Concrete implementation of *t*-out-of-*n* threshold lattice signatures

MAX GUSTAFSSON & MATTIAS PETERSSON

MASTER´S THESIS

DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Concrete implementation of t-out-of-n threshold lattice signatures

Department of Electrical and Information Technology
Lund University

Max Gustafsson
ma5517gu-s@student.lu.se

Mattias Petersson
ma8884pe-s@student.lu.se

Supervisor: Paul Stankovski Wagner

Assistant supervisor: Denis Nabokov

Examiner: Thomas Johansson

June 15, 2024

Abstract

In preparation for quantum attacks on current cryptographic primitives, efforts are being made to find ways to construct new primitives. While quantum computers are not yet capable of breaking the keys used in modern parameter selection, efforts are still being made to establish new primitives not vulnerable to any known quantum attacks. These newly proposed solutions range from theoretical constructions to concrete implementations. In this paper, we implement a lattice-based linearly homomorphic t-out-of-n threshold signature scheme based on lattices in an effort to verify its functionality and gather detailed performance data. We do this using fully homomorphic encryption schemes and Shamir's Secret Sharing method. Our work proves that both the proposed passive and active security constructions in the paper works in practice using our implementation. We provide insights to the communication between participants and the number of messages being sent, as well as the size of each message during an actively secure run of our scheme. We also show some results on key sizes and total execution time for different parameters, as well as computations for the number of mathematical operations used in the algorithms. We further used our implementation to determine how runtime scales for key generation and signature generation for increasing values of (t, n). The main bottleneck was determined to be the Shamir's Secret Sharing component of the algorithm. Our results also show that key generation, which was the most expensive algorithm overall, achieves maximum time-cost per participant when t is equal to $\frac{n+1}{2}$.

Keywords: Lattices – Threshold Signatures – Homomorphic Encryption – Threshold Encryption.

Popular Science Summary

There is a current challenge to develop new cryptographic standards that are secure against quantum algorithms. With the knowledge that a sufficiently powerful quantum computer can break current asymmetric cryptographic schemes, it is important to redesign such protocols. This thesis explores the practical implementation of a threshold-signature scheme that is secure even with access to a quantum computer.

Digital signatures are schemes utilizing asymmetric cryptography, allowing an entity to create a public/private key pair. The public key can then be distributed freely, while the private key can be used to sign messages, documents, software and more. Anyone with access to the public key can verify that the signature is legitimate, i.e. the signature was created by the holder of the private key. This scheme can be extended to a threshold scheme, where multiple participants are required to create a valid signature. Threshold-signature schemes are typically called t-out-of-n signature schemes, where t denotes the number of participants required to create a valid signature, and n denotes the total number of participants.

This is achieved in our scheme with two underlying homomorphic schemes for commitments and encryption. The homomorphic property allows for mathematical operations such as addition of encrypted data, e.g. ciphertexts, where adding multiple ciphertexts is equivalent to the sum of the encrypted corresponding plaintext.

Lattices are a popular candidate going forward for constructing quantum-safe algorithms. Additionally, homomorphic schemes were first realized in lattice-based cryptography. A lattice is a set of points in a multi-dimensional space, created by taking the linear combinations of a set of basis vectors.

There exists various theoretical implementations of lattice-based t-out-of-n schemes, but publicly available practical implementations are lacking. Our implementation serves as a first prototype to demonstrate the feasibility of implementing these schemes. We evaluate execution time, sizes of messages sent, and number of mathematical operations in the protocol for different values of (t, n). We found that our system scaled exponentially, with Shamir's Secret Sharing method requiring the most time out of all algorithms. We also discovered that key generation, which was the most expensive algorithm, was the most expensive for each participant when t was equal to $\frac{n+1}{2}$.

Acknowledgments

We would like to thank our supervisors, Paul and Denis, for helpful feedback and support, enabling us to do the best work we could. The idea for this thesis came from Paul and was a perfect fit for our interests, and for that we are appreciative.

We would also like to thank our friends and family that has been there for us throughout our entire time at LTH. Without your constant encouragement, we would not be where we are. We would like to give special thanks to William Emami, for providing useful feedback on this paper while we were in the process of writing it.

List of abbreviations

BDLOP18	Efficient homomorphic commitment scheme
BGV11	Fully homomorphic encryption scheme
CRYSTALS	Cryptographic Suite For Algebraic Lattices
DOTT20	n-out-of-n threshold signature scheme
FHE	Fully Homomorphic Encryption
GKS23	t-out-of-n threshold signature scheme
HVZKP	Honest Verifier Zero-knowledge Proof
LDB23	t-out-of-n threshold signature scheme
LPN	Learning Parity with Noise
LWE	Learning With Errors
MP12	Trapdoors for lattices
NIST	National Institute of Standards and Technology
NIZKP	Non-interactive Zero-knowledge Proof
ROM	Random Oracle Model
RSA	Asymmetric encryption algorithm by Rivest, Shamir and Adleman
SIS	Short Integer Solution
SVP	Shortest Vector Problem
ZKP	Zero-Knowledge Proof

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

This paper is an exploration of a practical implementation of the recently published GKS23 scheme for t-out-of-n threshold signature cryptography based on the R-LWE problem. The originally published paper described the scheme, and gave suggestions for a future implementation, including commitment and zero-knowledge proof schemes which would be required. The goal was to follow these suggestions as closely as possible to evaluate the scheme, identify bottlenecks for performance and establish a springboard from which further investigations into the scheme could start.

1.1 Motivation

The state of cryptography is evolving rapidly due to the threat of quantum algorithms breaking commonly used cryptographic primitives. This means that there are several new schemes proposed every year, many lacking any amount of practical testing. This leads to interesting opportunities to study the process of implementing new schemes. Advanced cryptographic schemes are often built on top of other schemes, and while getting an objective view how how efficient a scheme is, it can be useful to identify what aspects of execution are the most costly, since that can help direct future efforts of optimization.

Our paper and implementation is done to evaluate performance, and should provide valuable insights into the efficiency and scalability of GKS23. We also lay the groundwork to identify bottlenecks in performance, which could improve the practical viability of the scheme itself.

1.2 Task

The goal of this paper is to implement a t-out-of-n threshold signature scheme from [GKS23], which had previously not been implemented. Due to a recent revision of the article, the implementation covers both the original and the revised schemes. When applicable we cite these separately, the revised version can be seen in [GKS24]. Our analysis focuses on execution time, signature and public key size, and the number of polynomial additions and multiplications. Different values of

(t, n)-threshold signatures are analyzed to observe how the scheme scales. Additionally, our analysis of $(3, 5)$-threshold signatures is used to compare signature and public key sizes with those presented in [GKS24].

1.3 Relevant Values to Study in Implementation

There are many variables to consider when comparing cryptographic schemes that make directly comparing time to execute in seconds or clock cycles difficult. Choices of language, exact implementation of underlying mathematical operations and optimization techniques can all drastically alter the execution time. As such, making a direct comparison of performance required controlling for these variables, which is difficult to accomplish without implementations done with identical design choices. While trying to compare the execution time of an implementation of GKS23 with existing schemes would not be particularly illuminating, there were still useful comparisons to be made. Variables like key size, signature size and the number of messages sent were aspects of the scheme itself and were therefore independent of our implementation.

Execution time could be of interest to provide comparative values for future research. Additionally, it served as a benchmark for performing modifications done to our implementation, as the relative change could be studied. It also allowed for the study of separate aspects of the implementation, allowing us to determine what aspects of the scheme are most useful to optimize or replace with faster alternatives.

Aside from execution time, which varies between exact implementation and the power of the computer, the number of certain costly operations in the scheme can be easily counted in a practical implementation. In GKS, these include the mathematical operations on large polynomials, the number of which are determined by the settings for t and n, additionally the number of Shamir secret reconstructions are also determined by the participant parameters. These can be used along with the size of coefficients and the degree of polynomials to compare the implemented scheme to others relying on similar mathematical operations directly, without having to compare implementations.

Chapter 2

Background

2.1 Preliminaries

This section aims to give an overview of notations and mathematical concepts that will be used throughout this paper. Then, we give an introduction to two algebraic structures known as groups and rings. We also define polynomials to be able to present polynomial rings, which are used extensively throughout this paper.

2.1.1 General Notation

Algorithms stemming from research papers will be denoted by first letter of each author's last name, as well as the year the paper was published. As an example, this thesis is implementing the scheme proposed by Gur, Katz, and Silde in [GKS23]. The scheme is subsequently called GKS23. Vectors and matrices are denoted by boldface lower- and uppercase letters, respectively. $\langle \boldsymbol{u}, \boldsymbol{v} \rangle$ denotes the cross product of vectors $\boldsymbol{u}, \boldsymbol{v}$. All vectors are column vectors by default. We write $:=$ for deterministic assignment, and $\overset{\$}{\leftarrow}$ for uniform random distribution from a set. For instance, we write $\boldsymbol{e} \overset{\$}{\leftarrow} \mathcal{X}^m$ as a vector of length m where each e_i is sampled from the distribution $\mathcal{X}$.

2.1.2 Polynomials

A polynomial is an expression that can be written in the form

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0.$$

A constant is a special case of polynomial known as a constant polynomial, containing only an a_0 term. In this paper, we are generally not interested in evaluating polynomials, aside from in the implementation of Shamir Secret Sharing. Instead, one could think of the x^n denoting the coefficient in the n-th position. A polynomial is irreducible if it cannot be factored into a product of two polynomials, excluding constant polynomials.

Norm of Polynomial

We use Borwein & Erdélyi's definition in [BE95] for the ℓ_p-norm of a polynomial p as

$$\|p\|_{\ell_p} = \left(\sum_{k=0}^{n} |a_k|^p \right)^{1/p}$$

for $p \geq 1$. A special case is the ℓ_∞-norm, defined as

$$\ell_\infty = \max_k \{|a_k|\}.$$

The norms used in our implementation are the ℓ_∞-norm, ℓ_1-norm, and ℓ_2-norm. We denote these norms, using the definitions above, as

$$\|p\|_1 = \sum_{k=0}^{n} |a_k|,$$
$$\|p\|_2 = \sqrt{\sum_{k=0}^{n} |a_k|^2},$$
$$\|p\|_\infty = \max_k \{|a_k|\}.$$

2.1.3 Algebraic Structures

In this section, we present two different types of algebraic structures known as groups and rings. We then expand on these concepts to introduce polynomial rings.

Definition 2.1.1 (Group)**.** A group $(G, *)$ consists of a set G, with a binary operation $*$ defined on G with the following properties.

- Closure: $x * y \in G, \forall x, y \in G$,
- Associativity: $(x * y) * z = x * (y * z), \forall x, y, z \in G$,
- Identity: $\exists e \in G : e * x = x * e = x, \forall x \in G$,
- Inverse: $\exists x' \in G : x * x' = x' * x = e, \forall x \in G$.

One additional property that is often desirable in groups is commutativity, i.e. $a * b = b * a$. This is not a strict requirement, a group that is commutative is an abelian group. Addition over the reals, denoted $(\mathbb{R}, +)$, is an example of an abelian group.

Definition 2.1.2 (Ring)**.** A ring $(R, +, \times)$ consists of a set R, where two binary operations $+, \times$ are defined with the following properties.

- $(R, +)$ is an abelian group,
- The $\times$ operator has closure, associativity, and identity as defined for groups,

- The distributive laws hold, i.e. $\forall a, b, c \in R$ we have

$$a \times (b + c) = (a \times b) + (a \times c),$$
$$(a + b) \times c = (a \times c) + (b \times c).$$

The $+$ operator is required to be commutative, the same does not apply to the $\times$ operator. Rings that are commutative for both operations are known as commutative rings. For example, addition and multiplication over the integers, denoted as $(\mathbb{Z}, +, \times)$, form a commutative ring. Throughout this paper, we frequently use the ring of integers modulo q, denoted by $\mathbb{Z}_q$.

2.1.4 Polynomial Rings

Rings can be formed using polynomials by fixing a polynomial $f(x)$ and an integer q. Elements in the ring are all polynomials of degree less than the degree of $f(x)$, with all coefficients $a_i \in \{-\frac{q-1}{2}, ..., \frac{q-1}{2}\}$. Addition is defined as polynomial addition, with coefficients reduced modulo q. Multiplication is likewise defined as polynomial multiplication, with the product reduced modulo $f(x)$ and q. We select $f(x)$ to be an irreducible polynomial in the ring and q to be prime, which makes the ring a commutative ring. As is common in literature, we let $f(x)$ be in the form $x^N + 1$, where N is a power of two. This type of ring has some desirable properties, one of which is that every non-zero element in the ring has a multiplicative inverse. This property is a requirement for many of the schemes implemented throughout this work. We denote a polynomial ring as

$$R_q = \mathbb{Z}_q[X]/\langle f(x) \rangle.$$

2.2 Threshold Signatures

Asymmetric encryption schemes can be used to provide authentication in the form of digital signatures. Algorithms for key generation, signing, and verification are required for a complete digital signature scheme. The key generation algorithm takes some input parameters and computes a public key and a secret key. The signing algorithm uses the secret key to create a valid signature. The verification step should not require the secret key, rather it should use the public key and thus be usable by any user, and will verify that the signature was indeed made by someone holding the secret key.

Threshold signatures are a modified version of digital signatures where the secret key is divided into multiple parts during key generation, where each user then holds only a part of the secret key. Signing then requires several users to combine their shares of the secret key in order to properly create a signature [Des92]. We write that there are n total keys, with t the minimum required number of keys to sign, which create a t-out-of-n scheme, $t \leq n$. If $t = n$, we call it an n-out-of-n scheme. In practice, this is often constructed so that a signer $\mathcal{S}_i \in \mathcal{U}, |\mathcal{U}| = n$ produces a partial signature that can be combined with at least $t - 1$ other participants in $\mathcal{U}$ to produce a valid signature.

2.3 Shamir Secret Sharing

At its most basic level, threshold cryptography is often based on Shamir's Secret Sharing method in [Sha73], where one splits a secret value into parts such that a subset of a certain size is required to reconstruct it. The scheme works as follows.

