EINAR HAEGER SOLFJELL
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Homomorphic Signatures: Implementation and Performance Evaluation

Master's Thesis - Computer Science - June 2022

```
func (m *Manager) NewConfiguration(opts ...gorums.ConfigOption) (c *Configuration, err error) {
	if len(opts) < 1 || len(opts) > 2 {
		return nil, fmt.Errorf("wrong number of options: %d", len(opts))
	}
	c = &Configuration{}
	for _, opt := range opts {
		switch v := opt.(type) {
		case gorums.NodeListOption:
			c.Configuration, err = gorums.NewConfiguration(m.Manager, v)
			if err ≠ nil {
				return nil, err
			}
		case QuorumSpec:
			// Must be last since v may match QuorumSpec if it is interface{}
			c.qspec = v
		default:
			return nil, fmt.Errorf("unknown option type: %v", v)
		}
	}
	// return an error if the QuorumSpec interface is not empty and no implementati
	var test interface{} = struct{}{}
	if _, empty := test.(QuorumSpec); !empty && c.qspec == nil {
		return nil, fmt.Errorf("missing required QuorumSpec")
	}
	return c, nil
}
```

I, **Einar Haeger Solfjell**, declare that this thesis titled, “Homomorphic Signatures: Implementation and Performance Evaluation” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master’s degree at the University of Stavanger.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Abstract

Homomorphic signatures allow users to outsource computation on their data while ensuring the integrity of the results, and to prove certain facts about official documents to third parties without sharing those documents. The signature scheme is capable of efficiently calculating several data analytical functions, including the average and standard deviation, on signed data and produce a signature for the result.

We present a fully functional implementation of the homomorphic signature scheme for polynomial functions by Boneh and Freeman (2011). We give a performance evaluation of our implementation, and measure the effects of two major performance improvements.

Acknowledgements

I would like to thank my supervisors for their enthusiasm and help with writing this thesis.

Contents

Chapter 1

Introduction

Homomorphic signatures allow users to outsource computation on their data while ensuring the integrity of the results, and to prove certain facts about official documents to third parties without sharing those documents. We provide a Go implementation of the homomorphic signature scheme by Boneh and Freeman (2011) and evaluate the performance of our implementation.

Using homomorphic signatures, companies can outsource computation on their data to a third party and be sure that it has returned the correct result without needing to trust this third party. Homomorphic signatures also let users prove that a calculation on their data is correct without also sharing the underlying data set. This feature is useful for companies and individuals who wish to share some facts about official documents without sharing the documents themselves. For example, a student who wishes to apply for a student loan could convince the loan agency that his grade point average is what he says it is, without sharing his actual grades and without needing a citation from the university that issued him those grades.

The primary goal of this thesis is to produce an implementation of the homomorphic signature scheme for homomorphic functions by Boneh and Freeman (2011) in Go. This signature scheme achieves homomorphic properties by expressing its signatures in terms of polynomials, which when evaluated on a specific value of x behave the same way as an integer would under addition, subtraction and multiplication. For a set of signed values $\mathbf{x}$ and corresponding signatures $\mathbf{S}$, performing a series of additions, subtractions and multiplications on $\mathbf{S}$ produces a new valid signature on the result of the same calculations on $\mathbf{x}$.

	A	B	$A+B$	$A \cdot B$
value	2	3	5	6
signature	$x^2 + x$	$4x - 1$	$x^2 + 5x - 1$	$4x^3 + 3x^2 - x$
$x = 1$	$1^2 + 1 = 2$	$4 \cdot 1 - 1 = 3$	$1^2 + 5 \cdot 1 - 1 = 5$	$4 \cdot 1^3 + 3 \cdot 1^2 - 1 = 6$

Table 1.1: Example of polynomials as homomorphic signatures

A simple example demonstrating these mathematical properties of signatures expressed as polynomials is shown in Table 1.1. The example shows two values A and B, each signed with a polynomial and a value of x for which the polynomial evaluates to the signed value. The sum and product of these two signatures can be calculated like the sum and product of the values themselves, and produce new polynomials that evaluate to the new results for the same value of x.

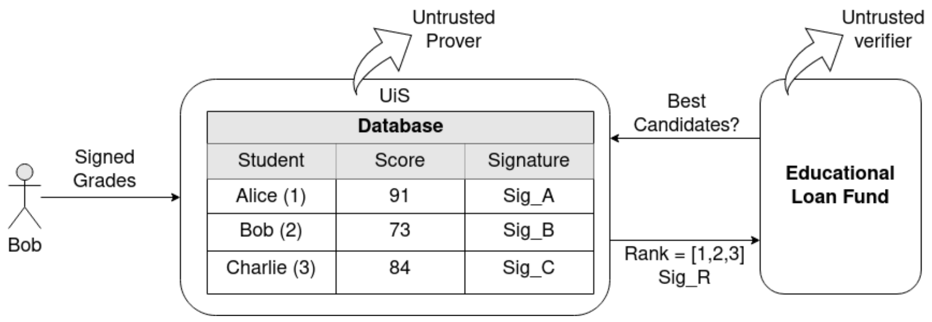
Anyone with access to $\mathbf{S}$ can generate a new homomorphic signature on any function applied to $\mathbf{x}$. The party generating this new signature can do so without needing access to the secret key used to sign values, nor the public key used to verify signatures, nor the values $\mathbf{x}$ that were signed.

As an example, consider a student who has graduated from university, and wants to apply for a student loan in another country. To be eligible for this loan, the student needs to prove that his average grades are above a certain limit. The student could prove his grades simply by giving the loan agency access to his grades so they can calculate the average. This however exposes more data than necessary, has higher overhead in terms of access management, and increases the risk of data leaks. Alternatively, the student could calculate the average himself and use homomorphic signatures to convince the loan agency that the result is credible. The process could be as follows:

- Bob receives his diploma, including homomorphic signatures for his grade in each course, from the university;
- Bob calculates the average of his grades, and also derives a homomorphic signature certifying the result;
- Bob applies for a student loan, and includes his grade point average as well as the homomorphic signature;
- The loan agency verifies the signature and is convinced that Bob's grade point average is what Bob stated it to be.

In this scenario, the university has allowed the student Bob, an untrusted source, to make verifiable statements about his grades, without revealing those grades to the loan agency and without consulting the university.

This work was commissioned for the Verifiable Ranking (vrank) project by my supervisors Meling and Saramago. Vrank seeks to ensure the integrity of rankings of university students based on their grades, eliminating vulnerabilities such as bribery from affecting the rankings. Vrank intends to use homomorphic signatures to let the university, which in this scenario is considered untrusted, calculate students' scores based on their grades as well as homomorphic signatures verifying these scores, then produce an official ranking of these scores. This official ranking can then be verified by verifying the homomorphic signature of each entry, which confirms both that the score itself is correct and that the correct function was used to calculate that score. This process is visualized in figure 1.1.



Database		
Student	Score	Signature
Alice (1)	91	Sig_A
Bob (2)	73	Sig_B
Charlie (3)	84	Sig_C

Figure 1.1: Verifiable ranking using homomorphic signatures

1.1 Approach and Contributions

We present a fully functional implementation of the homomorphic signature scheme for polynomial functions by Boneh and Freeman (2011). We give a performance evaluation of our implementation, and measure the effects of two major performance improvements.

1.2 Related Work

Jong (2011) made an effort to implement the signature scheme discussed in this thesis. While Jong has correctly implemented a majority of the signature scheme, his implementation appears to be incomplete. Jong's implementation deviates from the protocol designed by Boneh and Freeman (2011) in ways that compromise both the security of its signatures and the scheme's context hiding property[1].

