use for end-user
R2	The system must be efficient and fast
R3	The system must remove the need for a password to authenticate
R4	The underlying protocol must function as an identification scheme
R5	The underlying protocol must leak no sensitive information about the user
R6	The underlying protocol must use quantum-resistant cryptography as building block

3.1.3 Treatment Validation

As the treatment is not to be implemented in this thesis, validation is the last step of the iterative cycle. The goal of this process is to "justify that it would contribute to stakeholders' goals if implemented" [Wie14]. A key part of this process is to investigate the effects and evaluate if they satisfy the requirements specified during treatment design. A model of the problem context will be established, and the interaction between a prototype of the artifact and the problem context will be

studied. To validate the treatment, this thesis will create a model of the problem context similar to the real-world implementation. The implementation would be to deploy the authentication system to a service, where interaction between users and the system could be conducted. This is emulated by the authors by viewing the developed artifact as a test environment. This enables the authors to test the interaction between the authentication system and the problem context.

By comparing the results of the validation with both the stakeholder goals and the requirements presented in Table 3.1, one is able to determine a *design theory*. This theory may then be used to predict what an implementation would look like. This will be done by completing research objective 4. The objective states "Test and validate the implemented solution in terms of performance.". If the stakeholder goals or the requirements are not satisfied, new iterations of the design cycle can be initiated until the desired results are reached. As the problem context is emulated, the goal of the validation is to simulate the outcome of applying the treatment in a real-world situation. Thus, having clear requirements is a prerequisite for evaluating the outcome.

3.2 Tools and resources

This section will explore tools and resources to be used when creating the envisioned artifact, i.e. the authentication system. Both software and hardware will be presented, along with an explanation of why specific tools are chosen.

3.2.1 Python

The authors chose Python as the main programming language for the test environment. It is a high-level programming language with a focus on readability and simplicity. It is an interpreted language, which allows for easy prototyping. Python was chosen because of the vast amount of libraries available, in addition to both authors being familiar with it. Python version 3.11.2 was used along with the Integrated Development Environment (IDE) Visual Studio Code (VS Code). A range of Python libraries were used as building blocks during the development. Below follows a list of the most important ones along with a short explanation of them.

Python Libraries

Flask [Pal23] A framework for developing web applications. Used for Hypertext Transfer Protocol (HTTP) handling and routing requests to correct Application Programming Interface (API).

JavaScript Object Notation (JSON) [Pyt23b] Used as the format for all data sent between components in the test environment. Enables encoding and decoding of data in a simple manner.

PyMongo [Mon23] Allows a Python application to interact with the database MongoDB.

NumPy [Num23] The main Python library in the proposed solution. It is used for numerical computing and enables key functionality such as array and matrix operations on both integers and polynomials.

Hashlib [Pyt23a] Gives access to various cryptographic hashing algorithms, including popular ones like SHA-256, SHAKE-256, etc.

os [Pyt23c] Gives Python the ability to interact with the operating system.

time [23b] A time measurement tool in Python. May be used to measure the time between two time instances with nanosecond precision.

3.2.2 Swift

Swift is a high-level, general-purpose compiled programming language developed by Apple. To run Swift programs, the Objective-C run-time library is still used, which allows for C, C++, Objective-C, and Swift to run together in the same program [23g]. Swift was chosen as a programming language because it is the preferred language for iOS development, and is actively developed and maintained by Apple. It was chosen since the authors were familiar with it. Swift has the option of downloading and utilizing reusable components known as Swift packages. Below follows a list of the Swift packages used along with an explanation of them.

Swift Packages

NumPy-iOS [Ahn23a] A package that enables the use of the NumPy library, explained in Section 3.2.1, in iOS applications.

Python-iOS [Ahn23b] A package that enables use of Python modules in iOS applications. This is the underlying software used by NumPy-iOS to enable the use of NumPy.

PythonKit [Vie22] A Swift framework that enables interaction between a Swift program and the local Python interpreter. Python-iOS, thus also NumPy-iOS, relies on PythonKit for NumPy, and other Python libraries, to be used in iOS applications.

3.2.3 Apple Keychain

The Apple Keychain, hereinafter denoted as keychain, is an SQLite database offered by Apple to securely store keychain objects, i.e., passwords, security keys, or any other sensitive data. These objects are either stored locally on the Apple device or on the iCloud[1] keychain, which makes keychain objects accessible to all Apple devices synchronized to iCloud. Two separate AES-256-GCM[2] keys are used to store keychain objects securely. Both keys are protected by Secure Enclave which is a reserved, secure subsystem integrated into Apple's System on Chip (SoC) [3], isolated from the main processor to provide an extra layer of security. Apple Keychain was chosen by the authors to store the user's private key securely.

Keychain objects are classified into accessibility classes. The strictest class, denoted by the global variable **kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly** [23d], restricts access to the keychain object only when the application is in the foreground and when the device is unlocked. This class is only available for devices that have enabled passcode. If the passcode is removed or disabled, both security keys tied to the keychain object are deleted, making the keychain object virtually useless. Keychain objects stored in this class are not synchronized to the iCloud keychain and are not included in security backups.

3.2.4 Xcode

Xcode is Apple's IDE for macOS, mainly used for the development of macOS, iOS, iPadOS, tvOS, and watchOS applications. Xcode supports multiple programming languages, such as Swift, Objective-C, Python, Java, etc. The authors chose Xcode as one of the IDEs used for developing the test environment, mainly because Xcode is the only IDE that supports compilation and building iOS applications through the iOS Software Development Kit (SDK) [23i].

3.2.5 React

React is an open-source front-end JavaScript library used for web development, maintained by Meta Platforms, Inc. [23e]. React uses components, which are independent and reusable bits of code, to create user interfaces. The authors chose React when developing the client application because it is a well-documented front-end library that the authors were already relatively familiar with.

[1] iCloud is Apple's cloud service which enables users to store and synchronize data across Apple devices.

[2] AES-256-GCM is the symmetric-key encryption scheme "Advanced Encryption Standard" with 256-bit key and Galois counter mode as mode of operation.

[3] An integrated circuit where multiple components, i.e., the application processor, Secure Enclave, etc., are combined on a single Central Processing Unit (CPU) chip.

3.2.6 MongoDB

MongoDB is a NoSQL[4] database that is document-oriented. Data is stored in documents similar to JSON and allows for high performance, a flexible schema, and scalability. Along with easy setup, initialization, and interoperability with Python, this resulted in MongoDB database being chosen for the test environment. MongoDB follows a *document data model*. Data is stored in flexible documents rather than fixed columns and rows. The database is built of *collections*, which contains groups of *documents*. A collection is often used to group documents with similar contents, but there are no requirements that the documents must have the same fields. A document is an entry in the database that stores data in field-value pairs. JSON is the most used format for documents, but others like Binary Encoded JavaScript Object Notation (BSON) and Extensible Markup Language (XML) are supported.

3.2.7 Git

Git is a Distributed Version Control System (DVCS) which enables authors to track changes during software development. The authors chose GitHub for this task. GitHub allows for easy collaboration on the same software development project, code review, and serves as backup of source code.

[4]Non-relational

Chapter 4
Proposed Solution

This chapter presents the proposed solution developed by the authors. The solution consists of a post-quantum digital signature scheme and a test environment. The test environment is an authentication system inspired by FIDO2, where an instance of the digital signature scheme presented in Section 2.2.4 will facilitate authentication. Functional and quality requirements for the proposed solution will be presented, alongside a specification of the digital signature from Section 2.2.4. The architecture of the test environment will then be presented before its components and their function in the system are explained.

4.1 Specification

4.1.1 Requirements

To perform tests and validate the digital signature, a test environment is needed. This test environment will enable the authors to implement the digital signature in a system that emulates a real-world system. To facilitate development and create a clear path toward the goals of this thesis, a set of requirements is defined. These are split between *functional* and *quality* requirements. A functional requirement defines the behavior and features the system possesses, while quality requirements describe characteristics the system should have. This could be aspects such as usability, security, performance, etc.

Functional requirements

Table 4.1: Functional requirements for the test environment.

ID	Requirement
FR1	The user interface should be simple and intuitive. Users should be able to register an account without a password.
FR2	Users should be able to authenticate themselves without a password, using the provided authenticator iOS application.
FR3	Users should not be able to register an account with a username already in use.
FR4	Multiple users should not be able to share the same authenticator for the same RP.

Quality requirements

Table 4.2: Quality requirements for the test environment.

ID	Requirement
QR1	The system should implement strong security measures to prevent unauthorized access to user data.
QR2	The system should be able to handle multiple concurrent users without this affecting the performance of the system.
QR3	The system should provide fast and seamless authentication for the user.

4.1.2 Digital Signature

The quantum-resistant zero-knowledge digital signature scheme, presented in Section 2.2.4, will facilitate the authentication on behalf of the user. It will do so by proving knowledge of the private key corresponding to the public key generated during the registration ceremony, presented in Section 2.3.2. To prove knowledge of the private key, the user will, with the help of an authenticator, produce valid signatures on random messages chosen by the RP. The test environment requires three algorithms from the digital signature scheme, the *key generation* algorithm, the *signature* algorithm, and the *verification* algorithm. The key generation and signature algorithm will reside in the authenticator, while the verification algorithm resides in the RP server. Algorithm 4.1 presents the key generation algorithm. A 256-bit seed ζ is sampled to further generate two seeds (ρ, ρ') needed to generate the secret $(\mathbf{s_1}, \mathbf{s_2})$ and the matrix $\mathbf{A}$. The algorithm outputs $(sk = (\mathbf{s_1}, \mathbf{s_2}, \rho'), pk = (\rho', \mathbf{t}))$, where $\mathbf{t} = \mathbf{A}\mathbf{s_1} + \mathbf{s_2}$.

The functions `ExpandS` and `ExpandA` share similarities with the corresponding functions in Dilithium specification version 3.1 [DKL+21] and the original Dilithium specification [DKL+18] respectively. The goal of `ExpandS` is to map the input seed ρ to the secrets $(\mathbf{s_1}, \mathbf{s_2}) \in \mathcal{R}_{q,f}^{m}$, while the goal of `ExpandA` is to map the input seed ρ' to a matrix $\mathbf{A} \in \mathcal{R}_{q,f}^{n \times m}$. This is done differently by the two functions, but the overall methodology is similar. Both functions generate the coefficients of the polynomials they comprise, based on the input seed. The input seed along with a counter is absorbed by an eXtendable-Output Function (XOF). A block of bytes returned from the XOF is then interpreted in a specific way to represent an integer in a given range. Algorithm 4.5 shows how this works for mapping the seed ρ' to a matrix $\mathbf{A} \in \mathcal{R}_{q,f}^{n \times m}$. $\mathbf{A}$ is initialized as m column vectors of length n. $\mathcal{H}$ is instantiated as SHAKE-128. b_2' is b_2 with Most Significant Bit (MSB) equals to 0. `Polynomial(coefs)` maps a list of coefficients to a polynomial, i.e. $p(\mathbf{c}) = c_d x^{d-1} + c_{d-1} x^{d-2} + \ldots c_3 x^2 + c_2 x + c_1, \mathbf{c} \in (c_1, \ldots, c_d)$.

Algorithm 4.1 Key generation algorithm

1: `KeyGen`
2: $\zeta \leftarrow \{0,1\}^{256}$
3: $(\rho, \rho') \in \{0,1\}^{512} \times \{0,1\}^{256} := \mathcal{H}(\zeta)$ ▷ $\mathcal{H}$ is instantiated as SHAKE-256
4: $(\mathbf{s_1}, \mathbf{s_2}) \in \mathcal{R}_{q,f}^{m} \times \mathcal{R}_{q,f}^{n} :=$ `ExpandS`(ρ)
5: $\mathbf{A} \in \mathcal{R}_{q,f}^{n \times m} :=$ `ExpandA`(ρ')
6: $\mathbf{t} = \mathbf{A s_1} + \mathbf{s_2}$
7: **return** $(sk = (\mathbf{s_1}, \mathbf{s_2}, \rho'), pk = (\rho', \mathbf{t}))$

The signature algorithm, shown in Algorithm 4.2, regenerates $\mathbf{A}$ with the use of `ExpandA` and ρ', before sampling a random 512-bit seed ρ. This seed, together with κ, is mapped to the two masking vectors $(\mathbf{y_1}, \mathbf{y_2})$ with the use of `ExpandMask`. `ExpandMask` share similarities with the corresponding function in the original Dilithium specification [DKL+18], and operates similarly as `ExpandS` and `ExpandA`. `HashToBall` maps the hash digest c' to a polynomial $c \in \mathcal{C}$, presented in Equation 2.4. Algorithm 4.3 shows pseudo-code for `HashToBall`. The rest of the algorithm resembles the general signing procedure given in Section 2.2.4. The hash digest c' and ω are included in σ instead of c and $\mathbf{A y_1} + \mathbf{y_2}$ respectively, to keep the output small.

Similar to the signature algorithm, the verification algorithm, shown in Algorithm 4.4, starts with regenerating $\mathbf{A}$ with the use of `ExpandA` and ρ'. The challenge polynomial c is then regenerated with the hash digest c'', and the same checks as given in Section 2.2.4 are made.

The three functions `ExpandS`, `ExpandA`, and `ExpandMask`, used for generating the secret $(\mathbf{s_1}, \mathbf{s_2})$, the matrix $\mathbf{A}$, and the masking vectors $(\mathbf{y_1}, \mathbf{y_2})$ respectively, require

Algorithm 4.2 Signature algorithm

$\texttt{Sign}(sk = (\mathbf{s_1}, \mathbf{s_2}, \rho'), \mu)$
$\mathbf{A} \in \mathcal{R}_{q,f}^{n \times m} := \texttt{ExpandA}(\rho')$
$\mathbf{t} = \mathbf{A}\mathbf{s_1} + \mathbf{s_2}$
$\rho \leftarrow \{0,1\}^{512}$
$\kappa := 0, (\mathbf{z_1}, \mathbf{z_2}) :=\perp$
while $(\mathbf{z_1}, \mathbf{z_2}) :=\perp$ **do**
    $(\mathbf{y_1}, \mathbf{y_2}) \in \mathcal{R}_{q,f}^{m} \times \mathcal{R}_{q,f}^{n} := \texttt{ExpandMask}(\rho, \kappa)$
    $\omega \in \{0,1\}^{384} := \mathcal{H}(\mathbf{A}\mathbf{y_1} + \mathbf{y_2})$ ▷ $\mathcal{H}$ is instantiated as SHAKE-256
    $c' \in \{0,1\}^{384} := \mathcal{H}(\mathbf{A} \mid\mid \mathbf{t} \mid\mid \omega \mid\mid \mu)$
    $c \in \mathcal{C} \subset \mathcal{R}_{q,f} := \texttt{HashToBall}(c')$
    $\mathbf{z_1} = c\mathbf{s_1} + \mathbf{y_1}$
    $\mathbf{z_2} = c\mathbf{s_2} + \mathbf{y_2}$
    if $||\mathbf{z_1}||_\infty > \bar{\beta}$ `or` $||\mathbf{z_2}||_\infty > \bar{\beta}$ **then**
        $(\mathbf{z_1}, \mathbf{z_2}) :=\perp$
    end if
    $\kappa = \kappa + n$
end while
return $\sigma = (\mathbf{z_1}, \mathbf{z_2}, c', \omega)$

Algorithm 4.3 HashToBall algorithm.

$\texttt{HashToBall}(c')$
$\mathbf{c} := (c_1, \ldots, c_d)$
$s \in \{0,1\}^{\eta} := \mathcal{H}(c')$ ▷ $\mathcal{H}$ is instantiated as SHAKE-256
for $i = d - \eta$ `to` d **do**
    $j \leftarrow \{0, \ldots, i\} := \mathcal{H}(c'||i)$
    $c[i] := c[j]$
    $c[j] := -1^{s[i-d+\eta]}$
end for
return $\texttt{Polynomial}(c)$

an input seed that is truly random. The sampling of a truly random seed takes place in the authenticator, which is an iOS application written in Swift. The developed iOS application will be implemented and tested on an iPhone XR running iOS 16.3 as Operative System (OS). The recommended approach to sample random bytes securely is through the Swift function `SecRandomCopyBytes(_:_:_:)`, which returns an array of cryptographically secure random bytes [23f]. If `kSecRandomDefault` is passed as the argument, Apple's kernel Cryptographic Pseudo-Random Number Generator (CPRNG) is used, which is the Cryptographically Secure Pseudo-Random Number Generator (CSPRNG) called Fortuna [FS03]. Fortuna produces cryptographically secure pseudo-random numbers by seeding itself during boot and throughout the device's lifetime with multiple high-quality entropy sources. These entropy sources

are accessed by using the kernel APIs `getrandom(2)` [20b] and `/dev/random` [19]. Another kernel API is `/dev/urandom`, which functions similar to `/dev/random`, but with one key difference. `/dev/random` will block access to the kernel's entropy pool if sufficient entropy is not gathered, and only return a number of bytes within the estimated number of bits of noise in the entropy pool [19]. `/dev/urandom` on the other hand, does not block access, making it susceptible to theoretical cryptographic attacks prior to boot up. `/dev/urandom` is therefore secure enough for emphemeral randomness, but `/dev/random` is preferred when generating long term keys. The authors have therefore concluded that the use of `SecRandomCopyBytes(_:_:_:)` in the digital signature scheme is secure.

Algorithm 4.4 Verification algorithm

$\texttt{Verify}(pk = (\rho', \mathbf{t}), \mu, \sigma = (\mathbf{z_1}, \mathbf{z_2}, c', \omega))$
$\mathbf{A} \in \mathcal{R}_{q,f}^{n \times m} := \texttt{ExpandA}(\rho')$
$c'' \in \{0,1\}^{384} := \mathcal{H}(\mathbf{A} \mid\mid \mathbf{t} \mid\mid \omega \mid\mid \mu)$ ▷ $\mathcal{H}$ is instantiated as SHAKE-256
$c \in \mathcal{C} \subset \mathcal{R}_{q,f} := \texttt{HashToBall}(c'')$
$\omega' \in \{0,1\}^{384} := \mathcal{H}(\mathbf{A}\mathbf{z_1} + \mathbf{z_2} - c\mathbf{t})$
return $[[\omega' = \omega]]$ `and` $[[c'' = c']]$ `and` $[[\ ||\mathbf{z_1}||_\infty \leq \bar{\beta}]]$ `and` $[[\ ||\mathbf{z_2}||_\infty \leq \bar{\beta}]]$

Algorithm 4.5 `ExpandA`

$\texttt{ExpandA}(\rho')$
$\mathbf{A} := (\mathbf{a_1}, \ldots, \mathbf{a_m})$ ▷ $\mathbf{a_i} = (a_{i,1}, \ldots, a_{i,n})$
`counter` $:= 0$
for $i = 1$ `to` n **do**
  for $j = 1$ `to` m **do**
    `coefs` $:= (c_1, \ldots, c_d)$
    $k = 1$
    while $k < d$ **do**
      $b_0, b_1, b_2 \in \{0,1\}^8 \times \{0,1\}^8 \times \{0,1\}^8 \leftarrow \mathcal{H}(\rho' || \texttt{counter})$
      ▷ $\mathcal{H}$ is instantiated as SHAKE-128
      `candidate` $\in \{0, 2^{23} - 1\} := b_2' \cdot 2^{16} + b_1 \cdot 2^8 + b_0$
      if `candidate` $< q$ **then**
        $c_k :=$ `candidate`
        $k := k + 1$
      end if
      `counter` $:=$ `counter` $+ 1$
    end while
    $a_{i,j} :=$ `Polynomial(coefs)`
  end for
end for
return A

The parameters for all three algorithms are listed in Table 4.3. The digital

signature scheme presented in this thesis is very similar to Dilithium. Modulus q and the degree d remains unchanged in Dilithium across security levels. The authors of this thesis have not implemented the NTT, which essentially is the Fast Fourier Transform (FFT) over $\mathbb{Z}_q^*$, to allow for more efficient polynomial multiplication, which is the most expensive operation in the digital signature scheme. Nevertheless, q and d have still been chosen to allow for the possibility of an efficient NTT implementation in the future. $q = 2^{23} - 2^{13} + 1 = 8380417$ and $d = 256$ enables the possibility of an efficient NTT implementation as it allows for a 512th root of unity $r \mod q$ to exists, e.g. $r = 1753$. $X^{256} + 1$ can thus be written as $(x - r)(x - r^3) \cdots (x - r^{2d-1})$, allowing for a polynomial $a \in \mathcal{R}_{q,f}$ to be written as $a = (a(r), a(r^3), \ldots, a(r^{2n-1}))$ with the Chinese Remainder Theorem (CRT). Polynomial multiplication can then be done pointwise [Lyu20] [DKL+18].

As the digital signature scheme relies on the hardness of both Module LWE and Module SIS, it makes sense to choose $n \neq m$. (n, m) is therefore chosen as $(5, 4)$ [Lyu20]. The choice of β impact the security of the scheme, as an increased β makes LWE harder while making SIS easier. $\beta = 5$ is chosen to make LWE and SIS equally hard [Lyu20].

As explained in Section 2.2.3 and given in Equation 2.4, the challenge space $\mathcal{C}$ is chosen to consist of all polynomials with η coefficients from the set $\{0, 1\}$ and the rest equal to 0. η is chosen as the smallest integer such that $\mathcal{C}$ consist of more than 2^{256} polynomials, i.e. $2^\eta \binom{d}{\eta} > 2^{256}$. As $d = 256$, we get $\eta = 60$ as the smallest integer.

The values for γ and $\bar{\beta}$ is from [Lyu20], and results in a scheme instance with 128-bit quantum security.

Table 4.3: Values used for the parameters in the implemented digital signature scheme with 128-bit quantum security [Lyu20].

Parameter	**Value**
q	$2^{23} - 2^{13} + 1$
d	256
$f(X)$	$X^d + 1$
(n, m)	$(5, 4)$
η	60
$\mathcal{C}$	$\{c \in [1] : \|\|c_1\|\| = \eta\}$
γ	275
$\bar{\beta}$	$q - 1/16$

In order to achieve the aforementioned 128-bit quantum security, 384-bit and

256-bit outputs have been chosen for collision resistance and 2^{nd}-preimage resistance respectively, as explained in Section 2.2.4.

4.2 Implementation

This section will present the proposed solution, whose task is to enable testing of the digital signature presented in Section 4.1.2. The source code for the proposed solution is published on GitHub[1]. The architecture of the test environment and its components will be introduced. Lastly, the registration and authentication ceremonies will be presented in detail to explain how each component serves its purpose to enable passwordless authentication with standards from FIDO2.

4.2.1 Architecture

To facilitate for *passwordless* authentication, a software architecture suited for this task had to be chosen. The authors chose an architecture inspired by FIDO2, presented in Section 2.3. To ensure that authentication is zero-knowledge as well as quantum-safe, the digital signature instance presented in Section 4.1.2 is used. The architecture consists of four main components: the RP server, which is a Flask application written in Python, the client application, which is a React application written in JavaScript, the polling server, a Flask application written in Python, and the authenticator application, which is an iOS application written in Swift. Figure 4.1 presents the physical view of the test environment. One thing to notice is that the authors have chosen to not deploy the proposed solution, and instead test the proposed solution by hosting the RP server, client application, and polling server locally, as well as building the authenticator application on the same computer. The following subsections present all four components that comprise the proposed solution.