For an integer value D, construct a random polynomial $q(x)$ of degree $k-1$, with constant $a_0 = D$. The partial values are constructed as $q(i) = D_i, 1 \leq i \leq n$. With k or more of these values and their indices known, it is trivial to calculate a_0, but $k-1$ or fewer known values do not provide any information to help solve for a_0.

As discussed in [GKS23], an extension of Shamir's scheme to allow for values of $D \in R_q$ can be done. The method is mostly identical to the aforementioned scheme, except that the coefficients of the polynomial are sampled from R_q. Additionally, the points at which the polynomial is evaluated to derive the values of D_i must be selected more carefully. Specifically, the pairwise difference of the selected evaluation polynomials must all be invertible for the scheme to hold.

2.4 Commitment Schemes

Commitment schemes are a method for having clients commit to certain values that they cannot change at a later time. Said values should also be obscured, to ensure that the committed value cannot be derived easily. This is often done by adding some randomness ρ to a message x. We write that $x \in \mathbb{P}$ and $\rho \in \mathbb{R}$ for the message space and randomness space, respectively. Then, we define an abstract commitment scheme algorithm as $\texttt{Com}(x, \rho) := c$, where c is the commitment to a message x along with randomness ρ. A commitment scheme has the following two properties, formulated in [Sma16].

Definition 2.4.1 (Binding)**.** A commitment scheme is binding if an adversary cannot win the following game.

- The adversary chooses values $x \in \mathbb{P}$ and $\rho \in \mathbb{R}$.
- The adversary must then output a value $x' \neq x \in \mathbb{P}$ and $\rho' \in \mathbb{R}$ such that $\texttt{Com}(x, \rho) = \texttt{Com}(x', \rho')$.

Definition 2.4.2 (Hiding)**.** A commitment scheme is hiding if an adversary cannot win the following game.

- The adversary generates two different messages $x_0, x_1 \in \mathbb{P}$.
- The challenger generates $\rho \in \mathbb{R}$ and chooses one of the messages at random as $x \in \{x_0, x_1\}$.
- The challenger commits and computes $c := \texttt{Com}(x, \rho)$, then sends c to the adversary.
- The adversary has to accurately guess which of the messages x_0, x_1 was chosen as x.

The properties can be either information-theoretically or computationally bounded, depending on the assumed resources of the adversary. If the property is information-theoretical bounded, the adversary is assumed to have unlimited resources, while in the computational bound, the adversary's computational power is limited. The security of a commitment scheme is often based on the hardness of the weakest of these two properties. It is important to note that a commitment scheme cannot achieve information-theoretical security for both properties of hiding and binding. This can be shown by having a client commit to a value $c := \texttt{Com}(x, \rho)$. Now there has to exist some values $x' \neq x \in \mathbb{P}, \rho' \in \mathbb{R}$ such that $c := \texttt{Com}(x', \rho')$, otherwise the hiding property could be broken. If this has to exist, then a commitment can be opened to two messages which would break the binding property.

A commitment scheme is divided into two phases, commitment and opening. We illustrate these two phases with an example, where participants $\mathcal{P}_j$ commit to a value, send the commitment c, and then all other participants ensure that the commitments match the value that was committed to. We show it from a participant $\mathcal{P}_i$'s perspective. In the commitment phase, $\mathcal{P}_i$ computes $c_i := \texttt{Com}(x_i, \rho_i)$ using their message x_i and a randomly generated value ρ_i. The commitment c_i is then sent to all other participants. Once all participants have received all c_j from $\mathcal{P}_j, j \neq i$, they proceed to the opening phase. Participant $\mathcal{P}_i$ now sends their tuple (x_i, ρ_i) to all other participants. Finally, having received all such tuples, all participants can verify that $\texttt{Com}(x_j, \rho_j) = c_j$.

There is a subset of this type of scheme, known as a trapdoor commitment scheme. In this scheme, a participant with access to some special information, known as a trapdoor, can open a commitment ambiguously, thus overcoming the binding property [Fis01].

2.5 Homomorphic Encryption

Homomorphic encryption enables computations on encrypted data without the need to decrypt it first. For example, if a scheme allows multiplication, such that $Enc(a) \cdot Enc(b) = Enc(a \cdot b)$, where $Enc(x)$ is an arbitrary encryption function, we call the scheme multiplicatively homomorphic [Gen09]. Similarly, a scheme is considered additively homomorphic scheme if the relation holds for addition. Without the homomorphic property, performing an operation requires first decrypting the encrypted data a and b, performing the operation, then encrypting the result. A scheme which is both additively and multiplicatively homomorphic is a fully homomorphic encryption (FHE) scheme.

The first homomorphic encryption scheme was constructed in 2009 by Gentry in [Gen09] and introduces noise to the ciphertext in its construction. At some point, the accumulated noise from repeated operations might lead to decryption failure. Gentry in his original paper solves this problem by a technique known as bootstrapping, where the decryption algorithm is first converted into a circuit. Then, any ciphertext along with the public key of the scheme can be sent into this circuit, outputting the ciphertext without any noise. Bootstrapping is the component requiring the most computational resources in a FHE scheme [AP23], research since the original paper has been committed to develop more efficient

FHE schemes, given its numerous use cases.

One such use case is in distributed encryption, where each party has its own secret element randomly distributed from a set. Parties can then generate a mutual encrypted product, with each participant contributing their own element to the product without revealing their share of the secret. There is current work in establishing a standard for homomorphic encryption, this can be found in [ACC+18].

2.6 Classic Cryptography Hardness Problems

In order to accurately assess the security of encryption algorithms, it is common to try to reduce an algorithm in complexity to a known mathematical problem that is known to be hard, meaning it can not be solved in polynomial time. This reduction means that the security of an entire scheme can rely on the difficulty of a single mathematical problem. We define factorization and the discrete logarithm problems, both of which are essential for asymmetric cryptography. We also define the RSA problem, which can be reduced to the factorization problem. In other words, the RSA problem can be turned into an easy problem given that one can find the factors p, q [Sma16].

Definition 2.6.1 (Factorization Problem)**.** Given an integer N, known to be a product of two primes p and q, find p and q.

Definition 2.6.2 (RSA Problem)**.** Given public parameters (N, e), along with a ciphertext c, where e is chosen such that

$$\gcd(e, \phi(pq)) = 1$$

where ϕ denotes Euler's totient function. Find m such that

$$m^e = c \pmod{N}.$$

Definition 2.6.3 (Discrete Logarithm Problem)**.** Given g, h in a finite abelian group $(G, \cdot)$ of prime order q, find an integer $x \in [0, ..., q-1]$ such that

$$g^x = h$$

if such an integer x exists.

There are algorithms for finding solutions to these problems. For the factorization problem 2.6.1, the number field sieve is a typical example. For discrete logarithms, two examples are the Pohlig-Hellman and Pollard Rho algorithms [Sma16]. None of the aforementioned algorithms, nor any currently existing algorithm, can solve these three problems in polynomial time on classical computers.

2.7 Quantum Advancements

Developments in quantum computing has lead to new algorithms which can turn all three problems in 2.6.1, 2.6.2, 2.6.3 into easy problems; Shor's algorithm is perhaps the most well known of these [Sho94]. Any data encrypted using a scheme that can be reduced to one of these problems is no longer secure, given a quantum computer powerful enough to run these quantum algorithms. Craig Gidney and Martin Ekerå in [GE21] estimate, with their construction that includes adaptions of Shor's algorithm, that the upper bounded number of qubits required to break RSA-2048 is roughly twenty million. They also note that this number can change rapidly with advancements in constructing quantum circuits, handling error correcting codes, and the physical design of the qubits [GE21]. Algorithms which do not rely on the hardness of factoring, RSA, or discrete logarithms can also be affected by the progress of quantum computing, Grover's algorithm is a quantum algorithm that searches a list of size n in $\sqrt{n}$ operations [Gro96]. This is a more efficient exhaustive key search and a successful implementation could make smaller key sizes of AES not-secure.

In an attempt to find how far along quantum computing has come, one might be inclined to look at current records. Karamlou et al. in [KSK+21] have factored 1,099,551,473,989, and Li et al. in [LDC+17] have factored 291,311, respectively. However, these integers have a special form which makes them easy to factor, we show how this is done for the larger integer using Fermat's Factorization Method.

Example. *Find the factors of* 1,099,551,473,989.

$$n = 1{,}099{,}551{,}473{,}989$$
$$n = x^2 - y^2 = (x-y)(x+y)$$

For the first trial, take $\lceil \sqrt{n} \rceil$ *then compute the difference with our original* n *and look if the difference is a square.*

$$\begin{aligned}\lceil \sqrt{n} \rceil &= 1{,}048{,}595 \\ 1{,}048{,}595^2 - n = -36 \iff n &= 1{,}048{,}595 + 6^2 \\ n &= (1{,}048{,}595 - 6)(1{,}048{,}595 + 6) \\ &= (1{,}048{,}589)(1{,}048{,}561) \\ \{p, q\} &= \{1{,}048{,}589, 1{,}048{,}561\}\end{aligned}$$

This can be done for $n = 291{,}311$ and its respective factors are also found after one trial. Due to their special form, successfully factoring these integers on a quantum computer should not be seen as a projection of how close quantum computing is to breaking modern cryptography. Rather the papers show the possibility of factoring large integers that is possible with current quantum technology by performing pre-computations on classical computers. It is hard to say when or even if quantum computers will become capable of running quantum algorithms for relevant bit sizes. Despite this uncertainty it is best to take a proactive approach and look into quantum-safe alternatives, i.e. hardness problems that are not turned into easy problems given quantum algorithms. Some companies are investing heavily in quantum computing. IBM revealed a 433 qubit processor in late 2022, and are aiming to develop a 100,000 qubit machine within the next ten years [Bro23].

2.8 Zero-Knowledge Proof

A zero-knowledge proof (ZKP) consists of two parties, a prover and a verifier. We will name the prover Peggy and the verifier Victor, as is frequently done in other literature. Peggy then wishes to prove that she knows something, e.g. the secret key of an asymmetric scheme, to Victor. Additionally, Victor should not learn what the secret is. There are three criteria for a scheme to be a zero-knowledge proof, these are highlighted below. The properties of completeness and soundness include some small error margin due to ZKPs being probabilistic, the scheme can be designed to make this error negligible.

Definition 2.8.1 (Completeness)**.** If both Peggy and Victor are honest participants, then Victor will almost always accept Peggy's proof.

Definition 2.8.2 (Soundness)**.** If Peggy is disingenuous and Victor is honest, Victor should accept the proof only for some small probability.

Definition 2.8.3 (Zero-knowledge)**.** If Peggy is honest and the protocol is followed, Victor should learn nothing about the secret itself.

An efficient method of a ZKP is known as a Sigma protocol and consists of three steps. First Peggy issues a commitment, Victor then responds to said commitment with a challenge, and finally, Peggy provides a response. Victor is, after this three-step process, able to verify that Peggy knows what she claims to know. One oftentimes done simplification is to assume that Victor is an honest participant that behaves according to the protocol, such ZKPs are known as honest verifier zero knowledge proofs (HVZKP). Sigma protocols can be turned into a non-interactive proof, then referred to as non-interactive ZKP (NIZKP). In this paper, this is achieved by replacing the second step where Victor sends a challenge with a hash of some values in the scheme which both parties have access to, this way Peggy can create a complete proof on her own, and send it to Victor, and he can still be confident that the challenge used in the proof was chosen at random.

Schnorr's algorithm for signatures, which can be reduced to the discrete logarithm problem from 2.6.3, can be proven to be a Sigma protocol. We end this section with an example of this, where we abbreviate Peggy as P and Victor as V. We denote the direction data is sent, along with what data, with arrows and the name of the variable above it.

Example (Sigma protocol for the discrete logarithm problem)**.** *Peggy wants to prove to Victor that she knows x for $g^x = h$*

$$\begin{aligned}
&P \xrightarrow{r} V : r := g^k, k \overset{\$}{\leftarrow} G \\
&V \xrightarrow{e} P : e \overset{\$}{\leftarrow} G \\
&P \xrightarrow{s} V : s := k + x \cdot e \pmod q
\end{aligned}$$

Victor can verify that Peggy knows x by verifying that $r = g^s \cdot h^{-e}$, we will merely state that this fulfills all required criteria for a zero-knowledge proof.

2.9 Lattice-Based Cryptography

In an attempt to find a hardness problem that is quantum-safe, lattices are one of the potential candidates going forward. A lattice is a discrete version of a vector subspace [Sma16]. As an example, take two vectors

$$\boldsymbol{b}_1 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \text{ and } \boldsymbol{b}_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

and plot these with integers as coefficients for $\boldsymbol{b}_1, \boldsymbol{b}_2$ to create a two-dimensional lattice. See Figure 2.1. It is important to note that there are many possible bases for a lattice, the only requirement is that the vectors that form a basis are linearly independent.

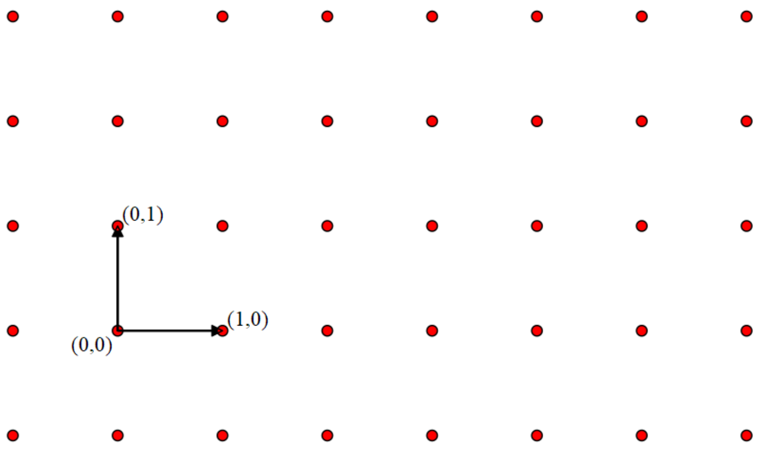


Figure 2.1: A two-dimensional lattice with basis vectors $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ and $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ marked out

We denote the non-zero minimum of any lattice L by

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

2.9.1 Shortest Vector Problem

The most well-known hard lattice problem is the Shortest Vector Problem (SVP). This problem is extensively studied and comes in several forms, in this paper we define only the basic version.

