Abstract

This dissertation is devoted to implement a homomorphic signature scheme that is capable of evaluating both linears and polynomials on signed data. Given the public key and a signatures set of messages, this scheme can efficiently produce a signature on the mean, standard deviation, and other statistics of the signed messages.

The implementation presented in this dissertation is guilded by [5]. There are four algorithms created in the implementation, namely Setup, Sign, Verify, and Evaluate. The process of their construction is deeply discussed with examples, as well as specific algorithms within them.

Various mathematical theories are investigated in this dissertation to achieve the implementation. The majoy ones are lattices and ideals. The usage of them can be seen as an analogous signature version of Gentrys' fully homomorphic encryption[15].

Experimental results of core procedures in the implementation are analysed to study their time complexity. They compare the average time used by core procedures and nested for-loops, and they are fitted into curves to obtain an intuitive view.

Although this dissertation reached certain point of success, there are still many open problems and improvement space. A brief discussion of open problems of homomorphic signatures are also given lastly.

Acknowledgements

I would like to express my deep gratitude to Prof. Nigel Smart for his help, interesting discussions and constant encouragement through all the stages.

I wish to acknowledge the continuing support from the people of MS51 of the Department of Computer Science of the University of Bristol.

I would like to express my deep appreciation to my family. Whilst I have not seen them for a whole year whilst doing this course, their encouragement and support always were always with me. I also thank them for the financial support without which I could have never taken this course.

I would like to thank Xiaomeng Zhang for his help in C++.

I would like to thank Kevin Qian, Jessie Shen, and Ryan Qian for their help in proof reading.

Finally, I appreciate Chenjun Bao. She always prays for me.

Contents

CHAPTER 1

Introduction

We propose a solution to the new open problem of constructing a *fully homomorphic signature scheme*. This notion was introduced by Johnson, Molnar, Song and Wagner[17] when homomorphic encryption schemes had been studied for some time. Homomorphic signature schemes can be considered as an analogous problem of homomorphic encryption schemes but with different security properties. For example, basic RSA is a multiplicatively homomorphic encryption scheme, and it can be used for signature schemes – i.e., given RSA public key $\mathrm{pk} = (N, e)$ and signatures $\{\psi_i \leftarrow m_i^d \bmod N\}$, one can efficiently compute $\Pi_i \psi_i = (\Pi_i \pi_i)^d \bmod N$, a signature that signs the product of the original messages. This useful property led a natural question: what can one do with a signature scheme that is *fully* homomorphic: a scheme $\mathcal{S}$ with an efficient algorithm $\mathsf{Evaluate}$ that, for any valid sign key, any signatures $\psi_i \leftarrow \mathsf{Sign}(sk, m_i)$, outputs

$$\psi \leftarrow \mathsf{Evaluate}(pk, \psi_1, \ldots, \psi_t),$$

a valid signature under sk? Boneh and Freeman[5] answered: one can arbitrarily *compute* on *signed data* – i.e., one can process signed data without the signed key. As an application, they suggested private data mining service. A user can store its data on an untrusted server with signatures. Later, it can send a query on the data to the server, whereupon the server can express this query as a function to be applied to the data, and use the $\mathsf{Evaluate}$ algorithm to construct a signature as the response to the user's query, which the user then verifies. The server's response here is desired to be more concise than the trivial solution, in which the server just sends all of the signatures back to the user to process on its own.

1.1. A Brief and Informal Overview of Our Construction

Firstly, an example application[5] of computing on signed data is given for explanation convenience: Alice has a numerical data set $m_1, \ldots, m_n$ of size n. She independently adds a tag and an index to each datum m_i, and signs the triple (tag, m_i, i) for $i = 1, \ldots, n$ and obtains n independent signatures $\sigma_1, \ldots, \sigma_n$. For convenience, write $\vec{\sigma} := (\sigma_1, \ldots, \sigma_n)$. The data set and the n signatures are stored on some untrusted remote server.

Later, when clients ask the server to compute some authenticated functions of the data, such as the mean or variance of subsets of the data, the server apply an algorithm $\mathsf{Evaluate}(\mathsf{pk}, \cdot, f, \vec{\sigma})$ that uses $\vec{\sigma}$ and f to derive a signature σ on the triple

$$(tag, m := f(f_1, \dots, m_n), \langle f \rangle)$$

where $\langle f \rangle$ is an encoding of the function f. Now, the server publishes (m, σ) to clients, and clients can verify that σ is a signature on the triple to check that the server correctly applied f to the data set.

Our system uses two n-dimensional integer lattices Λ_1 and Λ_2 to generate the vector space for signing both the data and a description of the function f applied on the data. The lattice Λ_1 that defines the message space of $\mathbb{Z}^n/\Lambda_1$ is considered as a vector space over $\mathbb{F}_p$ for some prime p.

A signature in our system that does the "dual-role"signing is a short vector σ in $\mathbb{Z}^n$ in the intersection of $\Lambda_1 + \mathbf{u_1}$ and $\Lambda_2 + \mathbf{u_2}$ for some $\mathbf{u_1}, \mathbf{u_2} \in \mathbb{Z}^n$, which indicates $\sigma = \mathbf{u_1} \bmod \Lambda_1$ and $\sigma = \mathbf{u_2} \bmod \Lambda_2$. This method of jointly signing two vectors $\mathbf{u_1}$ and $\mathbf{u_2}$ is designed for preventing an attack from generating a vector σ' from σ such that $\sigma = \sigma' \bmod \Lambda_1$ but $\sigma \neq \sigma' \bmod \Lambda_2$[5].