[1] https://github.com/larsore/TestPlatform

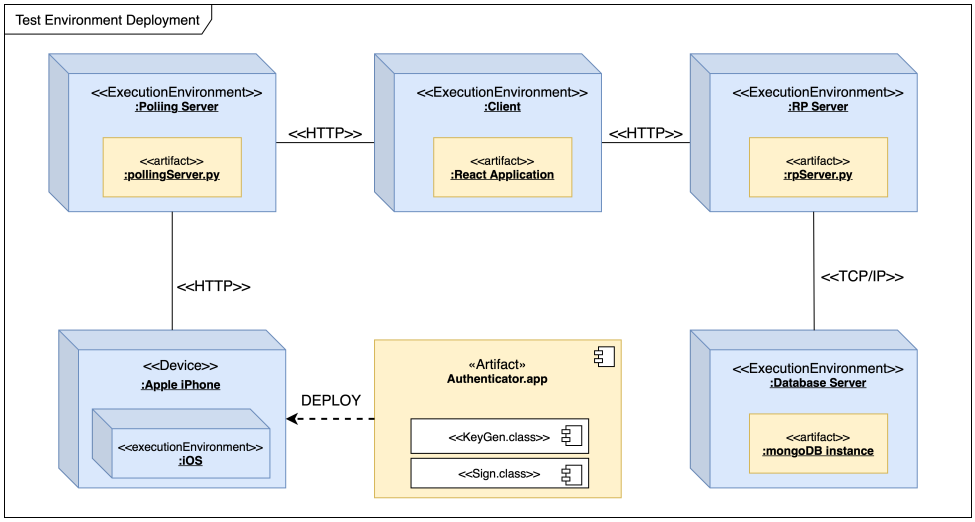


Figure 4.1: Physical view of the proposed solution.

RP Server

The RP server is responsible for handling registration and authentication, which includes handling user data and associated public keys, and verifying signatures generated by the authenticator. The verification is done by implementing the verification algorithm given in Algorithm 4.4. As the authors were mostly interested in the overall architecture of FIDO2 and used the system primarily as a test environment for the digital signature, communication via HTTP was chosen between the client application and RP server. To store usernames and public keys for registered users, the RP server utilizes a database, in this case, a local MongoDB instance.

The server consists of a Flask application, introduced in Section 3.2.1, and a handler class. The Flask application, located in `rpServer.py`, is responsible for three things: defining the API for the RP server, loading updated Internet Protocol (IP) address and digital signature parameters, presented in Section 4.1.2, and instantiating the handler class with the correct parameters. In the original WebAuthn specification [All23], the RP and authenticator negotiate which digital signature instance to use. As the proposed solution realizes a test environment for *one* digital signature scheme, this negotiation is omitted and replaced by storing the digital signature parameters in `para.txt`, a file located in the folder `Authenticator/Authenticator/Model`. Authenticator and RP server read the updated parameters from this file, thus eliminating the need for negotiating which digital signature instance to use. The updated IP address needs to be loaded and specified in the Flask application to enable Cross-Origin Resource Sharing (CORS).

The API for the RP consists of six endpoints which all allow HTTP POST requests: three for registration and three for authentication. The Flask framework checks if incoming HTTP requests have the correct path and method. All API endpoints defined in `rpServer.py` essentially perform the same tasks on incoming requests: checking if the request has payload keys matching a predefined set, and if that holds, delegates the handler class to handle the request. This can be seen in Listing 4.1, which is the API endpoint `/authenticate`, that handles incoming authentication requests from the client application. `rpServer.py` reads the updated digital signature parameters and instantiates the handler class with the correct parameters by reading the contents from `para.txt` and calling a setter method in the handler class.

The handler class, named `Handler`, located in `rpHandler.py`, is delegated by `rpServer.py` to handle and act on incoming requests. This includes updating the state of all users during registration and authentication attempts, storing and retrieving user information from a local MongoDB instance, and verifying signatures generated by authenticators on behalf of users. In production-grade software, these responsibilities would ideally be split up and delegated among several modules, thus increasing the modularity of the software and decreasing cohesion between the different modules. The goal of the authors was to quickly create a test environment where the digital signature scheme could be tested in a practical FIDO2 inspired environment, functioning as a PoC. As a result, the handler class takes up a lot of responsibility that ideally would be separated and distributed among several modules. Figure 4.2 shows a class diagram for the RP server.

```
104 # TestPlatform/rp-server/rpServer.py
105 @app.route("/authenticate", methods=['POST'])
106 @cross_origin(origins=[macClientUrl, iPhoneClientUrl, "http://localhost:3000"])
107 def clientLogin():
108     body = request.json
109     for key in body.keys():
110         body[key] = str(body[key])
111     requiredKeys = ["username"]
112     if not checkKeys(requiredKeys, list(body.keys())):
113         return json.dumps("The provided key is not correct. The correct key is " + ' '.join(requiredKeys))
114     response = responseHandler.handleLogin(body)
115     return response
```

Listing 4.1: API endpoint for `/authenticate`.

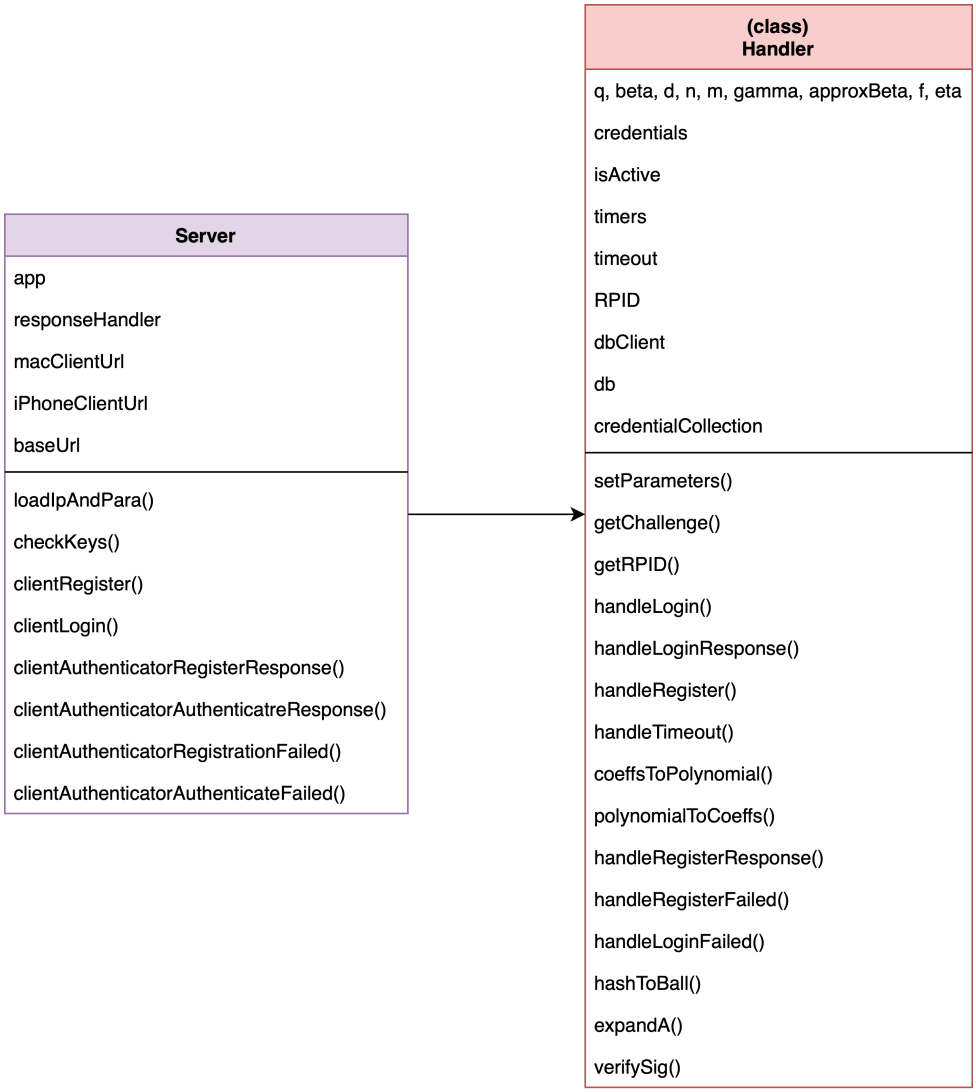


Figure 4.2: Class diagram for the RP server.

Client Application

The client application is the interface towards the users, and is accessed through a user agent, i.e., a browser, and is where registration and authentication requests reside from. The application consists of a React application which serves as a gateway between the RP and the authenticator. The client application has three main react class components: `App` located in `App.jsx`, `Register` located in `register.jsx`, and `Login` located in `login.jsx`. `App` is responsible for coordinating and managing the

main user interface, allowing users to toggle between registration and login, which is done by the method `changeState()`. `Register` and `Login` are responsible for the registration and login interface as well as handling registration and authentication attempts. The attempts are handled in `handleRegister()` and `handleLogin()`. A class diagram for the client application can be viewed in Figure 4.3.

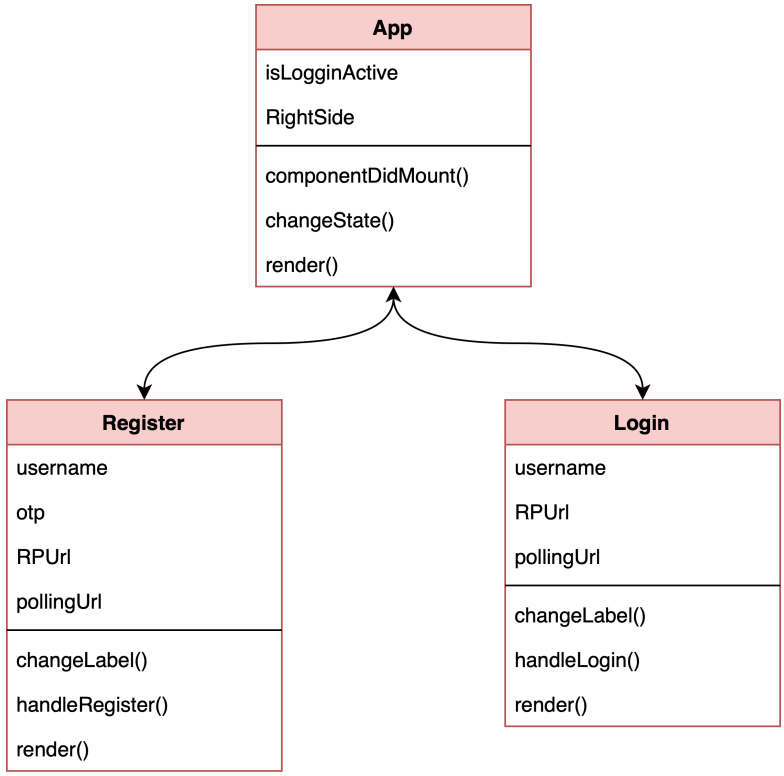


Figure 4.3: Class diagram for the client application.

Polling Server

This is the only component in the test environment that is not included in the FIDO2 specification. The reason for including this is a workaround to bypass Apple's Developer Program and its restrictions. Ideally, the client application should directly communicate with the authenticator application via CTAP. The authors chose to omit the use of CTAP and instead communicate directly with the authenticator through the Apple Push Notification Service (APN). However, to achieve this, one must be a part of the Apple Developer Program. Since the authors are not part of this program, a polling server was developed and placed between the client application and the authenticator as an intermediary. Communication between the client application and authenticator, via the polling server, is done through HTTP. The polling server

has two responsibilities: queueing requests sent from the client application until the authenticator associated with the request polls the server, and forwarding requests sent from the authenticator to the client application.

The polling server consists of a Flask application located in `pollingServer.py` and a handler class located in `pollingHandler.py`. The Flask application is responsible for defining the API for the polling server and loading the updated IP address to enable CORS for the client application. The API consists of eight endpoints which all allow HTTP POST methods: four endpoints dedicated to requests from the client application and four endpoints dedicated to requests from the authenticator. All API endpoints perform the same tasks as the API endpoints in the RP server, i.e., checking if incoming requests have payload keys matching a predefined set, and if that holds, delegate the handler class to handle the request. How this is done for a given API endpoint in this Flask application is similar to Listing 4.1.

Upon the arrival of incoming HTTP requests, the Flask application delegates the handler class to handle the request and provide it with the correct response. The handler class is responsible for coordinating and updating the state of authentication and registration attempts from the client application destined for specific authenticators. It also stores the authenticators and which RPs they are registered to in a local MongoDB instance. This responsibility is handled alone by the handler class at the expense of modularity, as the goal of the authors was to quickly establish a test environment for the digital signature scheme. Figure 4.4 shows a class diagram for the polling server.

Authenticator

The authenticator's main responsibilities is to generate key pairs during registration and authenticate users by signing challenges generated by the RP with the secret key. Key generation is done by implementing the key generation algorithm given in Algorithm 4.1, while signing is done by implementing the algorithm given in Algorithm 4.2. Figure 4.5 shows a class diagram for the authenticator application.

The struct `AuthenticatorApp` includes the entry point for the application, denoted by `@main`, and is responsible for the start-up of the application. It displays the view defined in the struct `authenticatorView` on the device display. `authenticatorView` is responsible for the visual part of the application, i.e., rendering the user interface based on user interaction, as well as scheduling a timer that polls the polling server for pending registration or authentication attempts via the `EventHandler`. It is also responsible for updating polling server via `EventHandler` with the current one-time code.

The class `EventHandler` is responsible for handling all events by the instruction

Figure 4.4: Class diagram for the polling server.

of `authenticatorView`. These events include polling the polling server for incoming registration or authentication attempts, updating the polling server with current one-time code, handle registration or authentication in the case where the user accepts the attempt, as well as handling the event where a user dismisses a registration or authentication attempt. `EventHandler` does not carry out any of these tasks itself, but delegates and coordinates other classes to perform the tasks in a certain sequence on behalf of `EventHandler`. The class is also responsible for reading the updated IP address and digital signature parameters from `para.txt`, to make network requests to the polling server and instantiate the digital signature implementation with updated parameters.

Everything related to performing network requests is delegated to the class `CommunicateWithServer`. It includes the public methods of sending the current one-time code, polling the polling server for incoming registration or authentication attempts, sending the response for either of the two cases, as well as sending a response in the case where a user chooses to dismiss the attempt. The actual payload for these responses is provided by `EventHandler`.

Everything related to the digital signature is delegated to the class `DilithiumLite`. It includes the public methods implementing the key generation algorithm, shown in Algorithm 4.1, and the signature algorithm, shown in Algorithm 4.2, as well as

private methods acting as support modules for the two public methods. Another public method in `DilithiumLite` is `getSecretKeyAsData(secretKey: SecretKey)`, which outputs an object of type `Data`. The type `Data` allows a simple byte buffer in memory to take on the behavior of Foundation objects [23a], i.e., objects of types defined in the Foundation framework, which is a framework that lets an application "access essential data types, collections, and operating-system services to define the base layer of functionality" [23c]. In order to store the private key in the keychain, the `EventHandler` needs to encode the private key, defined by the struct `SecretKey`, as an object of type `Data`.

The last class, `AccessKeychain`, is responsible for interacting with the keychain. This includes storing objects on the keychain, as well as retrieving them. Objects to be stored on the keychain needs to be of type `Data`, as well as being mapped to *an account* and *a service*. The `EventHandler` therefore encodes the private key as an object of type `Data` with the use of `getSecretKeyAsData(secretKey: SecretKey)`, before passing the encoded object to `AccessKeychain`'s public method `saveItem(account: String, service: String, item: Data)`. This method stores the private key mapped to a newly generated credential ID as the account, and the RPID as the service. For the `EventHandler` to retrieve a private key from the keychain, the corresponding credential ID and RPID needs to be included in a query in `AccessKeychain`'s public method `getItem(account: String, service: String)`, which returns the corresponding private key of type `Data`, which `EventHandler` decodes and uses to produce signatures.

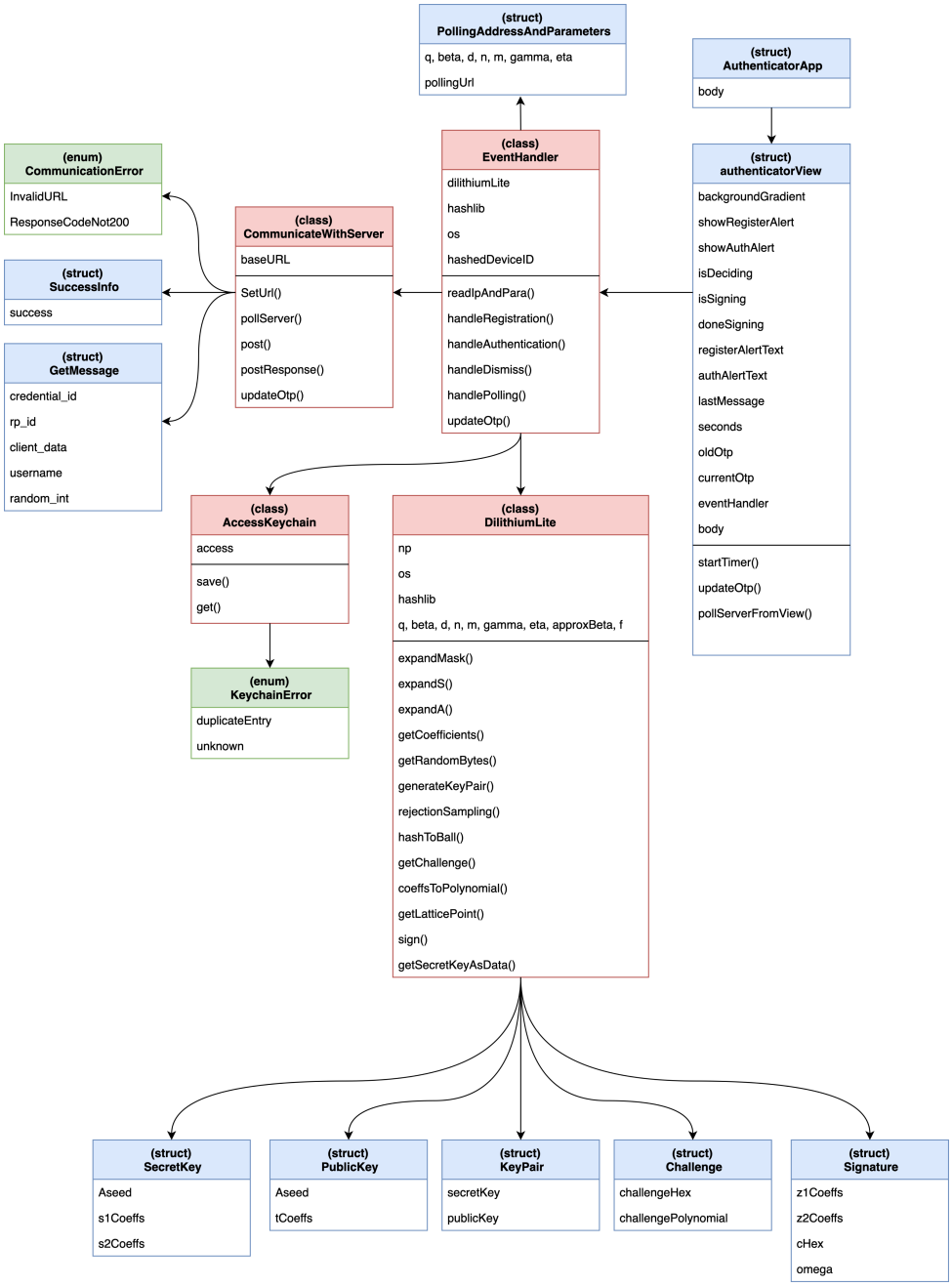


Figure 4.5: Class diagram for the authenticator application.

4.2.2 Registration Ceremony

The FIDO2 registration ceremony, as described in Section 2.3, comprises seven steps. These are all implemented in the test environment. Some simplifications are made, specifically related to the content of messages sent, as parts of the original messages were not considered relevant to the proposed solution. The flow of messages is still true to the specification. Figure 4.6 shows the registration ceremony for the proposed solution. The key generation algorithm in Algorithm 4.1 is used by the authenticator to generate a new key pair, where the public key is transmitted back to the RP and stored in a local MongoDB database.

The authors have decided to split up the ceremony into four phases to make it easier to follow and more readable. **Phase 1** covers the initial registration request made by the client application and the subsequent response from the RP. **Phase 2** entails the client application's creation of `clientData`. The phase also covers the transmission of the request containing `clientData` and additional information, as well as the handling of this request by the polling server. **Phase 3** focuses on the activities of the authenticator, which involves polling the polling server to retrieve pending registration requests, generating a new key pair, and storing the private key securely on the device's keychain. Additionally, this phase includes the transmission of the public key and additional information from the authenticator to the polling server. Lastly, **Phase 4** encompasses the polling server's reception of the response from the authenticator, it's forwarding of this information to the client application, the client application's subsequent forwarding of this data to the RP, the RP's verification of received `clientData` and storage of public key, and the RP's communication of result back to the client application. Details for the messages sent are omitted in Figure 4.6 to increase readability, but an overview of each message can be seen in Table 4.4.

Table 4.4: An overview of messages sent in the registration ceremony for the proposed solution.