ORide (Pham et al., 2017) is an example of another application using homomorphic encryption for its main functionality. ORide is a ride hailing service protocol. Their protocol uses homomorphic encryption to match riders and drivers while preserving privacy. Ride hailing services typically match users based on the distance between them. When a rider requests a ride, he is provided with a list of nearby drivers along with information about the distance between them, the driver's reputation score, etc. Typical ride hailing service protocols are able to inform riders of nearby drivers' locations because a central server keeps track of their 0. ORide seeks to provide privacy to both drivers and riders by implementing the driver matching feature with homomorphic encryption. Riders and drivers using ORide encrypt their location before sending it to the central server. The central server then homomorphically calculates the distance between the rider and several drivers before sending the encrypted results back to the rider for him to decrypt. With ORide, riders and drivers can make use of matchmaking features offered by a ride hailing service provider without revealing their location to the service provider.

[1]Signatures derived homomorphically using this scheme should reveal nothing about the original data set beyond the result of a given calculation.

Chapter 2

Background

This chapter provides the technological and mathematical background information we use to explain the homomorphic signature scheme by Boneh and Freeman (2011) and our implementation of this scheme.

2.1 Homomorphic Encryption and Signatures

Homomorphic encryption refers to encryption protocols that allow mathematic manipulation of data while it is encrypted. Homomorphic encryption allows users to outsource computation on their data without revealing the data itself to the one doing the computation, ensuring data privacy. This is illustrated in Figure 2.1.

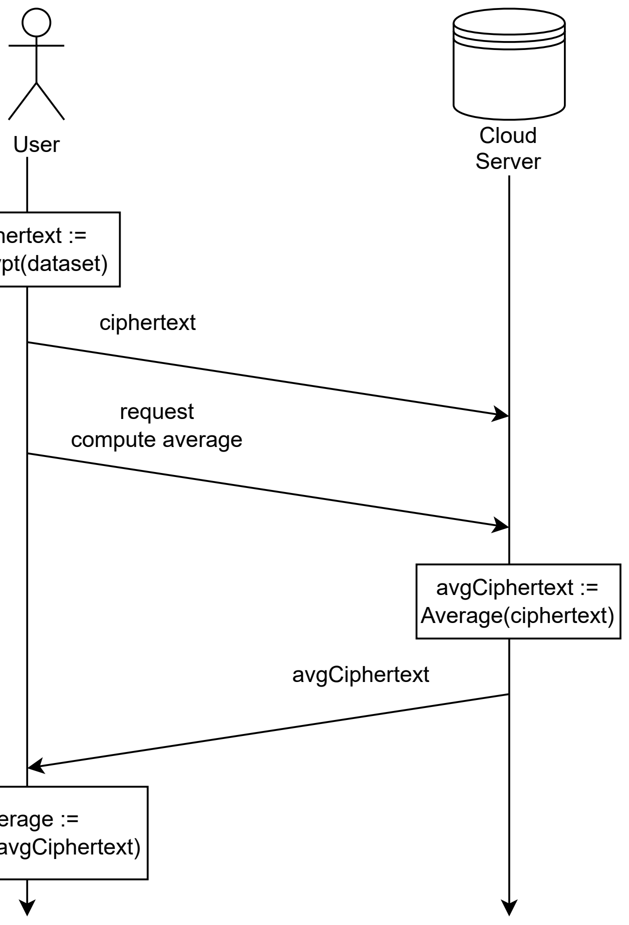

User
Cloud Server
ciphertext := Encrypt(dataset)
ciphertext
request compute average
avgCiphertext := Average(ciphertext)
avgCiphertext
average := Decrypt(avgCiphertext)

Figure 2.1: Delegating computation while ensuring privacy using homomorphic encryption

Homomorphic signatures allows anyone to perform calculations on signed data, and simultaneously generate a new signature certifying the result. This new signature is generated by performing the same calculations on the signatures as on the plaintext data. These calculations can be done without requiring access to the signing key, letting users outsource calculation and ensure that the calculation was done correctly by also producing a verifiable signature. This is illustrated in Figure 2.2.

The homomorphic signature scheme discussed in this thesis achieves its homomorphic properties by expressing signatures in terms of polynomials. These polynomials, when evaluated on specific values of x, equal the value that is being signed. Polynomials give homomorphic properties because polynomials evaluated on a specific value behave the same as integers under addition and multiplication.

Consider an example where we want to produce a homomorphic signature on the product of two integers. We first select our two integers, for example $a = 2$ and $b = 3$. We then produce a polynomial for a and b encoding their values, such as $A(x) = x^2 + 1$ and $B(x) = 2x + 1$, where $A(x)$ and $B(x)$ equal a and b respectively when evaluated on $x = 1$. We can easily evaluate that $A(1) = 1^2 + 1 = 2$ and $B(1) = 2 \cdot 1 + 1 = 3$. We can then calculate the product of the two polynomials $C(x) = A(x) \cdot B(x) = 2x^3 + x^2 + 2x + 1$, then evaluate $C(1) = 6$, which also equals the result of multiplying $a \cdot b$.

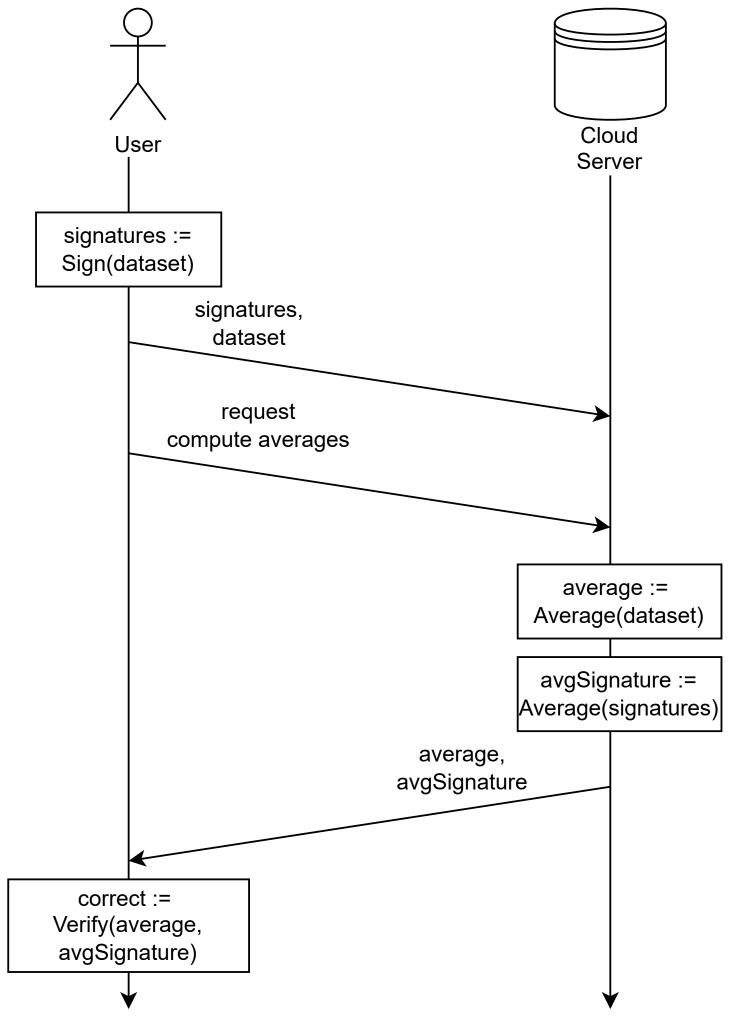

User
Cloud Server
signatures := Sign(dataset)
signatures, dataset
request compute averages
average := Average(dataset)
avgSignature := Average(signatures)
average, avgSignature
correct := Verify(average, avgSignature)

Figure 2.2: Delegating computation while ensuring integrity using homomorphic signatures

2.2 Integer Lattices

The signature scheme discussed in this thesis is a lattice-based signature scheme. The scheme uses integer lattices to generate short vectors, which when interpreted as polynomials make up the signatures used in this scheme. This section will explain what an integer lattice is, and describe some properties that are relevant to the signature scheme.

A lattice consists of every point in a coordinate space that can be reached by adding or subtracting together an integer number of a set of basis vectors. An integer lattice is a lattice whose basis vectors have integer coordinates. We generally refer to this set of basis vectors for a lattice simply as a ”basis”.