Precisely speaking, a signature σ on a triple is a short vector in $\mathbb{Z}^n$ that satisfies $\sigma = m \bmod \Lambda_1$ and $\sigma = \omega_\tau(\langle f \rangle) \bmod \Lambda_2$, where ω_τ is a hash function that is defined by tag τ which encodes functions into vectors in $\mathbb{Z}^n/\Lambda_2$. To this degree, it can be seen that the hash function ω_τ does two significant contributions to our system. One is supporting the homomorphic properties of our system, and the other is keeping our system from a chosen-message adversary[15].

It can be seen that these signatures are *additively* homomorphic. Let σ_1 be a signature on a triple $(\tau, m_1, \langle f_1 \rangle)$ and let σ_2 be a signature on $(\tau, m_2, \langle f_2 \rangle)$. It can be ensure that $\sigma_1 + \sigma_2$ is a signature on $(\tau, m_1 + m_2, \langle f_1 + f_2 \rangle)$ with a good hash function ω_τ[5]. If $p = 2$, which indicates $\Lambda_1 = (2\mathbb{Z})^n$, it turns out to be a similar and more efficient linearly homomorphic signature over $\mathbb{F}_2$ than previous known[4].

Then we focus on polynomially homomorphic properties. ideal lattices are used to support our signature system with polynomial functions. This use of ideal lattice is a signature analogue of Gentry's somewhat homomorphic system[14]. Let $g \in \mathbb{Z}$ be a polynomial of degree n and let R be the ring $\mathbb{Z}[x]/(g)$. Then R is isomorphic to $\mathbb{Z}^n$[5] and ideals in R correspond to integer lattices in $\mathbb{Z}^n$ under the "coefficient embedding"[22]. Two lattices Λ_1 and Λ_2 in our system is chosen to be prime ideals $\mathfrak{p}$ and $\mathfrak{q}$ in R and a signature on the triple (τ, m, f) to be a short element in R such that $\sigma = m \bmod \mathfrak{p}$ and $\sigma = \omega_\tau(\langle f \rangle) \bmod \mathfrak{q}$. Now two signatures σ_1 and σ_2 are obtained, which respectively signs two triple (τ, m_1, f_1) and (τ, m_2, f_2). Then with an appropriate hash function ω_τ, it can be seen that [5]

$$\sigma_1 + \sigma_2 \text{ is a signature on } (\tau, m_1 + m_2, \langle f_1 + f_2 \rangle) \text{ and}$$
$$\sigma_1 \cdot \sigma_2 \text{ is a signature on } (\tau, m_1 \cdot m_2, \langle f_1 \cdot f_2 \rangle).$$

From this, any bounded degree polynomial with small coefficients can be evaluated on signatures.

1.2. Related Work

Before diving into the details, we briefly present some of the research literature related to fully homomorphic signature scheme. Micali's computationally sound (CS) proofs[23] gives a solution to the problem discussed above: Alice sends the data set D and the corresponding signature σ to the server. Later, for some function f, the server computes the result of applying f to the data set D as t, and publishes the triple (σ, t, π) where π is a short proof that the original data set D of σ and t exists. It uses the full machinery of the PCP theorem to construct π and the soundness is in the random oracle model. After Micali's, Kilian[19] achieves the same without the random oracle model.

Compared with our approach, the proof π is eliminated. The server does not compute the result t of applying the function f to the data set D but a signature σ' that is derived from σ and authenticates both t and f. In the meanwhile, constructing σ' takes about the same work as computing $f(D)$ and anyone can get an authentication of the result of applying further functions to t more simply by homomorphic signature than CS proofs[34].

More recently Lyubashevsky and Micciancio [21] present a one-time signature scheme that is constructed by lattices directly. This signature scheme can sign $O(n)$-bit messages in $\tilde{O}(n)$. The efficiency of the scheme and its security are both guaranteed by a special family in lattice that is called *ideal lattice*. From the one-time scheme a full signature scheme with similar asymptotic efficiency can be obtained by in corporating it into a standard tree structure[15].

Another mentionable related work is "redactable" signatures[2, 32, 18, 16, 7, 8]. The objective of these schemes in common is that they construct a signature on a message so that anyone can derive a signature on subsets of the message. This featured property gives a solution to computing on a subset of signed data which can be seen as a particular case in our focus – computing arithmetic functions on independently authenticated data.

1.3. Security

Unforgeablity. Generally speaking, a forgery under a chosen message attack is that an adversary generates a valid signature on $(\tau, m, \langle f)$ but the message m does not equal to the result of applying f to the data set with tag τ. For example, let $m_1, \ldots, m_k$ be the data set signed using tag τ, and a message $m = f(m_1, \ldots, m_k)$. Then an adversary generates a signature σ such that σ is valid signature on $(\tau, \mathrm{pk}, m' \neq m, \sigma, f)$. If a forgery as above exists, it can be used to solve the Small Integer Solution (SIS) problem in the lattice Λ_2[5], which is as hard as standard worst case lattice problem[25].