Message	Content
RPresponse	{`challenge`, `rp_id`, `timeout`}
clientRequest	{`otp`, `rp_id`, `client_data`, `timeout`, `username`}
pollingResult	{`credential_id`, `rp_id`, `client_data`, `username`, `random_int`}}
authenticatorResponse	{`public_key`, `credential_id`, `rp_id`, `client_data`, `authenticator_id`}
pollingResponse	{`public_key`, `credential_id`, `client_data`, `authenticator_id`}
clientResponse	{`public_key`, `credential_id`, `client_data`, `username`, `authenticator_id`}

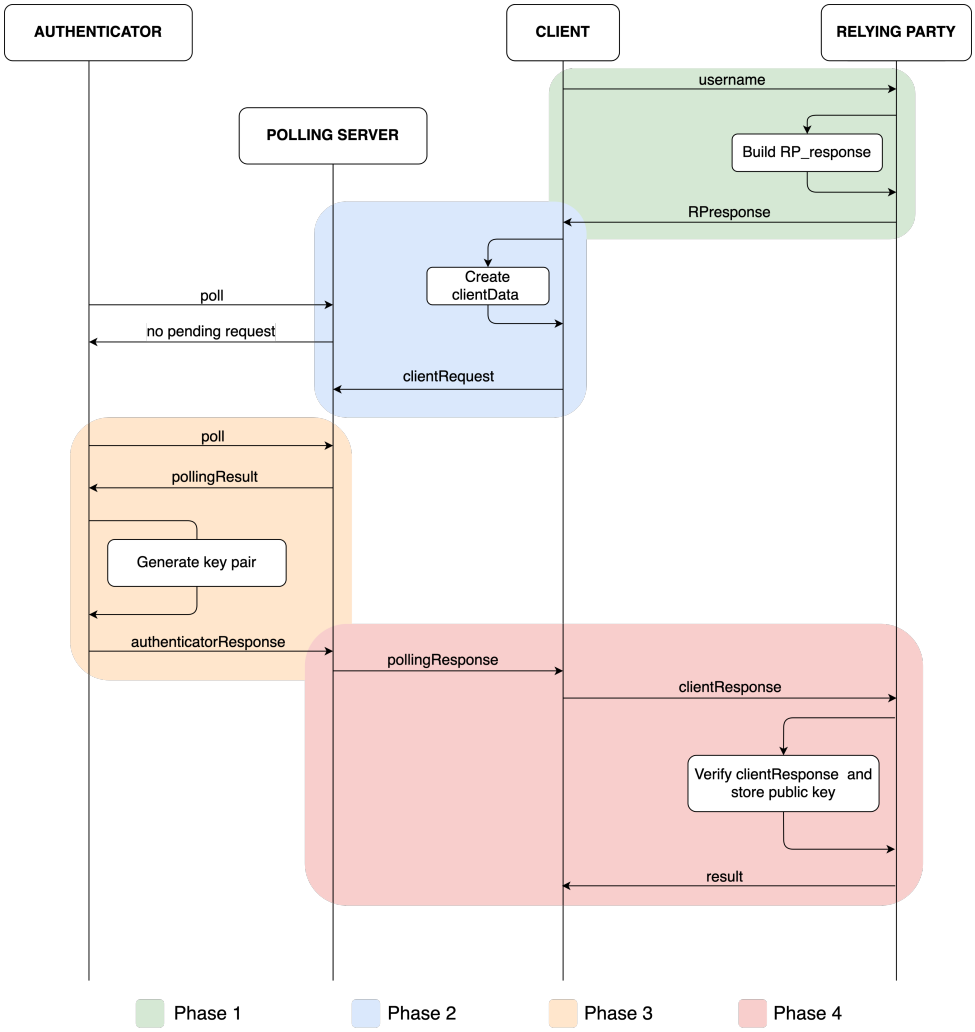
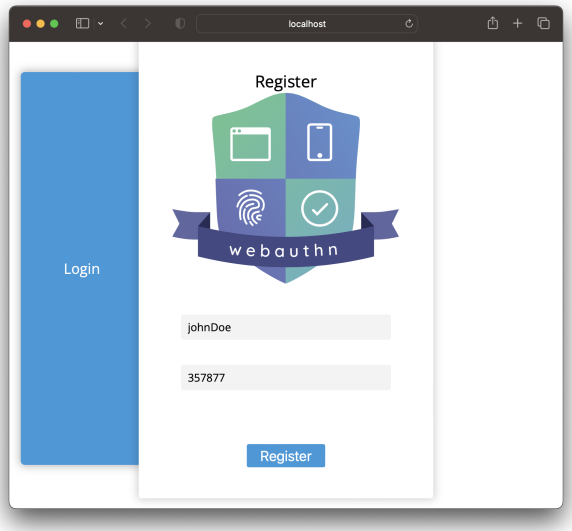


Figure 4.6: Registration ceremony for the proposed solution.

Phase 1

Phase 1 starts with the client application collecting username and a one-time code from the user, and forwarding the username to the RP. The one-time code can be seen on the authenticator application, as shown in Figure 4.7b. The client application collects username and one-time code through the user interface shown in Figure 4.7a. The username is chosen by the user and has to be unique. The one-time code is a random integer between 1 and 999999 inclusively, which is changed every 60th seconds. The authors chose a one-time code as an easy and intuitive way for the user to associate an authenticator with the client application during registration.

Associating an authenticator to the client application in FIDO2 is done through CTAP, which relies on Near-Field Communication (NFC), USB, or Bluetooth to communicate. As CTAP was not implemented by the authors, a one-time code became a reasonable option. The username has to be unique and is ensured by the RP during each registration attempt.



(a) Registration user interface.

(b) One-time code displayed in the authenticator application.

Figure 4.7

On reception of the registration request, `rpServer.py` receives a username and delegates the handler class to handle the request and provide `rpServer.py` with the correct response. The method `handleRegister(cls, body)`, shown in Listing 4.2, handles the request. The key takeaways from this method is the generation of a random 64 byte challenge and a timer, counting down from 30 seconds, getting started. The generation of a random 64 byte challenge happens on line 130 in Listing 4.2. Here, the static method `Handler.getChallenge()` is used, which returns `os.urandom(64).hex()`. The method `os.urandom(`N`)` is a method in the OS library, presented in Section 3.2.1, that samples N random bytes from the OS specific randomness source [Pyt23c]. The authors have developed and tested the RP server on Macbook Pros running macOS Ventura version 13.4. As a result, `os.urandom(`N`)` samples N random bytes via the file `/dev/urandom` [Pyt23c], which, as explained in Section 4.1.2, produces bytes random enough for ephemeral use. The bytes sampled with `os.urandom(`N`)` are thus secure enough as the challenge is only valid for a short timeout period, i.e., 30 seconds. If Windows had been used rather than macOS,

sampling random bytes with `os.urandom(`N`)` would be insecure. This is due to `os.urandom(`N`)` sampling via the deprecated function `CryptGenRandom()`, which is not beneficial as it is not regularly updated and revised, and may include dormant insecurities.

Line 143 instantiates a timer associated with a user, while line 144 starts the timer. The timer is instantiated with a 30 seconds timeout, denoted by `cls.timeout`, which is the maximum duration of both registration and authentication attempts. Line 145-149 returns the response to `rpServer.py`, which forwards it to the client application. The response is denoted as "RPresponse" in Figure 4.6. This concludes phase 1 of the registration ceremony.

```
126 # TestPlatform/rp-server/rpHandler.py
127 def handleRegister(cls, body):
...     # Line 128-129: Make sure the username is not already registered.
...
130     challenge = Handler.getChallenge()
...
...     # Line 131-142: Create a dictionary to keep track of the user's state
...
143     cls.timers[body["username"]] = Timer(cls.timeout, cls.handleTimeout, args=(body["username"], True ,))
144     cls.timers[body["username"]].start()
145     return json.dumps({
146         "challenge": challenge,
147         "rp_id": cls.RPID,
148         "timeout": cls.timeout
149     })
```

Listing 4.2: Method that handles incoming registration requests from client application.

Phase 2

Upon receipt of RPresponse, there are per the specification two tasks to be performed by the client application. One is to create the object `clientData` and send this to the polling server. `clientData` is defined as the hash of the concatenation of RPID and challenge, i.e., $\mathcal{H}$(`RP_ID`||`challenge`), where $\mathcal{H}$ is instantiated as SHA-256. It is the `clientData` the authenticator signs during authentication, and it is used by both the authenticator and RP throughout the registration and authentication ceremony. The other task is to verify the origin of the request, by checking if the HTTP Origin header matches the RPID. This check is not performed in the proposed solution, as the origin of the request varies from time to time due to it being tested locally. If the components were to be deployed, thus getting static IP-addresses, this check could be implemented. Listing 4.3 presents how the client receives this message, stores the content in variables, and creates the `clientData`.

```
48 // TestPlatform/client/src/components/login/register.jsx
49 const RPresponse = await fetch(Register.RPUrl+'/register', RPrequestOptions);
50 const RPdata = await RPresponse.json();
...
... //Line 52-56: Error handling and updating user interface of current progression in registration attempt
...
```

```
59 const rp_id = RPdata["rp"]["id"];
60 const challenge = RPdata["challenge"];
61 const timeout = RPdata["timeout"];
62
63 var clientData = sha256.create();
64 clientData.update(rp_id);
65 clientData.update(challenge);
66 clientData = clientData.hex();
```

Listing 4.3: Parsing received response from RP and generating `clientData`.

Line 67-77 in Listing 4.4 crafts the HTTP POST request to be sent to the polling server. This request is denoted as "clientRequest" in Figure 4.6. As seen on line 78, the request is sent to the API-endpoint `/client/register`.

```
66 // TestPlatform/client/src/components/login/register.jsx
67 const pollingRequestOptions = {
68           method: 'POST',
69           headers: { 'Content-Type': 'application/json' },
70           body: JSON.stringify({
71               "otp": otp,
72               "rp_id": rp_id,
73               "client_data": clientData,
74               "timeout": timeout,
75               "username": username
76           })
77       };
78       const pollingResponse = await fetch(Register.pollingUrl+'/client/register', pollingRequestOptions);
```

Listing 4.4: Forward clientData to polling server.

This request is received by the defined API endpoint for `/client/register` in `pollingServer.py`, which like the RP delegates the handler class in `pollingHandler.py` to handle incoming requests and return the correct response. The method `handlePOSTClientRegister(cls, registerRequest)`, shown in Listing 4.5, is responsible for handling the request.

The first action is to check whether or not the received one-time code is valid. The polling handler keeps track of the current mapping of one-time code and authenticator ID, which is an alphanumeric Universally Unique IDentifier (UUID) string that uniquely identifies an authenticator. The authenticator ID is used by the authenticator to poll the polling server for pending requests. As the current one-time code changes every 60th second, the authenticator sends a HTTP POST request to the polling server's API endpoint `/authenticator/update`, which ensures that the polling server has the correct and updated mapping between one-time code and authenticator ID. If the received one-time code is valid, the polling server retrieves the authenticator ID currently mapped to the one-time code, before conducting a set of verification checks. If the checks hold, a dictionary holding the state for each authenticator is updated with a new key-value pair. The content of the request is stored in a queue in `activeRequests` on lines 57-63. As the response returned by this method is dependent on the actions of the user with the authenticator, a countdown from 30 seconds is

started on line 71. This countdown checks two things every 0.5 second: if there lies a response from the authenticator ready to be sent to the client application, or if the authenticator chose to dismiss the registration attempt, shown on lines 77 and 74 respectively. If neither of these checks holds after 30 seconds, the authenticator is removed from the dictionary holding the state and a response indicating a timeout is returned to the client application. This concludes phase 2 of the registration ceremony.

```
39  # TestPlatform/polling-server/pollingHandler.py
40  def handlePOSTClientRegister(cls, registerRequest):
...     #Line 41-55: Retrieving the authenticator ID mapped to the received one-time code and checking if the
            authenticator is already registered or in the middle of a registration attempt. A dictionary holding
            the state for each authenticator is initialized if the checks hold.
...
56      if len(cls.activeRequests[authID]["R"]) == 0:
57          cls.activeRequests[authID]["R"].append({
58              "credential_id": "",
59              "rp_id": registerRequest["rp_id"],
60              "client_data": registerRequest["client_data"],
61              "username": registerRequest["username"],
62              "random_int": ""
63          })
...
...         #Line 64-67: Updating the state for the authenticator.
...
68          timeout = int(registerRequest["timeout"])
69          waitedTime = 0
70          interval = 0.1
71          while waitedTime <= timeout:
72              waitedTime += interval
73              time.sleep(interval)
74              if cls.activeRequests[authID]["dismissed"]:
75                  cls.activeRequests.pop(authID, None)
76                  return json.dumps("Authenticator chose to dismiss the registration attempt")
77              if authID in list(cls.responseToClient.keys()):
78                  response = cls.responseToClient.pop(authID, None)
79                  cls.isActive[authID]["R"] = False
80                  return json.dumps(response)
81          cls.activeRequests.pop(authID, None)
82          cls.isActive.pop(authID, None)
83          return json.dumps("Timeout")
84      return json.dumps("Pending registration already exists for the given authenticator")
```

Listing 4.5: Handle registration attempt from client application in polling server.

Phase 3

Phase 3 is initiated by the authenticator polling the polling server for pending registration requests. The authenticator polls the polling server every second by calling the function `startTimer()` when the application starts. This function schedules a timer that performs the same task every second. The task is to call the function `pollServerFromView()`, that instructs `EventHandler` to poll the polling server via the function `pollServer()` in `CommunicateWithServer`, shown in Listing 4.6. The polling is a HTTP POST request with the hashed authenticator ID as payload. The polling server delegates the method `handlePOSTAuthenticator(cls, body)`, shown in Listing 4.7, to handle the request upon receival of it. The polling server first checks

if the authenticator is registered on line 131, before retrieving the state stored for the specific authenticator on line 133. It then checks if the queue holding registration requests are non-empty, and returns a response, denoted as "pollingResult" in Figure 4.6, which includes an empty credential ID. An empty credential ID and random integer are included for pollingResult to be identical when registering and authenticating, making it easier for the authenticator to differentiate between them. The authenticator checks whether or not the value for the credential ID is empty, which indicates a registration attempt. Line 134-136 checks if the authenticator has a pending registration request and returns it. Line 137-139 does the same for authentication requests.

```swift
// TestPlatform/Authenticator/Authenticator/Model/communicateWithServer.swift
static func pollServer(hashedDeviceID: String) async throws -> GetMessage? {
    guard let baseUrl = CommunicateWithServer.baseURL else {
        print("BaseUrl not set")
        return nil
    }
    guard let url = URL(string: baseUrl + "/poll") else {
        throw CommunicationError.InvalidURL
    }
    let body: [String: Any] = [
        "authenticator_id":hashedDeviceID
    ]
    guard let data = try await CommunicateWithServer.post(url: url, body: body) else {
        print("Unable to get response from server")
        return nil
    }
    return try JSONDecoder().decode(GetMessage.self, from: data)
}
```

Listing 4.6: Function in `CommunicateWithServer` responsible for polling the server.

```python
# TestPlatform/polling-server/pollingHandler.py
def handlePOSTAuthenticator(cls, body):
    if body["authenticator_id"] not in list(cls.activeRequests.keys()):
        return json.dumps("Authenticator with specified ID has not been registered")
    activeRequests = cls.activeRequests[body["authenticator_id"]]
    if len(activeRequests["R"]) != 0:
        request = activeRequests["R"].pop()
        return json.dumps(request)
    elif len(activeRequests["A"]) != 0:
        request = activeRequests["A"].pop()
        return json.dumps(request)
    return json.dumps("No pending requests for authenticator")
```

Listing 4.7: Method in `pollingHandler.py` responsible for handling incoming polling requests from authenticators.

Figure 4.8 shows the alert that pops up when the authenticator polls the polling server and receives a registration request as a response. The user can either dismiss the registration attempt or accept it, denoted by the two choices **Dismiss** and **Register**. If the user chooses to dismiss the registration attempt, a response indicating dismissal is sent from the authenticator to the polling server, which will update the state for the authenticator. If this is done within the timeout period, the method in Listing 4.5 will notice the change in state for the authenticator and return a response to the

client application. The client application will then send a HTTP request to the API endpoint `/authenticator/register/failed` in the RP server to notify the RP of failed registration.

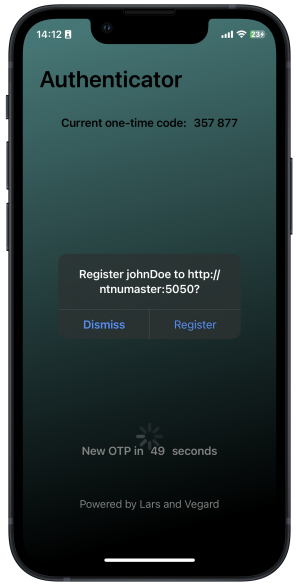


Figure 4.8: Alert that pops up when a user tries to register with an authenticator.

In the case where the user accepts the registration attempt, the `authenticatorView` in the authenticator application calls the method `handleRegistration(RP_ID: String, clientData: String)` in `EventHandler`, shown in Listing 4.8. This method performs the following four tasks: generating a key pair and a credential ID, storing the private key on the keychain, and sending the response to the polling server, denoted as "authenticatorResponse" in Figure 4.6. This response includes the generated public key and credential ID, the received `clientData`, as well as the hashed authenticator ID and RPID. Generation of a new key pair is done on line 95 by calling the method `generateKeyPair()` in `DilithiumLite`, shown in Listing 4.9, which returns a `KeyPair`. This method is the authors' implementation of the key generation algorithm shown in Algorithm 4.1.

```
93  // TestPlatform/Authenticator/Authenticator/Model/eventHandler.swift
94  func handleRegistration(RP_ID: String, clientData: String) {
95      guard let keyPair = dilithiumLite.generateKeyPair() else {
96          print("Unable to generate keypair...")
97          return
98      }
99      let credential_ID = UUID().uuidString
100     let encodedSecretKey = DilithiumLite.getSecretKeyAsData(secretKey: keyPair.secretKey)!
101     do {
102         try AccessKeychain.saveItem(account: credential_ID,
103                                     service: RP_ID,
```

```
104                         item: encodedSecretKey)
105     } catch {
106         print(error)
107         return
108     }
...
...     // Line 109-120: Instructing CommunicateWithServer to send response to the polling server.
...
121 }
```

Listing 4.8: `EventHandler`'s method for handling registration attempts.

```
200 // TestPlatform/Authenticator/Authenticator/Model/SignatureAlgs/dilithiumLite.swift
201 public func generateKeyPair() -> KeyPair? {
202     guard let zeta = self.getRandomBytes(count: 32) else {
203         print("Unable to sample zeta with getRandomBytes")
204         return nil
205     }
...
...     // Line 206-219: Generating the three seeds needed to generate the secrets (s1, s2) and the matrix A.
...
220     let s1 = self.expandS(seed: Python.str(rho1).encode(), noOfPoly: self.m)
221     let s2 = self.expandS(seed: Python.str(rho2).encode(), noOfPoly: self.n)
222     let sk = SecretKey(
223         s1Coeffs: self.getCoefficients(polyList: s1),
224         s2Coeffs: self.getCoefficients(polyList: s2),
225         Aseed: rhoprime
226     )
227     let A = self.expandA(seed: Python.str(sk.Aseed).encode())
228     let t = self.getLatticePoint(A: A, s: s1, e: s2)
229     let pk = PublicKey(
230         Aseed: sk.Aseed,
231         tCoeffs: self.getCoefficients(polyList: t)
232     )
233     return KeyPair(secretKey: sk, publicKey: pk)
234 }
```

Listing 4.9: Method implementing the key generation algorithm from Algorithm 4.1.

The method starts with sampling the seed ζ, denoted by the variable `zeta`. This is done through the method `getRandomBytes(count: Int)` shown in Listing 4.10, which uses the function `SecRandomCopyBytes()` to return an array of cryptographically secure random bytes [23f]. The default random number generator is used for this, as specified by passing `kSecRandomDefault` as an argument on line 192. The authors have considered this to be secure for generating key pairs, as explained in Section 4.1.2. The method `generateKeyPair()` then generates three 256-bit seeds used in `expandS(seed: PythonObject, noOfPoly: Int)` to generate $\mathbf{s_1}, \mathbf{s_2}$ and in `expandA(seed: PythonObject)` to generate $\mathbf{A}$. These seeds are generated by absorbing ζ in an XOF instantiated as SHAKE-256 and outputting 768 bits. `expandS(seed: PythonObject, noOfPoly: Int)` and `expandA(seed: PythonObject)` is the implementation of the corresponding functions in Algorithm 4.1. `expandS(seed: PythonObject, noOfPoly: Int)` outputs a NumPy array of `noOfPoly` NumPy polynomials $\in [\beta]$, where β is denoted as the variable `self.beta`. `expandA(seed: PythonObject)` outputs an $n \times m$ NumPy array consisting of NumPy polynomials $\in \mathcal{R}_{q,f}$.

```swift
// /TestPlatform/Authenticator/Authenticator/Model/SignatureAlgs/dilithiumLite.swift
private func getRandomBytes(count: Int) -> [Int8]? {
    var bytes = [Int8](repeating: 0, count: count)
    let status = SecRandomCopyBytes(kSecRandomDefault, bytes.count, &bytes)
    if status == errSecSuccess {
        return bytes
    } else {
        print("Unable to sample random bytes")
        return nil
    }
}
```

Listing 4.10: Method for sampling random bytes.

The authors chose to implement mathematical operations between vectors and matrices of polynomials with NumPy arrays and NumPy polynomials since they are fast and effective. To access these libraries from the iOS application, the packages NumPy-iOS, Python-iOS, and PythonKit, presented in Section 3.2.2, had to be used. These packages are wrappers that provide access to Python libraries, including NumPy, hashlib, os, etc. Python libraries can thus be imported in any Swift file, as shown for the class `DilithiumLite` in Listing 4.11. Objects instantiated with imported Python libraries are all of the type `PythonObject`. Swift has no knowledge of the rules guarding such objects, thus no additional type safety is ensured by XCode when performing actions on `PythonObject`s.

```swift
// TestPlatform/Authenticator/Authenticator/Model/SignatureAlgs/dilithiumLite.swift
PythonSupport.initialize()
NumPySupport.sitePackagesURL.insertPythonPath()
self.np = Python.import("numpy")
self.os = Python.import("os")
self.hashlib = Python.import("hashlib")
```

Listing 4.11: Importing Python libraries.

The secret $(\mathbf{s_1}, \mathbf{s_2})$ is generated on line 220 and 221 in Listing 4.9. A `SecretKey` struct containing the coefficients of $\mathbf{s_1}$ and $\mathbf{s_2}$, as well as ρ', denoted by `rhoprime`, is created on line 222-226. The coefficients of $\mathbf{s_1}$ and $\mathbf{s_2}$ are included in the struct instead of the NumPy polynomials because Swift has to have full knowledge of the type in order to encode and decode it. ρ' is included instead of $\mathbf{A}$ to reduce the size of the private key, thus generating $\mathbf{A}$ on demand. It must be included to be able to calculate the public lattice point $\mathbf{t} \in \Lambda_q^{\perp}(\mathbf{A})$, i.e. $\mathbf{t} = \mathbf{A}\mathbf{s_1} + \mathbf{s_2}$, when producing signatures. Line 227 and 228 generates $\mathbf{A}$ and $\mathbf{t}$ with the help of `expandA(seed: PythonObject)` and `getLatticePoint(A: PythonObject, s: PythonObject, e: PythonObject)` respectively. Lastly, line 229-233 creates a `PublicKey` struct consisting of ρ' and the coefficients of $\mathbf{t}$ and outputs a `KeyPair` struct consisting of the private and public key.

The next task for `EventHandler` is to generate a random credential ID, which is done on line 99 in Listing 4.8. To do this, a `UUID` struct is instantiated and the variable `UUID().uuidString` is accessed, which produces a random UUID string. The

`EventHandler` encodes the newly generated private key as `Data` with the use of `getSecretKeyAsData(secretKey: SecretKey)` in `DilithiumLite`. The private key is stored on the keychain with `AccessKeychains` method `saveItem(account: String, service: String, item: Data)` on line 107-114. A key takeaway from `AccessKeychain` is the variable `access` shown in Listing 4.12. This variable is included when storing and subsequently retrieving the private key from the keychain and specifies the access control for the keychain object. As explained in Section 3.2.3, `kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly` denotes the strictest class and restricts access to the keychain object only when the application is in the foreground and when the device is unlocked. The flag `.userPresence` means that the keychain object can only be retrieved if Face ID or passcode is provided on demand, thus realizing *user verification*, presented in Section 2.3.1, in the proposed solution.

```
11 // TestPlatform/Authenticator/Authenticator/Model/accessKeychain.swift
12 private static let access = SecAccessControlCreateWithFlags(nil, // Use the default allocator.
13                                         kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly,
14                                         .userPresence,
15                                         nil) // Ignore any error.
```

Listing 4.12: Accessability class for the stored private key.