The rank of a lattice refers to the number of independent vectors that make up the basis of that lattice. A set of vectors with the property of being linearly independent means that none of these vectors can be expressed in terms of an integer sum of the other vectors, or equivalently, that any non-trivial integer sum of these vectors can equal the null vector. The dimension of a lattice is equal to the number of coordinates in each of the lattice’s basis vectors. A lattice’s rank is independent of its dimensions. However for lattices used by the signature scheme discussed in this thesis, rank and dimension are chosen to be equal.

A common example of a lattice is the XY coordinate plane, where the basis consists of the vectors $[0, 1]$ and $[1, 0]$, and the intersection points between grid lines in this coordinate plane make up the lattice. This is an example of a two-dimensional lattice with unit vectors.

For cryptography applications however, we usually consider lattices with hundreds to thousands of dimensions , and with longer and more complex basis vectors (Peikert, 2016).

2.2.1 Prime Ideal Lattices

The signature scheme discussed in this thesis uses a specific type of lattices called prime ideal lattices. Prime ideal lattices are used because they are closed under multiplication, and because they can be described succinctly with the so-called ”two-element” representation.

The property of a set of numbers being closed under a mathematical operation means that performing the operation on any pair of numbers from that set will result in another number in the set. Lattices generally are closed under addition, since the points that are in the lattice are all points that can be reached by *adding* together any number of basis vectors. Prime ideal lattices are also closed under multiplication, meaning that multiplying two lattice points, when interpreted as polynomials, results in a new point that is also in the lattice.

This property naturally lets us perform both addition and multiplication on polynomials, which makes up the signatures for this signature scheme. By using addition and multiplication, we can evaluate any polynomial on both signatures and the values they sign, resulting in a polynomially homomorphic signature scheme.

Prime ideal lattices are structured in a way that makes it possible to describe them succinctly with the so-called ”two-element” representation. A prime ideal lattice can be described by how it relates to R, the lattice containing all points with integer coordinates. R can also be described as the lattice with unit basis vectors, i.e. vectors of length 1 where one coordinate is 1 and the rest are 0s. A prime ideal lattice λ can be described as $\lambda = p \cdot R + h(x) \cdot R$ for some prime p and some polynomial h of the form $h(x) = x - \alpha$. Thus the two-element representation consists of the prime p and the variable α which defines h. A more in-depth description of the two-element representation is given by Boneh and Freeman (2011) in chapter 5.

2.2.2 Evaluating Polynomials on a Lattice

Signatures in the signature scheme discussed in this thesis are polynomials which encode the signed message in that they are equal to the message when evaluated on two lattices. Evaluating a signature on one lattice returns the value that was signed, and evaluating it on another lattice returns a number corresponding to the function that was used to generate that value.

Given the two-element representation of a lattice (p, α), we evaluate a polynomial on a lattice by reducing the polynomial mod p and $h(x) = x - \alpha$. To take a polynomial A modulo another polynomial B, we set $B = 0$ then perform basic algebraic operations to find $x^d = rest$, and replace every instance of x^d in A with the $rest$. For example, if we want to take $A \mod B$ where A = $x^4 + 2x^3 + 1$ and $B = x^2 - 3$, we calculate $x^2 = 3$, replace every instance of x^2 in A with 3, and we get $A \mod q = 3^2 + 2x \cdot 3 + 1 = 6x + 10$. Since $h(x) = x - \alpha$, taking A mod $h(x)$ means to replace every instance of x with α, i.e. to evaluate the polynomial on $x = \alpha$. In summary, evaluating a polynomial A on a lattice (p, α) means evaluating $A(\alpha) \mod p$.

2.3 Gram-Schmidt Orthogonalization

Signatures in the homomorphic signature scheme discussed in this thesis consist of polynomials corresponding to short vectors generated relative to a lattice. We generate these short vectors by taking a long vector $target$ with certain properties, then finding a lattice point near $target$, and returning the difference between the two vectors. There exists an efficient, polynomial time algorithm called the Nearest Plane algorithm (Micciancio and Goldwasser (2002), figure 2.5) which finds lattice points near a given vector. The signature scheme uses a variant of this algorithm to generate its signatures. Both the Nearest Plane algorithm and the variant used to generate signatures depend on the Gram-Schmidt Orthogonalisation (GSO) of a basis for the lattice to find lattice points in.

The GSO of a set of vectors is a corresponding set of vectors consisting of the component of each vector that is orthogonal[1] to each previous vector in the set. We calculate the GSO b^* of a matrix b by setting $b_0^* = b_0$, then each subsequent vector b_i^* is first set equal to b_i, and for each previous vector in the GSO $b_j^*(j < i)$, the component of b_i^* parallel to b_j^* is subtracted from b_i^*.

2.4 Lattice-based cryptography

The security of the signature scheme discussed in this thesis is based on two central problems in lattice-based cryptography, the Shortest Independent Vectors Problem (SIVP) and the Small Integer Solution (SIS) problem. In the following we explain these problems in more detail.

The Shortest Independent Vectors Problem is defined by Micciancio and Goldwasser (2002) as:

> Given a basis $\mathbf{B}$ of rank n, find linearly independent lattice vectors $s_1, ..., s_n$ such that $||s_i|| \leq \gamma \cdot \lambda_n(L(\mathbf{B}))$ for all $i = 1, ..., n$.

In this definition, the rank of a basis refers to the number of linearly independent vectors it contains. The condition $||s_i|| \leq \gamma \cdot \lambda_n(L(\mathbf{B}))$ means that the euclidean length of each vector $||s_i||$ must be shorter than some limit determined by γ and $\mathbf{B}$.

[1]Orthogonal vectors can be thought of as vectors that form a 90 degree angle between each of them.

This problem is a hard problem because the vectors in $\mathbf{B}$ may be made up of complex combinations of vectors in the shortest possible basis, and untangling these is by no means straight-forward. There are deterministic, polynomial time algorithms that solve SIVP for "large" values of γ (Micciancio and Goldwasser (2002) chapter 2.2), but SIVP becomes a hard problem for smaller γ.

Micciancio and Regev (2007) showed that several worst-case hard lattice problems can be reduced to the average-case SIS problem. That is, the SIS problem in the average case is at least as hard as the worst case hardness of several lattice problems, including SIVP.

The Small Integer Solution problem is defined by Micciancio and Regev (2007) as:

> Given an integer q, a matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$ and a real β, find a nonzero integer vector $\mathbf{z} \in \mathbb{Z}^m \setminus \{\mathbf{0}\}$ such that $\mathbf{Az} = \mathbf{0} \mod q$ and $||\mathbf{z}|| \leq \beta$.

The problem consists of finding a non-zero integer sum z of basis vectors which produces a vector $\mathbf{s} = \mathbf{Az}$, such that $\mathbf{s} \mod q = \mathbf{0}$. A version of this problem with no constraints on $\mathbf{z}$ can be solved trivially by setting $\mathbf{z} = \{q, 0, 0, ...\}$. The problem can also be solved easily if the constraint β is large. SIS becomes a hard problem when β becomes small enough.

Chapter 3

Approach

The signatures used in this scheme are polynomials which encode both the value being signed and the mathematical function used to generate that value. Signature polynomials encode these values by the results of evaluating these polynomials on parameters of two lattices. Evaluating a signature polynomial on the first lattice returns the value being signed, and evaluating it on the second lattice returns a number representing the function.

Forging signatures in the signature scheme is as hard as the SIS problem due to the requirement that signature polynomials, when interpreted as vectors made up of the polynomials' coefficients, need to be short. To generate these short polynomials, we start by generating a long polynomial that evaluates to the value and function when evaluated on the two lattices. We then find a polynomial in the lattice near the long polynomial, and return the difference between the two.