Definition 2.9.1 (Shortest Vector Problem)**.** Given the basis of a lattice L, find the shortest non-zero vector $\boldsymbol{x}$ in L. In other words, find $\boldsymbol{x}$ such that $\|\boldsymbol{x}\| = \lambda_1(L)$.

It is easy to reason that $\boldsymbol{x}$ is not unique by observing the vectors in Figure 2.1. For randomized reductions, this problem can be shown to be NP-hard, meaning it **cannot** be solved in polynomial time, if P $\neq$ NP [Ajt98].

2.9.2 Short Integer Solution Problem

Another lattice-based problem is the Short Integer Solution (SIS) problem, first formulated in [Ajt96]. The parameters n, m, q, B, are all positive integers. Here, n denotes the security parameter, m is application specific, q denotes the modulus and B denotes the bound. We then define the problem as follows.

Definition 2.9.2 (Short Integer Solution Problem)**.** Given $\boldsymbol{A} \overset{\$}{\leftarrow} \mathbb{Z}_q^{n\times m}$, find a non-zero vector $\boldsymbol{x} \in \mathbb{Z}^m$ such that

$$\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{0} \pmod q \quad \text{and} \quad \|\boldsymbol{x}\|_\infty \leq B.$$

Generally the parameters are chosen such that $m \ll n$. Additionally, B is chosen such that $B \ll q$. The name comes from the vector $\boldsymbol{x}$ being short, often chosen such that $\boldsymbol{x} \in \{-1, 0, 1\}^m$.

This problem can be extended to polynomial rings, then called R-SIS. This problem is analogous to SIS over $\mathbb{Z}_q$, we show the ring variant below.

Definition 2.9.3 (R-SIS Problem)**.** Given $\boldsymbol{A} \overset{\$}{\leftarrow} R_q^m$, find a non-zero vector $\boldsymbol{x} \in R^m$ such that

$$\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{0} \pmod q \quad \text{and} \quad \|\boldsymbol{x}\| \leq B.$$

2.9.3 Learning With Errors

One known lattice-based problem is the Learning with Errors problem (LWE) first formulated by Regev in [Reg09]. We start by fixing a modulo $q \geq 2$ and a dimension $n \geq 1$. We let m denote the number of samples, with $n > m$. We then define an error distribution $\mathcal{X}$ over $\mathbb{Z}_q$ [Reg10]. This distribution is generally described as either a discrete Gaussian distribution or a rounded continuous Gaussian distribution, with zero mean and a small standard deviation σ. There are then two variants of the LWE problem, we start by defining the LWE search problem.

Definition 2.9.4 (LWE Search Problem)**.** Let $\boldsymbol{A} \overset{\$}{\leftarrow} \mathbb{Z}_q^{m\times n}, \boldsymbol{s}, \boldsymbol{b} \overset{\$}{\leftarrow} \mathbb{Z}_q^n, \boldsymbol{e} \overset{\$}{\leftarrow} \mathcal{X}^m$. The problem is then to, given $(\boldsymbol{a}, \boldsymbol{b})$, find $\boldsymbol{s}$ such that

$$\boldsymbol{b} := \boldsymbol{A} \cdot \boldsymbol{s} + \boldsymbol{e} \pmod q.$$

The idea behind this problem is to introduce noise into an otherwise linear systems of equations. In Section 2.5 when we were talking about homomorphic encryption, the noise comes from Regev's construction based on LWE. Without this added noise part the problems could be easily solved using Gaussian elimination. Because $n > m$, the matrix has more rows than columns and will either have no solutions or one solution. The parameters are chosen such that it will have a unique solution.

The second variant is the LWE decision problem.

Definition 2.9.5 (LWE Decision Problem)**.** Let $\boldsymbol{e} \overset{\$}{\leftarrow} \mathcal{X}^m$. Given $(\boldsymbol{A}, \boldsymbol{b})$, where $\boldsymbol{A} \overset{\$}{\leftarrow} \mathbb{Z}_q^{m\times n}$, $\boldsymbol{s}$, and $\boldsymbol{b} \overset{\$}{\leftarrow} \mathbb{Z}_q^n$, determine if either

1. $\boldsymbol{b} := \boldsymbol{A} \cdot \boldsymbol{s} + \boldsymbol{e} \pmod q$.
2. $\boldsymbol{b} \overset{\$}{\leftarrow} \mathbb{Z}_q^n$.

To realize why LWE is believed to be hard, Regev in [Reg10] provides the following list of arguments.

1. Regev proves in [Reg09] that LWE is worst-case hard if the shortest vector problem (SVP) is hard. As was mentioned earlier, SVP is a well-studied problem, it and its variants are believed to be hard.
2. LWE is an extension of the learning parity with noise (LPN) mathematical problem [Reg10], which is also believed to be hard.
3. The most well-known algorithm for solving LPN is provided by Blum, Kalai and Wasserman in [BKW03] and runs in subexponential time. This algorithm has been applied to LWE in [ACF+15], further improvements to the original algorithm can be seen in [BGJ+21], [QLGZ20], [KF15].

2.9.4 Ring-Learning With Errors

Encryption schemes often use a variant of LWE based on polynomial rings, called R-LWE defined in [LPR12]. This variant was created due to the original LWE being inefficient for any practical means, simple primitives requiring quadratic overhead for key sizes and computation times [LPR12]. We begin by defining R_q as was done in the preliminaries, as

$$R_q = \mathbb{Z}_q[X]/\langle f(x) \rangle = \mathbb{Z}_q[X]/\langle x^N + 1 \rangle.$$

Where $N = 2^k$ for some positive integer k. We then define an error distribution $\mathcal{X}$ similarly to how we did for regular LWE, now outputting polynomials of degree smaller than the degree of N. Analogous to the general case, the coefficients are often distributed from either a discrete Gaussian distribution or a rounded continuous Gaussian distribution, with mean zero and a small standard deviation σ [ACC+18]. We can then define the ring version of the two LWE problems as follows.

Definition 2.9.6 (R-LWE Search Problem)**.** Let $a, s, b \leftarrow R_q$, and $e \leftarrow \mathcal{X}$. The problem is then to, given (a, b), find s such that

$$b := a \cdot s + e.$$

Definition 2.9.7 (R-LWE Decision Problem)**.** Let $a, s, b \overset{\$}{\leftarrow} R_q$, and $e \overset{\$}{\leftarrow} \mathcal{X}$. Determine if either

1. $b := a \cdot s + e$.
2. $b \overset{\$}{\leftarrow} R_q$.

We will in a subsequent section describe how the parameters and distributions are chosen, as many schemes used throughout this paper are based on R-LWE.

2.9.5 Implementations

Finding an implementation of a lattice-based threshold encryption fit for standardization is an ongoing process. The National Institute of Standards and Technology (NIST) initiated this process by issuing a request for post-quantum cryptographic algorithms as early as December 2016 [Nat16]. For key exchange and signatures, Kyber [BDK+18] and Dilithium [DKL+18] respectively are the leading candidates. Both of these algorithms are based on the hardness of LWE [Bou22], and are components of the Cryptographic Suite for Algebraic Lattices (CRYSTALS) suite. Sometimes the algorithms are referred to as CRYSTALS-Kyber & CRYSTALS-Dilithium.

2.10 Current State of Lattice-Based Threshold Signatures

In this section we aim to give an overview of GKS23 as presented in [GKS23] by providing details on the required underlying schemes. We include schemes here that were not implemented by us, and explain why that is the case. Then, we move on to briefly give some detail on two of these underlying schemes before ending this section by discussing similar work to GKS23.

2.10.1 GKS23

GKS23 is a t-out-of-n scheme requiring three underlying schemes, see Figure 2.2. The figure also shows which algorithms were used in this paper for the underlying schemes. GKS23 recommends using DOTT20 by Damgård et al. in [DOTT20] for homomorphic trapdoor commitment. For the three required ZKP, GKS23 recommends three different schemes. For proofs of linear relation, they recommend using BDLOP18 by Baum et al. in [BDL+18]. The remaining proofs are a proof of shortness by Lyubashevsky et al. in [LNP22], and a proof for straight-line extractability by Katsumata in [Kat21]. Lastly, GKS23 recommends using BGV11, by Brakerski et al. in [BGV11] for threshold homomorphic encryption.

Due to overall complexity of implementing the trapdoor functionality of DOTT20, our implementation does not include this scheme. Instead, we rely on BDLOP18, which is an efficient homomorphic commitment scheme without the trapdoor functionality. Our work also does not include a proof of shortness nor a proof for straight-line extractability. These were not included in order to prioritize on other aspects of implementation within the given time frame. Details on the security implications of not implementing DOTT20, as well as the proof of shortness and proof for straight-line extractability will be provided in subsequent chapters. In Figure 2.2, the missing schemes are denoted by a dotted border, and implemented schemes with a full border. There is a dotted line between the homomorphic trapdoor commitment scheme and BDLOP18 as it is missing the trapdoor functionality, thus only providing some of the necessary features.

GKS23 is evaluated in [GKS23] by estimations of performance, public key size and signature size, leaving a concrete implementation as follow-up work.

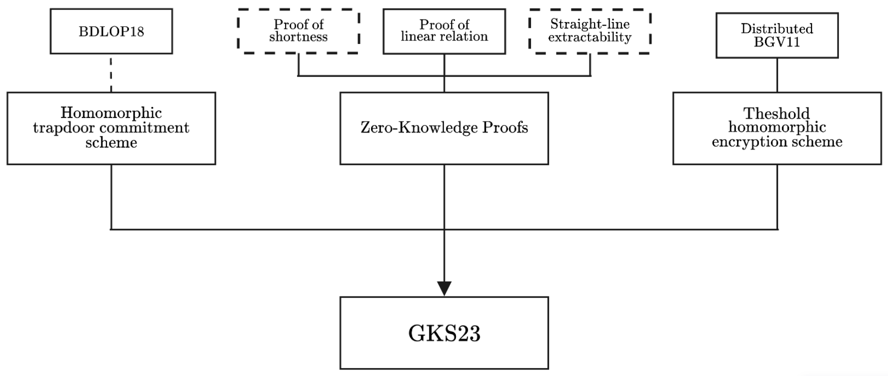

BDLOP18
Proof of shortness
Proof of linear relation
Straight-line extractability
Distributed BGV11
Homomorphic trapdoor commitment scheme
Zero-Knowledge Proofs
Theshold homomorphic encryption scheme
GKS23

Figure 2.2: GKS23 and its required underlying schemes. Which algorithm was used can be seen above each underlying scheme.

2.10.2 BDLOP18

BDLOP18 is an efficient additively homomorphic commitment scheme. The scheme allows commitments to vectors over R_q. BDLOP18 is the recommended commitment scheme in [GKS23] where trapdoor functionality is not required, as well as for various NIZKP. As our implementation does not have a trapdoor commitment scheme, BDLOP18 will be the only commitment scheme we implement.

2.10.3 BGV11

BGV11 is a FHE scheme that works for both LWE & R-LWE; our focus will be on the R-LWE version. The scheme leverages techniques from prior research to get efficient FHE without bootstrapping [BGV11]. As mentioned in Section 2.5, the bootstrapping is computationally heavy and adds a layer of complexity to FHE schemes. BGV11 instead reduces noise by performing modulus switching between two integers p and q. The BGV11 scheme is not a threshold scheme as presented in [BGV11], modifications can be made to support distributed key generation and threshold decryption, without loss of security, as can be seen in [GKS23].

2.10.4 Similar work

While no implementation of GKS23 has been published, Leevik et al. proposed and implemented a t-out-of-n scheme, LDB23, in [LDB23]. GKS23 and LDB23 differ in several key design choices. LDB23 utilizes Dilithium-G for its underlying signature scheme, similar to DOTT20's n-out-of-n solution, whereas GKS23 uses Lyubashevsky-type signatures [LDB23], [DOTT20], [GKS23]. Aside from these underlying signature schemes, LDB23 and GKS23 use the same commitment scheme and rely on the same Shamir Secret Sharing method to produce a t-out-of-n signature scheme.

Chapter 3

Methodology

For detailed performance data, such as measuring the execution time as well as the number of operations for various operations such as key generation and signature generation, an accurate model of the entire system was needed. This section aims to explain these modeling choices that were done for our concrete instantiation of GKS23. The implementation was created in Python, with everything other than polynomial operations implemented from the ground up. This was done to give us the most fine-grained analysis of what parts of the code took the most time to execute, as well as to exclude the possibility of different implementations of parts of the scheme utilizing different optimization strategies.

We begin by describing overarching initialization values, such as how polynomials were represented, defining distributions, and how we made a hash function that outputs a polynomial. We then present the underlying protocols that were needed, along with the optimal parameters for each protocol. Lastly, we describe how all of these components were integrated in order to create a complete version of GKS23.

3.1 Initialization

This section aims to provide a baseline for the schemes that were implemented. We first describe how our polynomial representation was done. Then, we define three types of distributions that were used in our schemes. Finally, we show our implementation of a hash function that maps a polynomial or array of polynomials into a deterministic polynomial d.

3.1.1 Polynomials

Representing polynomials was done using the CyPari2 library in [FDDK24], which is a Python interface to PARI/GP at [PAR23]. Utilizing this library allowed us to perform addition and multiplication for polynomials in R_q. Additionally, it supported vector and matrix addition and multiplication for large values of q and N in R_q.

3.1.2 Distributions

We generally required three distributions for our algorithms to sample polynomials from. One was a uniform distribution, one was discrete Gaussian, and one was ternary. In order to create a polynomial in R_q we sampled N independent one-dimensional samples as the coefficients of the polynomial.

Creating a discrete Gaussian distribution is a complex problem. In [ACC+18], it is stated that it is sufficient to choose each coefficient from a rounded continuous Gaussian distribution. Our rounding was done to the nearest integer, for values at exactly the halfway point we rounded to the nearest even value. It is a slight misnomer to call this implementation a discrete Gaussian distribution, as a specific integer $i \in R$ is not proportional to $\frac{e^{-(x-v)^2}}{2\sigma^2}$, though the coefficients of a polynomial will still have a bell-like shape. We plot these coefficients and overlay a normal distribution, both with standard deviation $\sigma = 4$ in Figure 3.1.