The last task for the `EventHandler` is to send the newly generated public key and credential ID, as well as the `clientData`, RPID, and authenticator ID to the polling server. This is done on lines 109-120 in Listing 4.8, which instructs `CommunicateWithServer` to send the response as a JSON object to the polling server. The transmission of this response concludes phase 3.

Phase 4

Phase 4 is initiated by the polling server's receipt of the authenticatorResponse, which is sent to the API endpoint `/authenticator/register`. The polling server delegates the method `handlePOSTAuthenticatorRegister(cls, registerRequest)`, shown in Listing 4.13, to handle the response. The method starts with checking whether or not the registration attempt has timed out. It updates the dictionary holding the state for the authenticator with the newly registered RP on line 160 if this is not the case. It then updates a dictionary holding the response for each authenticator with a pollingResonse on lines 162-168. This is the same dictionary that `handlePOSTClientRegister(cls, registerRequest)` checks if contains a response on line 77 in Listing 4.5. The method `handlePOSTAuthenticatorRegister(cls, registerRequest)` then queries the MongoDB instance for the document storing information for the authenticator on line 169. If the document exists, it is updated with the RPID of the RP on line 171. This realizes that an authenticator may be registered to multiple RPs. As the authors have not implemented an additional RP, this code will never be executed, but it was included to illustrate the possibility for an authenticator to register to multiple RPs. If the document does not exist, a

new document is created and inserted into the collection on lines 173-177, before returning a response indicating success to the authenticator.

```
156 # TestPlatform/polling-server/pollingHandler.py
157 def handlePOSTAuthenticatorRegister(cls, registerRequest):
... # Line 158-159: Check if the registration attempt has timed out.
...
160     cls.activeRequests[registerRequest["authenticator_id"]]["RPs"].append(registerRequest["rp_id"])
161     if registerRequest["authenticator_id"] not in list(cls.responseToClient.keys()):
162         cls.responseToClient[registerRequest["authenticator_id"]] = {
163             "credential_id": registerRequest["credential_id"],
164             "public_key_t": registerRequest["public_key_t"],
165             "public_key_seed": registerRequest["public_key_seed"],
166             "client_data": registerRequest["client_data"],
167             "authenticator_id": registerRequest["authenticator_id"]
168         }
169     docs = cls.authenticatorCollection.find({"_id": registerRequest["authenticator_id"]})
170     if len(list(docs)) > 0:
171         cls.authenticatorCollection.update_one({"_id": registerRequest["authenticator_id"]}, {"$push": {"RPs",
                registerRequest["rp_id"]}})
172     else:
173         newDoc = {
174             "_id": registerRequest["authenticator_id"],
175             "RPs": [registerRequest["rp_id"]]
176         }
177         cls.authenticatorCollection.insert_one(newDoc)
178     return json.dumps({"success": "NS Auth Reg"})
```

Listing 4.13: Method in `pollingHandler.py` that handles registration responses from authenticators.

Line 77 in Listing 4.5 makes sure that the response, denoted as "pollingResponse" in Table 4.4, is forwarded to the client application. The client application's main task is to forward this to the RP. Listing 4.14 shows what the client application does when receiving the response from the polling server. If the user dismissed the registration attempt, the client application will send a request to the RP indicating this, or that an error occurred, on line 89. If the response is not erroneous, an HTTP POST request containing the pollingResponse and the username, denoted as "clientResponse" in Figure 4.6, is crafted and forwarded to the RP on line 96-108.

```
78  // TestPlatform/client/src/components/login/register.jsx
79  const pollingData = await pollingResponse.json();
...
...     // Line 81-93: Checks if the response indicates a dismissal or an error, and crafts a HTTP POST requests to
            be sent to RP server. Logging and updating the user interface
...
96  const RPresponseOptions = {
97      method: 'POST',
98      headers: { 'Content-Type': 'application/json' },
99      body: JSON.stringify({
100         "username": username,
101         "credential_id": pollingData["credential_id"],
102         "public_key_t": pollingData["public_key_t"],
103         "public_key_seed": pollingData["public_key_seed"],
104         "client_data": pollingData["client_data"],
105         "authenticator_id": pollingData["authenticator_id"]
106     })
107 };
108 const RPresponseResponse = await fetch(Register.RPUrl+'/authenticator/register', RPresponseOptions);
```

Listing 4.14: clientResponse sent by client application during registration.

When the RP receives the final request from the client application, it has two tasks: verify the correctness of `clientData` and store the received public key, credential ID, username, and authenticator ID in the user collection. Listing 4.15 presents the method `handleRegisterResponse(cls, body)`, which is responsible for carrying out the two tasks. Before any of the two tasks can be initiated, the method checks whether or not the registration attempt has timed out. To verify the correctness of `clientData`, the RP first need to create the expected value of `clientData`. During the first part of the registration attempt, a challenge was created by the RP server. A hash of the concatenation of challenge and `rpid` defines the variable `clientData`, i.e. $\mathcal{H}(\texttt{challenge}||\texttt{RPID})$, where $\mathcal{H}$ was instantiated as SHA-256. This expected value is created in Listing 4.15 on lines 188-190, while the comparison with the received `clientData` takes place on line 191. In the event that these two values differ, it signifies tampering with the received variable at some point during the registration attempt. Consequently, the registration attempt is terminated, and an error message is transmitted back to the client application on line 219. If the verification of `clientData` holds, the received public key is stored along with username, authenticator ID, and credential ID, as seen on lines 195-204. A response indicating successful registration is returned to `rpServer.py`, which in turn sends this to the client application which displays this to the user. This concludes the registration ceremony.

```
180 # TestPlatform/rp-server/rpHandler.py
181 def handleRegisterResponse(cls, body):
... 
...     # Line 182-187: Checks if the registration attempt has timed out and updates the user state if this is not
            the case.
... 
188     expectedHash = sha256()
189     expectedHash.update(cls.RPID.encode())
190     expectedHash.update(cls.credentials[body["username"]]["challenge"].encode())
191     if expectedHash.hexdigest() == body["client_data"]:
192         cls.timers[body["username"]].cancel()
193         docs = cls.credentialCollection.find({"username": body["username"]})
194         if len(list(docs)) == 0:
195             doc = {
196                 "username":body["username"],
197                 "authenticator_id":body["authenticator_id"],
198                 "credential_id":body["credential_id"],
199                 "pubKey":{
200                     "t": json.loads(body["public_key_t"]),
201                     "Aseed": str(body["public_key_seed"])
202                 }
203             }
204             cls.credentialCollection.insert_one(doc)
... 
...             # Line 205-211: Updates the state for the newly registered user.
... 
212             return json.dumps(body["username"]+" is now registered!")
213         cls.credentials.pop(body["username"], None)
214         return json.dumps({
215             "msg": "User already registered for some reason",
216             "reason": "userAlreadyRegistered"
217             })
218     cls.credentials.pop(body["username"], None)
219     return json.dumps({
220         "msg": "Not the same clientData!",
```

```
221         "reason": "cryptoVerificationFailure"})
```

Listing 4.15: Verification of `clientData` and storage of public key.

4.2.3 Authentication Ceremony

Figure 4.9 shows the authentication ceremony. Like the registration ceremony, the content of messages is simplified to only contain information relevant to the proposed solution. The figure gives insight into the responsibilities each component has during authentication. The signature algorithm in Algorithm 4.2 is used by the authenticator during the authentication ceremony to produce signatures, while the verification algorithm in Algorithm 4.4 is used by the RP to verify them. Figure 4.9 shows the authentication ceremony for the proposed solution.

The authentication ceremony has been divided into four phases, similar to the registration ceremony. **Phase 1** covers the initial authentication request initiated by the client application and the subsequent response received from the RP. **Phase 2** focuses on the reception of this response and the creation of `clientData`. Additionally, this phase involves the transmission of clientRequest by the client application and its subsequent receipt by the polling server. **Phase 3** revolves around the activities of the authenticator, including polling for authentication requests, the signing of `clientData` using the securely stored private key, and the transmission of the authenticatorResponse to the polling server. Finally, **Phase 4** encompasses the polling server's receipt of the authenticatorResponse and the subsequent transmission of a response back to the client application. This phase also covers the forwarding of this response from the client application to the RP, the RP's verification of the signature, and the communication of the result back to the client application. As for the registration ceremony, details for the messages sent during the authentication ceremony are omitted in Figure 4.9 to increase readability, but an overview of each message can be seen in Table 4.5.

Table 4.5: An overview of messages sent in the authentication ceremony for the proposed solution.

Message	**Content**
RPresponse	{`challenge`, `rp_id`, `timeout`, `authenticator_id`, `credential_id`, `random_int`}
clientRequest	{`authenticator_id`, `rp_id`, `client_data`, `timeout`, `username`, `credential_id`, `random_int`}}
pollingResult	{`credential_id`, `rp_id`, `client_data`, `username`, `random_int`}}
authenticatorResponse	{`signature`, `authenticator_data`, `client_data`, `authenticator_id`, `random_int`}}
pollingResponse	{`signature`, `authenticator_data`, `client_data`, `random_int`}}
clientResponse	{`signature`, `authenticator_data`, `client_data`, `username`}

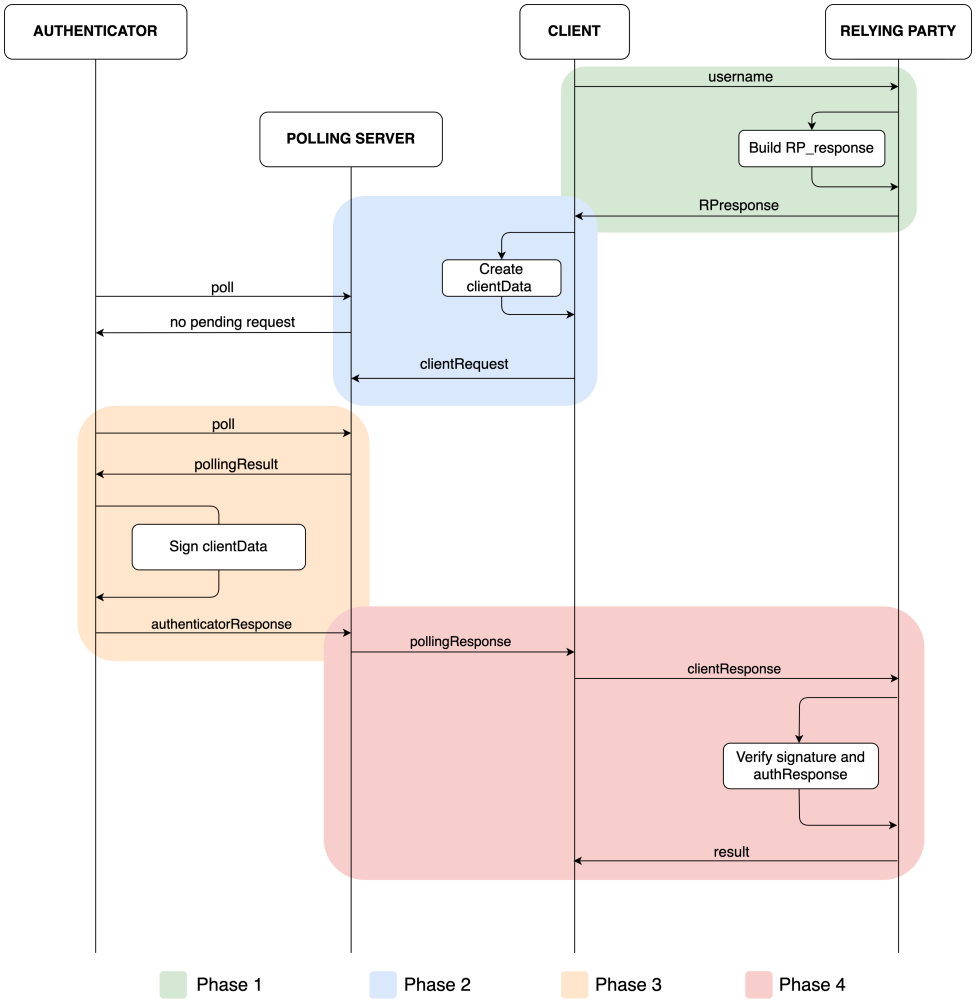


Figure 4.9: Authentication ceremony for the proposed solution.

Phase 1

Figure 4.10 presents the user interface used during authentication, which only requires the username from the user. After entering their unique username, an authentication request is sent to the RP.

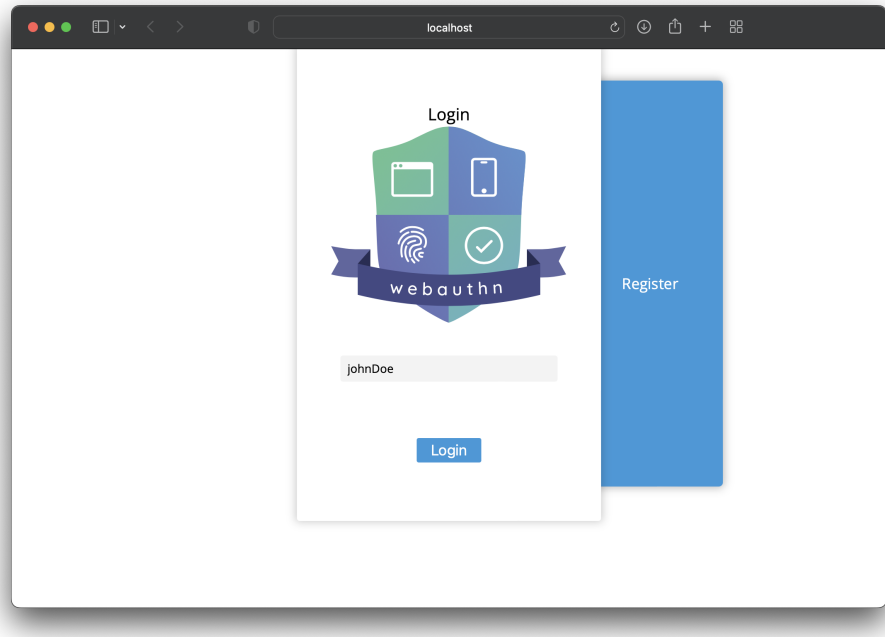


Figure 4.10: Authentication user interface.

Listing 4.16 shows the method of handling incoming authentication requests on behalf of the RP server. When the RP receives the request, it needs to check whether or not this user has previously registered to the RP and if the user is currently in the middle of another authentication attempt. If these checks hold, a response is built and returned to the client application. This starts with generating a new 64-byte random challenge by calling `Handler.getChallenge()` on line 80, which is the same method called by the RP during the registration ceremony. The method updates the state of the user with the newly generated challenge on line 81, thus being able to retrieve this when verifying the received `clientData` in the last request from the client application, denoted as "clientResponse" in Figure 4.9. The credential ID and authenticator ID stored with the user during registration is retrieved on line 82 and 83. The method then updates the state of the user to reflect an active authentication attempt before the response on lines 88-95, denoted as "RPresponse", is returned to `rpServer.py`, which forwards it to the client application. A random integer is also returned to the client application. This is used as a countermeasure against session hijacking, as the proposed solution is vulnerable to this as a consequence of the authenticator and client application communicating via HTTP. In FIDO2, the communication between the authenticator and the client application is handled by CTAP, which eliminates the possibility of session hijacking by restricting the physical distance

between the authenticator and the client application. This is not the case with the proposed solution, where the authenticator and client application can communicate from anywhere in the world via HTTP. As a result, a random integer must be passed back to the client application and relayed to the authenticator. Both the client application and authenticator can then display this integer to ensure the user that the correct client is authenticated when accepting the authentication requests in the authenticator application. This concludes phase 1 of the authentication ceremony.

```
74  # TestPlatform/rp-server/rpHandler.py
75  def handleLogin(cls, body):
...
...     # Line 76-79: Check that the user with the given username is registered and that the user is not in the
              middle of another authentication procedure.
...
80      challenge = Handler.getChallenge()
81      cls.credentials[body["username"]]["challenge"] = challenge
82      credID = cls.credentials[body["username"]]["A"]["credential_id"]
83      authID = cls.credentials[body["username"]]["authenticator_id"]
84      cls.timers[body["username"]] = Timer(cls.timeout, cls.handleTimeout, args=(body["username"], False ,))
85      cls.timers[body["username"]].start()
...
...     # Line 86-87: Updates the state of the user.
...
88      return json.dumps({
89          "rp_id":cls.RPID,
90          "challenge":challenge,
91          "credential_id":credID,
92          "timeout":cls.timeout,
93          "authenticator_id":authID,
94          "random_int": str(np.random.randint(low=1, high=100000))
95      })
```

Listing 4.16: Handle authentication request

Phase 2

Upon receipt of the RPresponse, the client application generates `clientData` in the same way as during the registration ceremony, shown in Listing 4.3. After this, it forwards a request to the polling server, denoted as "clientRequest" in Figure 4.9. The request is forwarded to the API endpoint `/client/authenticate` in the polling server.

The method `handlePOSTClientAuthenticate(cls, authenticateRequest)` handles clientRequest on behalf of `pollingServer.py`. This method runs a series of checks to verify that the user and authenticator are registered to the given RP and that the given authenticator is not in an ongoing authentication procedure. If these checks hold, a polling response, denoted as "pollingResult" in Figure 4.9, is queued on line 95-101. A timer counting down from 30 seconds, i.e., the received timeout duration, is started in the same way as for a registration attempt. Similar to registration attempts, this timer checks if the authenticator has dismissed the attempt on line 110, or if a response from the authenticator has been received on line 113. If the latter holds, a response, denoted as "pollingResponse" in Figure 4.9, is

returned to the client application, which takes place on line 116. The polling server's handling of clientRequest concludes phase 2 of the authentication ceremony.

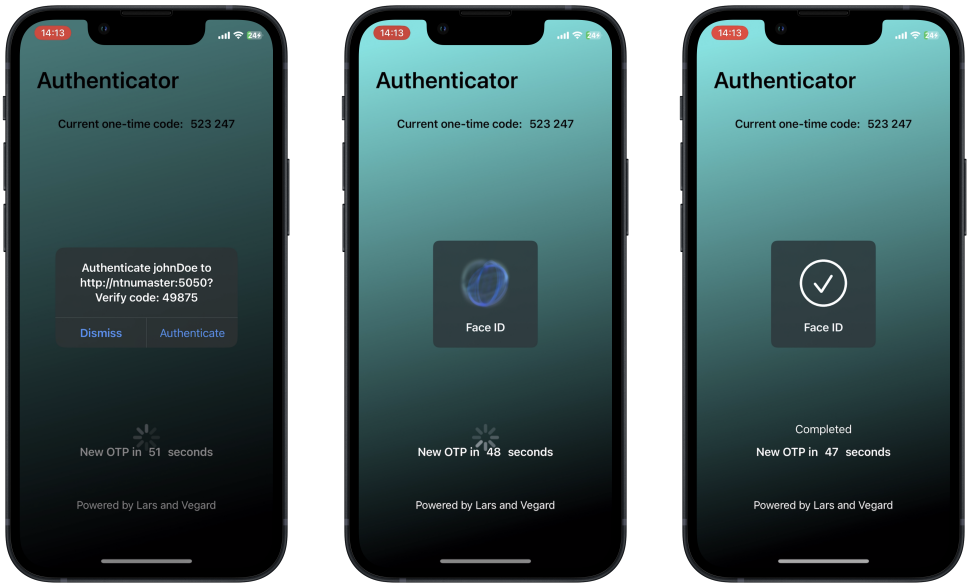
```
86  # TestPlatform/polling-server/pollingHandler.py
87  def handlePOSTClientAuthenticate(cls, authenticateRequest):
...     # Line 88-93: Checks if the authenticator is in the middle of an ongoing authentication attempt, if the
              authenticator has registered, and if the authenticator is registered to the given RP.
...
94      if len(cls.activeRequests[authenticateRequest["authenticator_id"]]["A"]) == 0:
95          cls.activeRequests[authenticateRequest["authenticator_id"]]["A"].append({
96              "credential_id": authenticateRequest["credential_id"],
97              "rp_id": authenticateRequest["rp_id"],
98              "client_data": authenticateRequest["client_data"],
99              "username": authenticateRequest["username"],
100             "random_int": authenticateRequest["random_int"]
101         })
...
...         # Line 102-103: Updating the state of the user currently in an ongoing authentication attempt.
...
104         timeout = int(authenticateRequest["timeout"])
105         waitedTime = 0
106         interval = 0.1
107         while waitedTime <= timeout:
108             waitedTime += interval
109             time.sleep(interval)
110             if cls.activeRequests[authenticateRequest["authenticator_id"]]["dismissed"]:
111                 cls.activeRequests[authenticateRequest["authenticator_id"]]["dismissed"] = False
112                 return json.dumps("Authenticator chose to dismiss the authentication attempt")
113             if authenticateRequest["authenticator_id"] in list(cls.responseToClient.keys()):
114                 response = cls.responseToClient.pop(authenticateRequest["authenticator_id"], None)
115                 cls.isActive[authenticateRequest["authenticator_id"]]["A"] = False
116                 return json.dumps(response)
...
...         # Line 117-119: Updating the state of the user if the authentication attempt timed out.
...
120         return json.dumps("Timeout")
121     return json.dumps("Pending authentication request already exists for the given authenticator")
```

Listing 4.17: Handle clientRequest.

Phase 3

Phase 3 is initiated with the authenticator polling the polling server for pending registration or authentication requests. The authenticator instructs a scheduled timer to poll the polling server every second, as explained in Section 4.2.2. As polling is done regardless of whether or not a registration or authentication request is pending, the same method in the polling server is delegated by `pollingServer.py` to handle the polling request from the authenticator, which can be viewed in Listing 4.7. If the queue mapping authenticator IDs to pending authentication attempts are filled with a pending authentication attempt, the queue mapping to pending registration attempts will always be empty, as it is not possible to receive a registration request from the client application if a user has already registered with the given authenticator. The check on line 138 in Listing 4.7 therefore holds, and an authentication request is returned to the authenticator. As mentioned in Section 4.2.2, the key values in the payload of the response from the polling server are identical when returning a registration or authentication request. One difference is whether or not a credential

ID is filled with a value or empty, where a value for the credential ID indicates an authentication request. The authenticator application will notice that the value for credential ID is non-empty, thus displaying an alert notifying the user about this, as shown in Figure 4.11a. The displayed alert also includes the received random integer. The user can then verify that the received integer is the same as the integer shown in the user interface for the client application, as shown in Figure 4.12. A user can either dismiss the authentication attempt, denoted by **Dismiss** or allow it, denoted by **Authenticate**.