Privacy. Some applications need the signatures derived by Evaluate to be private. That is a signature on a message m that is the result of applying some function f on messages $m_1, \ldots, m_k$ should reveal no information about $m_1, \ldots, m_k$ besides what has been already revealed by m.

On one hand, in linearly homomorphic signatures, our construct satisfy a privacy property called *weak context hiding*[5, 4], which indicates the signature derived from Evaluate for linearly functions is still private.

On the other hand, in polynomially homomorphic signatures, it is difficult to tell whether our construct is private or not. It seems that the product $\sigma = \sigma_1 \cdot \sigma_2$ leaks information about the components σ_1 and σ_1 and also about the original messages. It is still an open problem either to find some reasonable computational assumption to support our construct to be private, or to design a polynomially homomorphic signature scheme which is also private.

Length efficiency. There is an additional requirement for homomorphic signature schemes that the length of derived signatures is not much longer than the original signatures from which they were derived.

CHAPTER 2

Definitions related to Homomorphic Signature

2.1. Basic Definitions

A conventional digital signature scheme $\mathcal{S}$ consists of three algorithms: $\mathsf{KeyGen}_{\mathcal{S}}$, $\mathsf{Sign}_{\mathcal{S}}$ and $\mathsf{Verify}_{\mathcal{S}}$. $\mathsf{KeyGen}_{\mathcal{S}}$ is a randomized algorithm that takes a security parameter n as input, and outputs a secret key sk as sign key and a public key pk as verify key; pk defines a message space $\mathcal{M}$ and a signature space Σ. $\mathsf{Sign}_{\mathcal{S}}$ is a randomized algorithm that takes sk and a message $m \in \mathcal{M}$ as input, and outputs a signature $\sigma \in \Sigma$. $\mathsf{Verify}_{\mathcal{S}}$ takes a message m, pk and a signature $\sigma \in \Sigma$ as input, and outputs 1(accept) or 0(reject) which indicates whether the signature σ authenticates the message m. The computational complexity of these algorithms must be polynomial in n. A minimal requirement of these algorithms is *correctness.*

Definition 2.1.1. *We say that a digital signature scheme is correct if, for any key-pair* (sk, pk) *generated by* $\mathsf{KeyGen}_{\mathcal{S}}(n)$*, any message* $m \in \mathcal{M}$*,*

$$\text{if } \sigma \leftarrow \mathsf{Sign}(sk, m), \text{ then } \mathsf{Verify}(pk, m, \sigma) = 1.$$

A homomorphic signature scheme comprises the usual algorithms KeyGen, Sign, Verify as well as an additional algorithm $\mathsf{Evaluate}$ that returns how functions applied on signatures rather than messages. The objective processed in a homomorphic signature scheme is also slightly different from a conventional one. Homomorphic signature schemes deal with data sets of messages, while conventional ones sign single messages. If $\vec{\sigma}$ is a valid set of signatures on messages $\vec{m}$, then $\mathsf{Evaluate}(f, \vec{\sigma})$ should be a valid signature for $f(\vec{m})$.

Additionally, since many data sets are processed in homomorphic signature schemes, especially when evaluating functions, the Sign, Verify and $\mathsf{Evaluate}$ algorithms take an extra "tag" as input[5]. The tag is used to bind messages from the same data set together so that they will not be mixed with messages from different data sets by those algorithms. There are many other ways to binding messages from the same data set. To avoid the tag, one could generate a new public key for each data set, but a new tag for each data set is more convenient.

Formally, a homomorphic signature scheme is as follows:

Definition 2.1.2. [5, Definition 2.1] *A homomorphic signature scheme is a tuple of probabilistic,*

polynomial-time algorithms (Setup, Sign, Verify, Evaluate) with the following
functionality[5]*:*

- $\mathsf{Setup}(1^n, k)$.
  Takes a security parameter n and a data set size k. Output a public key pk and a secret key sk. The public key pk defines a message space $\mathcal{M}$, a signature space Σ, and a set $\mathcal{F}$ of admissible functions $f : \mathcal{M}^k \to \mathcal{M}$.
  Let $\pi_i : \mathcal{M}^k \to \mathcal{M}$ be the function that projects onto the ith component; for every pk output by $\mathsf{Setup}(1^n, k)$, there is $\pi_i \in \mathcal{F}$ for $i = 1, ..., k$.
- $\mathsf{Sign}(\mathsf{sk}, \tau, m, i)$.
  Takes a secret key sk, a tag $\tau \in 0, 1^n$, a message $m \in \mathcal{M}$ and an index $i\ in \{1, ..., k\}$, and outputs a signature $\sigma \in \Sigma$.
- $\mathsf{Verify}(\mathsf{pk}, \tau, m, \sigma, f)$.
  Takes a public key pk, a tag $\tau \in \{0, 1\}^n$, a message $m \in \mathcal{M}$ and a signature $\sigma \in \Sigma$, and a function $f \in \mathcal{F}$, and outputs either 0 (reject) or 1 (accept).
- $\mathsf{Evaluate}(\mathsf{pk}, \tau, f, \sigma)$.
  Takes a public key pk, a tag $\tau \in \{0, 1\}^n$, a function $f \in \mathcal{F}$, and a tuple of signatures $\vec{\sigma} \in \Sigma^k$, and outputs a signature $\sigma' \in \Sigma$.