The signature scheme is capable of generating valid signatures verifying the result of any polynomial function on signed data. Polynomial functions are mathematical functions that use only addition, subtraction and multiplication. The signature scheme can be used to verify the results of several commonly used data analysis functions, including the average and standard deviation.

Signatures encode both a numerical value and the function used to generate that value. Signatures are verified by checking three conditions:

- the polynomial must equal the value when evaluated on the first lattice.
- the polynomial must equal a number corresponding to the function when evaluated on the second lattice.

- the vector whose coordinates equal the coefficients of the polynomial must be shorter than some limit.

3.1 Existing Approaches/Baselines

While developing our code, we used a C++ implementation of the same homomorphic signature scheme by Jong (2011) as a reference. We eventually discovered this implementation to be incomplete, but it served as a good baseline to guide us through developing our own implementation in Go. Jong's implementation also contained correct implementations of several algorithms described in the paper by Boneh and Freeman, allowing us to take inspiration from their implementation.

3.2 Proposed Solution

While implementing the homomorphic signature scheme, we followed the description of the polynomially homomorphic scheme given on page 19 of the paper by Boneh and Freeman (2011). The code for our implementation can be found on github (Solfjell et al., 2022).

The scheme contains four functions: Generate Key, Sign, Verify and Evaluate. Generate Key generates the public and secret key pair used for generating, verifying and calculating functions on signatures. The public key contains two lattices given in the two-element representation, as well as the number of dimensions the lattices span which equals the security parameter n. Signatures are generated relative to these lattices, and signatures are verified by evaluating them on these lattices. The public key also contains parameters that define which functions can be calculated over signatures using this public key. The set of admissible functions is limited to a maximal degree and a maximal size of the coefficient of each term in the function. These parameters, as well as the security parameter n, are given as parameters to the GenerateKey function. GenerateKey also generates a secret key, which is needed for generating signatures. This secret key contains a lattice basis for the union of the two lattices described in the public key. The signatures in this scheme consist in part of a linear sum of vectors from this lattice basis.

Sign samples a signature in the form of a polynomial corresponding to a short vector in the lattice. This polynomial is generated so that it corresponds to the message that was signed when evaluated on the lattices in the public key.

Verify simply checks that the signature polynomial evaluates to the message, and that the vector corresponding to the signature is short enough. To verify that the signature polynomial evaluates to the message, we evaluate it on the two-element representation (p, α) of the two lattices in the public key. We evaluate a polynomial A on the two-element representation by evaluating $A(\alpha) \mod p$. If the signature is correct, evaluating it on the first lattice returns the number that was signed, and evaluating it on the second lattice returns a number corresponding to the function used to generate the signature. The maximal length of a valid signature is calculated based on parameters in the public key. This maximal length is derived from a theoretical analysis on the distribution of polynomials output by the algorithms used to generate signatures.

For a set of message-signature pairs, Evaluate generates a signature for the result of a function on these messages. This new signature is generated by performing the same additions and multiplications as on the messages themselves. The signature scheme can be used to calculate any polynomial function, such as the average or standard deviation of a set of numbers.

3.2.1 Generate Key

To produce and verify signatures in this signature scheme we first need a pair of public and secret keys. GenerateKey uses an algorithm proposed by Smart and Vercauteren (2010), referred to as PrincGen, to generate prime ideal lattices. Princgen generates a prime ideal lattice, and returns its two-element representation as well as a generator polynomial for this lattice. We use PrincGen to generate two lattices, then use the two generator polynomials to generate a lattice basis for the union of these lattices. The two lattices along with some parameters that define the admissible functions for this key pair make up the public key, and the lattice basis for the union of the two lattices makes up the secret key.

Generating Lattices

The signature scheme uses prime ideal lattices for their property of being closed under both addition and multiplication, and for their concise two-element representation. GenerateKey uses PrincGen to generate the two-element representation of and a generator polynomial for two such lattices.

PrincGen works by checking if random polynomials G generate lattices that are prime ideal lattices. The first element p of the two-element representation of a lattice generated by G is equal to the resultant between G and an irreducible polynomial $irre$[1]. If p is prime, then G generates a prime ideal lattice.

Algorithm 1 Prime Ideal Lattice generator

```
function PrincipalPrime(polynomial irre)
    $p \leftarrow 0$
    while $p$ is not prime do
        $S \leftarrow$ random polynomial
        $G \leftarrow 2S + 1$
        $p \leftarrow$ Resultant($G$, irre)
    $D \leftarrow$ GCD(irre, $G$) mod $p$
    $\alpha \leftarrow$ unique root of $D$
    return $G, (p, \alpha)$
```

UnionBasis

Using a generator polynomial G, we can generate a set of linearly independent vectors that span the lattice, forming a basis for that lattice. The coordinates of the vectors in this lattice basis are the same as the coefficients of G, rotated left one spot for each vector in the basis. We generate this lattice basis by multiplying G by powers of x (i.e. $1, x, x^2, ..., x^{n-1}$), and take the results mod $irre$. To generate a basis for the union of two lattices, we replace G with the product of the generator polynomials of the two lattices. Pseudocode for UnionBasis is shown in Algorithm 2.

3.2.2 Sign

Sign generates a signature for each message to be signed in the form of a polynomial that equals that message when evaluated on parameters of the lattice. Each message to sign is defined by a value, a tag, and the message's index in the set of

[1]$irre$ is an irreducible polynomial defining the number field our lattices are in. We select $irre := x^n - 1$ based on the security parameter n. Polynomials constructed in this manner are irreducible if n is a power of 2.

Algorithm 2 Two-lattice Union Basis Generator

```
function UnionBasis(polynomials $pGen, qGen, irre$, int $n$)
    $basis \leftarrow n \times n$ matrix
    for $i \leftarrow 0, ..., n-1$ do
        $basis[i] \leftarrow pGen \cdot qGen \cdot x^i \mod irre$
    return $basis$
```

messages to sign. The value is a number, such as a student's grade in a given subject. The tag is a text string describing the set of messages. Each message in the set is given a unique identifier, generated by hashing the tag and the message's index.

A signature has three requirements to be considered valid. First, it needs to be equal to the message's value when evaluated on the first lattice in the public key. Second, it needs to be equal to the message's unique identifier when evaluated on the second lattice. Third, the vector with coordinates corresponding to the polynomial's coefficients needs to be short. Signature polynomials need to be short because generating short vectors in a lattice is difficult without knowing a basis for that lattice with short vectors.

Sampling Short Polynomials

To sample vectors from the lattice we use an algorithm by Gentry et al. (2008), referred to as "SamplePre" by Boneh and Freeman (2011). SamplePre produces samples from a Gaussian distribution over a given lattice, with a given center and variance. We produce short vectors by sampling a vector *sample* near a given vector *target*, and returning *target* − *sample*.

The signature scheme uses SamplePre to sample vectors from the union of the two lattices in the public key. Vectors sampled by SamplePre, when interpreted as polynomials, equal zero when evaluated on the two-element representation of either of the two lattices making up the lattice we're sampling from. To make sure that sampled polynomials are both short and evaluate to the right numbers, we first produce a polynomial *target*, independent of the lattice, which evaluates to the right numbers. We then generate a vector *sample* from a distribution over the lattice centered on *target*, and subtract *sample* from *target*. The result is a short vector, since it is *target* minus a vector within a short distance from *target*, and when interpreted as a polynomial it evaluates to the right numbers since *target* evaluates to the right numbers and any sample evaluates to zero.

SamplePre is a randomized version of the Nearest Plane algorithm (Micciancio and Goldwasser (2002), figure 2.5) for solving the closest vector problem. Instead of adding a deterministic multiple c_i of each basis vector b_i to the result, it samples integers z_i from a distribution centered on c_i and adds $z_i \cdot b_i$ to the result.