(a) **(b)** **(c)**

Figure 4.11: The steps of an authentication process. (a) Verify one-time code and accept authentication attempts. (b) User verification with biometrics. (c) Authentication completed.

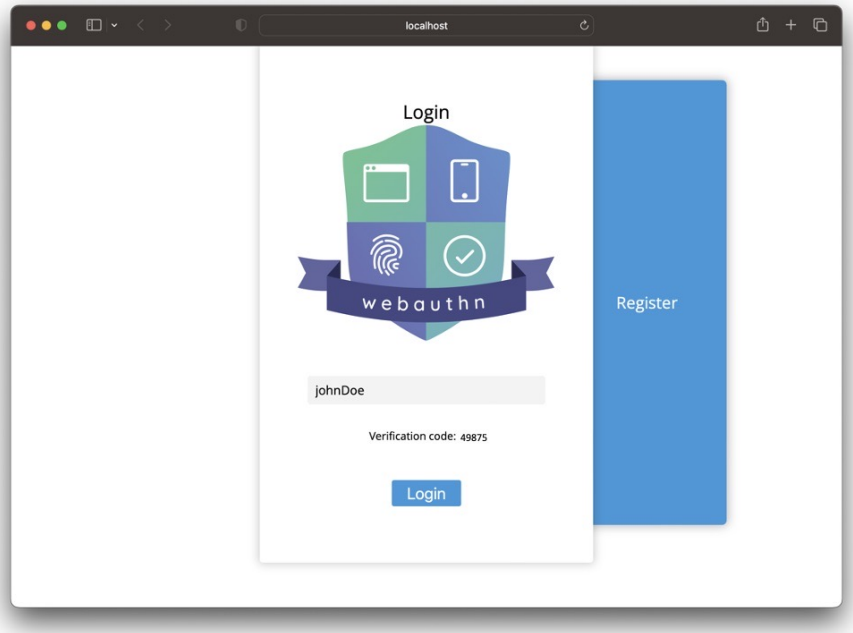


Figure 4.12: Verification code displayed in the client application when authenticating.

If a user chooses to allow and click on **Authenticate**, the `authenticatorView` will delegate `EventHandler`'s method `handleAuthentication(credential_ID: String, RP_ID: String, clientData: String, randomInt: String)`, shown in Listing 4.18, to handle the event. This method performs the most crucial task in the proposed solution, i.e., facilitates authentication by signing `clientData` with the stored private key corresponding to the public key stored by the RP during the registration ceremony. To do this, the method first has to retrieve the stored private key, by calling the method `getItem(account: String, service: String)` defined by the class `AccessKeychain` on line 124-126. This method queries the keychain for a keychain object stored with the RPID as service, and the received credential ID as account. The method also queries with the same accessibility class defined when storing the private key, presented in Section 4.2.2. As explained, this class requires user verification to retrieve the stored keychain object. This is shown in Figure 4.11b, where biometric authentication, i.e., Face ID, is required to retrieve the private key. If FaceID is not an option on the iPhone, the application will prompt the user to type in their passcode. The method returns the retrieved private key as an object of type `Data` and decodes it to the format defined by the struct `SecretKey` on lines 131-134.

```
122 // TestPlatform/Authenticator/Authenticator/Model/eventHandler.swift
123 func handleAuthentication(credential_ID: String, RP_ID: String, clientData: String, randomInt: String) {
124     guard let data = AccessKeychain.getItem(
125         account: credential_ID,
126         service: RP_ID
127     ) else {
128         print("Failed to read secret key from keychain")
129         return
130     }
131     guard let secretKey = try? JSONDecoder().decode(DilithiumLite.SecretKey.self, from: data) else {
132         print("Unable to decode secret key")
133         return
134     }
135     let sig = dilithiumLite.sign(sk: secretKey, message: clientData)
136
137     guard let authenticatorData = String(hashlib.sha256(Python.str(RP_ID).encode()).hexdigest()) else {
138         print("Unable to convert authenticatorData python hash to a SWIFT String")
139         return
140     }
...
...     // Line 141-153: Instructing CommunicateWithServer to post the authenticatorResposne to the polling server.
...
154 }
```

Listing 4.18: `EventHandler`'s method for handling authentication attempts.

The next action for the `EventHandler` is to sign the received `clientData` with the newly retrieved and decoded private key. The method `sign(sk: SecretKey, message: String)`, shown in Listing 4.19, defined by the class `DilithiumLite` is responsible for this. This method is the authors' implementation of Algorithm 4.2. The method starts with transforming the coefficients of $\mathbf{s_1}, \mathbf{s_2}$ to vectors of polynomials on lines 328 and 329, with the helper method `coeffsToPolynomial(listOfCoeffs: [[Int]])`. The matrix A is then generated on line 330 by inputting the seed ρ', denoted as `sk.Aseed`, into `expandA(seed: PythonObject)`, and the public lattice point $\mathbf{t}$ is calculated with `getLatticePoint(A: PythonObject, s: PythonObject, e: PythonObject)` on line 331. According to Algorithm 4.2, a random 512-bit random seed ρ is sampled to generate the two masking vectors $\mathbf{y_1}$ and $\mathbf{y_2}$. This seed is denoted as `rho1` and `rho2` on lines 332 and 333, and are they 32 bytes, i.e. 256 bits. These seeds are sampled with `getRandomBytes(count: Int)` shown in Listing 4.10, and their randomness is explained in Section 4.1.2. κ, denoted as `kappa`, is then defined on line 348 and ensures that a new commitment is generated in each loop by including it in the method `expandMask(seed: PythonObject, kappa: Int, noOfPoly: Int)`. Said loop starts on line 336 and continues until a valid signature is generated, i.e., the rejection sampling algorithm "accepts" the opening $(\mathbf{z_1}, \mathbf{z_2})$. The first step of this loop is to generate the two masking vectors $\mathbf{y_1}$ and $\mathbf{y_2}$ by the use of `expandMask(seed: PythonObject, kappa: Int, noOfPoly: Int)`, which in a similar manner as `expandA` and `expandS` maps a random seed to a vector of polynomials $\in [\gamma + \bar{\beta}]$. A commitment ω, denoted as `omega`, is then generated by calculating $\omega \in \{0,1\}^{384} := \mathcal{H}(\mathbf{Ay_1} + \mathbf{y_2})$ on line 340, where $\mathcal{H}$ is instantiated as SHAKE-256.

```
326 // TestPlatform/Authenticator/Authenticator/Model/SignatureAlgs/dilithiumLite.swift
327 func sign(sk: SecretKey, message: String) -> Signature {
```

```
328     let s1 = self.coeffsToPolynomial(listOfCoeffs: sk.s1Coeffs)
329     let s2 = self.coeffsToPolynomial(listOfCoeffs: sk.s2Coeffs)
330     let A = self.expandA(seed: Python.str(sk.Aseed).encode())
331     let t = self.getLatticePoint(A: A, s: s1, e: s2)
332     let rho1 = self.getRandomBytes(count: 32)
333     let rho2 = self.getRandomBytes(count: 32)
334     var kappa = 0
335     var k = 1
336     while true {
337         let y1 = self.expandMask(seed: self.np.array(rho1).tobytes(), kappa: kappa, noOfPoly: self.m)
338         let y2 = self.expandMask(seed: self.np.array(rho2).tobytes(), kappa: kappa, noOfPoly: self.n)
339         let w = self.getLatticePoint(A: A, s: y1, e: y2)
340         let omega = self.hashlib.shake_256(self.np.array(self.getCoefficients(polyList: w)).tobytes())
341         let c = self.getChallenge(A: A, t: t, omega: omega.hexdigest(48).encode(), message:
                Python.str(message).encode())
342         var z1: [PythonObject] = []
343         for i in 0..<Int(s1.size)! {
344             z1.append(self.np.polynomial.Polynomial(
345                 self.np.polynomial.polynomial.polydiv(
346                     self.np.inner(c.challengePolynomial,s1[i]).coef, self.f.coef)[1])+y1[i])
347         }
348         var z2: [PythonObject] = []
349         for i in 0..<Int(s2.size)! {
350             z2.append(self.np.polynomial.Polynomial(
351                 self.np.polynomial.polynomial.polydiv(
352                     self.np.inner(c.challengePolynomial,s2[i]).coef, self.f.coef)[1])+y2[i])
353         }
354         if rejectionSampling(z1: self.np.array(z1), z2: self.np.array(z2)) {
355             return Signature(
356                 z1Coeffs: self.getCoefficients(polyList: self.np.array(z1)),
357                 z2Coeffs: self.getCoefficients(polyList: self.np.array(z2)),
358                 cHex: c.challengeHex,
359                 omega: String(omega.hexdigest(48))!
360             )
361         }
362         k += 1
363         kappa += self.n
364     }
365 }
```

Listing 4.19: The authors' Swift implementation of the signature algorithm of the digital signature scheme.

The matrix $\mathbf{A}$, the public lattice point $\mathbf{t}$, the commitment ω, and the `clientData`, denoted as `message`, is included as input in the method `getChallenge(A: PythonObject, t: PythonObject, omega: PythonObject, message: PythonObject)`, shown in Listing 4.20, to generate a challenge, denoted as c. The output of this method is a `Challenge` struct, which defines $c' \in \{0,1\}^{384} := \mathcal{H}(\mathbf{A} \,||\mathbf{t}\, ||\omega\, ||\mu)$ from Algorithm 4.2, where c' is denoted as `challengeHex` and μ is the `clientData`, denoted as `message`. The struct also defines the challenge polynomial $c \in \mathcal{C} \subset \mathcal{R}_{q,f} :=$ `HashToBall`(c') from Algorithm 4.2, where c is denoted as `challengePolynomial`. `HashToBall` is denoted as the method `hashToBall(seed: PythonObject)`, and is the implementation of Algorithm 4.3, which maps a seed c' to a polynomial $\in \mathcal{C}$ with η coefficients $\in \{-1, 1\}$ and the rest equal to 0. The coefficients of $\mathbf{A}$ and $\mathbf{t}$, the commitment ω and the message to be signed are absorbed by SHAKE-256 on line 289-302 in Listing 4.20, before the variable `challengeHex`, i.e., c', is defined on line 303. A `Challenge` struct is returned on lines 304-307.

```
286 // TestPlatform/Authenticator/Authenticator/Model/SignatureAlgs/dilithiumLite.swift
287 func getChallenge(A: PythonObject, t: PythonObject, omega: PythonObject, message: PythonObject) -> Challenge {
288     let h = self.hashlib.shake_256()
...
...     // Line 289-302: The hash function h absorbs the coefficients of A and t, the commitment omega and the
            message to be signed, i.e. clientData.
...
303     let challengeHex = String(h.hexdigest(48))!
304     return Challenge(
305         challengeHex: challengeHex,
306         challengePolynomial: self.hashToBall(seed: h.hexdigest(48).encode())
307     )
308 }
```

Listing 4.20: Method that outputs a 384-bit hash output c' from SHAKE-256 instantiation absorbing $\mathbf{A}, \mathbf{t}, \omega$, `clientData`, as well as the polynomial c defined as `HashToBall`(c').

The signature implementation on Listing 4.19 continues with calculating the opening $(\mathbf{z_1}, \mathbf{z_2})$ with the newly generated challenge polynomial, where $\mathbf{z_1} = c \cdot \mathbf{s_1} + \mathbf{y_1}$ and $\mathbf{z_2} = c \cdot \mathbf{s_2} + \mathbf{y_2}$ on line 342-353, before checking if $\mathbf{z_1}$ or $\mathbf{z_2}$ has a coefficient $> \bar{\beta}$. This check is carried out by the method `rejectionSampling(z1: PythonObject, z2: PythonObject)`. If that is the case, the loop continues until `rejectionSampling(z1: PythonObject, z2: PythonObject)` "accepts" the opening $(\mathbf{z_1}, \mathbf{z_2})$, which upon a `Signature` struct containing the coefficients of $\mathbf{z_1}$ and $\mathbf{z_2}$, as well as the hash outputs c' and ω is returned to the `EventHandler`. The coefficients of $\mathbf{z_1}$ and $\mathbf{z_2}$ are returned instead of the polynomials for Swift to be able to encode them and transmit them to the polling server.

After a signature is generated by `sign(sk: SecretKey, message: String)` on behalf of the `EventHandler`, the authenticator data, defined in Section 2.3.3, is generated as the SHA-256 hash output of the RPID on line 137-140. The `EventHandler` instructs `CommunicateWithServer` to send the signature, authenticator data, `clientData`, random integer, and authenticator ID, denoted as "authenticatorResponse" in Figure 4.9 on line 141-153 in Listing 4.18. This concludes the actions of the authenticator during an authentication attempt, and therefore also phase 3.

Phase 4

When the polling server receives the authenticatorResponse, it first verifies that the authentication attempts have not timed out. If this is not the case, a response is added to a dictionary holding responses ready to be sent back to the client application. The response is sent when the countdown in Listing 4.17 notices that a response from the given authenticator is received. The client application will forward the response from the polling server as well as the username back to the RP upon reception of pollingResponse.

The RP has the following responsibilities when receiving the clientResponse: verify the correctness of received `clientData` and authenticator data, and verify the signature. The method shown in Listing 4.21 does this. The correctness of `clientData` can be verified by creating the expected hash output, as presented in Section 4.2.2. The expected hash output is generated on lines 105-107 in Listing 4.21 and compared on line 119. The received authenticator data is also compared with the SHA-256 hash of the RPID the RP has access to on line 119.

```
97  # TestPlatform/rp-server/rpHandler.py
98  def handleLoginResponse(cls, body):
...
...     # Line 99-104: Check that the authentication attempt has not timed out, and update the user state if this is
            not the case.
...
105     expectedHash = sha256()
106     expectedHash.update(cls.RPID.encode())
107     expectedHash.update(cls.credentials[body["username"]]["challenge"].encode())
108     pubKey = cls.credentials[body["username"]]["A"]["pubKey"]
109     pubKeyVerify = {
110         "t": Handler.coeffsToPolynomial(np.array(pubKey["t"])),
111         "Aseed": pubKey["Aseed"]
112     }
113     signature = {
114         "omega": str(body["omega"]),
115         "c": str(body["c"]),
116         "z1": Handler.coeffsToPolynomial(np.array(json.loads(body["z1"]), dtype=int)),
117         "z2": Handler.coeffsToPolynomial(np.array(json.loads(body["z2"]), dtype=int))
118     }
119     if expectedHash.hexdigest() == body["client_data"] and sha256(cls.RPID.encode()).hexdigest() ==
            body["authenticator_data"] and Handler.verifySig(pubKey=pubKeyVerify, sig=signature,
            clientData=expectedHash.hexdigest()):
120         cls.timers[body["username"]].cancel()
121         return json.dumps("Successfully logged in as "+body["username"])
122     return json.dumps({
123         "msg": "clientDataJSON, authData or signature failed!",
124         "reason": "cryptoVerificationFailure"})
```

Listing 4.21: RP: Handle authentication request

Lastly, the received signature needs to be verified. To do this, the public key associated with the user has to be retrieved. The retrieved public key component $\mathbf{t}$ consists of the coefficients of the polynomials that comprise $\mathbf{t}$. Before verifying the signature, $\mathbf{t}$ has to be transformed into a vector of polynomials based on its coefficients, which is done on line 110. The same goes for $\mathbf{z_1}$ and $\mathbf{z_2}$ in the received signature, which happens on lines 116 and 117. The signature is now ready to be verified by the method `verifySig(cls, pubKey, sig, clientData)`, which is the authors' implementation of Algorithm 4.4.

The method `verifySig(cls, pubKey, sig, clientData)` can be seen in Listing 4.22. The RP also need an implementation of `ExpandA` from Algorithm 4.5, as the stored public key only includes ρ'. The matrix $\mathbf{A} \in \mathcal{R}_{q,f}^{n \times m} := \texttt{ExpandA}(\rho')$ is generated on line 286 by the method `expandA(cls, seed)`, which maps the seed ρ' to a matrix $\in \mathcal{R}_{q,f}^{n \times m}$. To verify the signature, the RP needs to perform three checks. It has to check if the received challenge hash output c is correct by computing its own and comparing

it to the received one. It has to make sure that both $\mathbf{z_1}$ and $\mathbf{z_2}$ consist of short-norm polynomials, i.e., $\leq \bar{\beta}$. Lastly, it has to verify the correctness of the signature by making sure the received commitment ω equals $\mathcal{H}(\mathbf{Az_1} + \mathbf{z_2} - c\mathbf{t}) \in \{0,1\}^{384}$. The method starts with RP generating c'', denoted as `expectedHash.hexdigest(48)`, which happens on line 298-302 and compares it to the received challenge $c' \in \{0,1\}^{384}$, which happens on line 303. If this comparison holds, a challenge polynomial $c \in \mathcal{C}$, denoted as `cPoly`, is generated on line 305 with the use of RPs own implementation of `HashToBall`. The RP then calculates $\omega' := \mathcal{H}(\mathbf{Az_1} + \mathbf{z_2} - c\mathbf{t})$ on line 306-308 and compares it to the received commitment ω on line 309. Lastly, the RP concatenates the coefficients of $\mathbf{z_1}$ and $\mathbf{z_2}$ into a single list and checks if the absolute value of any of them is larger than $\bar{\beta}$, denoted by `cls.approxBeta`. If these three checks hold, `verifySig(cls, pubKey, sig, clientData)` will output `true`, indicating that this is a valid signature on `clientData` with the private key corresponding to the public key received when registering. The method `handleLoginResponse(cls, body)` in Listing 4.21 will then return the result to be sent back to the client application, which indicates successful authentication for the user. This concludes phase 4 and the authentication ceremony.

```
284 # TestPlatform/rp-server/rpHandler.py
285 def verifySig(cls, pubKey, sig, clientData):
286     A = cls.expandA(pubKey["Aseed"].encode())
...
...     # Line 287-297: Assigning the components of the signature and public lattice point t to variables, as well as
            extracting the coefficients of the polynomials in A and storing them in the list ACoeffs.
...
298     expectedHash = shake_256()
299     expectedHash.update(np.array(ACoeffs).tobytes())
300     expectedHash.update(np.array(Handler.polynomialToCoeffs(t)).tobytes())
301     expectedHash.update(omega.encode())
302     expectedHash.update(clientData.encode())
303     if expectedHash.hexdigest(48) != c:
304         return False
305     cPoly = cls.hashToBall(expectedHash.hexdigest(48).encode())
306     ct = np.array([cPoly*p for p in t])
307     omegaprime = np.inner(A, z1)+z2-ct
308     omegaprime = np.array([Polynomial((p % cls.f).coef % cls.q) for p in omegaprime])
309     if not shake_256(np.array(Handler.polynomialToCoeffs(omegaprime), dtype=int).tobytes()).hexdigest(48) ==
            omega:
310         return False
311     concatenatedList = np.array(Handler.polynomialToCoeffs(z1) + Handler.polynomialToCoeffs(z2)).flatten()
312     if np.any(np.absolute(concatenatedList) > cls.approxBeta):
313         return False
314     return True
```

Listing 4.22: Implementation of verification algorithm.

4.2.4 Process View

Figure 4.13 presents a high-level complete process view, which includes all possible actions and subsequent reactions in the proposed solution. The figure focuses on the run-time behavior of the system, the processes within the system, and how components communicate.

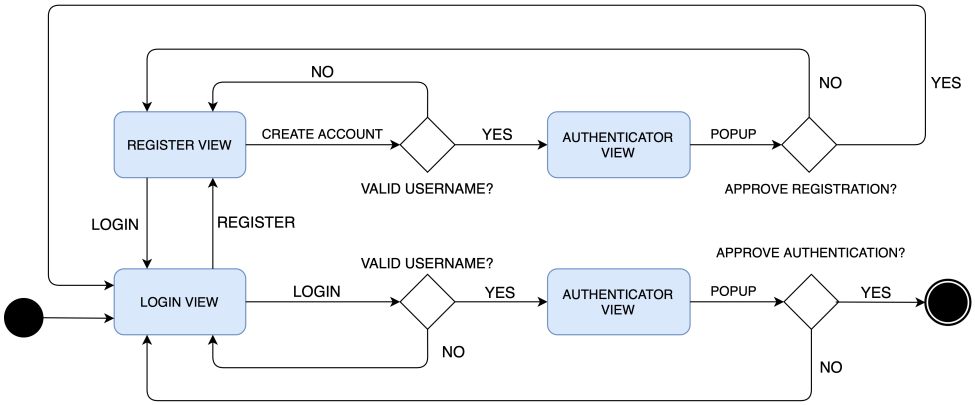


Figure 4.13: Process View for the proposed solution.

Chapter 5
Performance and Discussion

This chapter presents the performance of the proposed solution introduced in Chapter 4. The performance is evaluated against a set of metrics, including key and signature sizes, as well as efficiency. The proposed solution will be compared to similar existing state-of-the-art solutions on the measured metrics. The functional and quality requirements from Section 5.1.5 will be revisited and reviewed. Lastly, a discussion on how this master's thesis answers the research questions defined in Section 1.4 is carried out.

5.1 Performance

The evaluation of the proposed solution focuses on the performance of the implemented digital signature scheme from Section 2.2.4. The authors have tested how different values for the dimension (n, m) affect the key and signature sizes of the scheme, as well as the performance of the implemented digital signature algorithms from Section 4.1.2. (n, m) was chosen because it directly affects the performance of the scheme by governing the number of operations performed when multiplying and adding matrices and vectors of polynomials. It was also chosen since it affects the hardness of the underlying hard problems, i.e., LWE and SIS, thus affecting the overall security of the scheme. Different values for β were also tested when evaluating the performance of the signature algorithm, to identify any dependencies between β, i.e., the secret $(\mathbf{s_1}, \mathbf{s_2})$, and the run-time of the signature algorithm.

The following presents the results from each test and discusses them in light of the presented background and related work from Chapter 2. The digital signature implemented by the authors in the proposed solution will be compared with the state of the art within lattice-based zero-knowledge digital signatures, i.e., Dilithium, which was presented in Section 2.4.2. Table 5.1 and 5.2 shows a comparison of key and signature size in bytes, and the performance respectively, between Dilithium2, an instance of Dilithium with equivalent security, i.e., 128-bit quantum security, and

the digital signature implemented in the proposed solution. The data for Dilithium2 is gathered from Table 2.1 and 2.2 in Section 2.4.2, while the data for the digital signature implemented in the proposed solution is gathered from Table A.1, A.2, A.4, A.3, and A.6 in Appendix A, which contains the test results. In Table 5.2, the key generation algorithm is denoted as **KeyGen**, the signature algorithm is denoted as **Sign**, and the verification algorithm is denoted as **Verify**. The authors have decided that the measurements of performance should include the average time usage for all three algorithms. The average number of attempts before a valid signature is produced will also be measured for the signature algorithm. The key generation and signature algorithm were tested in a Swift program executed on an iPhone XR with an A12 Bionic chip, as these algorithms are to be executed by the authenticator during registration and authentication. The verification algorithm was tested with a Python script running on a Macbook Pro 2017 with a 2.3 GHz dual-core Intel Core i5.