Correctness is still a minimal requirement which a homomorphic signature need to hold.

Definition 2.1.3. *(Correctness of Homomorphic Signature). We say that a homomorphic signature scheme $\mathcal{S}$ is correct for functions in $\mathcal{F}_{\mathcal{S}}$ if, for any key-pair(sk,pk) output by $\mathsf{Setup}_{\mathcal{S}}(1^n, k)$, any function f in $\mathcal{F}_{\mathcal{S}}$, any tag $\tau \in \{0, 1\}^n$, any message $m \in \mathcal{M}_{\mathcal{S}}$,any tuples $\vec{m} = (m_1, ..., m_k) \in \mathcal{M}^k_{\mathcal{S}}$, and any index $i \in \{1, \ldots, k\}$, it is the case that:*
if $\sigma \leftarrow \mathsf{Sign}(\mathrm{sk}, \tau, m, i)$, then

$$\mathsf{Verify}(\mathrm{pk}, \tau, m, \sigma, \pi_i) = 1,$$

if $\sigma_i \leftarrow \mathsf{Sign}(sk, \tau, m_i, i)$, then

$$\mathsf{Verify}(\mathrm{pk}, \tau, f(\vec{m}), \mathsf{Evaluate}(\mathrm{pk}, \tau, f, (\sigma_1, ..., \sigma_k)), f) = 1.$$

We also say a signature scheme as above is *$\mathcal{F}$-homomorphic*, or *homomorphic* with respect to $\mathcal{F}$[5].

Nevertheless, the Evaluate algorithm can take as input derived signatures themselves produced by Evaluate, by lots of iterations which will eventually leads to a point where the input signatures are valid, but the output signature is not. Therefore, a limited requirement of correctness property has been given here to simplify the discussion that Evaluate produce valid output when given as input signatures $\vec{\sigma}$ produced by the Sign algorithm.

Eventually, the definition of a almost-fully homomorphic signature scheme rises.

Definition 2.1.4. *(Almost-Fully Homomorphic Signature). We say that a homomorphic signature scheme $\mathcal{S}$ is fully homomorphic if it evaluates both linear functions and polynomial functions.*

2.2. Computational Security Definitions

Unforgeability. Unforgeability, informally, means it is computationally infeasible for an adaptive attacker to masquerade an honest sender in creating an authentic signed text that can be accepted by the verifying algorithm.

To be specific, when an adversary tries to get involved in homomorphic signatures, she is allowed to make adjustable signature queries on data sets of her choosing, each containing up to k messages, with the signer randomly choosing the tag τ for each data set queried.

After a sufficient preparation, the adversary produces a message-signature pare (m^*, σ^*) as well as an acceptable function f and tag τ^*. The winning condition corresponds two types of forgeries. In a *type 1 forgery*, the pair (m^*, σ^*) verifies for some data set not queried to the signer; this is the common notion of signature forgery. In a *type 2 forgery*, the pair (m^*, σ^*) verifies for some data set that was queried to the signer before, but m^* does not match the result of f applied to messages queried. In other words, a type 2 forgery indicates that the signature σ^* authenticates m^* as $f(\vec{m})$ which in fact is not true.

Definition 2.2.1. [5, Definition 2.2] *We say a homomorphic signature scheme $\mathcal{S}$ is unforgeable if for all k and all probabilistic polynomial-time adversaries $\mathcal{A}$, the advantage of $\mathcal{A}$ in the following game is negligible in the security parameter n:*

- **Setup:** The challenger runs $\mathsf{Setup}(1^n, k)$ to obtain (pk,sk) and publishes pk to $\mathcal{A}$. The public key pk defines a message space $\mathcal{M}$, a signature space Σ, and a set $\mathcal{F}$ of acceptable functions $f : \mathcal{M}^k \to \mathcal{M}$.
- **Queries:** $\mathcal{A}$ specifies a sequence of data sets $\vec{m}_i \in \mathcal{M}^k$ and send them to the challenger. In response, The challenger chooses τ_i uniformly from $0, n^n$ for each i, computes the signatures $\sigma_i j \leftarrow \mathsf{Sign}(sk, \tau_i, m_i j, j)$ for $j = 1, \dots, k$, and sends both of them to $\mathcal{A}$.
- **Output:** $\mathcal{A}$ generates a tag $\tau^* \in 0, 1^n$, a message $m^* \in \mathcal{M}$, and a signature $\sigma^* \in \Sigma$ for a function $f \in \mathcal{F}$, and send them to the challenger to verify.
- **Limitation:** The adversary $\mathcal{A}$ cannot request signatures on new messages m after seeing signatures on other messages in the same data set.

The adversary *wins* if $\mathsf{Verify}(pk, \tau^*, m^*, \sigma^*, f) = 1$, either

(1) $\tau^* \neq \tau_i$ for all i (a *type 1 forgery*), or

(2) $\tau^* = \tau_i$ for certain i but $m^* \neq f(\vec{m}_i)$(a *type 2 forgery*).