In each iteration, the algorithm produces a sample z_i from a Gaussian distribution over the integers, and adds the i-th basis vector b_i multiplied by z_i to the output. The parameters for the Gaussian distribution over the integers are based on the Gram-Schmidt orthogonalization of the lattice's basis b^*, and the *target* and *variance* parameters given as inputs to the function. With each iteration, the algorithm uses the component of the *target* vector that is parallel to the i-th Gram-Schmidt vector b_i^* to determine the center of the Gaussian distribution over the integers. Then it subtracts b_i multiplied by the sample z_i from the *target* vector before calculating the center for the next iteration. The variance for the Gaussian distribution over the integers is simply the parameter *variance* divided by the euclidean length of the i-th Gram-Schmidt vector $||b_i^*||$.

Algorithm 3 Sampling Gaussian Distribution Over Lattice

```
1:  function SampleGaussLattice(matrix $b$, $b^*$, polynomial target, int variance)
2:      $v_n \leftarrow 0$
3:      $c_n \leftarrow$ target
4:      for $i \leftarrow n, ..., 1$ do
5:          $c_i' \leftarrow \langle c_i, b_i^* \rangle / \langle b_i^*, b_i^* \rangle$
6:          $s_i' \leftarrow$ variance$/||b_i^*||$
7:          $z_i \leftarrow$ Sample$(\mathbb{Z}, s_i', c_i')$
8:          $c_{i-1} \leftarrow c_i - z_i \cdot b_i$
9:          $v_{i-1} \leftarrow v_i + z_i \cdot b_i$
10:     return $v_0$
```

Constructing a polynomial evaluating to the right numbers

SamplePre outputs polynomials in the lattice near a given vector. The signature scheme uses SamplePre to generate *sample* vectors near a *target* vector which evaluates to the right numbers. The *sample* vector is then subtracted from the *target* vector, producing a short vector. Because *target* encodes the signed message's value and unique identifier, the vector *target* – *sample* also encodes the message's value and unique identifier.

The polynomial corresponding to the *target* vector needs to equal the message's value v when evaluated on the public key's first lattice, and to equal the message's unique identifier id when evaluated on the second lattice. We produce such a polynomial in the form of a degree zero polynomial, i.e. a constant. Given the two-element representations of two lattices (p, α), (q, β), we select a constant *target* such that *target* $\mod p = v$ and *target* $\mod q = id$. A simple way of constructing *target* would be to start by setting *target* $= id$, then iteratively adding q until *target* $\mod p = v$. This process however would take a long time to finish when p and q become large. Instead of constructing *target* iteratively, we use modular arithmetic to calculate the number x of times we need to add q, as shown below:

$$\begin{aligned} id + x \cdot q &= v \mod p \\ x \cdot q &= v - id \mod p \\ x \cdot q \cdot q^{-1} &= (v - id) \cdot q^{-1} \mod p \\ x &= (v - id) \cdot q^{p-2} \mod p \end{aligned}$$

Now that we know x, we can set *target* $\leftarrow id + x \cdot q$. We know that *target* $\mod p = v$ by the first line of the above equations. We also know that *target* $\mod q = id$ since $id + x \cdot q \mod q = id + 0$. Since *target* $\mod p = v$ and *target* $\mod q = id$, we have the target vector we need to sample signature polynomials.

Algorithm 4 Constructing a polynomial evaluating to the right numbers

```
function targetVector(int $v$, $id$, $p$, $q$)
    $x \leftarrow (v - id) \cdot q^{p-2} \mod p$
    return $id + x \cdot q$
```

3.2.3 Verify

Verify simply verifies that a signature σ meets the three conditions to consider a signature valid. To verify that σ evaluates to the message's value v and unique identifier id when evaluated on the first and second lattice respectively, we evaluate σ on these and compare the results to the message. Given the two-element representations of two lattices (p, α) and (q, β), if $\sigma(\alpha) \mod p = v$, the first condition is verified. If $\sigma(\beta) \mod q = id$, the second condition is verified. To verify the signature's length, we calculate a maximum valid length based on parameters from the public key that define the set of admissible functions. Signatures may be longer if the public key allows functions with higher coefficients, with more variables or with higher maximal degree of terms in the function. If σ is shorter than this limit, the third condition is verified.

3.2.4 Evaluate

Evaluate calculates functions on a set of signatures, producing a new signature for the result of the same function on the original signed values. The signature scheme is capable of calculating any polynomial function, including commonly used data analytical functions like the average and standard deviation. The resulting signature is verified in the same way as a signature on a single value, by evaluating the signature on the two lattices in the public key. Given the two-element representations of two lattices (p, α), (q, β) and a signature σ, $\sigma(\alpha) \mod p$ equals the result of the function on the signed values, and $\sigma(\beta) \mod q$ is equal to a number uniquely corresponding to the function.

Evaluate takes a set of signatures *S*, a set of integers *c* specifying the coefficients of the function to calculate over *S*, and an integer d specifying the highest degree of the function. The function consists of the sum of every way of multiplying together d or fewer signatures, multiplied by the corresponding coefficient in *c*. If we denote each signature in *S* as an independent variable (like $x, y, z, ...$), then a "way of multiplying d or fewer signatures" can be denoted as a monomial of degree up to d. The coefficient c_i is applied to the i-th such monomial listed in lexicographic order[2].

Consider an example where *S* $= \{x, y, z\}$ and $d = 2$. In this case, all possible monomials are:

$$x, y, z, x^2, xy, xz, y^2, yz, z^2$$

To calculate a function of degree $d = 2$ over *S*, Evaluate needs a set of coefficients *c* with 9 elements. To calculate the function $(x + 2xz + 3yz)$, we set

[2] All monomials of degree d appear before monomials of degree $d+1$, and monomials with more x-s appear before monomials with more y-s which appear before monomials with more z-s, and so on.

$c = \{1, 0, 0, 0, 0, 2, 0, 3, 0\}$ and call Evaluate($S, c, 2$).

Monomial Iterator

To generate this set of all possible monomials given a number of distinct variables k and a maximal degree d, in lexicographic order, we implemented a monomial iterator type. This type holds the parameters k and d, as well as an encoding of the ”current” monomial. The ordered set of monomials is generated through the iterator’s Next() method. The encoding of the current monomial is in the form of a non-decreasing sequence[3] of numbers of length d, where each number represents an instance of one of the variables in the monomial. For example, the sequence $(0, 0, 1, 1, 2)$ represents the monomial x^2y in a monomial with degree at most 5. For monomials in k variables and max degree d, the Next() method treats the monomial encoding as a base-k, d-digit integer, and repeatedly ”adds one” to this base-k integer until the resulting sequence is non-decreasing. Pseudocode for the monomial iterator’s Next and AddOne functions is given in Algorithm 5 and 6.

Algorithm 5

```
function Next(int $k$, []int monomial)
    AddOne(monomial, $k + 1$)
    while !IsNonDecreasing(monomial) do
        AddOne(monomial, $k + 1$)
```

Algorithm 6

```
function AddOne([]int monomial, int $k$)
    $l \leftarrow$ len(monomial)
    monomial[$l$] $\leftarrow$ (monomial[$l$] $+ 1$)   mod  $k$
    for $i \leftarrow l, ..., 1$ do
        if monomial[$i$] $\neq 0$ then
            break
        monomial[$i - 1$] $\leftarrow$ (monomial[$i - 1$] $+ 1$)   mod  $k$
```

[3]each number is greater than or equal to the previous number

Chapter 4

Experimental Evaluation

In this chapter we give a performance evaluation for our implementation of the homomorphic signature scheme for polynomial functions by Boneh and Freeman (2011). We describe our performance testing procedure and experimental setup, and present our results. First we show the overall performance of our implementation with all optimizations, then we explain each optimization individually and show its impact on the performance of the relevant function.

The experimental results can be found in the github repository (Solfjell et al., 2022) hosting our code. The results are in the directory titled ”data”, which at the time of this writing is located in the branch titled ”performance_data”.