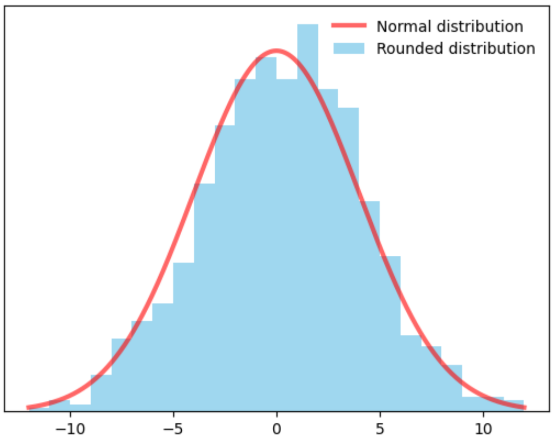


Figure 3.1: Rounded Gaussian distribution overlaid with a normal distribution, both with mean zero and standard deviation 4.

Our choice of $\sigma = 4 = \lceil \frac{8}{\sqrt{2\pi}} \rceil$ was based on the values in [ACC+18]. This ensured that no attacks were known against our scheme. Unless otherwise specified, $\sigma = 4$ was used as the standard deviation for our discrete Gaussian distributions. This value could change as new attacks are found or with time as understanding of the error standard deviation improves [ACC+18].

Our ternary distribution was used when we required small elements. We let all coefficients in a ternary polynomial be $-1, 0,$ or 1. Furthermore, we also limited the ℓ_1-norm of a polynomial to be equal to κ. We write the set of polynomials that we sample ternary polynomials from as

$$\mathcal{C} = \{c \in R_q : \|c\|_\infty = 1, \|c\|_1 = \kappa\}.$$

3.1.3 Hash Function

A hashing function was required for participants to be able to provide a NIZKP, mentioned in Section 2.8, as well as for some communication in GKS23. As we were working in polynomial rings, this function needed to work in a polynomial ring setting. We created a hash function H that takes as input a polynomial array of arbitrary size. This allowed for hashing single polynomials as well as matrices of polynomials, outputting a polynomial d deterministically, as shown in Figure 3.2.

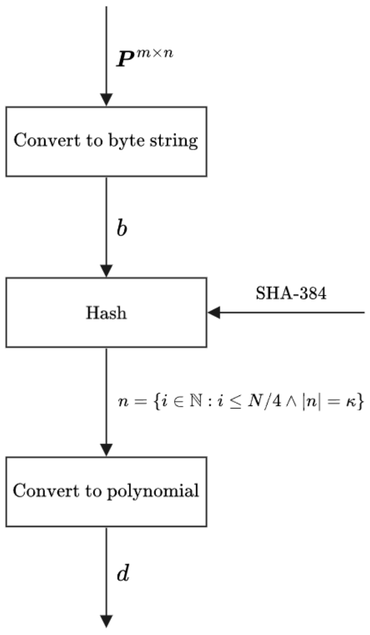

$\boldsymbol{P}^{m\times n}$
Convert to byte string
b
Hash
SHA-384
$n = \{i \in \mathbb{N} : i \leq N/4 \wedge |n| = \kappa\}$
Convert to polynomial
d

Figure 3.2: Our hash function $H(x)$, outputting a polynomial d from an input polynomial array $\boldsymbol{P}$.

The string representation of the input was encoded using UTF-8 into a sequence of bytes, which was then hashed into a 48-byte output using SHA-384. We chose to use SHA-384 as it provided 192-bit collision resistance security and 384-bit preimage resistance security in accordance with [Nat23]. We then took the 48-byte output and converted this to its integer representation, this was used as the seed for generating randomness to create a polynomial. For randomness, we used a 128-bit implementation of the permuted congruential generator defined in [One14]. Using this random generator seeded with the hash, we generated a sparse polynomial of at most degree $\frac{N}{4}$. The deterministic polynomial d was created such that it had an ℓ_1-norm κ, and ℓ_∞-norm 1.

3.2 BDLOP18

BDLOP18 was required for the commitment scheme as well as for the NIZKP proof of linear relations. From the commitment scheme we also required a proof of opening for a commitment, which cover the majority of required proofs. These proofs are interactive in the method presented in [BDL+18]. To make them non-interactive, the authors in [BDL+18] recommend using the Fiat-Shamir transform presented in [FS87]. This transform meant that we replaced the challenge value selected by the verifier, as was shown in Section 2.8, with a hash of values that were known to both the participant creating the NIZKP, and the participant who later had to verify that the proof was valid.

Parameters were initialized using recommended values from [BDL+18]. We needed a prime $q \equiv 2d+1 \pmod{4d}$ for adequate security, and to guarantee that all small elements (ℓ_2 or ℓ_∞) were invertible. The q used in BDLOP18 was the same as the prime modulo of the polynomial to be commited, as such the values of q and Q, explained in 3.4.1 had to be selected to ensure this held. The value of N was similarly the same as N in GKS23 and BGV11, that being the order of polynomials. Additionally, let a set of challenges that were small be defined as $\bar{\mathcal{C}}$. An element in $\bar{\mathcal{C}}$ was the difference of two polynomials c, c' sampled from $\mathcal{C}$, with $c \neq c'$. Any element $c - c'$ had an ℓ_∞-norm of at most 2, and would be invertible in R_q [BDL+18]. All instance parameters, along with a brief description and their values can be found in Table 3.1.

Parameter	Brief description	Value
q	The ring modulus (prime)	Set by GKS23
N	Degree of the ring polynomial	Set by GKS23
k	Width of the commitment matrices $\boldsymbol{A}_1, \boldsymbol{A}_2$	3
n	Height of $\boldsymbol{A}_1$	1
l	Message space dimension	1
κ	ℓ_1-norm for any $c \in \mathcal{C}$	36
$\bar{\mathcal{C}}$	$\{c - c' : c \neq c' \in \mathcal{C}\}$	N/A
β	Maximum ℓ_∞-norm for small vectors	1
S_β	$\{x \in R : \|x\|_\infty \leq \beta\}$	N/A
$\mathcal{N}$	Gaussian distribution for NIZKP	N/A
σ	Standard deviation	$11 \cdot \kappa \cdot \beta \cdot \sqrt{kN}$

Table 3.1: BDLOP18 parameters, a brief description, and their values.

These values were selected from the optimal table provided in [BDL+18]. They are considered optimal due to ensuring the same level of hardness for both hiding and binding properties. It is important to again note that the security of the commitment scheme is only as strong as the hardness of the weakest of the two properties, which was maximized using these values. The hardness of the two properties relied on the hardness of two mathematical problems, one for each property. The problem for the binding property was dependent on hardness of module-LWE, and for the hiding property on module-SIS. The module prefix to these problems is simply replacing singular ring elements from the definitions in 2.9.3, 2.9.6, 2.9.7 with module elements over the same ring, as discussed in [AD17].

We summarize the functions `Keygen`, `Com` and `Open` below.

Keygen: Create public commitment matrices $\boldsymbol{A}_1 \in R_q^{n\times k}, \boldsymbol{A}_2 \in R_q^{\ell \times k}$ as

$$\boldsymbol{A}_1 = \begin{bmatrix}\boldsymbol{I}_n & \boldsymbol{A}_1'\end{bmatrix}, \boldsymbol{A}_1' \xleftarrow{\$} R_q^{n\times(k-n)} \tag{3.1}$$

$$\boldsymbol{A}_2 = \begin{bmatrix}\boldsymbol{0}^{\ell\times n} & \boldsymbol{I}_\ell & \boldsymbol{A}_2'\end{bmatrix}, \boldsymbol{A}_2' \xleftarrow{\$} R_q^{\ell\times(k-n-\ell)} \tag{3.2}$$

with $\boldsymbol{I}_n$ denoting an identity matrix of dimension n over R_q. Similarly $\boldsymbol{0}^{\ell\times n}$ denotes a zero matrix of dimension $\ell \times n$ over R_q.

Commit: Commit to a message $\boldsymbol{x} \in R_q^\ell$ by first choosing a small commitment randomness $\boldsymbol{r} \xleftarrow{\$} S_\beta^k$, then output $\boldsymbol{c}$ as

$$\texttt{Com}(\boldsymbol{x}, \boldsymbol{r}) := \boldsymbol{c} = \begin{bmatrix}\boldsymbol{c}_1\\ \boldsymbol{c}_2\end{bmatrix} = \begin{bmatrix}\boldsymbol{A}_1\\ \boldsymbol{A}_2\end{bmatrix}\cdot \boldsymbol{r} + \begin{bmatrix}\boldsymbol{0}^n\\ \boldsymbol{x}\end{bmatrix} \tag{3.3}$$

Open: An opening consist of a commitment $\boldsymbol{c}$, a message $\boldsymbol{x} \in R_q^\ell$, randomness $\boldsymbol{r} \in R_q^k$, and a small polynomial $f \in \bar{\mathcal{C}}$. The opening is valid if the following holds.

$$f \cdot \boldsymbol{c} = \begin{bmatrix}\boldsymbol{A}_1\\ \boldsymbol{A}_2\end{bmatrix}\cdot \boldsymbol{r} + f \cdot \begin{bmatrix}\boldsymbol{0}^n\\ \boldsymbol{x}\end{bmatrix} \wedge \|r_i\|_2 \leq 4\sigma\sqrt{N}, \forall i. \tag{3.4}$$

It is important to distinguish the commitment randomness from the opening randomness, both denoted $\boldsymbol{r}$. This difference arises due to it being easier to use the ℓ_∞-norm when committing, but most efficient ZKPs demonstrate knowledge of small vectors in the ℓ_2-norm [BDL+18]. We show how the proofs of linear relation for opening and proof of sum were constructed in Figures 3.3 and 3.4.

With these two proofs, the vast majority of required proofs for the GKS23 and distributed BGV11 schemes could be implemented.

3.3 Distributed BGV11

A distributed version of the BGV11 scheme from [BGV11] was needed for threshold homomorphic encryption. We begin by showing non-distributed BGV11 and its three algorithms (`KGen`, `Enc`, `Dec`), and verify that this scheme is correct for a plaintext $m \in R_q$. We then move on to present a distributed version as presented in [GKS23].

Let q, Q be two primes, with $q \ll Q$. We denote two polynomial rings R_q, R_Q for a fixed dimension N. In this scheme, q is referred to as the plaintext modulo, while Q is referred to as the ciphertext modulo. The naming of these parameters may seem odd, but is done to simplify the understanding of the relation between BGV11 and GKS23. In our presentation of BGV11, R_q is the plaintext modulo, NIZKP proof of opening

Prerequisites: $\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_1 \\ \boldsymbol{A}_2 \end{bmatrix}$ as in 3.1, 3.2. $\boldsymbol{r} \in S_\beta^k$. $\boldsymbol{c} = \begin{bmatrix} \boldsymbol{c}_1 \\ \boldsymbol{c}_2 \end{bmatrix}$ as in 3.3.

Commitment:

$$\boldsymbol{y} \overset{\$}{\leftarrow} \mathcal{N}_\sigma^k$$
$$t := \boldsymbol{A}_1 \cdot \boldsymbol{y}$$

Challenge: For a hash function H outputting a polynomial, compute

$$d := H(t)$$
$$\boldsymbol{z} = \boldsymbol{y} + d \cdot \boldsymbol{r}$$

Response: Verify that

$$\boldsymbol{A}_1 \cdot \boldsymbol{z} = \boldsymbol{t} + d \cdot \boldsymbol{c}_1$$
$$\|z_i\|_2 \leq 2\sigma\sqrt{N}, \forall i.$$

Figure 3.3: NIZKP proof of opening from BDLOP18 utilizing a Fiat-Shamir transform.

NIZKP proof of sum

Prerequisites: $\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_1 \\ \boldsymbol{A}_2 \end{bmatrix}$ as in 3.1, 3.2. $\boldsymbol{r}, \boldsymbol{r}', \boldsymbol{r}'' \in S_\beta^k$.

$\boldsymbol{c} = \begin{bmatrix} \boldsymbol{c}_1 \\ \boldsymbol{c}_2 \end{bmatrix} = \texttt{Com}(x, r)$, $\boldsymbol{c}' = \begin{bmatrix} \boldsymbol{c}_1' \\ \boldsymbol{c}_2' \end{bmatrix} = \texttt{Com}(x', r')$,

$\boldsymbol{c}'' = \begin{bmatrix} \boldsymbol{c}_1'' \\ \boldsymbol{c}_2'' \end{bmatrix} = \texttt{Com}(g \cdot x + g' \cdot x', r'')$ as in 3.3.

Commitment:

$$\begin{gathered}
\boldsymbol{y}, \boldsymbol{y}', \boldsymbol{y}'' \xleftarrow{\$} \mathcal{N}_\sigma^k \\
t := \boldsymbol{A}_1 \cdot \boldsymbol{y} \\
t' := \boldsymbol{A}_1 \cdot \boldsymbol{y}' \\
t'' := \boldsymbol{A}_1 \cdot \boldsymbol{y}'' \\
\boldsymbol{u} := g \cdot \boldsymbol{A}_2 \cdot \boldsymbol{y} + g' \cdot \boldsymbol{A}_2 \cdot \boldsymbol{y}' - \boldsymbol{A}_2 \cdot \boldsymbol{y}''
\end{gathered}$$

Challenge: For a hash function H outputting a polynomial, compute

$$\begin{gathered}
d := H(t, t', t'') \\
\boldsymbol{z} = \boldsymbol{y} + d \cdot \boldsymbol{r} \\
\boldsymbol{z}' = \boldsymbol{y}' + d \cdot \boldsymbol{r}' \\
\boldsymbol{z}'' = \boldsymbol{y}'' + d \cdot \boldsymbol{r}''
\end{gathered}$$

Response: Verify that

$$\begin{gathered}
\boldsymbol{A}_1 \cdot \boldsymbol{z} = \boldsymbol{t} + d \cdot \boldsymbol{c}_1 \\
\boldsymbol{A}_1 \cdot \boldsymbol{z}' = \boldsymbol{t}' + d \cdot \boldsymbol{c}_1' \\
\boldsymbol{A}_1 \cdot \boldsymbol{z}'' = \boldsymbol{t}'' + d \cdot \boldsymbol{c}_1'' \\
g \cdot \boldsymbol{A}_2 \cdot \boldsymbol{z} + g' \cdot \boldsymbol{A}_2 \cdot \boldsymbol{z}' - \boldsymbol{A}_2 \cdot \boldsymbol{z}'' = (g \cdot \boldsymbol{c}_2 + g' \cdot \boldsymbol{c}_2' - \boldsymbol{c}_2'') \cdot d + u \\
\|z_i\|_2 \leq 2\sigma\sqrt{N}, \forall i.
\end{gathered}$$

Figure 3.4: NIZKP proof of sum, based on proof of linear relation from BDLOP18 utilizing a Fiat-Shamir transform.

and in GKS23 it is the ciphertext modulo. The choice for this is that aspects of the signature itself need to be encrypted in the process of generating a signature, and as such the encryption plaintext modulo must be the same size as the signature ciphertext modulo.