The advantage of $\mathcal{A}$ is defined to be the probability that $\mathcal{A}$ wins the security game.

Privacy. Privacy, generally speaking in cryptography, means it should be computationally infeasible for an adaptive attacker to gain any partial information on the contents of messages. In this settings, privacy means that even given signatures on a data set $\vec{m} \in \mathcal{M}^k$, the derived signatures on results of $f_1(\vec{m}), \dots, f_t(\vec{m})$ don't give away any information about $\vec{m}$ besides what has been leaked by the results[4].

More specifically, privacy for homomorphic signatures is defined by the idea of "adversaries can not distinguish"[5]. The concept is similar to witness indistinguishability [12] from which the attacker is given signatures on many messages derived from two different data sets such that she can not tell which data set the derived signatures came from. This privacy property is *weakly context hiding*, which indicates the fact that derivation took place is not hidden. In addition, another limitation of this privacy property is that the original signatures are assumed as not public.

Definition 2.2.2. [5, Definition 2.3] *We say that a homomorphic signature scheme $\mathcal{S}$ is weakly context hiding if for any fixed value of k the advantage of any probabilistic, polynomial-time adversary $\mathcal{A}$ in the following game is negligible in the security parameter n:*

Setup: The challenger runs $\mathsf{Setup}(1^n, k)$ to obtain (pk,sk) and publishes (pk,sk) to $\mathcal{A}$. The public key pk defines a message space $\mathcal{M}$, a signature space Σ, and a set $\mathcal{F}$ of acceptable functions $f : \mathcal{M}^k \to \mathcal{M}$.

Challenge: $\mathcal{A}$ generates two data sets $(\vec{m}_0^*, \vec{m}_0^*) \in \mathcal{M}^k$, then picks many functions $(f_1, \dots, f_s) \in \mathcal{F}$, which satisfy

$$f_i(\vec{m}_0^*) = f_i(\vec{m}_0^*) \quad \text{for all} i = 1, \dots, s.$$

$\mathcal{A}$ sends $(\vec{m}_0^*, \vec{m}_0^*, f_1, \dots, f_s)$ to the challenger. In response, the challenger flips a fair coin $b \in \{0, 1\}$ and pick a random tag. Then, k signatures in a vector $\vec{\sigma}$ are signed from messages in $\vec{m}_b^*$ with the random tag. Next, the challenger derives a signature σ_i on $f_i(\vec{m}_b^*$ from $\vec{\sigma}$ for $i = 1, \dots, s$, and send s signatures to $\mathcal{A}$.

Output: $\mathcal{A}$ outputs a bit b'.

The adversary $\mathcal{A}$ wins the game if $b = b'$. The advantage of $\mathcal{A}$ is the probability that A wins the game.

If an adversary won the challenge game, it meant that the adversary could tell the difference between signatures of a response from signatures on $\vec{m}_0^*$ or $\vec{m}_i^*$.

Length efficiency. Length efficiency is a special criterion for homomorphic signatures, which means there is some limitation on the length of derived signatures. More precisely, it has been pointed out that the length of derived signatures should depends only logarithmically on the size k of the data set[5].

Definition 2.2.3. [5, Definition 2.4] *We say that a homomorphic signature scheme $\mathcal{S}$ is length efficient if there is some function $\mu : \mathbb{N} \to \mathbb{R}$ such that for all (pk,sk) output by $\mathsf{Setup}(1^n, k)$, for all $\vec{m} = (m_1, \ldots, m_k) \in \mathcal{M}^k$ ($\mathcal{M}^k$ is defined by pk), for all tags $\tau \in \{0,1\}^n$, and all functions $f \in \mathcal{F}$ ($\mathcal{F}$ is defined by pk), if*

$$\sigma_i \leftarrow \mathsf{Sign}(\mathrm{pk}, \tau, m_i, i) \quad \text{for } i = 1, \ldots, k, \text{ and } \quad \vec{\sigma} := (\sigma_1, \ldots, \sigma_k)$$

then the bit length of the derived signature σ output by $\mathsf{Evaluate}$(pk, $\tau, f, \vec{\sigma}$) is at most

$$length(\sigma) \leq \mu(n) \cdot \log k$$

with overwhelming probability for all $k > 0$.

CHAPTER 3

Background on Lattices and Ideal Lattices

In this Chapter, we provide some basic background material needed to instantiate what is lattice, what is ideal lattice, and how to generate them. They are used to construct our abstract homomorphic signature scheme in next chapter.

3.1. Basic Background on Lattice

Let $\mathbb{R}$ denote the real numbers, and $\mathbb{Z}$ denote the integers. Vectors are written in column form using bold lower-case letters, e.g. $\mathbf{v}$; Matrices are written as bold capital letters, e.g., $\mathbf{B}$; $\mathbf{b}_i$ is the i-th column. $\|\mathbf{v}\|$ is used to denote the Euclidean length of a vector $\mathbf{v}$. For matrix $\mathbf{B}$, $\|\mathbf{B}\|$ is used to denote the length of the longest column vector in $\mathbf{B}$.

In our system, a *lattice* has been concerned as a discrete additive subgroups of $\mathbb{Z}^m$ having finite index. The determinant of Λ is the cardinality of the quotient group $\mathbb{Z}^m/\Lambda$. Geometrically, the determinant is used to describe how sparse the lattice is.