Table 5.1: A comparison of key and signature sizes, in bytes, between the implemented digital signature and Dilithium2.

	Private key	**Public key**	**Signature**
Dilithium2	2544	1312	2440
Proposed solution	5645	10102	15179

Table 5.2: A comparison of performance between the implemented digital signature and Dilithium2.

	KeyGen (ms)	**Sign**		**Verify (ms)**
		Attempts	**Time (ms)**	
Dilithium2	2.04	4.25	11.9	2.21
Proposed solution	349.63	3.12	431.17	19.36

5.1.1 Key and Signature Size

Key and signature sizes of the scheme were evaluated in the test environment presented in Chapter 4. The key and signature sizes have therefore been evaluated by a Swift program added to the authenticator, which instructed the class `DilithiumLite` to generate new key pairs and sign a standardized message of type `String`. The test program can be viewed in Listing A.1 in Appendix A. The authors implemented two additional methods in `DilithiumLite` to encode `PublicKey` and `Signature` structs as objects of type `Data`, similar to `getSecretKeyAsData(secretKey: SecretKey)`. After encoding key pairs and signatures as objects of type `Data`, their sizes in bytes can be accessed through `Data.count`. Table 5.3 shows the size of private keys, public keys, and signatures with $(n, m) = \{(4, 3), (5, 4), (6, 5)\}$.

Table 5.3: Sizes in bytes for the private key, public key, and signature for the implemented digital signature.

$(\mathbf{n}, \mathbf{m})$	**Private Key Size**	**Public Key Size**	**Signature Size**
$(4, 3)$	4405	8090	11471
$(5, 4)$	5645	10102	15179
$(6, 5)$	6870	12098	18899

Table 5.3 illustrates a rather obvious fact for the keys and signature, their size increases linearly as (n, m) increase, as shown in Figure 5.1. The data used to generate this plot can be viewed in Table A.1 in Appendix A. The private key implemented in the proposed solution consists of two lists, `s1Coeffs` and `s2Coeffs`, containing the coefficients for the polynomials that comprise $\mathbf{s_1}$ and $\mathbf{s_2}$, as well as a 256-bit seed ρ', which is used to generate the matrix $\mathbf{A}$. The size of ρ' will remain unchanged independent of (n, m), while the total number of lists of coefficients will be $n + m$. The linear increase in private key size as a function of (n, m) is the result of the increase in the number of lists of coefficients for $\mathbf{s_1}$ and $\mathbf{s_2}$.

The public key used in the proposed solution consists of a list, `tCoeffs`, containing n lists of d coefficients for the polynomials that comprise $\mathbf{t} = \mathbf{As_1} + \mathbf{s_2}$, as well as the seed ρ', used to generate the matrix $\mathbf{A}$. The number of lists in `tCoeffs` will increase as n increases, while ρ' will remain unchanged independent of (n, m). As a result, the public key size increase linearly as (n, m) increase.

The signature implemented in the proposed solution consists of two lists `z1Coeffs` and `z2Coeffs`, as well as two hash outputs `cHex` and `omega`. The two lists are the list of coefficients for the polynomials that comprise $\mathbf{z_1}$ and $\mathbf{z_2}$ respectively. The two hash outputs are the 384-bit hash digest c' and the 384-bit commitment ω, presented in Algorithm 4.2. The hash digest c' and commitment ω remain unchanged independent of (n, m), while `z1Coeffs` and `z2Coeffs` increase linearly as a function of (n, m). The total number of lists of coefficients in a signature is $n + m$.

One should notice from Table 5.3 and Figure 5.1 that even though the private key contains m more lists of coefficients than the public key, the size of the public key is ≈1.8 times larger than the private key. This indicates that the size of the coefficients plays an important role when encoding them, as $\beta << q$. The same goes for the size of the signature, which contains $m + n$ list of relatively large coefficients, i.e., $\beta << \bar{\beta}$. The signature is ≈1.5 times larger than the public key and ≈2.7 times larger than the private key.

Table 5.1 shows a comparison of key and signature size between the implemented digital signature and Dilithium2. From the table, it can be read that for Dilithium2,

the private key is the largest, ≈1.9 larger than the public key and slightly larger than the signature. This is due to techniques explained in [Lyu20] and [DKL+18] that reduce the size of the public key and the proof of the underlying Σ-protocol. As a result, the private key for the digital signature implemented in the proposed solution is ≈2.22 larger than the private key used in Dilithium2. As for the public key and signature implemented in the proposed solution, their size is substantially larger than the public key and signature used in Dilithium2, ≈7.7 and ≈6.3 times larger respectively. This is expected as the techniques presented in [Lyu20] and [DKL+18] were not part of the digital signature scheme implemented in the proposed solution.

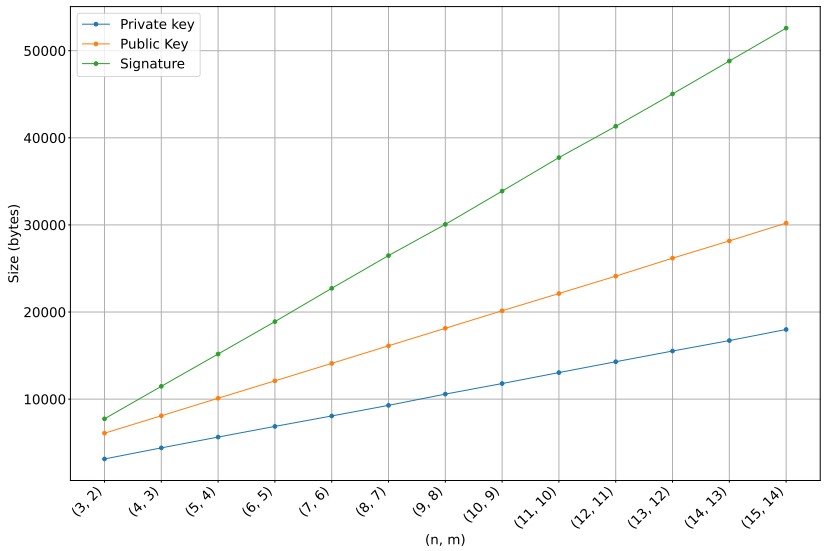


Figure 5.1: The size of the private key, public key, and signature with $\beta = 5$ and $(n, m) = ((3, 2), \ldots, (15, 14))$.

5.1.2 Key Generation

The key generation algorithm was evaluated in a similar manner as for key and signature sizes of the digital signature scheme, i.e., a Swift program was added to the authenticator iOS application. The program instructed `DilithiumLite` to generate key pairs via the method `generateKeyPair()` while varying (n, m) and β. The time spent on executing the method was measured. The program can be viewed in Listing A.4 in Appendix A. 100 key pairs were generated for each value of (n, m, β) to get an average with minimized variance. Time measurements were implemented with `DispatchTime.now().uptimeNanoseconds` [23b] [23h], an instance property returning the number of nanoseconds since system boot. Figure 5.2 shows a plot of the average

time spent, in milliseconds, to generate keys with $(n, m) = ((3, 2), (4, 3), \ldots, (15, 14))$ for $\beta = 5$. The data set used to generate the plot can be viewed in Table A.2 in Appendix A.

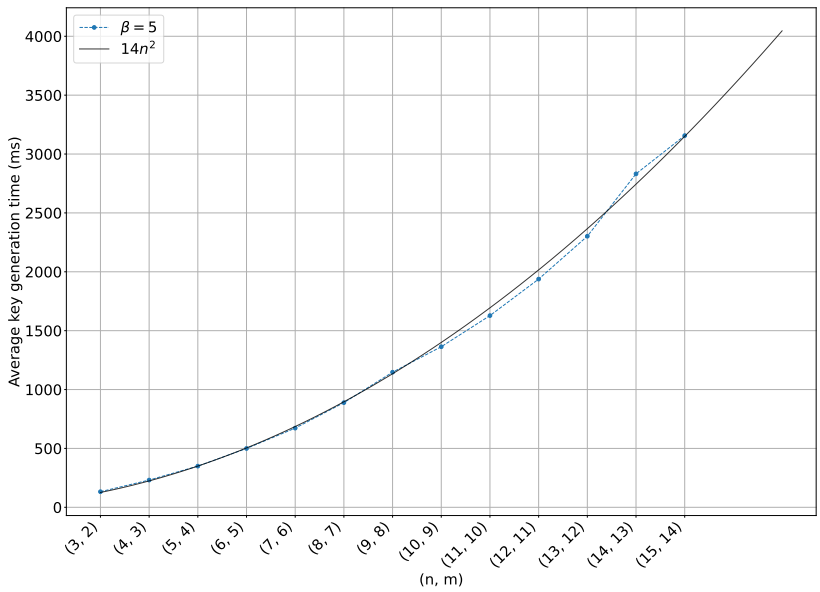


Figure 5.2: Average time in milliseconds needed to generate key pairs.

From each plot, it can be observed that the average time needed to generate a new key pair tends to grow polynomially as (n, m) increases. As presented in Algorithm 4.1, key generation includes sampling a seed ζ to further generate the two seeds ρ, ρ', which are used to generate the secret $(\mathbf{s_1}, \mathbf{s_2})$ and the public matrix $\mathbf{A}$ respectively. The last step is to calculate the public lattice point $\mathbf{t} = \mathbf{As_1} + \mathbf{s_2}$, which is an expensive operation in the key generation algorithm, especially as NTT was not implemented by the authors. Acquiring the asymptotic run-time complexity of the key generation algorithm involves inspecting the operations executed. In all tests carried out by the authors, (n, m) is such that $m = n - 1$. The run-time complexity of generating the secret $(\mathbf{s_1}, \mathbf{s_2})$ from ζ is $O(n + m) = O(n + (n - 1)) = O(2n)$, while the run-time complexity of sampling the matrix $\mathbf{A}$ is $O(n{\cdot}m) = O(n{\cdot}(n{-}1)) = O(n^2)$. The last step, calculating the public lattice point, requires $n \cdot m$ polynomial multiplications and $n \cdot m$ polynomial additions. This is due to $\mathbf{A} \cdot \mathbf{s_1}$ involving $n \cdot m$ polynomial multiplications and $n(m-1)$ polynomial additions, while adding $\mathbf{As_1}$ with $\mathbf{s_2}$ involves n additions. A total amount of $n{\cdot}m$ polynomial multiplications and $n(m{-}1){+}n = n{\cdot}m{-}n{+}n = n{\cdot}m$ polynomial additions are needed to calculate $\mathbf{t}$, resulting in a run-time complexity of

$O(n^2)$. An upper bound for the run-time complexity of the key generation algorithm is thus $O(2n)+O(n^2)+O(n^2) = O(2n^2+2n) = O(n^2)$. The plot for $14 \cdot n^2$ is plotted in Figure 5.2, and shows that the time usage grows polynomially.

Table 5.2 shows a comparison of performance between Dilithium2 and the implemented digital signature. The average time usage of the key generation algorithm for Dilithium2 is 2.04 ms, while the average time usage for the implemented digital signature is 349.63 ms, which is more than for Dilithium2 by a factor of ≈171. One reason for this difference is the authors' inefficient implementation of the functions `expandS` and `expandA`, which are used to generate the secret $(\mathbf{s_1}, \mathbf{s_2})$ and $\mathbf{A}$ respectively. In the implemented solution, coefficients are generated sequentially by interpreting bytes from a SHAKE instance as integers. The authors of Dilithium have implemented this in a vectorized form, allowing multiple coefficients to be generated in parallel [DKL+18]. In Dilithium specification 3.1 [DKL+21], the authors have opted to use Advanced Encryption Standard (AES) in `expandS` and `expandA`, allowing already made hardware implementations of AES to be utilized for an additional speed-up. Another reason for the difference in performance is the fact that the digital signature in the proposed solution was implemented in Swift, which is a high-level language. As a high-level language, developers are not granted fine-grained access to hardware, as in low-level programming languages such as C or Assembly. Low-level programming languages grant developers direct access to hardware resources, allowing them to take advantage of low-level optimization which high-level languages such as Swift simply cannot do. The performance metrics presented in Table 5.2 are for an optimized implementation of Dilitihum written in C [pq-17].

The authors have not considered the time usage for key generation as a large problem, as the use case for the implemented digital signature is to facilitate passwordless authentication. A time variance in the range of a couple of hundred milliseconds is not noticeable for the end user, especially when the key generation algorithm is only executed for a user when registering to a service.

5.1.3 Signing

The evaluation of the signature algorithm was carried out in a similar manner as for the key generation algorithm. The test aimed to reflect the real-life performance of the signature algorithm when authenticating to a service with the authenticator, and a Swift program was therefore added to the iOS application. The program instructed `dilithiumLite` to generate a key pair and sign a message. The time used to sign the message with the newly created private key was then measured in a similar manner as for the key generation algorithm. The number of attempts needed to produce a valid signature was also measured, as the time usage for the signature algorithm heavily depends on the number of attempts. The parameters (n, m) and β were

changed, and 50 tests were executed for each combination of (β, n, m) to minimize the variance of the estimated average. The authors chose to *only* execute 50 tests, as the tests became time-consuming when (n, m) approached $(8, 7)$. The program implemented to test the signature algorithm is presented in Listing A.3 in Appendix A.

Figure 5.3 and 5.4 shows plots for the average number of signature attempts needed until a valid signature was generated, and the average time used, in milliseconds, for the signature algorithm respectively. The plots in Figure 5.3 were generated with the data set from Table A.4 in Appendix A, while the plots in Figure 5.4 were generated with the data set from Table A.3 in Appendix A. Both figures show the average number of attempts and the average time usage for $(n, m) = ((3, 2), \ldots, (15, 14))$, and both figures include 5 plots each for $\beta = (3, 4, 5, 6, 7)$.

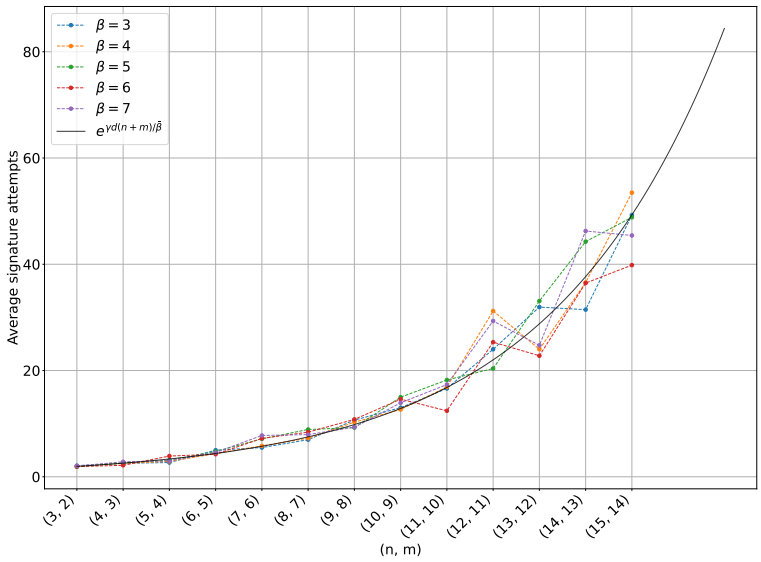


Figure 5.3: Average signature attempts.

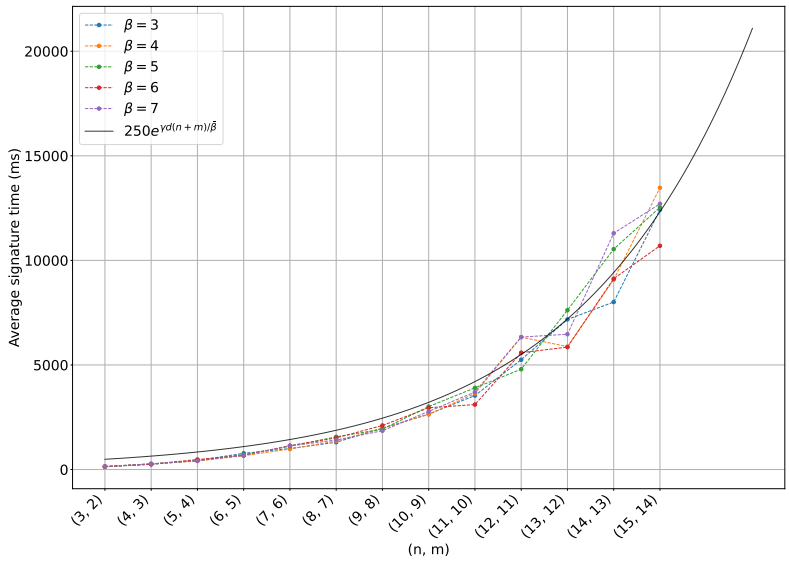


Figure 5.4: Average signature time usage (ms).

Figures 5.3 and 5.4 illustrate how the time usage for the signature algorithm is dominated by the number of attempts needed before a valid signature is generated. Equation 2.5 shows the probability for a signature to be accepted by the rejection sampling algorithm. As shown in [Lyu20], the probability of accepting a signature can be approximated in the following way:

$$\left(\frac{2\bar{\beta}+1}{2(\bar{\beta}+\gamma)+1}\right)^{d(m+n)} > \left(\frac{\bar{\beta}}{\bar{\beta}+\gamma}\right) = \left(1+\frac{\gamma}{\bar{\beta}}\right)^{-d(n+m)} \approx e^{-\frac{\gamma d(n+m)}{\bar{\beta}}} \tag{5.1}$$

The implemented signature procedure resembles “Bernoulli trials”, as mentioned in Section 2.2.4. X is defined as the number of signature attempts until a valid signature is produced, including the attempt to produce the valid signature. Equation 2.6 shows the expected number of attempts before a valid signature is produced. Substituting the probability for success with the approximate probability of success given in Equation 5.1 yields an approximated expected number of attempts.

$$\mathbb{E}[X] = \frac{1}{\Pr\limits_{\mathbf{y_1},\mathbf{y_2}}[(\mathbf{z_1},\mathbf{z_2}) \neq \perp]}$$
$$\mathbb{E}[X] = \frac{1}{\left(\frac{2\bar{\beta}+1}{2(\bar{\beta}+\gamma)+1}\right)^{d(m+n)}}$$
$$\mathbb{E}[X] \approx \frac{1}{e^{-\frac{\gamma d(n+m)}{\bar{\beta}}}} = e^{\frac{\gamma d(n+m)}{\bar{\beta}}} \tag{5.2}$$

The number of signature attempts grows exponentially following Equation 5.2 when (n, m) increases. This is illustrated in Figure 5.3, where $e^{\gamma d(n+m)/\bar{\beta}}$ is plotted alongside the other plots. In this specific plot, the authors chose the same values for the parameters as presented in Section 4.1.2, i.e., $\gamma = 275, d = 256, \bar{\beta} = \frac{q-1}{16}$, where $q = 8380417$.

As the time usage for the signature procedure is heavily dominated by the number of signature attempts, time usage also grows exponentially following $c \cdot e^{\gamma d(n+m)/\bar{\beta}}, c \in \mathbb{R}^+$. This is illustrated in Figure 5.4, where $250e^{\gamma d(n+m)/\bar{\beta}}$ is plotted alongside the other plots.

Equation 5.2, which governs the number of signature attempts, and thus also the time usage for the signature procedure, does not depend on the value of β. The plots in Figures 5.3 and 5.4 illustrate this, as no significant patterns emerge for different β values. As mentioned in Section 2.2.3, this is an important property as any dependency between the secret $(\mathbf{s_1}, \mathbf{s_2})$ and the run-time of the algorithm would lead to the possibility of side-channel attacks. This is further illustrated in Figure 5.5, which includes a scatter plot for the average number of attempts needed to generate a valid signature for $\beta = (1, 2, \ldots, 20)$, as well as a linear graph generated by applying linear regression on the test results. To produce this, the authors chose $(n, m) = (5, 4)$ and generated 100 signatures for each value of β. Listing A.2 in Appendix A shows the test program implemented on the authenticator, while Table A.5 in Appendix A shows the test results used to generate the plots in Figure 5.5. The linear graph emerging from applying regression is $-0.0037 \cdot \beta + 3.39$. A slope as close to 0 as -0.0037 indicates that no dependency between the value of β and the number of attempts exists. Repeating the test $N = \infty$ times would result in the graph $0 \cdot \beta + \mathbb{E}[X]$ after applying linear regression, where X is the number of attempts before a valid signature is produced.

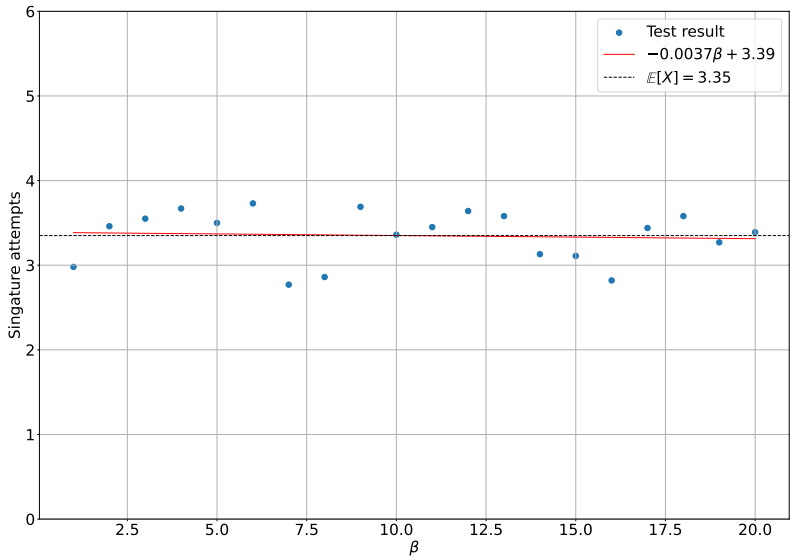


Figure 5.5: Scatter plot and linear regression graph for the average number of attempts needed before an accepted signature is produced for $\beta = (1, 2, \ldots, 20)$.

Table 5.2 shows that the Dilithium instance with a similar security level, i.e., Dilithium2, needs an average of 4.25 attempts to produce a valid signature. The expected number of attempts for the implemented scheme can be calculated by substituting the (n, m) in Equation 5.2 with $(5, 4)$, resulting in $\mathbb{E}[X] \approx 3.35$. Inspecting Table 5.2 shows that the average number of attempts achieved from the test of the implemented algorithm, i.e., $(n, m) = (5, 4), \beta = 5$, is 3.12. This is the only test parameter where the implemented scheme outperforms Dilithium. The average number of attempts for Dilithium is the result of an increase in efficiency by reducing the size of the public key and signature [DKL+18]. This is also presented in [Lyu20] and compared to the implemented scheme. The slight increase in the number of attempts needed to produce a valid signature is compensated by the decreased time usage for the signature algorithm, which for Dilithium2 is 11.9 ms, as shown in Table 5.2. This is the most time-consuming procedure for Dilithium, similar to the digital signature implemented in the proposed solution, which on average uses 431.17 ms to produce a signature. This is larger than the average time usage for Dilithium2 by a factor of ≈36. The main reason for this difference is the difference in implementation language and Dilithium's focus on low-level optimizations by implementing the scheme in C [pq-17]. Another reason for the time difference is the lack of NTT in the authors' implementation. This affects the performance of

polynomial multiplication when regenerating the public lattice point $\mathbf{t} = \mathbf{As_1} + \mathbf{s_2}$, when generating the commitment $\mathcal{H}(\mathbf{Ay_1} + \mathbf{y_2})$, and when generating the opening $(\mathbf{z_1} = c\mathbf{s_1} + \mathbf{y_1}, \mathbf{z_2} = c\mathbf{s_2} + \mathbf{y_2})$. The last reason is the inefficient implementation of `expandMask` compared with Dilithium's implementation, which utilizes a vectorized implementation of `expandMask` to speed up the generation of $(\mathbf{y_1}, \mathbf{y_2})$.