Let D_{Enc} be a distribution over R_Q with an ℓ_∞-norm bounded by predefined values B_{Enc}. The calculation of B_{Enc} is presented in [GKS23], for brevity we will simply state that for our chosen security parameters, the distribution should be bounded by 2^7. The standard deviation should be then be approximately 2^5.

Keygen: Sample a uniform element $a \in R_q, s, e \leftarrow \mathcal{C}$, output public key $\mathfrak{pk} := (a, b) = (a, as + pe)$, secret key $\mathfrak{sk} := s$.

Enc: Input $\mathfrak{pk}, m \in R_q$. Sample $r, e', e'' \leftarrow D_{Enc}$, output ciphertext $(u, v) = (ar + pe', br + pe'' + m)$. Also computes a ZKP π_{enc} proving that r, e', e'' are all bounded by D_{Enc}, that m is bounded by q, that $u = ar + pe'$ and that $v = br + pe'' + m$. Implementation for proof of sum are covered in figure 3.4 and for proof of bounded values is discussed in 4.2.2.

Dec: Input $\mathfrak{sk}, (u, v)$. Output $(v - \mathfrak{sk} \cdot u \bmod Q) \bmod q$

We verify that the scheme is correct by showing that it works for decryption, which it will iff $\|v - su\|_\infty \leq B_{Dec} < \frac{q}{2}$ [GKS23]. We omit modulus operations in all intermediate steps for brevity.

$$\begin{aligned}
&(v - \mathfrak{sk} \cdot u \bmod Q) \bmod q \\
&= v - su \\
&= br + qe'' + m - su \\
&= (as + qe)r + qe'' + m - su \\
&= (as + qe)r + qe'' + m - s(ar + qe') \\
&= (qer + qe'' + m - sqe' \bmod Q) \bmod q \\
&= m
\end{aligned}$$

We then define $\mathcal{E}_\mathcal{T} = (\texttt{DKGen}, \texttt{Enc}, \texttt{TDec}, \texttt{Comb})$ as the algorithms for the distributed extension of BGV11 with threshold decryption. We have the same parameters as for the non-distributed version, with the addition of $\mathcal{U}$ denoting a set of participants in the scheme of at least size t. `Enc` in $\mathcal{E}_\mathcal{T}$ is identical to the non-distributed version above, as such it is omitted from the description.

Committing to a value was done using the BDLOP18 commitment scheme, and each commitment got a uniformly randomly distributed element from S_β defined in Section 3.2, denoted ρ_x for a message x. We show the protocol for a participant $\mathcal{P}_i$, and denote all participants as $j \in [n]$. Additionally, if $\mathcal{P}_i$ receives a value from all other participants, we write $\mathcal{P}_j, j \neq i$.

DKGen$_{\mathcal{E}_\mathcal{T}}$:

1. Sample $a_i \overset{\$}{\leftarrow} R_Q$, compute $h_{a_i} := H(a_i)$ and broadcast to all other parties. When $\mathcal{P}_i$ has received h_{a_j} from all participants $\mathcal{P}_j, j \neq i$, they broadcast a_i. They verify that $h_{a_j} := H(a_j)$, then define the public key as $\sum_{j \in [n]} a_j$.

2. Sample $s_i, e_i \overset{\$}{\leftarrow} \mathcal{C}$ as the decryption key share and error noise share, respectively. Set $b_i = as_i + qe_i$. Then compute and broadcast $h_{b_i} := H(b_i)$.
3. Commit to s_i as c_{s_i} and e_i as c_{e_i}. Compute t-out-of-n Shamir secret shares $\{s_{i,j}\}$ and $\{e_{i,j}\}$ of s_i and e_i, respectively. Set $b_{i,j} := as_{i,j} + qe_{i,j}$ for all $j \in [n]$.
4. $\mathcal{P}_i$ commits to all individual key shares as $c_{s_{i,j}} := \texttt{Com}(s_{i,j}, \rho_{e_{i,j}})$ and $c_{e_{i,j}} := \texttt{Com}(e_{i,j}, \rho_{e_{i,j}})$ for all $j \in [n]$.
5. $\mathcal{P}_i$ now computes a proof π_{sk_i} for the key s_i and error e_i, proving that both values are bounded, that b_i is indeed the sum of as_i and qe_i, and that each $b_{i,j}$ is the sum of $as_{i,j}$ and $qe_{i,j}$. $\mathcal{P}_i$ then broadcasts π_{sk_i}, c_{s_i}, c_{e_i}, $\{c_{s_{i,j}}\}$, $\{c_{e_{i,j}}\}$, b and $\{b_{i,j}\}$ to all other participants.
6. For each other party j, $\mathcal{P}_i$ sends $(s_{i,j}, \rho_{s_{i,j}})$ over a secure channel.
7. After receiving all data from the previous two steps from $\mathcal{P}_j, j \neq i$, $\mathcal{P}_i$ verifies that $h_{b_j} := H(b_j)$ for $j \in [n]$ and that a reconstruction of $b_{i,j}$ outputs b_i. They abort if any of these fail.
8. $\mathcal{P}_i$ verifies that all proofs π_{sk_j} are valid, and that $c_{s_{j,i}}$ is a valid commitment for $s_{j,i}$, aborting otherwise.
9. Lastly, $\mathcal{P}_i$ computes $b := \sum_{j \in [n]} b_j$, $sk_i := \sum_{j \in [n]} s_{j,i}$, $\rho_i := \sum_{j \in [n]} \rho_{j,i}$, and $c_j := \sum_{k \in [n]} c_{s_{j,k}}$. $\mathcal{P}_i$ now outputs the public key $\mathfrak{pk} := (a, b, \{c_j\}_{j \in [n]})$ and the secret key $\mathfrak{sk}_i := (s_i, \rho_{s_i})$.

TDec$_{\mathcal{E}_\mathcal{T}}$ **:** Input $\mathfrak{sk}_i$, (u, v), $\mathcal{U}$. Sample $E_i \overset{\$}{\leftarrow} R_Q$, $\|E_i\|_\infty \leq 2^{sec} B_{Dec}$ for statistical security parameter sec and noise-bound B_{Dec}. With λ_i the Lagrange coefficient for $\mathcal{P}_i$ with respect to $\mathcal{U}$, compute $d_i := \lambda_i s_i u + qE_i$. $\mathcal{P}_i$ commits to sk_i and E_i as c_{sk_i} and c_{E_i}, respectively. Then, $\mathcal{P}_i$ computes a proof π_{ds_i} for a proof of relation with respect to s_i and E_i, and that E_i is correctly bounded. Output $ds_i := (d_i, \pi_{ds_i}, c_{E_i}, c_{sk_i})$

Comb$_{\mathcal{E}_\mathcal{T}}$ **:** Input v from `Enc` and partial decryption shares $\{ds_j\}_{j \in \mathcal{U}}$. Ensure all proofs π_{ds_j} are valid, aborting with output 0 otherwise. Then output plaintext $ptx := (v - \sum_{j \in \mathcal{U}} d_j) \mod q$.

There was a small change to the underlying mathematics that had to be made to ensure that our implementation worked correctly. If a small number n is multiplied by $q \pmod{Q}$, such that $n \cdot q < Q$, then $(n \cdot q \pmod{Q}) \pmod{q} \equiv 0$. However, this condition will not hold if $0 < n \cdot q < Q$ is not satisfied, which is why the error shares e_i must be small. Generally, integers modulo Q is done in the range $[-\frac{Q-1}{2}, \frac{Q-1}{2}]$. Due to how Python handles modulo operations, our range was instead $[0, Q-1]$ for the integer coefficients. Using the interval $[-\frac{Q-1}{2}, \frac{Q-1}{2}]$ allowed for negative error shares to be used without the equivalence breaking, as the size requirement was changed from $0 < n \cdot q < Q$ to $-\frac{Q-1}{2} < n \cdot q < \frac{Q-1}{2}$. To allow us to still use negative error shares, we added $\frac{Q-1}{2}$ to each coefficient of the result of computing $(v - \sum_{j \in \mathcal{U}} d_j)$ before reducing it by modulo q. This added an error of $Q \pmod q$ to each coefficient, which was then subtracted.

3.3.1 Assumption for NIZKP

For the proof π_{sk_i} calculated in `DKGen` step 5, an assumption had to be made regarding completeness. We made the assumption that if b is proven reconstructable by any subset of size t in a set $\{b_i\}$ of size n, as well as being properly computed from corresponding s and e. Then each b_i is proven to be properly computed from corresponding s_i and e_i, and each s and e were assumed to be correctly secret shared into s_i and e_i. This assumption was made as there is otherwise simply not enough information sent between parties to prove the reconstruction of s and e, and the security of the scheme would be affected by sending other information. In the revised publication of GKS23 at [GKS24], an argument is provided for how the knowledge available is enough to construct a proof, but it does not construct such a proof and the argument is not fully explained. For the purpose of our implementation, however, we assumed that the argument is correct and that this proof was sufficient.

3.3.2 Communication Between Participants

Communication between participants was handled by a centralized controller issuing a request to all contributing participants sequentially. Each participant sent their data along with an identifier of who they were to the controller. The controller then split the data into the parts as the step dictated. We will illustrate this process using two examples, both from steps in `DKGen` above.

In the first example, when sharing a hash of each participant's $h_i = H(a_i)$, $0 \leq i \leq n$, the controller request for all n participants for their hashed value and put all of these in a list. Then, the controller sends the list of hashes back to all participants, where all participants now have access to all hashes. This is repeated for a, and each participant can now verify that $H(a_i) = h_i$, using the identifier to map which values belong together. Our second example of communication was for secret sharing and was done as follows.

1. Each participant $\mathcal{P}_j, j \in [n]$ generated their secret shares for s_j, e_j, which were sent to the controller.
2. The controller sends one share to each participant, along with a participant identifier, showing whose s_j this is a secret share of.
3. This process is repeated for all participants, which led to all participants holding one share of each s_j, e_j.
4. At a later step, these shares are to be reconstructed. The controller begin by issuing a request to all participants for all shares they were holding.
5. The controller mapped all shares back to their original owner, adding an identifier of who the holder was.

6. Each participant would then attempt to reconstruct their s_j, e_j, using all legitimate combinations.
7. If any combination failed in the reconstruction, the program terminated and raised an error, detailing which user attempted to reconstruct, as well as whose shares failed. Otherwise, the reconstruction succeeded and the program continued as normal.

As showcased by these examples, communication was not accurately modeled, as no trusted third party would be involved in the key generation process in a real, distributed implementation. However the details of participant to participant communication of the protocol (outside of the size of messages) was not considered to be an area of interest to this report or implementation.

3.4 GKS23

With all underlying algorithms explained, we can now define the actively secure t-out-of-n threshold signature scheme that was implemented and tested.

3.4.1 Parameters

We began by setting up what parameters were needed for GKS23, as well as a brief description parameters that were used in Table 3.2. Then, in Table 3.3, we show how these values were instantiated for two different scenarios for a bounded number of signatures as described in [GKS23].

Parameter	Brief description
q	The GKS ciphertext ring modulus (prime)
p	The GKS plaintext ring modulus (prime $\ll q$)
Q	The BGV ciphertext ring modulus (prime $\gg q$)
N	Degree of the ring polynomial
σ_r	Standard deviation for distribution D_r
κ	ℓ_1-norm for any $c \in \mathcal{C}$

Table 3.2: Parameter notation for GKS23 and distributed BGV11.

The first scenario is for a single signature generation for a given key generation, meaning a signature is only used once per key generation. This type of signature is used for Bitcoin transactions. The second scenario allows 365 signatures for a given key, or one signature per day for a whole year before requiring a new instantiation of the scheme. This scenario can be used in a variety of ways where a service is used at most once a day, such as when authenticating a user.

The values for q, N, and σ_r were selected to match those recommended in [GKS24] for setting a secure scheme for a bounded number of signatures with the same key. The value for p, being the size of the plaintext to sign, was selected to be as large as possible while still maintaining a correct scheme. Our κ was set to 36, equivalent to κ for BDLOP18. This selection of parameters was chosen to make the

Parameter	Value (One signature)	Value (365 signatures)
q	$2^{20} - 143$	$2^{24} - 879$
p	$2^{16} - 39$	$2^{16} - 39$
Q	$2^{35} - 975$	$2^{36} - 527$
N	2^{10}	2^{10}
σ_r	$\approx 2^{15.2}$	2^{11}
κ	36	36

Table 3.3: Parameter initialization for GKS23 and distributed BGV11. Values are set for one signature generation per key, or for 365 signature generations per key.

underlying R-SIS and R-LWE problems hard. This instantiation provided roughly 128 bits of security [GKS24]. Q was set to be sufficiently large to allow the BGV scheme to function with q as plaintext modulo. It was selected to be as small as possible without causing encryption to fail, and a value of $2^{36} - 5$ was found to be sufficient for 365 signatures per key. A larger Q can be used, but increasing its size will negatively impact the execution time of the scheme and have no benefit to the scheme, as such the smallest possible Q was desirable.

3.4.2 Threshold Signature Scheme

We define $\mathcal{TS}$ as our scheme with algorithms (`KGen`, `Sign`, `Vrfy`), utilizing $\mathcal{E}_\mathcal{T}$ from Section 3.3. We use the same notation for broadcasts, sharing of data, and commitment randomness as in the distributed BGV11 case. All steps for how the protocol works for a signer $\mathcal{S}_i$ is shown below.