Changing the point of view, a lattice Λ can be considered as a set of all integer linear combinations of m linearly independent basis vectors $\mathbf{B}= \{\mathbf{b}_1, \ldots, \mathbf{b}_m\} \subset \mathbb{Z}^m$[3]:

$$\Lambda = \mathcal{L}(B) = \left\{ \mathbf{Bc} = \sum_{i\in[m]} c_i \mathbf{b}_i : \mathbf{c} \in \mathbb{Z}^m \right\}.$$

For example, let

$$B = \begin{bmatrix} p & 0 & \cdots & \cdots & 0 \\ 0 & p & 0 & \cdots & \vdots \\ \vdots & \vdots & \ddots & \cdots & \vdots \\ \vdots & \vdots & 0 & p & 0 \\ 0 & \cdots & \cdots & 0 & p \end{bmatrix},$$

and Λ can be calculated as

$$\Lambda = p\mathbb{Z}^n = \{p \cdot (z_1, \ldots, z_n) : z_i \in \mathbb{Z}\}$$

$$= \{z_1 \cdot (p, 0, \ldots, 0) + z_2 \cdot (0, p, 0, \ldots, 0) + \ldots + z_n(0, \ldots, 0, p) : z_i \in \mathbb{Z}\}$$

$$= \left\{ \begin{pmatrix} p & 0 & \cdots & \cdots & 0 \\ 0 & p & 0 & \cdots & \vdots \\ \vdots & \vdots & \ddots & \cdots & \vdots \\ \vdots & \vdots & 0 & p & 0 \\ 0 & \cdots & \cdots & 0 & p \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_{n-1} \\ z_n \end{pmatrix} : z_i \in \mathbb{Z} \right\}$$

One thing here worth stating is that there are infinitely many bases those which can generate the same lattice when $m \geq 2$. Even though the absolute value of the determinant of the vectors $\mathbf{b}_i$ is uniquely determined by Λ, and is denoted by $det(\Lambda)$. These bases are related to each other by unimodular transformations – $\mathbf{B} = \mathbf{B}' \cdot \mathbf{U}$ that always exists for some unimodular $\mathbf{U} \in \mathbb{Z}^{m \times m}$.

Hermite Normal Form of A Lattice. As mentioned above, every lattice has an infinite number of lattice bases. Informally, these bases can be devided into two groups[14] – a "good" group and a "bad" group. The bases in "good" group have reasonably short and nearly orthogonal vectors. For any basis $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$, it holds that $\Pi_{i=1}^{n} \|\mathbf{b}_i\| \geq det(\Lambda)$; $\Pi_{i=1}^{n} \|\mathbf{b}_i\|$ of "good" bases is more likely to reach the equality to $det(\Lambda)$.

Among all of these infinite many bases of a lattice Λ, there exists a *worst* basis that is the Hermite Normal Form of Λ. In linear algebra, the Hermite normal form is an analogue of reduced echelon form for matrices over $\mathbb{Z}$, and it should have following mathematical properties:

(1) $\mathbf{H}$ is an *Upper Trianguluar Matrix*
(2) For $i \in [m]$, $h_{i,i} > 0$
(3) For $j \in [m]$, $\sum_{i \in [j]} h_{i,j} \cdot \mathbf{a}_i = \mathbf{0} \in \mathbb{Z}_q^n$
(4) For $i \in [m]$, $h_{i,i} \leq q$, proved by Section 2.4.1 in [3]
(5) $\mathbf{H}' = \mathbf{H} - \mathbf{I}$

Given any basis $\mathbf{B}$ of a lattice Λ, one can compute HNF(Λ) efficiently in time poly(n, lg($\|\mathbf{B}\|$)). The reason for "worst" is that HNF(Λ) does not leak more information about the structure of Λ than any other basis. Therefore, HNF(Λ) is a good choice to be the public lattice basis in public key[24].

Hard Random Lattices. A certain family of lattices in $\mathbb{Z}^n$ is peculiarly concerned, which is often unusually specified by a matrix $\mathbf{A} \in \mathbb{Z}_q^{l \times n}$ for some positive integer l and positive integer modulus q. The latice associated with $\mathbf{A}$ is defined as

$$\Lambda^{\perp}(\mathbf{A}) = \left\{ \mathbf{x} \in \mathbb{Z}^n : \mathbf{A}\mathbf{x} = \sum_{j \in [n]} x_j \cdot \mathbf{a}_j = 0 \in \mathbb{Z}_q^l \right\}.$$

It can be seen that $\Lambda^{\perp}(\mathbf{A})$ is closed under negation and addition, as well as it is ‘q-ary’.