The authors have deemed that the signature algorithm performs at a satisfactory level, given the use case for the digital signature scheme. The digital signature is implemented in a proposed solution to facilitate authentication, thus operating in an environment not sensitive to time variances in the range of a couple of hundred milliseconds.

5.1.4 Verification

The authors chose to evaluate the performance of the verification algorithm by testing it in a similar environment as presented in Chapter 4. The test was implemented as a Python script executing the verification algorithm implemented on the RP server. The test involved measuring the time usage of the implemented verification algorithm, which can be viewed in Listing A.5 in Appendix A. 100 tests were executed for each value of (n, m) to get an estimated average value of time usage with minimized variance. The time usage for the verification algorithm was then measured via the Python library `time`, introduced in Section 3.2.1, and the method `time.time_ns()` was used to output the number of nanoseconds used to execute the verification algorithm. The test results can be viewed in Table A.6 in Appendix A, and Figure 5.6 illustrates how the time usage grows as a function of (n, m).

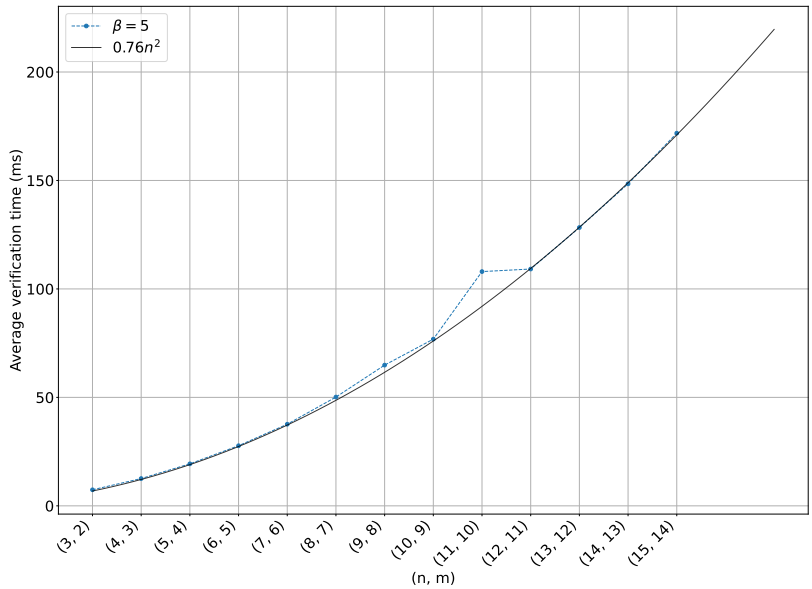


Figure 5.6: Average verification time (ms)

Figure 5.6 indicates that the average time usage for executing the verification algorithm grows polynomially as (n, m) increases. The steps involved with verification can be observed in Algorithm 4.4. The computation of $\omega' = \mathcal{H}(\mathbf{Az_1} + \mathbf{z_2} - c\mathbf{t})$ and the regeneration of the public matrix $\mathbf{A}$ via `expandA` are the steps that depend on (n, m). The run-time complexity for regenerating $\mathbf{A}$ with `expandA` is $O(n \cdot m)$, but since the value of m is such that $m = n - 1$, a run-time complexity of $O(n^2)$ is obtained. Computing ω' requires $n \cdot m + n$ polynomial multiplications and $n(m-1) + 2n$ polynomial additions. As the value of m is such that $m = n - 1$, the resulting run time complexity is $O(n^2 + n + n^2 + 2n) = O(n^2)$. This results in a total run-time complexity of $O(n^2)$ for the verification algorithm. This is illustrated in Figure 5.6, where $0.76n^2$ is plotted along with the average time usage.

Table 5.2 shows a comparison of Dilithium2 and the implemented digital signature in terms of verification performance. The time usage for Dilithium2 is 2.21 ms compared to 19.35 ms in the authors' implementation. That is a difference by a factor of $\approx$8.76. Dilithium has a more efficient implementation of `expandA`, which results in a faster generation of $\mathbf{A}$. Operations involving polynomial multiplication are more efficient due to Dilithium's optimized implementation of NTT, whereas the authors' implementation on the other hand is relatively computationally expensive as it lacks an implementation of NTT. Lastly, the creators of Dilithium have focused on

an implementation in a low-level language, which increases the overall performance of the scheme.

5.1.5 Requirements

In Chapter 4, a set of functional and quality requirements were presented. The goal of these was to act as guidelines during development and set clear goals for the proposed solution. Now that the proposed solution is created, a review of these requirements will be carried out.

Table 4.1 lists four functional requirements the system should possess. These requirements relate to the behavior of the system and actions a user should be able to complete. A *test report* is created for the functional requirements, which presents the results, time usage, and additional comments regarding the test. This is located in Appendix B. FR1 and FR2 relate directly to actions the user should be able to perform. Testing these two requirements involve registering a new user, and authenticating this user without the use of passwords. User testing revealed that the user interface is not as intuitive as desired, but the core functionality is present in the system. Both registration and authentication are carried out without the use of passwords with the help of the authenticator iOS application. The last two functional requirements, FR3 and FR4, encompass desired system behavior. Such logic is handled by the RP server, which has access to the user database. These requirements were tested as described in Appendix B.

The three quality requirements, listed in Table 4.2, relate to system characteristics. These are not testable in the same manner as the functional requirements, but a review of them will be carried out nonetheless. QR1 relates to the stored user data, which in the case of the proposed solution is stored in a local MongoDB instance. By default, this data is not encrypted, but MongoDB does provide the opportunity to do so with an enterprise version of its platform. The information stored by the RP is username, hashed authenticator ID, credential ID, and public key for registered users, thus not storing any sensitive information such as full names, passwords, or any other identifiable information. It should be mentioned that the research scope for this thesis focuses on implementing a lattice-based zero-knowledge digital signature and evaluating it in a passwordless authentication system, not the encryption of stored data. Anyway, secure storage of any user data is recommended.

The proposed solution does handle multiple concurrent users, but to what degree the performance is affected by this is not measured. Thus, QR2 is partly achieved. However, deploying the proposed solution in the real world requires additional modifications to the RP and polling server to handle scalability, as this was not a design goal defined by the authors. This is covered more in future work presented in Section 6.1. The last quality requirement, QR3, states that the system should

provide fast and seamless authentication for the user. The proposed solution is more time-consuming than traditional authentication with the use of usernames and passwords, but it provides authentication in a similar time frame compared to authentication methods that involve authenticators, e.g. MFA systems. The action carried out by the authenticator during authentication, i.e., signing `clientData`, is done in less than 0.5 seconds. The additional time usage stems from propagating messages between the components in the system, as well as waiting for user input. Anyway, the whole authentication sequence is completed within the limit set by the RP.

5.2 Research Questions

At the start of this thesis, three research questions were defined, along with four objectives. A review of how this thesis answered the proposed research questions is presented below.

5.2.1 Research Question 1

How can instances of *hard* problems within lattice cryptography be used to construct a zero-knowledge protocol?

By following the work of [Lyu20] and [DKL+18], the authors were able to utilize two instances of hard problems within lattice cryptography to construct a Σ-protocol that is zero-knowledge. The instances are SIS and LWE, which together provided the authors with the following one-way function

$$\mathbf{As_1} + \mathbf{s_2} = \mathbf{t} \tag{5.3}$$

where $(\mathbf{s_1}, \mathbf{s_2})$ served as private information and $(\mathbf{A}, \mathbf{t})$ served as public information. The one-way function was employed in a protocol similar to the interactive Schnorr protocol, presented in Section 2.1.6, enabling a prover to prove knowledge of a relaxed solution to Equation 5.3, i.e., proving knowledge of $(\bar{\mathbf{s_1}}, \bar{\mathbf{s_2}})$ in a somewhat larger interval than β [Lyu20]. The protocol differs from interactive Schnorr by introducing rejection sampling. This technique is vital in the protocol as it ensures the opening $(\mathbf{z_1}, \mathbf{z_2})$ does not leak any information about the secret $(\mathbf{s_1}, \mathbf{s_2})$, as explained in Paragraph 2.2.3.

The interactive Σ-protocol served as a zero-knowledge proof as it satisfied all three properties, correctness, soundness, and zero-knowledge, as explained in Section 2.2.3. With this, research question 1 was answered through the realization of research objective 1.

5.2.2 Research Question 2

How can such a protocol enable passwordless authentication?

To answer this question, research objectives 2 and 3 had to be achieved. Research objective 2 pursued the creation of "a test environment which implements passwordless authentication". Section 2.3 presents the FIDO2 standard, which defines multiple protocol specifications for enabling passwordless authentication. Chapter 4 presents the development and implementation of a proposed solution inspired by the overall architecture and message exchange presented in FIDO2. The proposed solution comprised four components: an RP server, a client application, an authenticator, and a polling server, which worked together in order to authenticate a user, without the need for a password.

Research objective 3 pursued the implementation of "the constructed quantum-resistant zero-knowledge protocol in said test environment". The proposed solution resembling FIDO2 laid the groundwork for passwordless authentication by introducing secure and convenient authentication methods. However, to fully enable passwordless authentication, the implementation of a digital signature was necessary, to ensure the integrity and non-repudiation of user credentials. Section 2.2.3 presented a lattice-based interactive zero-knowledge protocol designed for a prover to prove knowledge of a secret $(\mathbf{s_1}, \mathbf{s_2})$. However, for the protocol to enable passwordless authentication, it needed to be transformed into a digital signature scheme. Section 2.1.4 presents the Fiat-Shamir transform, which is a technique used to transform an interactive proof of knowledge into a non-interactive one, by replacing step 2 in an interactive Σ-protocol with the prover instead querying a random oracle. Section 2.2.4 presents a combination of rejection sampling and the Fiat-Shamir transform called "Fiat-Shamir with Aborts", which was used to transform the interactive lattice-based zero-knowledge protocol into a non-interactive one. The non-interactive protocol could then be used as a digital signature by including the message to be signed in the random oracle, as shown in Section 2.2.4. An instance of the digital signature scheme introduced in Section 2.2.4 with 128-bit quantum security [Lyu20] is presented in Section 4.1.2, while the implementation of this scheme is presented in Section 4.2.2 and 4.2.3. With this, research objective 3 was achieved. A user could now authenticate without the need for a password by demonstrating possession of the private key $(\mathbf{s_1}, \mathbf{s_2}, \rho')$ associated with a specific public key $(\mathbf{t}, \rho')$. This is achieved by generating signatures on the random messages generated by the RP using the private key corresponding to the public key, effectively proving ownership and enabling authentication without the need for further interactive exchanges. With research objectives 2 and 3 achieved, research question 2 was answered.

5.2.3 Research Question 3

How does the performance of the implemented passwordless authentication system, incorporating the zero-knowledge protocol, compare to similar state-of-the-art solutions?

The authors chose to focus on the implemented lattice-based digital signature when evaluating the performance of the proposed solution. This evaluation was carried out and presented in Section 5.1, thus achieving research objective 4. Section 2.4.2 presents the state of the art within the lattice-based digital signature, i.e., Dilithium, which resembles the implemented digital signature scheme, as both rely on the hardness of SIS and LWE. The performance of Dilithium is presented and includes the metrics used when comparing the performance of Dilithium with the digital signature scheme implemented by the authors. The comparison is presented alongside the performance evaluation of the implemented scheme in Section 5.1, and reveals that even though the implemented scheme is outperformed by Dilithium in terms of performance, it is more than efficient enough for the specific use case that is passwordless authentication. With this, research question 3 was answered.

Chapter 6
Conclusion and Future Work

This thesis aimed to develop a PoC for a passwordless authentication system using a lattice-based digital signature. The proposed solution successfully achieved this objective by implementing a digital signature based on the combination of SIS and LWE. An in-depth literature study explored the use of lattice-based zero-knowledge protocols and digital signatures.

Passwordless authentication was realized by creating a test environment inspired by FIDO2. This test environment utilizes the lattice-based digital signature to facilitate authentication. The digital signature was transformed from an interactive lattice-based zero-knowledge protocol with the use of "Fiat-Shamir with Aborts". This resulted in the proposed solution, whose performance was to be tested. The experimental evaluation of the proposed solution demonstrated its effectiveness in terms of key sizes, signature size, and the performance of the key generation, signature, and verification algorithms. The tests indicated that the implemented digital signature is feasible in a passwordless authentication system.

In conclusion, this thesis successfully demonstrated the feasibility of a passwordless authentication system based on a lattice-based digital signature. The system's ability to authenticate users without traditional passwords enhances user convenience and reduces the risk of password-related vulnerabilities, such as phishing attacks and password breaches. Further improvements and refinements, as outlined in the following Section, would contribute to a more robust and efficient system suitable for real-world implementation.

6.1 Future Work

The following presents the future work recommended by the authors.

TLS Per the WebAuthn specification, communication between the client application and RP should be protected by TLS. In the proposed solution, this communication

is carried out over HTTP for simplicity. For the proposed solution to follow the WebAuthn specification more closely, the implementation of TLS is recommended, as it provides key security features such as encryption, data integrity, and mutual authentication.

CTAP FIDO2 defines CTAP as the application layer protocol in charge of communication between the client application and the authenticator. This protocol is not implemented in the proposed solution. The use of CTAP ensures secure and standardized communication between the two components and mitigates several possible vulnerabilities. As CTAP works on top of transport protocols such as NFC and Bluetooth, this ensures that the authenticator and client are in close proximity to each other, thus mitigating the possibility of e.g., session hijacking.

NTT A key element to making lattice-based protocols efficient is NTT, as it provides an efficient way to multiply polynomials, which is one of the most computationally expensive operations in lattice-based schemes. An efficient implementation of NTT is present in Dilithium, which is part of the reason why Dilithium is one of the most efficient post-quantum schemes [DKL+18]. As the digital signature implemented in the proposed solution works with vectors and matrices of polynomials, implementing NTT would be a natural next step to improve performance.

AES The authors of this thesis took inspiration from the authors of Dilithium and implemented the three algorithms `expandS`, `expandA`, and `expandMask` for expanding the secret vectors $(\mathbf{s_1}, \mathbf{s_2})$, the public matrix $\mathbf{A}$, and the two masking vectors $(\mathbf{y_1}, \mathbf{y_2})$ from three randomly sampled seeds. The implementation of these algorithms in the proposed solution is not as efficient as Dilithiium's implementation, as Dilithium employs a vectorized representation of the XOF SHAKE, which allows them to sample multiple coefficients in parallel [DKL+18]. As presented in the Dilithium specification 3.1 [DKL+21], these algorithms have been implemented to support the use of AES to generate the coefficients of the polynomials instead of an XOF instantiated as SHAKE. This way, already made hardware implementations of AES could be utilized for an additional speed up for these algorithms. The implemented authenticator iOS application runs on iPhones, which grants them access to Apple's Secure Enclave AES Engine, a hardware block designed to execute operations based on AES [22]. Implementing `expandS`, `expandA`, and `expandMask` to utilize Secure Enclave's AES Engine would result in increased performance of the implemented digital signature scheme.

Parameters When evaluating the implemented scheme, the focus was to investigating how the parameters (n, m) and β affected its performance. For future work, testing the parameters γ and $\bar{\beta}$ is recommended. Given Equation 5.2, it is clear how

these parameters affect the performance of the signature algorithm. However, further testing would unveil the optimal parameter values that provide the highest security level without significantly reducing the efficiency of the scheme.

Scalability For this system to be implemented in the real world setting, the server should be able to handle a vast amount of concurrent requests. As of now, the proposed solution includes two servers, i.e., RP server and polling server, created using Flask. The servers are not designed to scale well and handle large amounts of simultaneous requests, as it falls outside the scope of the thesis. However, future work on the proposed solution should include further development of the RP server to able it to handle a higher amount of concurrent requests. This is not the case with the polling server, as it is obsolete if CTAP is implemented in future work with the system.

Converging Towards Dilithium The digital signature implemented in the proposed solution resembles Dilithium but lacks the optimized implementations presented in [Lyu20] and [DKL+18]. The authors recommend implementing Dilithium as the digital signature scheme of choice in the proposed solution, as it is the result of years of research and is soon to be standardized by NIST [NIS22].

References

[19] *Random – the entropy device*, Online; accessed 2-June-2023, FreeBSD Documentation Project, 2019.

[20a] *Ebay's journey to passwordless with fido*, https://media.fidoalliance.org/wp-content/uploads/2021/02/Fido-ebay.pdf, Online; accessed 24-May-2023, FIDO Alliance, 2020.

[20b] *Getrandom – get random data*, FreeBSD System Calls Manual, FreeBSD Documentation Project, 2020. [Online]. Available: %5Curl%7Bhttps://man.freebsd.org/cgi/man.cgi?query=getrandom&sektion=2&n=1%7D (last visited: Jun. 4, 2023).

[20c] *Solokeys*, https://github.com/solokeys, Online; accessed 25-May-2023, SoloKeys, 2020.

[22] *Apple platform security*, https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf, Online; accessed 16-May-2023, Apple Inc., 2022.

[23a] *Data*, https://developer.apple.com/documentation/foundation/data, Online; accessed 2-June-2023, Apple Inc., 2023.

[23b] *Dispatchtime*, https://developer.apple.com/documentation/dispatch/dispatchtime, Online; accessed 12-June-2023, Apple Inc., 2023.

[23c] *Foundation*, https://developer.apple.com/documentation/foundation, Online; accessed 2-June-2023, Apple Inc., 2023.

[23d] *Ksecattraccessiblewhenpasscodesetthisdeviceonly*, https://developer.apple.com/documentation/security/ksecattraccessiblewhenpasscodesetthisdeviceonly, Online; accessed 16-May-2023, 2023.

[23e] *React - the library for web and native user interfaces*, https://react.dev, Online; accessed 16-May-2023, Meta Open Source, 2023.

[23f] *Secrandomcopybytes(...)*, https://developer.apple.com/documentation/security/1399291-secrandomcopybytes, Online; accessed 4-June-2023, Apple Inc., 2023.

[23g] *Swift*, https://developer.apple.com/swift/, Online; accessed 16-May-2023, Apple Inc., 2023.

[23h] *Uptimenanoseconds*, https://developer.apple.com/documentation/dispatch/dispatchtime/2300047-uptimenanoseconds, Online; accessed 12-June-2023, Apple Inc., 2023.

[23i] *Xcode ide*, https://developer.apple.com/xcode/features/, Online; accessed 16-May-2023, Apple Inc., 2023.

[Ahn23a] C. Ahn, *Numpy-ios*, https://github.com/kewlbear/NumPy-iOS, Online; accessed 16-May-2023, 2023.

[Ahn23b] C. Ahn, *Python-ios*, https://github.com/kewlbear/Python-iOS, Online; accessed 16-May-2023, 2023.

[Ajt96] M. Ajtai, «Generating hard instances of lattice problems», in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 99–108.

[All] F. Alliance, *Apple, google, and microsoft are pushing passkeys: Password-less future?*, https://fidoalliance.org/tech-times-apple-google-and-microsoft-are-pushing-passkeys-password-less-future/, Online; accessed 25-May-2023.

[All23] F. Alliance, *Fido2: Web authentication (webauthn)*, https://fidoalliance.org/fido2-2/fido2-web-authentication-webauthn/, [Online; accessed 13-April-2022], 2023.

[Ama] I. Amazon Web Services, *Enabling a fido security key (console)*, https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_mfa_enable_fido.html, Online; accessed 24-May-2023.

[AR05] D. Aharonov and O. Regev, «Lattice problems in np∩ conp», *Journal of the ACM (JACM)*, vol. 52, no. 5, pp. 749–765, 2005.

[AS20] H. Amiri and A. Shahbahrami, «Simd programming using intel vector extensions», *Journal of Parallel and Distributed Computing*, vol. 135, pp. 83–100, 2020.

[BDF+11] D. Boneh, Ö. Dagdelen, *et al.*, «Random oracles in a quantum world», 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 17*, Springer, 2011, pp. 41–69.

[BHH+15] D. J. Bernstein, D. Hopwood, *et al.*, «Sphincs: Practical stateless hash-based signatures», in *Advances in Cryptology–EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I 34*, Springer, 2015, pp. 368–397.

[BNG22] L. Beckwith, D. T. Nguyen, and K. Gaj, «High-performance hardware implementation of lattice-based digital signatures», *Cryptology ePrint Archive*, 2022.

[BR93] M. Bellare and P. Rogaway, «Random oracles are practical: A paradigm for designing efficient protocols», in *Proceedings of the 1st ACM Conference on Computer and Communications Security*, 1993, pp. 62–73.

[Con15] W. W. W. Consortium, *Fido 2.0: Key attestation format*, https://www.w3.org/Submission/2015/SUBM-fido-key-attestation-20151120/%23signaturef, [Online; accessed 26-April-2022], 2015.

[Con19] W. W. W. Consortium, *Web authentication: An api for accessing public key credentials*, https://www.w3.org/TR/webauthn-2/, [Online; accessed 13-April-2022], 2019.

[Con23] M. Contributors, *Attestation and assertion*, 2023. [Online]. Available: %5Curl %7Bhttps://developer.mozilla.org/en-US/docs/Web/API/Web_Authentication_API/Attestation_and_Assertion%7D (last visited: Jun. 7, 2023).

[Dam02] I. Damgård, «On Σ-protocols», *Lecture Notes, University of Aarhus, Department for Computer Science*, p. 84, 2002.

[DFG13] Ö. Dagdelen, M. Fischlin, and T. Gagliardoni, «The fiat–shamir transformation in a quantum world», in *Advances in Cryptology-ASIACRYPT 2013: 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part II 19*, Springer, 2013, pp. 62–81.

[DFMS19] J. Don, S. Fehr, *et al.*, «Security of the fiat-shamir transformation in the quantum random-oracle model», in *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part II 39*, Springer, 2019, pp. 356–383.

[DFPS23] J. Devevey, P. Fallahpour, *et al.*, *A detailed analysis of fiat-shamir with aborts*, Cryptology ePrint Archive, Paper 2023/245, https://eprint.iacr.org/2023/245, 2023. [Online]. Available: https://eprint.iacr.org/2023/245.

[Din] P. Dingle, *All about fido2, ctap2 and webauthn*, https://techcommunity.microsoft.com/t5/security-compliance-and-identity/all-about-fido2-ctap2-and-webauthn/ba-p/288910, Online; accessed 24-May-2023.