4.1 Experimental Setup

We gathered performance data by executing each of the four main functions of the signature scheme with random data, and recorded the execution time of each function. We ran tests using values of n equal to powers of two from 4 to 128, and a set of 5, 10 or 20 values to sign. For each set of testing parameters, we repeated this testing procedure for 30 repetitions or for two minutes, whichever took longer. Finally we calculated the average and standard deviations of the results for each set of testing parameters for display in the graphs in this chapter. Unless otherwise stated, the results presented in this chapter were gathered from tests with 20 signed values.

The results displayed in this chapter were generated using Windows Subsystem for Linux (WSL) on a Windows 10 computer with an Intel Core i3-8350K

CPU at 4.00 GHz. WSL had access to 308.6 MB of RAM.

4.2 Experimental Results

We were not able to make our implementation efficient enough to use the ”hundreds to low thousands of dimensions” of lattices as mentioned by Peikert (2016) on the computer we used for performance testing. The results of our performance testing of the four main functions in the signature scheme with a range of values of n can be seen in Figure 4.1.

The highest value of the security parameter n we could use and get reasonably short execution times for GenerateKey and Sign is $n = 64$. With $n = 64$, our implementation was able to complete GenerateKey in an average of 15 seconds, and generate signatures in an average of 0.75 seconds per signature. Our implementation is however able to complete Evaluate and Verify much faster, taking an average of 3 milliseconds to compute a linear function over 20 signatures, and an average of 0.6 milliseconds to verify the signature validating the result of this calculation, also with $n = 64$.

The time it takes to execute each main function in the signature scheme is dependent on the security parameter n. While we have not analysed the running times of the algorithms themselves, we can make estimates based on imperial data. Of the four main functions, GenerateKey and Sign take the longest to calculate for values of n between 8 and 128. The time to execute these functions also scales significantly faster with n than Evaluate and Verify. The execution time of GenerateKey scales faster than that of Sign, with GenerateKey being roughly $O(n^4)$ where Sign is roughly $O(n^3)$. Evaluate and Verify scale more slowly with n, where Evaluate takes roughly $O(n)$ time. While Verify appears to take roughly $O(n)$ for small values of n, its average execution time jumps somewhat for $n = 64$ and $n = 128$. This jump appears to be due to a slowdown in the big numbers library for Go when calculating high exponents of large numbers.

Certain factors other than n affect the execution speeds of Sign, Evaluate and Verify. Sign takes a roughly constant amount of time for each signature with little overhead for signing a batch of values, thus time to execute Sign depends linearly on the number of values k to be signed. Evaluate and Verify also depend on k, and also depend on the complexity of the function. A function over k messages with degree d is encoded by a set of coefficients with a length of $l = \binom{k+d}{d}$. For each of these coefficients, Evaluate and Verify multiply together the appropriate number of polynomials and integers respectively, and add the result to the output.

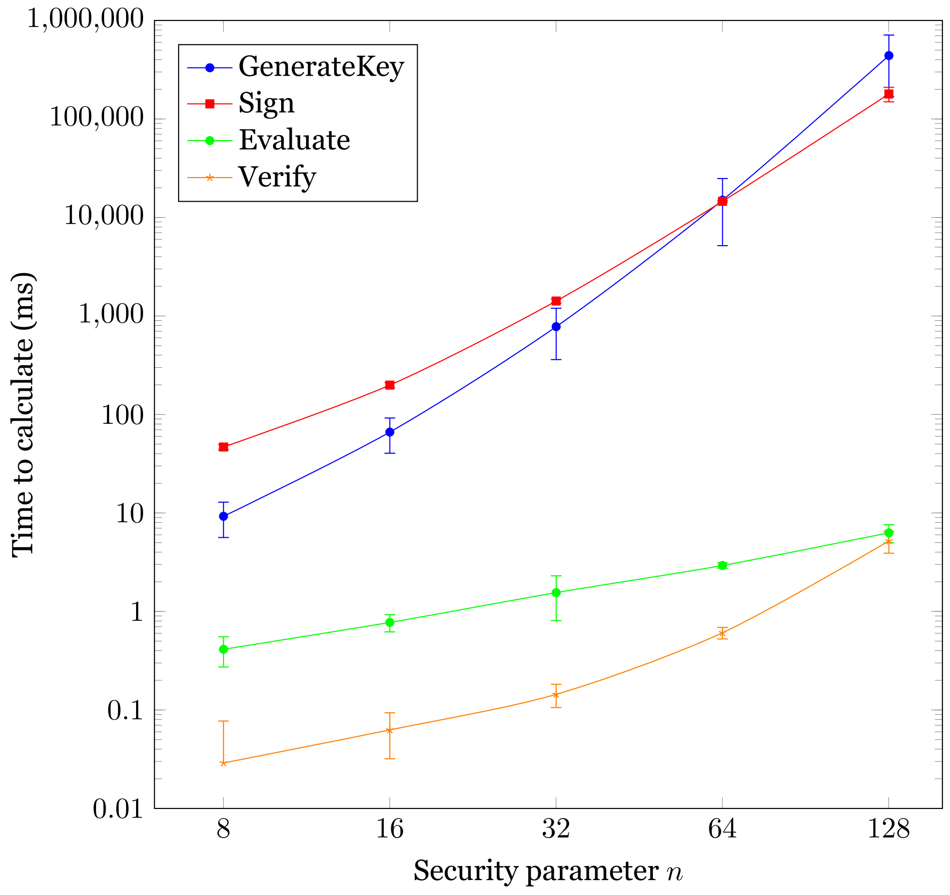


Figure 4.1: Performance of each main function

4.3 Bottlenecks and Optimizations

After completing a working implementation of the signature scheme, we discovered some performance bottlenecks. Lattice-based cryptography systems usually operate in lattices with hundreds to low thousands of dimensions (Peikert, 2016). However, our initial implementation was only able to use lattices of up to 32 dimensions and still achieve acceptable execution times.

The largest bottlenecks were due to repeated division and addition of rational numbers from the Go big numbers library. Rational numbers from the big numbers library store numbers with perfect precision as a ratio of two relatively prime numbers. The problem occurs when rational division results are carried over to the next iteration of a loop, increasing the numbers' numerator and denominator by a few orders of magnitude with each iteration, making each iteration take longer and longer to complete.

This bottleneck affected two of the main functions of the signature scheme, GenerateKey and Sign. GenerateKey encountered the bottleneck in the algorithm for calculating the resultant between two polynomials. Sign encountered the bottleneck in the algorithm we use to calculate the Gram-Schmidt Orthogonalization of a lattice basis. We were able to work around both of these bottlenecks and achieve significant improvements in performance for the affected functions.

4.3.1 Modular Resultant Calculation

To get around the bottleneck in GenerateKey of calculating the resultant between two polynomials, we took inspiration from the Number Theory Library (NTL) by Steuer and Shoup (2021). NTL calculates the resultant using modular arithmetic. Expressing the algorithm in terms of modular arithmetic allows us to replace division with multiplying by the modular inverse of the divisor, thus circumventing the bottleneck. Calculating a resultant mod some prime number won't always produce the correct result however, and the result may change for different moduli. Here we can use the Chinese Remainder Theorem (Pei et al., 1996) to retrieve the correct resultant from the results of several modular resultant calculations with different moduli.

Performance improvement

The results of implementing the modular resultant algorithm are shown in Figure 4.2. Implementing the resultant algorithm using modular arithmetic and the Chinese Remainder theorem resulted in GenerateKey taking slightly longer for $n \leq 16$, but taking much less time for $n \geq 64$. The modular variant starts out taking longer to complete, but the time it takes increases more slowly with n than the non-modular variant. At $n = 64$, GenerateKey using the modular resultant algorithm completes on average almost 10 times faster than using the non-modular resultant algorithm.

The basic resultant algorithm was the biggest bottleneck affecting our implementation, as such it took a long time to run the basic resultant experiment with high values of n. The data shown in Figure 4.2 of the basic resultant algorithm with $n = 128$ is based on a single trial of the experiment, as opposed to the 30 or more trials we used for most other experiments.