KGen$_{\mathcal{TS}}$ **:**

1. $\mathcal{S}_i$ invokes `DKGen` from $\mathcal{E}$, with input t, n, d. They obtain $\mathfrak{pk}_\mathcal{E}$ and $\mathfrak{sk}_i$ from $\mathcal{E}$.

2. Sample $a_i \xleftarrow{\$} R_q$ uniformly, compute $h_{a_i} := H(a_i)$ and broadcast h_{a_i}. After $\mathcal{S}_i$ receives h_{a_j} from signers $\mathcal{S}_j, j \neq i$, they broadcast a_i. They then verify that all $h_{a_j} = H(a_j), j \in [n]$, aborting if any fail with output j. Otherwise, compute $a := \sum_{j \in [n]} a_j$ and $\boldsymbol{a} := \begin{bmatrix} a & 1 \end{bmatrix}$.

3. Sample $s_{i,1}, s_{i,2}$ as short signing key pieces. Set $\boldsymbol{s}_i := \begin{bmatrix} s_{i,1} & s_{i,2} \end{bmatrix}$. Compute $y_i := \langle \boldsymbol{a}, \boldsymbol{s}_i \rangle$ and its corresponding hash $h_{y_i} := H(y_i)$, proceed to broadcast h_{y_i}. Upon receiving h_{y_j} from signers $\mathcal{S}_j, j \neq i$, $\mathcal{S}_i$ encrypts $\boldsymbol{s}_i$ as $ctx_{\boldsymbol{s}_i} := \texttt{Enc}(\mathfrak{pk}_\mathcal{E}, \boldsymbol{s}_i)$. They compute a proof $\pi_{\boldsymbol{s}_i}$ proving that $\boldsymbol{s}_i$ is correctly bounded and the same value used to calculate y_i and broadcast the tuple $(y_i, ctx_{\boldsymbol{s}_i}, \pi_{\boldsymbol{s}_i})$.

4. Having received tuples from the step above for all $\mathcal{S}_j, j \neq i, \mathcal{S}_i$ verifies that $h_{y_j} = H(y_j)$ and that their proof $\pi_{\boldsymbol{s}_j}$ is valid for $(ctx_{\boldsymbol{s}_j}, y_j)$. They abort if any of these verification steps fail with output j. Otherwise, compute

$y := \sum_{j\in[n]} y_j$ and $ctx_{\boldsymbol{s}} = \sum_{j\in[n]} ctx_{\boldsymbol{s}_j}$, then define the public key $\mathfrak{pk} = (\boldsymbol{a}, y)$, the secret key $\mathfrak{sk}_i$, and auxiliary information $aux = (aux_{\mathcal{E}}, \mathfrak{pk}_{\mathcal{E}}, ctx_{\boldsymbol{s}})$.

Sign$_{\mathcal{TS}}$ **:**

1. Input a message μ to be signed, with $\mathcal{S}_i \in \mathcal{U}$. Sample signature randomness $r_{i,1}, r_{i,2} \overset{\$}{\leftarrow} D_r$ along with a commitment randomness ρ_i. $\mathcal{S}_i$ then computes $\boldsymbol{r}_i := \begin{bmatrix} r_{i,1} & r_{i,2} \end{bmatrix}, w_i := \langle \boldsymbol{a}, \boldsymbol{r}_i \rangle$, and commitment $c_i := \texttt{Com}(w_i, \rho_i)$. They then encrypt $\boldsymbol{r}_i$ as $ctx_{\boldsymbol{r}_i} := \texttt{Enc}(\mathfrak{pk}_{\mathcal{E}}, \boldsymbol{r}_i)$ and compute a proof $\pi_{\boldsymbol{r}_i}$ proving that $\boldsymbol{r}_i$ is correctly bounded and the same value used to calculate w_i. Lastly $\mathcal{S}_i$ broadcasts the tuple $(ctx_{\boldsymbol{r}_i}, com_i, \pi_{\boldsymbol{r}_i})$ to all $j \in \mathcal{U}\backslash\{i\}$.

2. Having received tuples from the step above for all users $j \in \mathcal{U}\backslash\{i\}, \mathcal{S}_i$ verifies that $\pi_{\boldsymbol{r}_j}$ is valid for $(ctx_{\boldsymbol{r}_j}, c_j)$, aborting if any fail with output j. If all succeed, proceed and define $c := \sum_{j\in\mathcal{U}} c_j$, the challenge $c := H(c, \mathfrak{pk}, \mu)$. Then, compute the encryption of the signature as $ctx_{\boldsymbol{z}} := c \cdot ctx_{\boldsymbol{s}} + \sum_{j\in\mathcal{U}} ctx_{\boldsymbol{r}_j}$, and decrypt its own share $ds_i := \texttt{TDec}(ctx_{\boldsymbol{z}}, \mathfrak{sk}_i, \mathcal{U})$. Broadcast (ds_i, w_i, ρ_i) to all $j \in \mathcal{U}\backslash\{i\}$.

3. With $\mathcal{S}_i$ having all $(ds_j, w_j, \rho_j)\forall j \in \mathcal{U}\backslash\{i\}$, they verify that $\texttt{Open}(c_j, w_j, \rho_j) = 1$ and abort with output j if any fail. $\mathcal{S}_i$ then attempts to combine decryptions as $\boldsymbol{z} := \texttt{Comb}(ctx_{\boldsymbol{z}}, \{ds_j\}_{j\in\mathcal{U}})$, aborting with output j if $\boldsymbol{z} = \perp$ and $\texttt{Comb}$ aborts with output j. $\mathcal{S}_i$ computes $\rho := \sum_{j\in\mathcal{U}} \rho_j$, then outputs $\sigma := (c, \boldsymbol{z}, p)$ as their signature.

Vrfy$_{\mathcal{TS}}$ **:** Input σ and μ, verify that $\|\boldsymbol{z}\| \leq B_{\boldsymbol{z}}$ and $\|\rho\| \leq B_\rho$. Compute $w^* := \langle \boldsymbol{a}, \boldsymbol{z} \rangle - cy$ and derive $c^* := H(\texttt{Com}(w^*, \rho), \mathfrak{pk}, \mu)$. Output 1 iff all checks hold and if $c = c^*$, 0 otherwise.

3.4.3 Implementation

The steps of key generation, signing and verification was implemented in the same way as the functions of distributed BGV11, creating an expanded participant object capable of both distributed encryption and signature creation.

The short signing key pieces $s_{i,1}, s_{i,2}$ in step 3 of `KGen` were polynomials with coefficients distributed from our discrete Gaussian distribution, with $\sigma = 4$.

One change done to the implementation was the manner in which π_{ctx} of was verified during signature generation. Rather than adding the zero-knowledge proofs together and taking into account the multiplication with c that occurs when calculating ctx_z, which is a complicated set of operations both to implement and to execute, the zero-knowledge proof of a ciphertext is verified when it is used in a homomorphic calculation of a new ciphertext, and no zero knowledge proof of this new ciphertext is calculated. This should have no impact on security assuming that the homomorphic operations on ciphertexts are correct. All internal linear relations will still hold and, since no ciphertext that is transmitted to another participant is ever calculated from other ciphertexts, the only thing the proofs of opening would be used for would be to verify that all proofs of opening in the original ciphertexts held.

The one exception, the only proof in a calculated ciphertext that may fail even if the same proof holds in all ciphertexts used to calculate it, would be the proof of shortness. Since the variables checked to ensure shortness would be added together, and as such the variables in the calculated ciphertext may break shortness. However, since our implementation lacks an implementation of proofs of shortness, this has no impact on this implementation, and is only noted for completeness.

Chapter 4

Results

As all underlying schemes and GKS23 interoperate tightly, our implementation included all of the underlying schemes described in the previous section. This allowed a comprehensive analysis of GKS23 for all selected parameters. Furthermore, it allowed us to avoid relying on an imported scheme in a black-box fashion. Due to the required interoperability, importing a scheme could be detrimental and could possibly skew our analysis. Additionally, implementing the scheme in its entirety allowed us to ensure consistent handling of polynomials, which was helpful when troubleshooting and ensuring correctness of the schemes.

4.1 Implementation

The full implementation is available at [GP24].

4.2 Missing Schemes

In this section, we highlight what could not be achieved due to unimplemented schemes. We begin by providing an insight into what the missing scheme does. Then, we detail what consequences the absence of the scheme had on our implementation.

4.2.1 DOTT20

DOTT20 was the suggested trapdoor homomorphic commitment scheme, as mentioned in Section 2.10. They accomplish this by expanding on BDLOP18 [BDL+18], with new parameters and adding a trapdoor scheme for lattices. They recommend a scheme by Micciancio and Peikert in [MP12], known as MP12 for trapdoor functionality.

DOTT20 was utilized in signature generation for GKS23, in step 1 of $\texttt{Sign}_{\mathcal{TS}}$. In the original scheme, a commitment key was computed as $ck = H(\mathfrak{pk}, \mu)$. This commitment key was then used to commit to w_i, ρ_i in the same step, replacing $c_i := \texttt{Com}(w_i, \rho_i)$ with $c_i := \texttt{Com}_{ck}(w_i, \rho_i)$.

The absence of DOTT20 makes our implementation lack the equivocation property, present in the original GKS23 scheme. This property is required to ensure that dishonest participants can not pretend to have access to certain or different information than the information they are holding. A number of security games are presented in [GKS24], without DOTT20 our implementation would not hold for the scenario presented in $\boldsymbol{G}_3$.

4.2.2 ZKP Proof of Shortness

We required a proof of shortness for a complete GKS23 scheme to ensure that secret shares were correctly bounded. Unlike the proof of opening and proof of sum, a proof of shortness cannot easily be constructed using BDLOP18. Instead, [GKS23] recommends the proof of shortness scheme by Lyubashevsky et al. in [LNP22]. Implementing this scheme would have required a significant portion of the time available, as it differs substantially from all other schemes.

Without this proof a dishonest participant could run the scheme with long secret and error shares, which for a real-world application would obviously be unacceptable. The missing proof of shortness also resulted in less data transmitted, which led to our implementation requiring less communication than would be necessary in a real-world implementation.

4.2.3 ZKP Straight-Line Extractability

The property of straight-line extractability was required for concurrent security [GKS23]. This was needed due to our use of the Fiat-Shamir transform to turn ZKP into NIZKP. However, this transform is only secure in the classical random oracle model (ROM), not in a quantum random oracle model. In a ROM, a function is modeled as a black box that responds to every query with uniformly random output. This method is commonly used to analyze the security of a system. The authors of GKS23 recommend the technique proposed by Katsumata in [Kat21] to achieve this.

Katsumata's technique is based on existing efficient NIZKP to make them provably quantum safe, using smaller overhead compared to other algorithms [Kat21]. This technique was meant to be utilized in combination with the aforementioned proof of shortness. As our implementation did not include the straight-line extractability property, our implementation could not be said to be provably quantum-safe using a quantum ROM.

4.3 Communication Between Participants

Transmission analysis was done by counting messages exchanged by the participants during $\texttt{KGen}_{\mathcal{TS}}$ and $\texttt{Sign}_{\mathcal{TS}}$. The number and size of messages sent by each participant during key generation depends on the number of participants in the scheme n. During signing it instead depends on the threshold value t. The messages sent between participants are made up of polynomials, as even derived message types like ciphertexts and commitments are made up of one or more polynomials. These polynomials were either in R_Q, R_q or R_p, which sets limitations on their size in bits as $N \cdot log_2(Q)$, $N \cdot log_2(q)$ and $N \cdot log_2(p)$ respectively.

During $\texttt{KGen}_{\mathcal{TS}}$, a total of eight message exchanges take place, where each participant sends a message to each other participant, and does not continue with execution of the scheme until messages from each other participant has been received. During $\texttt{Sign}_{\mathcal{TS}}$, four such message exchanges occur. These exchanges are near-instant in our implementation as all participants are running on the same computer, in a real world implementation they would be much slower, as messages would need to traverse the internet.

4.3.1 Message Size

The total size of messages exchanged during $\texttt{KGen}_{\mathcal{TS}}$ and $\texttt{Sign}_{\mathcal{TS}}$ was calculated. Each message consists of a number of polynomials, usually of degree N and of a prime modulus. We did not consider any overhead from our polynomial implementation, nor did we consider any potential compression. The size of a polynomial was calculated as the product of its degree and the base 2 logarithm of its modulus. This was done for all polynomials except for commitment randomnesses as well as d, which was the output of our hash function H, as these were known to be bounded. The commitment randomness consisted of binary coefficients, and could be represented using one bit for each coefficient. Similarly, the hash output was ternary, requiring only two bits. We calculated the total number and size of each type of a message, and then multiplied these with our selected parameters to find the total size. We show the results for a $(3, 5)$-threshold for $\texttt{KGen}_{\mathcal{TS}}$ in Table 4.1 and the results for $\texttt{Sign}_{\mathcal{TS}}$ in Table 4.2. It is important to note that many of these types can be sent as parts of one larger message. Each type of message is sent only once in Table 4.2.

Value type	Number	Size per Message	Total Size
Hashes	4	$\frac{N}{4} \cdot log_2(4)$	2 Kib
Polynomials (R_q)	$(4 + n)$	$N \cdot \log_2(q)$	216 Kib
Polynomials (R_p)	1	$N \cdot \log_2(p)$	16 Kib
Commitments	$(2 + 2n)$	$2N \cdot \log_2(Q)$	864 Kib
ρ	1	N	1 Kib
ZKP π_{si}	1	$11N \cdot \log_2(q)$	264 Kib
ZKP π_{sk}	1	$7N(n + 1) \cdot \log_2(Q)$	1512 Kib
BGV ciphertext	1	$26N \cdot \log_2(Q)$	936 Kib

Table 4.1: Number of transmissions of each type sent by a participant to each other participant running $\texttt{KGen}_{\mathcal{TS}}$. The total size is calculated for a $(3, 5)$ scheme with parameters for 365 signatures presented in 3.3.

The output signature that is generated consists of one hash, a commitment randomness, and two polynomials in R_q. Using the values from the tables above, these require a total of 49.5 Kib. The public key of the scheme consists of two polynomials in R_q. In the public key, $\boldsymbol{a}$ is technically a vector of length two. As the second value is set to 1, it can be disregarded both for storage and transmission.