[DKL+18] L. Ducas, E. Kiltz, *et al.*, «Crystals-dilithium: A lattice-based digital signature scheme», *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 238–268, 2018.

[DKL+21] L. Ducas, E. Kiltz, *et al.*, *Crystals-dilithium, algorithm specifications and supporting documentation (version 3.1)*, 2021.

[FHK+18] P.-A. Fouque, J. Hoffstein, *et al.*, «Falcon: Fast-fourier lattice-based compact signatures over ntru», *Submission to the NIST's post-quantum cryptography standardization process*, vol. 36, no. 5, 2018.

[FS03] N. Ferguson and B. Schneier, *Practical cryptography*. Wiley New York, 2003, vol. 141.

[FS87] A. Fiat and A. Shamir, «How to prove yourself: Practical solutions to identification and signature problems», in *Advances in Cryptology—CRYPTO'86: Proceedings 6*, Springer, 1987, pp. 186–194.

[GHLY16] L. Groot Bruinderink, A. Hülsing, *et al.*, «Flush, gauss, and reload–a cache attack on the bliss lattice-based signature scheme», in *Cryptographic Hardware and Embedded Systems–CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings 18*, Springer, 2016, pp. 323–345.

[GMSS99] O. Goldreich, D. Micciancio, *et al.*, «Approximating shortest lattice vectors is not harder than approximating closest lattice vectors», *Information Processing Letters*, vol. 71, no. 2, pp. 55–61, 1999.

[HLHL10] C. Hazay, Y. Lindell, *et al.*, «Sigma protocols and efficient zero-knowledge», *Efficient Secure Two-Party Protocols: Techniques and Constructions*, pp. 147–175, 2010.

[Hou20] R. Housley, *Use of the HSS/LMS Hash-Based Signature Algorithm in the Cryptographic Message Syntax (CMS)*, RFC 8708, Feb. 2020. [Online]. Available: https://www.rfc-editor.org/info/rfc8708.

[IBM] IBM, *What is quantum computing?*, https://www.ibm.com/topics/quantum-computing, Online; accessed 21-May-2023.

[KP20] M. Kumar and P. Pattnaik, «Post quantum cryptography (pqc)-an overview», in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, 2020, pp. 1–9.

[LEMT22] M. Levent Doğan, A. A. Ergür, *et al.*, «The multivariate schwartz–zippel lemma», *SIAM Journal on Discrete Mathematics*, vol. 36, no. 2, pp. 888–910, 2022.

[LJL+10] T. D. Ladd, F. Jelezko, *et al.*, «Quantum computers», *nature*, vol. 464, no. 7285, pp. 45–53, 2010.

[Lyu09] V. Lyubashevsky, «Fiat-shamir with aborts: Applications to lattice and factoring-based signatures», in *Advances in Cryptology–ASIACRYPT 2009: 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings 15*, Springer, 2009, pp. 598–616.

[Lyu20] V. Lyubashevsky, *Basic lattice cryptography: Encryption and fiat-shamir signatures*, https://drive.google.com/file/d/1JTdW5ryznp-dUBBjN12QbvWz9R41NDGU/view, Online; accessed 27-April-2023, 2020.

[LZ19] Q. Liu and M. Zhandry, «Revisiting post-quantum fiat-shamir», in *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part II 39*, Springer, 2019, pp. 326–355.

[LZ22] Z. Liang and Y. Zhao, «Number theoretic transform and its applications in lattice-based cryptosystems: A survey», *arXiv preprint arXiv:2211.13546*, 2022.

[Mao03] W. Mao, *Modern cryptography: theory and practice*. Pearson Education India, 2003.

[Mer06] D. Mermin, «Breaking rsa encryption with a quantum computer: Shor's factoring algorithm», *Lecture notes on Quantum computation*, pp. 481–681, 2006.

[Mic07] D. Micciancio, «Generalized compact knapsacks, cyclic lattices, and efficient one-way functions», *computational complexity*, vol. 16, pp. 365–411, 2007.

[Mon23] MongoDB, *Mongo python driver*, https://github.com/mongodb/mongo-python-driver, 2023.

[NIS17] NIST, *Post-quantum cryptography standardization*, https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization, [Online; accessed 3-November-2022], 2017.

[NIS22] NIST, *Selected algorithms 2022*, https://csrc.nist.gov/projects/post-quantum-cryptography/selected-algorithms-2022, [Online; accessed 19-May-2023], 2022.

[NS22] V. Nyeng and L. Sørensen, «The implementation and use of cryptographic zero-knowledge protocols», Norwegian University of Science and Technology, Project report in TTM4502, Dec. 2022.

[Num23] NumPy, *Numpy*, https://github.com/numpy/numpy, 2023.

[Pal23] Pallets, *Flask*, https://github.com/pallets/flask, 2023.

[Pei+16] C. Peikert *et al.*, «A decade of lattice cryptography», *Foundations and Trends® in Theoretical Computer Science*, vol. 10, no. 4, pp. 283–424, 2016.

[PKLN22] S. Paul, Y. Kuzovkova, *et al.*, «Mixed certificate chains for the transition to post-quantum authentication in tls 1.3», in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022, pp. 727–740.

[pq-17] pq-crystals, *Dilithium*, , Online; accessed 16-May-2023, 2017.

[PU02] A. Papoulis and S. Unnikrishna Pillai, *Probability, random variables and stochastic processes*. 2002.

[Pyt23a] Python, *Hashlib*, https://docs.python.org/3/library/hashlib.html, 2023.

[Pyt23b] Python, *Json*, https://docs.python.org/3/library/json.html, 2023.

[Pyt23c] Python, *Os*, https://docs.python.org/3/library/os.html, 2023.

[PZ04] J. Proos and C. Zalka, *Shor's discrete logarithm quantum algorithm for elliptic curves*, 2004.

[QQQ+01] J.-J. Quisquater, M. Quisquater, *et al.*, «How to explain zero-knowledge protocols to your children», in *Advances in Cryptology—CRYPTO'89 Proceedings*, Springer, 2001, pp. 628–631.

[Reg09] O. Regev, «On lattices, learning with errors, random linear codes, and cryptography», *Journal of the ACM (JACM)*, vol. 56, no. 6, pp. 1–40, 2009.

[Sco16] C. Scott, *Protecting our members*, https://blog.linkedin.com/2016/05/18/protecting-our-members, [Online; accessed 4-November-2022], 2016.

[Sho97] P. W. Shor, «Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer», *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, 1997. [Online]. Available: https://doi.org/10.1137/S0097539795293172.

[TJB12] L. Thijs, v. d. P. Joop, and d. W. Benne, *Solving hard lattice problems and the security of lattice-based cryptosystems*, Cryptology ePrint Archive, Paper 2012/533, https://eprint.iacr.org/2012/533, 2012. [Online]. Available: https://eprint.iacr.org/2012/533.

[Unr17] D. Unruh, «Post-quantum security of fiat-shamir», in *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, Springer, 2017, pp. 65–95.

[Ver22] Verizon, «Data breach investigations report», 2022, [Online; accessed 3-November-2022], p. 37.

[Vie22] P. J. P. Vieito, *Pythonkit*, https://github.com/pvieito/PythonKit, Online; accessed 16-May-2023, 2022.

[VJ14] H. C. Van Tilborg and S. Jajodia, *Encyclopedia of cryptography and security*. Springer Science & Business Media, 2014.

[Wie14] R. J. Wieringa, «Design science methodology for information systems and software engineering», in R. J. Wieringa, Ed. Berlin, Heidelberg: Springer Berlin, Heidelberg, 2014, p. 27. [Online]. Available: https://doi.org/10.1007/978-3-662-43839-8.

Appendix A
Performance Data

This appendix contains the scripts used to collect data used in the evaluation of the performance of the proposed solution, and the data collected. The plots presented in Chapter 5 are plotted based on this data.

A.1 Data Collection Scripts

```
for i in 3...15 {
    print("Sizes for (n,m) = (\(i), \(i-1))")
    dilithiumLite.changeNM(n: i, m: i-1)
    let keypair = dilithiumLite.generateKeyPair()
    print("Secret key: \(DilithiumLite.getSecretKeyAsData(secretKey: keypair!.secretKey)?.count)")
    print("Public key: \(DilithiumLite.getPublicKeyAsData(publicKey: keypair!.publicKey)?.count)")
    let sig = dilithiumLite.sign(sk: keypair!.secretKey, message: "SFM's disciples")
    print("Signature: \(DilithiumLite.getSignatureAsData(signature: sig)?.count)")
}
```

Listing A.1: Script for collecting size of private key, public key and signature.

```
let tests = 100
for beta in 1...20 {
    dilithiumLite.changeBeta(beta: beta)
    let n = 5
    let m = 4
    print("Testing for beta = \(beta)")
    dilithiumLite.changeNM(n: n, m: m)
    var attempts: [Int] = []
    for _ in 1...tests {
        let keypair = dilithiumLite.generateKeyPair()
        let sig = dilithiumLite.sign(sk: keypair!.secretKey, message: "SFM's disciples")
        attempts.append(sig.attempts)
    }
    print("Average attempts: \(attempts.reduce(0, +)/attempts.count)")
    print()
}
```

Listing A.2: Script for collecting the average number of attempts until a valid signature is generated, while varying β. $(n, m) = (5, 4)$.

```
let tests = 50
for beta in 3...7 {
    dilithiumLite.changeBeta(beta: beta)
    var n: Int
```

```
    var m: Int
    for i in 3...15 {
        n = i
        m = i-1
        print("Testing for (n, m, beta) = (\(n), \(m), \(beta))")
        dilithiumLite.changeNM(n: n, m: m)
        var sigTimes: [Int] = []
        var attempts: [Int] = []
        for _ in 1...tests {
            let keypair = dilithiumLite.generateKeyPair()
            let startSig = DispatchTime.now()
            let sig = dilithiumLite.sign(sk: keypair!.secretKey, message: "SFM's disciples")
            let stopSig = DispatchTime.now()
            sigTimes.append(Int(stopSig.uptimeNanoseconds-startSig.uptimeNanoseconds))
            attempts.append(sig.attempts)
        }
        print("Average attempts sign: \(attempts.reduce(0, +)/attempts.count)")
        print("Average time sign: \(sigTimes.reduce(0, +)/sigTimes.count)")
        print()
    }
}
```

Listing A.3: Script for collecting the average time usage and average number of attempts for the signature algorithm, while varying (n, m) and β.

```
let tests = 100
for beta in 3...7 {
    dilithiumLite.changeBeta(beta: beta)
    var n: Int
    var m: Int
    for i in 3...15 {
        n = i
        m = i-1
        print("Testing for (n, m, beta) = (\(n), \(m), \(beta))")
        dilithiumLite.changeNM(n: n, m: m)
        var keyGenTimes: [Int] = []
        for _ in 1...tests {
            let startKeyGen = DispatchTime.now()
            let keypair = dilithiumLite.generateKeyPair()
            let stopKeyGen = DispatchTime.now()
            keyGenTimes.append(Int(stopKeyGen.uptimeNanoseconds-startKeyGen.uptimeNanoseconds))
        }
        print("Average time keygen: \(keyGenTimes.reduce(0, +)/keyGenTimes.count)")
        print()
    }
}
```

Listing A.4: Script for collecting the average key generation time while varying (n, m) and β.

```
def test():
    print("TESTING STARTED")
    tests = 100
    for i in range(3,8):
        for j in range(3,16):
            print("Testing for beta = "+str(i)+", n = "+str(j)+", m = "+str(j-1))
            times = []
            for q in range(tests):
                kp = keyGen(i, j, j-1)
                sk = kp[0]
                pk = kp[1]
                message = "SFM's disciples"
                sig = sign(sk, message, j, j-1)
                start = time.time_ns()
                verifySig(pubKey=pk, sig=sig, clientData=message, n=j, m=j-1)
                times.append(time.time_ns()-start)
```

```
17            print("Average time: "+str(np.average(np.array(times))))
18            print()
19    print("TESTING DONE")
```

Listing A.5: Script for collecting the time usage for the implemented verification algorithm.

A.2 Key and Signature Sizes

Table A.1: Sizes in bytes for private key, public key, and signature.

$(\mathbf{n}, \mathbf{m})$	**Private Key Size**	**Public Key Size**	**Signature Size**
$(3, 2)$	3133	6090	7741
$(4, 3)$	4405	8090	11471
$(5, 4)$	5645	10102	15179
$(6, 5)$	6870	12098	18899
$(7, 6)$	8067	14102	22722
$(8, 7)$	9285	16125	26480
$(9, 8)$	10577	18132	30060
$(10, 9)$	11790	20142	33881
$(11, 10)$	13048	22124	37730
$(12, 11)$	14293	24119	41320
$(13, 12)$	15522	26182	45034
$(14, 13)$	16723	28165	48813
$(15, 14)$	17996	30209	52591

A.3 Key generation

Table A.2: Average key generation time (ms)

$(\mathbf{n}, \mathbf{m})$	$\beta = 3$	$\beta = 4$	$\beta = 5$	$\beta = 6$	$\beta = 7$
(3,2)	130.57	143.16	132.46	134.23	120.45
(4,3)	226.81	246.28	231.21	235.46	212.31
(5,4)	350.18	390.71	349.63	367.79	327.50
(6,5)	496.61	538.77	499.62	503.64	466.08
(7,6)	676.40	700.66	672.34	670.57	636.85
(8,7)	883.20	894.93	888.63	856.46	841.74
(9,8)	1116.19	1127.25	1147.23	1087.25	1052.71
(10,9)	1369.36	1364.01	1363.59	1531.21	1292.58
(11,10)	1647.19	1646.91	1627.85	1611.15	1652.95
(12,11)	1952.07	1945.12	1937.79	1887.86	1944.03
(13,12)	2279.65	2281.80	2301.95	2188.01	2369.22
(14,13)	2638.84	2648.87	2831.75	2565.86	2654.40
(15,14)	3107.22	3015.45	3156.46	2940.96	3030.88

A.4 Signing

Table A.3: Average signature time (ms)

$(\mathbf{n}, \mathbf{m})$	$\beta = 3$	$\beta = 4$	$\beta = 5$	$\beta = 6$	$\beta = 7$
(3,2)	131.31	140.42	140.60	143.46	145.55
(4,3)	248.09	261.05	272.54	253.08	275.83
(5,4)	432.74	406.32	431.17	478.79	416.27
(6,5)	775.55	661.51	713.10	674.05	680.49
(7,6)	1003.54	981.36	1115.22	1142.55	1129.25
(8,7)	1303.50	1363.01	1563.68	1512.24	1411.70
(9,8)	1975.78	1974.99	1928.60	2106.15	1853.62
(10,9)	2648.48	2634.71	3011.58	2951.92	2777.47
(11,10)	3538.44	3638.07	3904.50	3106.28	3708.05
(12,11)	5249.34	6329.85	4797.03	5587.59	6332.86
(13,12)	7177.48	5878.73	7617.43	5852.59	6469.68
(14,13)	8009.11	9081.39	10537.19	9124.83	11297.02
(15,14)	12419.83	13473.25	12534.46	10698.88	12704.96

Table A.4: Average signature attempts for varying (n,m) and β

$(\mathbf{n}, \mathbf{m})$	$\boldsymbol{\beta = 3}$	$\boldsymbol{\beta = 4}$	$\boldsymbol{\beta = 5}$	$\boldsymbol{\beta = 6}$	$\boldsymbol{\beta = 7}$
(3,2)	2.10	1.98	1.86	1.92	2.02
(4,3)	2.54	2.52	2.62	2.16	2.82
(5,4)	2.68	2.90	3.12	3.90	3.00
(6,5)	5.00	4.36	4.76	4.24	4.58
(7,6)	5.50	5.82	7.14	7.22	7.78
(8,7)	6.98	7.26	8.90	8.40	7.94
(9,8)	10.66	10.28	9.26	10.80	9.32
(10,9)	13.00	12.66	14.98	14.60	13.92
(11,10)	16.62	16.90	18.22	12.42	17.36
(12,11)	24.00	31.20	20.38	25.34	29.32
(13,12)	31.94	24.00	33.08	22.78	24.74
(14,13)	31.46	36.56	44.26	36.46	46.26

Table A.5: Average number of signature attempts for different β values with $(n, m) = (5, 4)$.

β	**Attempts**
1	2.98
2	3.46
3	3.55
4	3.67
5	3.50
6	3.73
7	2.77
8	2.86
9	3.69
10	3.36
11	3.45
12	3.64
13	3.58
14	3.13
15	3.11
16	2.82
17	3.44
18	3.58
19	3.27
20	3.39

A.5 Verification

Table A.6: Average verification time (ms)

$(\mathbf{n}, \mathbf{m})$	$\boldsymbol{\beta = 3}$	$\boldsymbol{\beta = 4}$	$\boldsymbol{\beta = 5}$	$\boldsymbol{\beta = 6}$	$\boldsymbol{\beta = 7}$
(3,2)	7.35	8.49	7.40	7.48	7.38
(4,3)	12.47	16.76	12.58	12.59	12.79
(5,4)	21.53	19.31	19.36	19.37	19.45
(6,5)	32.62	31.07	27.72	27.60	28.26
(7,6)	38.75	39.37	37.63	37.82	37.78
(8,7)	54.27	52.66	50.15	48.62	48.62
(9,8)	78.05	68.83	64.87	61.08	61.44
(10,9)	86.74	86.74	76.86	76.12	76.12
(11,10)	116.40	103.24	107.99	92.25	92.25
(12,11)	128.94	120.34	109.18	113.07	111.40
(13,12)	151.99	136.22	128.30	128.35	127.43
(14,13)	183.95	162.89	148.45	159.69	154.39
(15,14)	173.10	190.64	171.81	171.70	184.60

Appendix B
Test Report

This appendix contains the test reports for the functional requirements.

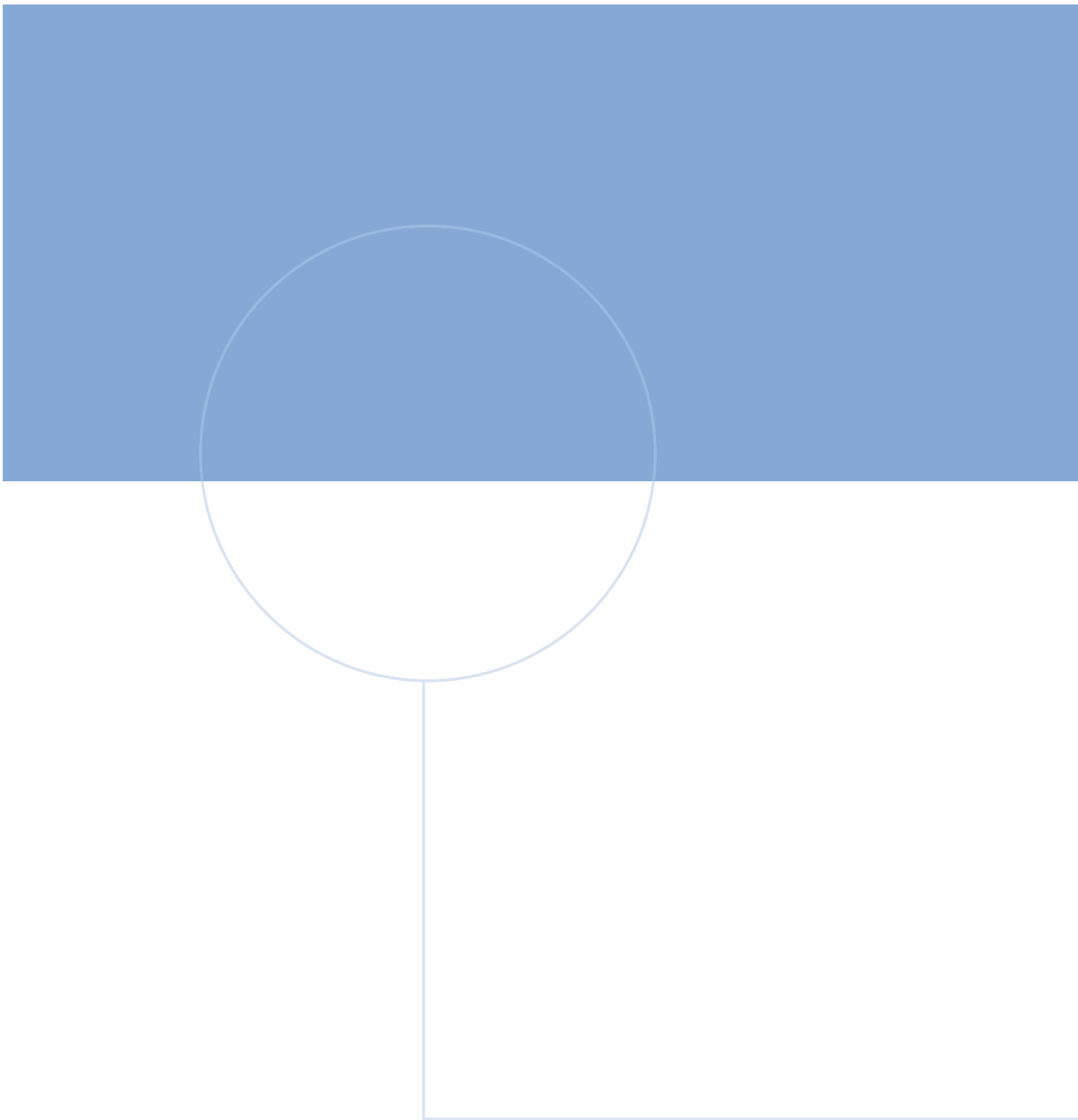
B.1 Functional Requirements

FR1: The user interface should be simple and intuitive. Users should be able to register an account without a password	
Executor:	VN
Date:	14.06.2023
Time used:	<1 minute
Evaluation:	Success
Comment:	When accessing the client website, two options are presented: *register* and *login*. After choosing register, a unique username and one-time code is required to fill in. After clicking "Register", an alert for registration pops up on the authenticator app. Upon accepting this, registration is completed.

FR2: Users should be able to authenticate themselves without a password, using the provided authenticator application	
Executor:	VN
Date:	14.06.2023
Time used:	<1 minute
Evaluation:	Success
Comment:	After choosing login, your username is required. Entering this and clicking "Login" results in a pop-up alert shown on the authenticator app. After accepting this request for authentication, authentication is completed and the client website communicates this to the user.

FR3: Users should not be able to register an account with a username already in use	
Executor:	VN & LS
Date:	14.06.2023
Time used:	2 minutes
Evaluation:	Success
Comment:	First registering user_1 to the RP, and afterwards trying to register a user with the same username using another authenticator. An alert from the client responds with "user_1 already registered".

FR4: Multiple users should not be able to share the same authenticator for the same Relying Party	
Executor:	VN & LS
Date:	14.06.2023
Time used:	2 minute
Evaluation:	Success
Comment:	Registered user_1 to the RP with authenticator_1. Next, user_2 tried to register with the RP using authenticator_1. Received the following feedback from the client: "Authenticator is already registered to another user".



NTNU

Norwegian University of
Science and Technology