For instance, let $l = 3, n = 4, q = 29$, and

$$A = \begin{bmatrix} 4 & 7 & 11 & 10 \\ 8 & 3 & 2 & 1 \\ 25 & 7 & 13 & 8 \end{bmatrix}.$$

When we try to compute $\Lambda^{\perp}(\mathbf{A})$, firstly the kernel of $\mathbf{A}$ in modulus q is needed to compute. That is, in linear algebra, the set of all vectors $\mathbf{x}$ for which $\mathbf{Ax} = \mathbf{0}$, where $\mathbf{0}$ denotes the zero vector with n components. That is

$$\mathrm{Ker}(\mathbf{A}) = \begin{bmatrix} 15 \\ 6 \\ 3 \\ 1 \end{bmatrix}.$$

Then a lattice basis of "$\mathbf{Ax} = \mathbf{0}$" is about to form. However, Lattices are over $\mathbb{Z}$, while the matrix $\mathrm{Ker}(\mathbf{A})$ is over $\mathbb{F}_q$. Thus, the kernel matrix $\mathrm{Ker}(\mathbf{A})$ is expanded with another n columns to $\left(\mathrm{Ker}(\mathbf{A}) \| q \cdot \mathbf{I}_n\right)$. It generates (by the columns)a basis of the lattice in $\mathbb{Z}^n$.

$$\left(\mathrm{Ker}(\mathbf{A}) \| q \cdot \mathbf{I}_n\right) = \begin{bmatrix} 15 & 0 & 0 & 0 & 0 \\ 6 & 29 & 0 & 0 & 0 \\ 3 & 0 & 29 & 0 & 0 \\ 1 & 0 & 0 & 29 & 0 \end{bmatrix}.$$

Finally, calculate its Hermite Normal Form.

$$\mathrm{HNF}\left(\mathrm{Ker}(\mathbf{A}) \| q \cdot \mathbf{I}_n\right) = \begin{bmatrix} 29 & 0 & 0 & 15 \\ 0 & 29 & 0 & 6 \\ 0 & 0 & 29 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

3.2. Basic Background on Ideal Lattice

In out system, the homomorphic signature scheme is based on lattices. According to [5], replacing general lattices to ideal lattices in linearly homomorphic signature scheme will improve it from for linear functions to for polynomial functions – the ultimate goal of this dissertation. Therefore, a survey of *ideal lattice* is needed, and the difference between them leads to the key of this thesis.

Ideal, Number Ring and Ideal Lattice. Generally, ideal lattices are a special class of lattices and a generalization of cyclic lattices[20], which unfortunately is not sufficient to our goal. According to [31], ideal lattice is considered as a special subset of lattices that posses the computationally interesting property of being related to structured matrices and polynomials.

Let a number field K represented as $\mathbb{Q}[x]/f(x)$ be a finite-degree algebraic extension of the rational numbers $\mathbb{Q}$. Its identification leads to a multiplicative structure on $\mathbb{Q}^n$ in addition to the usual additive structure[5]. Then how much the multiplication increases should be determined. Define a parameter[5, 31]:

$$\gamma_f := \sup_{u,v \in K} \frac{\|u \cdot v\|}{\|u\| \cdot \|v\|}.$$

This parameter bounds the length of the product increased by the multiplication, relative to the product of the length of the factors. In this implementation, the requirement is that $\gamma_f = \text{poly}(n)$. Let us focus on the function $f(x) = x^n + 1$ and n is a power of 2. Then, according to *Lemma 7.4.3* in [14] it can be obtained that $\gamma_f \leq \sqrt{n}$, which implies this $f(x)$ is a good choice.

Every number field has a ring of integers denoted by $\mathcal{O}_K$, and there is the subring $R = \mathbb{Z}[x]/f(x)$ inside $\mathcal{O}_K$. In other words, R is a proper sublattice of $\mathcal{O}_K$.

Let I be an ideal of R, i.e. a subset of $\mathbb{Z}[x]/f(x)$, that is closed under addition and multiplication by elements of $\mathbb{Z}[x]/f(x)$. By the identification of R with $\mathbb{Z}^n$, I is a sublattice of R[14], therefore also called an ideal lattice.

There are some key points worth stating from [33]:

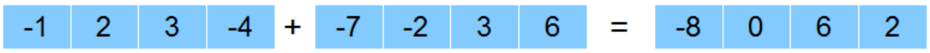
- The usage of "ideal lattice"to refer to a rank one R-module differs from that of [31] of which "ideal lattice"refers to R-modules of arbitrary rank, whereas many basic ideas are in common.
- An ideal lattice I is considered as *prime* if for $x, y \in R$, $xy \in I$ infers either $x \in I$ or $y \in I$.
- If $\mathfrak{p}$ is a *prime* ideal lattice of R, $\mathfrak{p}$ can be written as $\mathfrak{p} = (p + h(x)) \cdot R$, of which the reduction mod p of polynomial $h(x)$ is an irreducible factor of $f(x)$ mod p.

Basically, to go from the linear homomorphic signature scheme to the polynomial scheme the **keys** need to be replaced to more powerful keys– ideal lattices. In ideal lattice, the *multiply* operation can be feasible as well as the *add* operation, and its result remains in the lattice, which implies Evaluate can be extended to multiplication of signatures. This is the key of polynomial homomorphic signature schemes.

Example: Cyclic Lattices and Ideals in $\mathbb{Z}[x]/(x^n-1)$. Here an example of cyclic lattices is given as well as ideals in $\mathbb{Z}[x]/(x^n-1)$ to reveal some properties of ideal lattices more clearly.