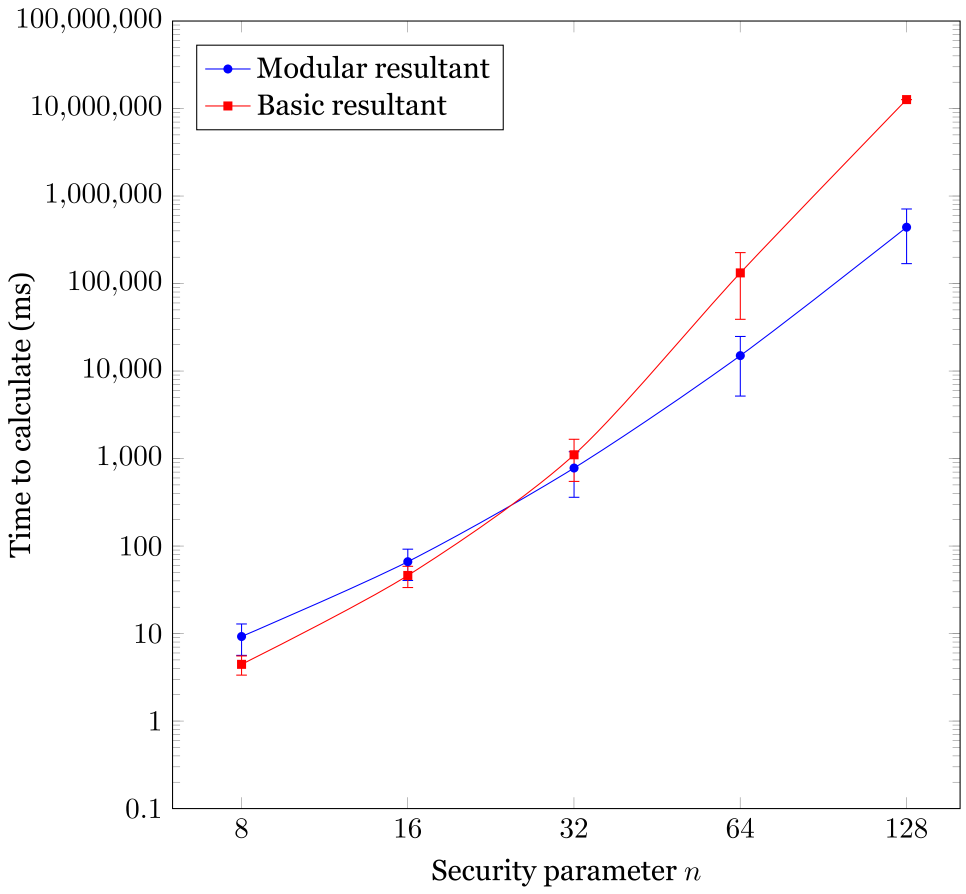


Figure 4.2: Resultant optimization: Time to GenerateKey

4.3.2 Floating Point Gram-Schmidt Orthogonalization

The Sign function uses SamplePre to generate signatures. SamplePre depends on the Gram-Schmidt Orthogonalization b^* of the basis b of the lattice it samples vectors from. The algorithm for calculating b^* is affected by the bottleneck of repeated division and addition with perfect precision.

The big numbers library for Go features three data types: *big.Int* for arbitrarily large integers, *big.Rat* for arbitrarily precise rational numbers, and *big.Float* for arbitrarily large numbers with limited precision. Simply expressing the algorithm for calculating b^* in terms of *big.Float* instead of *big.Rat* does indeed make the calculation swift, but this also changes the behaviour of SamplePre.

SamplePre outputs a vector *sample* in the lattice from a distribution with a given *variance* centered around a given *target* vector. To generate vectors from the correct distribution, SamplePre requires that b^* be given with perfect precision. If the error in b^* is too large, vectors output by SamplePre do not come from the correct distribution and are frequently too long to be valid signatures.

Specifically, the error in b^* affects the fifth line of Algorithm 3:

$$c_i' \leftarrow \langle c_i, b_i^* \rangle / \langle b_i^*, b_i^* \rangle$$

The variable c_i' defines the center of the distribution from which z_i is sampled from, which is the number of times the i-th basis vector is added to the output. The error in c_i' is proportional to both the error in b^* and the length of c_i, and if the error is large enough to significantly change the distribution of z_i, then SamplePre will output vectors with an incorrect distribution. The way we construct the initial *target* vector as explained in Section 3.2.2 results in a vector with length on the order of $p \cdot q$.

With security parameters as low as $n = 8$, *target* vectors constructed in this way are large enough to make errors in b^* on the order of 10^{-15} drastically alter the distribution of vectors output by SamplePre.

When we calculate the b^* using the *big.Float* number type instead of *big.Rat*, the result typically has errors on the order of 10^{-14}. This error is large enough for SamplePre to not output valid signatures, but the output distribution has another useful property. SamplePre given a GSO b^* with a small error consistently produces *sample* vectors closer to *target* than *target* is to the origin. The distance between *sample* and *target* appears to be proportional to both the length of *target* and the magnitude of the error. For example, given a *target* vector of length 10^{60} and an error in b^* on the order of 10^{-15}, the *sample* output by SamplePre will be approximately a distance of 10^{45} away from *target*. The vector *target – sample* will be shorter than *target*, and since any *sample* will evaluate to 0 on either of the two lattices in the public key, *target – sample* also encodes the same message that *target* does. Iteratively replacing *target* with *target* – *sample* in this way allows us to generate a *target* vector with otherwise the same properties that is short enough for the error in b^* to not significantly alter the output distribution from SamplePre. Thus we can sample valid signatures while cicrumventing the bottleneck of calculating b^* with perfect precision.

As mentioned previously, SamplePre is a randomized version of the Nearest Plane algorithm for producing vectors in a lattice near a given *target* vector. The distribution of vectors output from SamplePre has properties necessary for the security of the homomorphic signature scheme, but takes longer to execute than Nearest Plane. Thus we use the Nearest Plane algorithm to iteratively shorten the *target* vector to an acceptable length, then finally produce a *sample* vector from SamplePre using the shortened *target* vector.

Performance improvement

The results of implementing the imprecise GSO method are shown in Figure 4.3. Implementing the GSO algorithm using *big.Float* numbers in addition to using the iterative *target* shortening method resulted in Sign taking roughly half as long to complete for $n = 64$ compared to using a GSO with perfect precision. The variant of the Sign algorithm using an imprecise GSO and iterative *target* shortening appears to have achieved a reduction in execution time of roughly $O(n)$ after $n = 32$.