Value type	Size per Value	Total Size
Polynomials (R_q)	$N \cdot \log_2(q)$	24 Kib
Decryption shares	$11N \cdot \log_2(Q)$	396 Kib
ρ	N	1 Kib
Commitments	$2N \cdot \log_2(q)$	48 Kib
ZKP π_{ri}	$11N \cdot \log_2(q)$	264 Kib
BGV Ciphertext	$26N \cdot \log_2(Q)$	936 Kib

Table 4.2: Number of transmissions of each type sent by a participant to each other participant running $\texttt{Sign}_{\mathcal{TS}}$.

Effectively, the public key then requires 48 Kib.

Differences From Estimations in GKS23

In the revised paper, the authors present theoretical sizes of signatures and public keys for two scenarios [GKS24]. These scenarios are identical to the ones established in Section 3.4, where a signature can be used once or 365 times. In Table 4.3 we show the values they calculated.

	Single use signature		365 Signatures	
Type	Signature	Public key	Signature	Public key
Size	8.5 KB	2.6 KiB	10.4 KiB	3.1 KiB

Table 4.3: Theoretical values of the size of a signature and a public key for a single-use signature and 365 uses per signature. These values are from [GKS24].

A direct comparison can be made with the values we obtained for the 365 signature scenario. Above, we calculated that the signature requires 49.5 Kib and the public key 48 Kib. Converting our results to bytes instead gives 6.1875 KiB and 6 KiB, respectively.

The reason our signature is significantly smaller is due to the size of commitment randomness ρ differing between implementations. This difference is likely due to the increased complexity that would come from a fully functional DOTT20 scheme for commitments. Our public key is twice the size of the one proposed in [GKS24], this comes from differences in what we include as part of the public key. In our calculation, we include the vector $\boldsymbol{a}$, computed in $\texttt{KGen}_{\mathcal{TS}}$. They write that this value is instead deliverable by a ROM. We include $\boldsymbol{a}$ as our scheme supports both the revised version and the original in [GKS23], with a flag indicating which to run. These results indicate that the original authors' estimations are correct.

4.4 Execution Time and Complexity

The code was benchmarked on an Intel Core i7-10700K CPU. The code was implemented without any concurrency, leading to all participants executing each part of the code sequentially without any parallelization. As such, any calculation of execution time will represent the total execution time for all participants as well as the initialization of common objects like handlers for commitment schemes. Since in a real, distributed implementation each participant would need to initialize their own version of each of these, simply dividing the time required by the number or participants would not solve this concern. Instead, time analysis of specific parts of the code was used to calculate the time usage in a more detailed fashion.

In addition to the execution time analysis, the computational complexity was also analyzed by incrementing a counter for each instance of polynomial addition and multiplication. This is useful as the number of computations are a static property of the scheme itself, and as such is a more useful metric for comparison compared to the execution time.

4.4.1 Full Code Execution

For a given tuple (t, n), the number of multiplications and additions were constant. This was because all our polynomials, where a bound check was required, was verified upon generation of said polynomial. This meant that no additions or multiplications were done with any polynomial that was too large, as such no value failed a bounds check and needed to be regenerated.

To determine the best model for measuring time and the appropriate number of simulations, we ran 100 simulations for a $(3, 5)$-threshold and examined the lowest time, the average time, the highest time, and the variance. The results can be seen in Table 4.4.

	Lowest time	Average time	Highest time	Variance
$\texttt{KGen}_{\mathcal{TS}}$	18.857	19.272	20.016	0.0366
$\texttt{Sign}_{\mathcal{TS}}$	2.485	2.586	2.710	0.00123

Table 4.4: A measure of the average, lowest time, highest time, and estimated variance for key generation and signature generation. All units are in seconds.

Due to the low variance, we can reduce the number of simulations and still maintain an accurate average time. In future tables, we will instead use a lower number of simulations, and report the mean of these as the time to generate a key or a signature.

Our next analysis was done for key generation and signature generation for different configurations of (t, n). This was done to see how the computational complexity and execution time grew with the size of the distributed scheme. The results can be seen in Tables 4.5 and 4.6.

In Figure 4.1 we show how the complexity grew for a static $t = 3$ with an increasing n.

Key Generation				
t	n	Time (seconds)	Multiplications	Additions
2	4	17.5	3008	820
2	5	31	4895	1355
3	4	19.5	3008	820
3	5	44	4895	1355

Table 4.5: Measurements of execution time and number of multiplications and additions for $\mathtt{KGen}_{\mathcal{TS}}$ for different values of (t, n).

Signature generation				
t	n	Time (seconds)	Multiplications	Additions
2	4	3	647	182
2	5	3	647	182
3	4	5.3	1182	355
3	5	5.4	1182	355

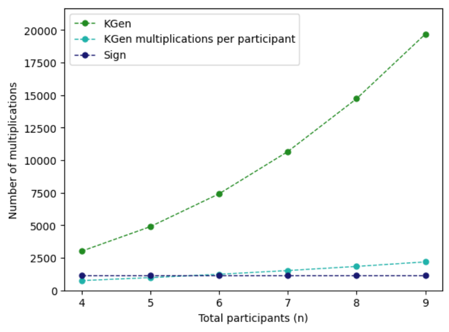
Table 4.6: Measurements of execution time and number of multiplications and additions for $\mathtt{Sign}_{\mathcal{TS}}$ for different values of (t, n).

From the figure, it is easy to see that the total time, number of additions and multiplications for $\mathtt{KGen}_{\mathcal{TS}}$ scaled with the number of total participants in the system. We also added a line showing the quotient of the $\mathtt{KGen}_{\mathcal{TS}}$ value and the number of participants. This showed that while the system grew to take approximately five minutes for nine participants, the time could be reduced to less than one minute if all participants had the same computing power and the system was fully parallelizable. $\mathtt{Sign}_{\mathcal{TS}}$ was static in these trials, which was expected, due to having a static $t = 3$.

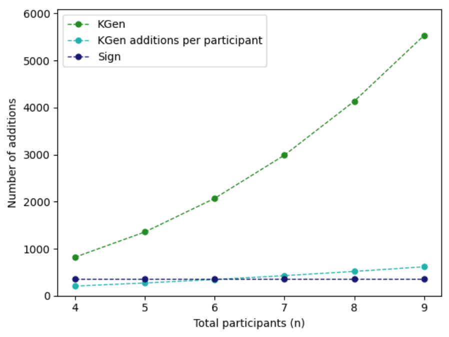
Next we examined how the system scaled for a static $n = 7$ and an increasing t. The results can be seen in Figure 4.2.

In the figure we see that the number of operations for $\mathtt{Sign}_{\mathcal{TS}}$ increases for larger t. Additionally, it can be seen that $\mathtt{KGen}_{\mathcal{TS}}$ requires more operations than $\mathtt{Sign}_{\mathcal{TS}}$, even when all participants are required, as in a n-out-of-n-threshold system. It is interesting to note that, as illustrated in Figure 4.2c, the time per participant for $\mathtt{KGen}_{\mathcal{TS}}$ reached its maximum when $t = \lceil n/2 \rceil$, for increasing values of t.

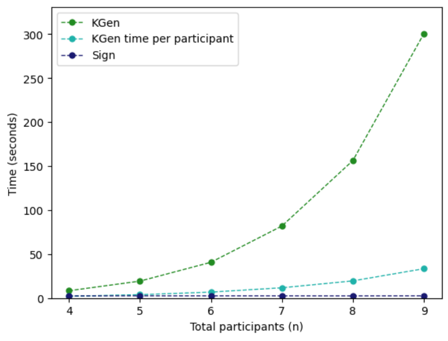
It may seem odd that despite the same number of operations in key generation, increasing the threshold value stills increased the execution time quite significantly. One must recognize that the operation count only accounts for operations between polynomials. The added time is made up of the additional constraints of secret sharing. At the end of key generation for $\mathcal{E}_{\mathcal{T}}$, detailed in Section 3.3. Here, each participant must evaluate that every possible subset of b_j of size t reconstruct into b. For each subset, this takes approximately 0.035 seconds with a $(2, 5)$-threshold, and 0.07 seconds with a $(3, 5)$-threshold. Part of the reason that this is less pronounced than the difference between $(2, 4)$ and $(3, 4)$ is that the number of



(a) Multiplications



(b) Additions



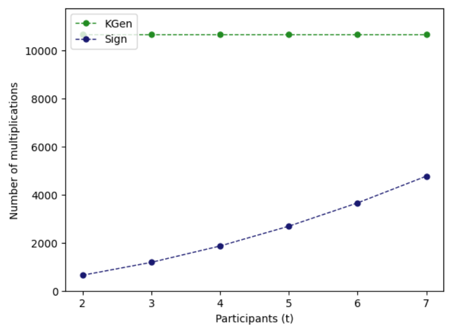
(c) Time

Figure 4.1: Measuring how our implementation scales for a static $t = 3$ and an increasing n.

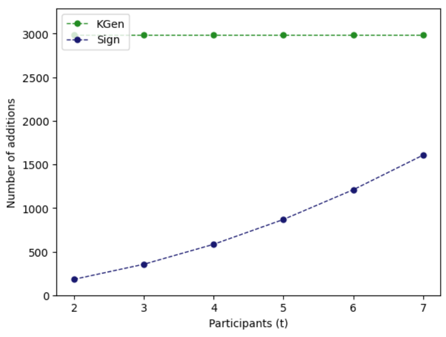
reconstructions required for each participant is directly correlated to the binomial coefficient of n over t. For $(2, 4)$ this coefficient is 6 and for $(3, 4)$ it is 4, therefore, the time to execute is only increased by a factor of approximately 1.33, rather than 2.

Full code execution was tested using a $(3, 5)$-threshold with the parameters for single use keys from Table 3.3, using a reduced number of tests. The difference in signature generation time was not statistically significant, but the time taken to generate keys differed by a full second. The cause of this was easy to ascertain. Using the Q-value for 365 signatures, along with all other parameters for single-use signatures, resulted in a key generation time that was identical to those where all parameters were set accounting for 365 signatures. It is quite clear that the main parameter limiting execution time is the value of Q. However, the other parameters are what determines the security of the scheme, and the value of Q is limited by them. It would have been interesting to run the scheme with parameters for boundless signatures as presented in [GKS24], but these were too large to function in our implementation.

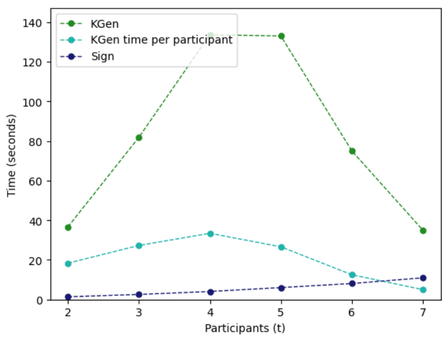
Another trial was done for a $(3, 5)$-threshold using $N = 2^{11}$ and the other



(a) Multiplications



(b) Additions



(c) Time

Figure 4.2: Measuring how our implementation scales for a static $n = 7$ and an increasing t.

parameters set to account for 365 signatures per key generation. This is not a rational selection of parameters, considering security, and should be seen as an investigation into how changing N would impact performance. Predictably, doubling the order of polynomials in the scheme resulted in a doubling of execution time for both $\texttt{KGen}_{\mathcal{TS}}$ and $\texttt{Sign}_{\mathcal{TS}}$, with $N = 2^{10}$.

4.4.2 Code Section Analysis

To ascertain what fraction of code execution time is taken up by each part of the code, separate timers were used for different segments to find the largest time sinks. This execution was run using parameters for a $(3, 5)$-threshold with parameters for 365 signatures using the same key. Many aspects of the code executed in just a few hundredths of a second, and were therefore not particularly interesting for the analysis. Among these were the initialization of commitment and ZKP schemes. These initializations are the only aspects of the scheme that are currently done by the controller, in a more realistic approach they would need to be performed by each participant. As these initializations take such a relatively small amount of time, much smaller even than the variance in execution time, they can safely be ignored. An accurate estimation of the time to execute the code for only one participant, excluding communication, can be done by simply dividing the execution time of $\texttt{KGen}_{\mathcal{TS}}$ by n and of $\texttt{Sign}_{\mathcal{TS}}$ by t.

During key generation the vast majority of time was taken up by the verification of values at the end of BGV key generation. As shown in Figure 4.4, key generation takes 19.27 seconds on average for $(3, 5)$, and verification of values took 14 seconds. Of this time, roughly 60% was taken up by the reconstruction of the secret shares of b. Verification of the commitment $c_{s_{i,j}}$ took at most a single percentage of the total time, with the remaining approximate one third of the time taken by the verification of the proof π_{sk}. A few other parts of key generation also took upwards of a second to execute. The section of code representing steps 3, 4, and 5 from $\texttt{DKGen}_{\mathcal{E}_{\mathcal{T}}}$ in 3.3, took roughly 1.3 seconds to execute, about half of which was taken up by the generation of π_{sk}. In total, the generation of the GKS23 specific keys took only about four seconds of the key generation. Roughly one second of this was taken up by the encryption of s_i as ctx_{s_i}, 0.8 seconds taken up by verifying the proof π_s and 1.8 seconds taken up by the summation of ciphertexts ctx_{s_i}, with smaller mathematical operations accounting for the remainder.

A short test was run to see if the relative differences in execution time remained consistent when changing the number of participants. $\texttt{KGen}_{\mathcal{TS}}$ for 365 signatures for a $(3, 9)$-threshold took 300.5 seconds, of which 284 seconds were taken up by the verification of values, or approximately $\frac{19}{20}$. This is a large difference from a $(3, 5)$-threshold, for which the verification took roughly $\frac{3}{4}$ of the time. The bulk of this increase was taken up by the reconstruction of secret shares, which went from taking less than 10 seconds to taking 252 seconds. This increase can be explained by the fact that each of the n participants have to reconstruct as many combinations as the binomial coefficient of t over n for each of the n b_j. For a $(3, 5)$-threshold, this totals 250 reconstructions between all participants, whereas for a $(3, 9)$-threshold it totals 6804. This led to an increase over the $(3, 5)$-threshold by a factor of 27.2.