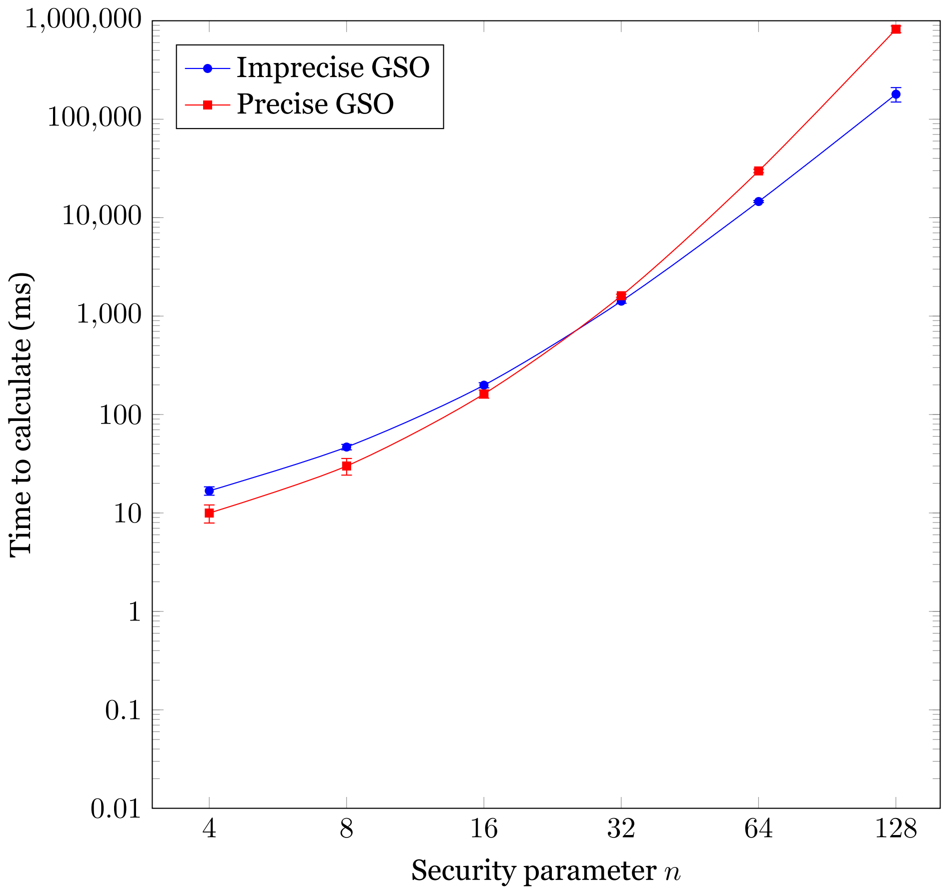


Figure 4.3: GSO optimization: Time to Sign

Interestingly, when we consider just the time it takes to sign 20 values without counting the additional overhead from calculating the GSO with perfect precision, both variants take almost the same amount of time to generate signatures. This is illustrated in Figure 4.4. The two variants take roughly the same amount of time and have roughly the same scaling with n despite the imprecise GSO version performing a series of transformations on *target*. This suggests that arithmetic involving rational numbers from the GSO with perfect precision carries an additional cost that is similar to that of iteratively shortening the *target* vector enough that an imprecise GSO is acceptable.

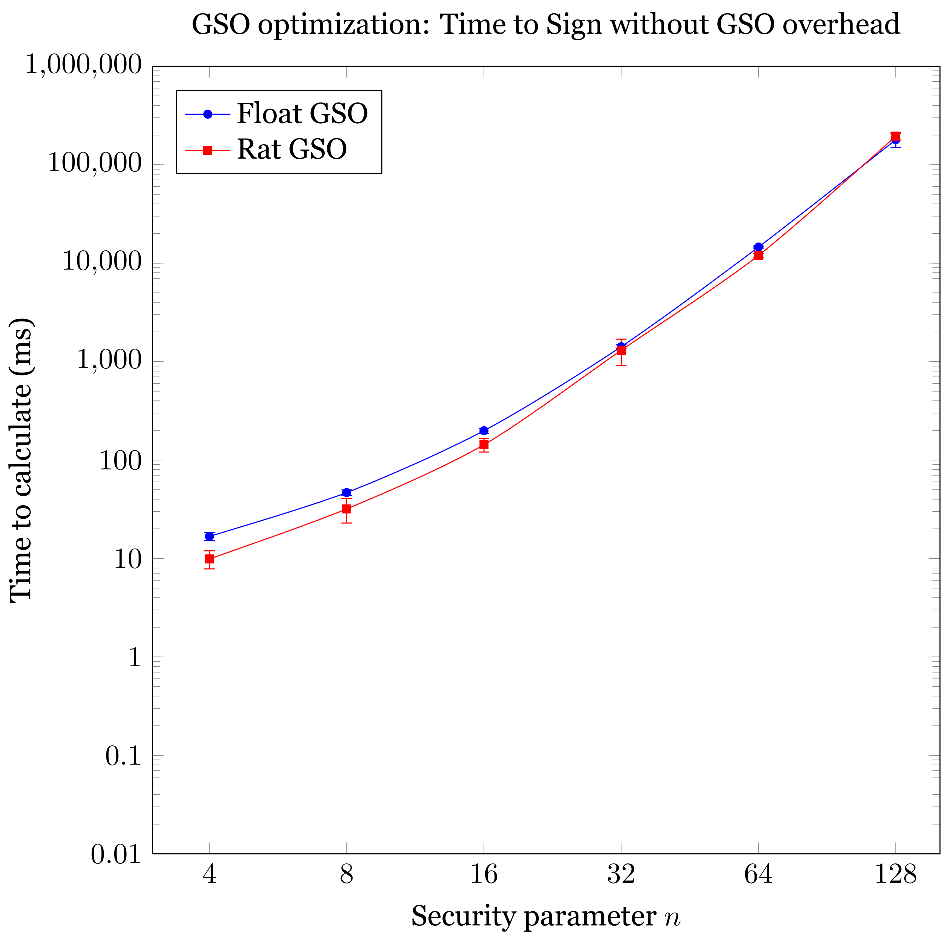


Figure 4.4: GSO optimization: Time to Sign without GSO overhead

Chapter 5

Conclusions and Further Work

We have presented our implementation of Boneh and Freeman's homomorphic signature scheme for polynomial functions in Go. We have given a description of key components of our implementation, including two significant performance optimizations. Lastly we have given a performance evaluation of our implementation, and measured the effects of our optimizations.

5.1 Further Work

We were not able to optimize our implementation enough to use more than 64-dimensional lattices and still get reasonable execution times on the computer we used for performance testing. Much work is yet to be done in making our implementation more efficient, particularly for the two slowest functions GenerateKey and Sign.

Multithreading

The parts of both GenerateKey and Sign that take the most time can fairly straightforwardly be made more efficient by making use of parallel processing.

GenerateKey spends a majority of the time it takes to execute searching for two prime ideal lattices. It does this by randomly sampling polynomials, then checking if these produce a prime ideal lattice. This process could be made parallel by starting a number of threads, each searching for prime ideal lattices in parallel. When two prime ideal lattices have been found, the threads would then be terminated before finishing the rest of GenerateKey.

Sign generates a signature for each value to be signed. Each signature is generated entirely independently from each other signature in the same batch. This process could be made parallel by simply spawning a new thread for each call of the function that generates signatures.

Bibliography

Dan Boneh and David Mandell Freeman. 2011. Homomorphic signatures for polynomial functions. https://theory.stanford.edu/ dfreeman/papers/homsigs.pdf

Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. 2008. Trapdoors for hard lattices and new cryptographic constructions. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*. 197–206.

Jason Y Jong. 2011. Homomorphic Signatures on Polynomial Functions. https://github.com/jasonyjong/Homomorphic-Signatures-for-Polynomial-Functions

Daniele Micciancio and Shafi Goldwasser. 2002. *Complexity of lattice problems: a cryptographic perspective*. Vol. 671. Springer Science & Business Media.

Daniele Micciancio and Oded Regev. 2007. Worst-case to average-case reductions based on Gaussian measures. *SIAM J. Comput.* 37, 1 (2007), 267–302.

Dingyi Pei, Arto Salomaa, and Cunsheng Ding. 1996. *Chinese remainder theorem: applications in computing, coding, cryptography*. World Scientific.

Chris Peikert. 2016. Lattice-Based Cryptography. https://www.youtube.com/watch?v=FVFw_qb1ZkY.

Anh Pham, Italo Dacosta, Guillaume Endignoux, Juan Ramon Troncoso Pastoriza, Kévin Huguenin, and Jean-Pierre Hubaux. 2017. {ORide}: A {Privacy-Preserving} yet Accountable {Ride-Hailing} Service. In *26th USENIX Security Symposium (USENIX Security 17)*. 1235–1252.

Nigel P Smart and Frederik Vercauteren. 2010. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *International Workshop on Public Key Cryptography*. Springer, 420–443.

Einar Haeger Solfjell, Hein Meling, and Rodrigo Queiroz Saramago. 2022. HSig. https://github.com/relab/hsig

Patrick Steuer and Victor Shoup. 2021. NTL – a library for doing numbery theory. https://github.com/libntl/ntl

University
of Stavanger