During $\texttt{Sign}_{\mathcal{TS}}$, there was no large secret sharing required, which substantially cut down the runtime, with an average time less than 2.6 seconds. Roughly 0.6 seconds were taken up by the encryption of r_i as ctx_{r_i}, 0.7 seconds were taken up by the calculation of ctx_z, and 0.6 seconds were taken up by by the combination of decryption shares into z. The remainder was taken up by smaller operations, the largest of which being about 0.25 seconds to verify the proof π_r.

Chapter 5
Discussion

We achieved the major goal of implementing GKS23, and found that for increasing values of (t, n), the complexity scaled faster than linearly. When accounting for possible parallelization between participants, the scaling was less pronounced. We found that $\texttt{KGen}_{\mathcal{TS}}$ was the most expensive algorithm, where most of the time was spent computing secret shares. It could also be seen that the time per each participant in $\texttt{KGen}_{\mathcal{TS}}$ was maximized for $t = \frac{n+1}{2}$, which could change how (t, n) should be set for an optimal scheme.

As there were no readily available resources that was compatible with our goals, we had to implement all of the underlying schemes ourselves. We hope that our implementation aids future work in post-quantum cryptographic schemes by enabling reuse and improvements, rather than having to create everything from scratch.

5.1 GKS Revision

The vast majority of the revisions in [GKS24] have made no change to the scheme itself other than changes to notation. The first revision included a change to $\texttt{DKGen}_{\mathcal{E}_{\mathcal{T}}}$ and $\texttt{KGen}_{\mathcal{TS}}$. The revision involves using a preset value for a rather than a sum of values from each participant. Our implementation included a toggle to use the revised version or the original. The execution time was negligible, as it was much smaller than the variance in execution time. This is unsurprising as the total number of saved operations was $2n^2$ additions during key generation, or $2n$ additions per participant. It is worth mentioning that the revised version requires a trusted exchange of a common reference string, which contains values for a and a_{ts}, and reduces the number of transmissions for each participant by $2(n-1)$ hashes, $(n-1)$ polynomials in R_q and $(n-1)$ polynomials in R_Q. Most important is that each of the saved transmissions are stand-alone messages, meaning that each one needs to be transmitted and copies of the message from each other participant received before the key generation can continue. In our implementation, with participants all existing in the same program, this time to transmit non-existent, but in a truly distributed scheme, with participants communicating over the internet, this could cause increases to the execution time measured in tens of milliseconds. This latency is not massive compared to the execution time of our implementation, but it would be static, meaning more processing power or optimized implementations would have no effect on the latency, which could make the removal of the delay a significant upgrade to the scheme.

5.2 Further Improvements

We highlight natural avenues to further investigate the scheme below.

Missing functionality. The implementation is, as mentioned in Section 4.2 lacking a trapdoor homomorphic commitment scheme as well as a proof of shortness and a straight-line extractability property. Implementing these and performing a similar analysis would be of interest for completeness. The addition of proofs of shortness are likely to add significant complexity to ZKP systems, which are already a significant part of execution, and as such are a must if one intends to compare this scheme to another. As these add complexity to the scheme, comparisons could be made with our execution times and number of operations as a whole.

Concurrency and latency for participants. All our participants as well as the controller are run locally on the same machine, as mentioned in Section 4.3. A more realistic approach would be to simulate latency when sending data and have the participants able to compute values concurrently. An analysis of the execution time and relevant comparisons with our implementation would be of interest.

Optimize Shamir Secret Sharing. The analysis of execution time by code sections shows that the majority of key generation time is taken up by the quite simplistic implementation of Shamir's Secret Sharing method. Finding a more optimized implementation or investigating parallelization of this aspect of the implementation could save significant time. A reduction of the execution time of Shamir Secret Sharing by 50% would cut the execution time of $\texttt{KGen}_{\mathcal{TS}}$ by one third.

Reduce message size. When calculating the size of different types of messages, as presented in Tables 4.1, 4.1, we highlight how commitment randomness and the output of our hash function were bounded and thus limited in size. However several other polynomials which are sent had coefficients which are bounded bounded by much values than their ring modulus, and in fact the scheme would not function if their coefficients were as large as the current messages can support, as such, optimized transmission of them could be made smaller than the theoretical size would imply.

Fine-grained parameter selection. The parameter selection for BDLOP18, BGV11, and GKS23 were selected by recommendations in the papers presenting BDLOP18 and GKS23. The large prime modulus Q was tested by trial until one was found that worked. More fine-tuned parameter selection is likely to lead to better efficiency in the implementation.

Bibliography

[ACC+18] M. Albrecht *et al.*, “Homomorphic Encryption Security Standard,” HomomorphicEncryption.org, Toronto, Canada, Tech. Rep., 2018.

[ACF+15] M. R. Albrecht, C. Cid, J.-C. Faugere, R. Fitzpatrick, and L. Perret, “On the Complexity of the BKW Algorithm on LWE,” *Designs, Codes and Cryptography*, vol. 74, pp. 325–354, 2015.

[AD17] M. R. Albrecht and A. Deo, “Large Modulus Ring-LWE $\geq$ Module-LWE,” in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer, 2017, pp. 267–296.

[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.

[Ajt98] M. Ajtai, “The Shortest Vector Problem in L2 is NP-hard for Randomized Reductions (extended abstract),” in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, ser. STOC ’98, Dallas, Texas, USA: Association for Computing Machinery, 1998, pp. 10–19, ISBN: 0897919629. DOI: 10.1145/276698.276705.

[AP23] A. Al Badawi and Y. Polyakov, “Demystifying Bootstrapping in Fully Homomorphic Encryption,” *Cryptology ePrint Archive*, 2023.

[BDK+18] J. Bos *et al.*, “CRYSTALS-Kyber: a CCA-Secure Module-Lattice-Based KEM,” in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2018, pp. 353–367.

[BDL+18] C. Baum, I. Damgård, V. Lyubashevsky, S. Oechsner, and C. Peikert, “More Efficient Commitments From Structured Lattice Assumptions,” in *International Conference on Security and Cryptography for Networks*, Springer, 2018, pp. 368–385.

[BE95] P. B. Borwein and T. Erdélyi, *Polynomials and Polynomial Inequalities.* (Graduate texts in mathematics: 161). Springer, 1995, ISBN: 0387945091.

[BGJ+21] A. Budroni, Q. Guo, T. Johansson, E. Mårtensson, and P. S. Wagner, “Improvements on Making BKW Practical for Solving LWE,” *Cryptography*, vol. 5, no. 4, 2021, ISSN: 2410-387X. DOI: `10.3390/cryptography5040031`.

[BGV11] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, *Fully Homomorphic Encryption Without Bootstrapping*, Cryptology ePrint Archive, Paper 2011/277, 2011. [Online]. Available: `https://eprint.iacr.org/2011/277`.

[BKW03] A. Blum, A. Kalai, and H. Wasserman, “Noise-Tolerant Learning, the Parity Problem, and the Statistical Query Model,” *Journal of the ACM (JACM)*, vol. 50, no. 4, pp. 506–519, 2003.

[Bou22] C. Boutin, “NIST Announces First Four Quantum-Resistant Cryptographic Algorithms,” *National Institute of Standards and Technology*, 2022.

[Bro23] M. Brooks, *IBM Wants to Build a 100,000-Qubit Quantum Computer*, `https://www.technologyreview.com/2023/05/25/1073606/ibm-wants-to-build-a-100000-qubit-quantum-computer/` (accessed June 15, 2024), 2023.

[Des92] Y. Desmedt, “Threshold Cryptosystems,” in *International Workshop on the Theory and Application of Cryptographic Techniques*, Springer, 1992, pp. 1–14.

[DKL+18] L. Ducas *et al.*, “CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 1, pp. 238–268, 2018. DOI: `10.13154/tches.v2018.i1.238-268`.

[DOTT20] I. Damgård, C. Orlandi, A. Takahashi, and M. Tibouchi, *Two-Round n-out-of-n and Multi-Signatures and Trapdoor Commitment from Lattices*, Cryptology ePrint Archive, Paper 2020/1110, 2020. [Online]. Available: `https://eprint.iacr.org/2020/1110`.

[FDDK24] L. D. Feo, V. Delecroix, J. Demeyer, and V. Klein, *CyPari 2*, `https://github.com/sagemath/cypari2`, 2024.

[Fis01] M. Fischlin, “Trapdoor Commitment Schemes and Their Applications,” Ph.D. dissertation, Frankfurt (Main), Univ., Diss., 2001, 2001. [Online]. Available: `https://core.ac.uk/download/pdf/14505426.pdf`.

[FS87] A. Fiat and A. Shamir, “How To Prove Yourself: Practical Solutions to Identification and Signature Problems,” in *Advances in Cryptology — CRYPTO’ 86*, A. M. Odlyzko, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 186–194, ISBN: 978-3-540-47721-1.

[GE21] C. Gidney and M. Ekerå, “How to Factor 2048 Bit RSA integers in 8 Hours Using 20 Million Noisy Qubits,” *Quantum*, vol. 5, p. 433, Apr. 2021, ISSN: 2521-327X. DOI: `10.22331/q-2021-04-15-433`.

[Gen09] C. Gentry, “A Fully Homomorphic Encryption Scheme,” Ph.D. dissertation, Stanford University, 2009. [Online]. Available: `https://crypto.stanford.edu/craig`.

[GKS23] K. D. Gur, J. Katz, and T. Silde, *Two-Round Threshold Lattice Signatures from Threshold Homomorphic Encryption*, Cryptology ePrint Archive, Paper 2023/1318, 2023. [Online]. Available: `https://eprint.iacr.org/archive/2023/1318/1693840365.pdf`.

[GKS24] K. D. Gur, J. Katz, and T. Silde, *Two-Round Threshold Lattice Signatures from Threshold Homomorphic Encryption*, Cryptology ePrint Archive, Paper 2023/1318, 2024. [Online]. Available: `https://eprint.iacr.org/2023/1318`.

[GP24] M. Gustafsson and M. Petersson, *Lattice-Based t-out-of-n Threshold Signatures*, `https://github.com/Mattias-Petersson/lattice-based-t-of-n-signature-python`, 2024.

[Gro96] L. K. Grover, *A Fast Quantum Mechanical Algorithm for Database Search*, 1996. arXiv: `quant-ph/9605043 [quant-ph]`.

[Kat21] S. Katsumata, “A New Simple Technique to Bootstrap Various Lattice Zero-Knowledge Proofs to QROM Secure NIZKs,” in *Annual International Cryptology Conference*, Springer, 2021, pp. 580–610.

[KF15] P. Kirchner and P.-A. Fouque, *An Improved BKW Algorithm for LWE with Applications to Cryptography and Lattices*, Cryptology ePrint Archive, Paper 2015/552, 2015. [Online]. Available: `https://eprint.iacr.org/2015/552`.

[KSK+21] A. H. Karamlou, W. A. Simon, A. Katabarwa, T. L. Scholten, B. Peropadre, and Y. Cao, "Analyzing the Performance of Variational Quantum Factoring on a Superconducting Quantum Processor," *npj Quantum Information*, vol. 7, no. 1, Oct. 2021, ISSN: 2056-6387. DOI: 10.1038/s41534-021-00478-z.

[LDB23] A. Leevik, V. Davydov, and S. Bezzateev, "Threshold Lattice-Based Signature Scheme for Authentication by Wearable Devices," *Cryptography*, vol. 7, no. 3, 2023, ISSN: 2410-387X. DOI: 10.3390/cryptography7030033.

[LDC+17] Z. Li *et al.*, *High-Fidelity Adiabatic Quantum Computation Using the Intrinsic Hamiltonian of a Spin System: Application to the Experimental Factorization of 291311*, 2017. arXiv: 1706.08061 [quant-ph].

[LNP22] V. Lyubashevsky, N. K. Nguyen, and M. Plançon, "Lattice-Based Zero-Knowledge Proofs and Applications: Shorter, Simpler, and More General," in *Annual International Cryptology Conference*, Springer, 2022, pp. 71–101.

[LPR12] V. Lyubashevsky, C. Peikert, and O. Regev, *On Ideal Lattices and Learning with Errors Over Rings*, Cryptology ePrint Archive, Paper 2012/230, 2012. [Online]. Available: https://eprint.iacr.org/2012/230.

[MP12] D. Micciancio and C. Peikert, "Trapdoors for Lattices: Simpler, Tighter, Faster, Smaller," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2012, pp. 700–718.

[Nat16] National Institute of Standards and Technology, "Announcing Request for Nominations for Public-Key Post-Quantum Cryptographic Algorithms," Computer Security Resource Center, Washington, D.C., Tech. Rep., 2016.

[Nat23] National Institute of Standards and Technology, *Hash Functions*, Washington, D.C., 2023. [Online]. Available: https://csrc.nist.gov/projects/hash-functions.

[One14] M. E. O'neill, "PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation," *ACM Transactions on Mathematical Software*, 2014.

[PAR23] PARI, *PARI/GP version 2.15.4*, available from http://pari.math.u-bordeaux.fr/, Univ. Bordeaux, 2023.

[QLGZ20] X. Qian, J. Liu, C. Gu, and Y. Zheng, “An Improved BKW Algorithm For LWE With Binary Uniform Errors,” in *2020 5th International Conference on Computer and Communication Systems (ICCCS)*, 2020, pp. 87–92. DOI: `10.1109/ICCCS49078.2020.9118492`.

[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.

[Reg10] O. Regev, “The Learning With Errors Problem,” *Invited survey in CCC*, vol. 7, no. 30, p. 11, 2010.

[Sha73] A. Shamir, “How to Share a Secret,” *Advances in Cryptology — CRYPTO’ 86*, 1973.

[Sho94] P. Shor, “Algorithms for Quantum Computation: Discrete Logarithms and Factoring,” in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134. DOI: `10.1109/SFCS.1994.365700`.

[Sma16] N. P. Smart, *Cryptography Made Simple*. Springer, 2016, ISBN: 9783319219363.

Printed by Tryckeriet i E-huset, Lund 2024

Series of Master´s theses
Department of Electrical and Information Technology
LU/LTH-EIT 2024-986
http://www.eit.lth.se