Let a lattice Λ_c be a cyclic lattice in $\mathbb{Z}^n$. Then it has three important properties:

(1) For all $\mathbf{v}, \mathbf{w}$ in Λ_c, $\mathbf{v}+\mathbf{w}$ is also in Λ_c:

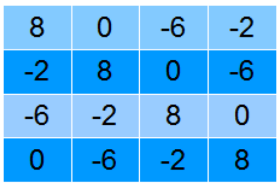


-1	2	3	-4	+	-7	-2	3	6	=	-8	0	6	2

(2) For all $\mathbf{v}$ in Λ_c, $-\mathbf{v}$ is also in Λ_c:

-8	0	6	2		8	0	-6	-2

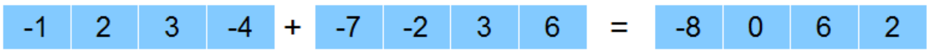
(3) For all $\mathbf{v}$ in Λ_c, a cyclic shift of $\mathbf{v}$ is also in Λ_c:



8	0	-6	-2
-2	8	0	-6
-6	-2	8	0
0	-6	-2	8

At the same time, we take another point of view and consider this lattice Λ_c as a ideal. We found that a cyclic lattice in $\mathbb{Z}^n$ is an ideal in $\mathbb{Z}[x]/(x^n-1)$. Then it still has three important properties:

(1) For all $\mathbf{v}, \mathbf{w}$ in Λ_c, $\mathbf{v}+\mathbf{w}$ is also in Λ_c:



-1	2	3	-4	+	-7	-2	3	6	=	-8	0	6	2

$$(-1+2x+3x^2-4x^3)+(-7-2x+3x^2+6x^3)=(-8+0x+6x^2+2x^3)$$

(2) For all $\mathbf{v}$ in Λ_c, $-\mathbf{v}$ is also in Λ_c:

(3) For all $\mathbf{v}$ in Λ_c, a cyclic shift of $\mathbf{v}$ is also in Λ_c:

-8	0	6	2

8	0	-6	-2

$(-1+2x+3x^2-4x^3)$ and $(1-2x-3x^2+4x^3)$

8	0	-6	-2
-2	8	0	-6
-6	-2	8	0
0	-6	-2	8

$$-1+2^x+3x^2-4x^3$$
$$(-1+2x+3x^2-4x^3)x=-4-x+2x^2+3x^3$$
$$(-1+2x+3x^2-4x^3)x^2=3-4x-x^2+2x^3$$
$$(-1+2x+3x^2-4x^3)x^3=2+3x-4x^2-x^3$$

CHAPTER 4

Homomorphic Signature Schemes

4.1. A Linearly Homomorphic Signature Scheme

The implemented homomorphic signature scheme in this dissertation are built on the “hash-and-sign”signatures of Gentry, Peikert, and Vaikuntanathan[15].

First of all, let us review how GPV signatures work. In KeyGen, a public key and a secret key are generated. The public key is a lattice $\Lambda \subset \mathbb{Z}^n$, and the secret key is the short basis of Λ. In Sign, a message m is hashed to an $H(m) \in \mathbb{Z}^n/\Lambda$, and a short vector σ is sampled from the coset of Λ defined by $H(m)$. In Verify, two things are checked: σ is short and $\sigma \bmod \Lambda = H(m)$.

Then some changes are made for adapting it to homomorphic signatures. The target which needs to be authenticated here is the triples $(\tau, m, \langle f \rangle)$, of which τ is a “tag”attached to a data set, m is a message in $\mathbb{F}_p^n$, and $\langle f \rangle$ is an encoding of a function f acting on k-tuples of messages. Now a formal description of the scheme from [5] is introduced, which is the base of this implementation:

The linearly homomorphic signature scheme.

$\mathsf{Setup}(1^n, k)$.
    (n– a security parameter,
    k– a data set size):

(1) Choose two primes $p, q = \text{poly}(n)$ with $q \geq (nkp)^2$. Define $l := \lfloor n/6 \log q \rfloor$.
(2) Set $\Lambda_1 := p\mathbb{Z}^n$.
(3) Use $\mathsf{TrapGen}(q, l, n)$ to generate a matrix $\mathbf{A} \in \mathbb{F}_q^{l \times n}$ along with a short basis T_q of $\Lambda_q^{\perp}(\mathbf{A})$. Define $\Lambda_2 := \Lambda_q^{\perp}(\mathbf{A})$ and $\mathbf{T} := p \cdot \mathbf{T}_q$. Note that $\mathbf{T}$ is a basis of $\Lambda_1 \cap \Lambda_2 = p\Lambda_2$.
(4) Set $v := p \cdot \sqrt{n \log q} \cdot \log n$.
(5) Let $H : \{0,1\}^* \to (\mathbb{F}_q^l)$ be a hash function.
(6) Output the public key $\mathsf{pk}:= (\Lambda_1, \Lambda_2, v, k, H)$ and the ssecret key $\mathsf{sk}=\mathbf{T}$.

$\mathsf{Sign}(\mathsf{sk}, \tau, m, i)$.
    (sk– a secret key,
    τ– a tag in $\{0,1\}^n$,
    m– a message in $\mathbb{F}_p^n$,
    i– an index):

(1) Compute $\alpha_i := H(\tau \| i) \in \mathbb{F}_q^l$. $\omega_\tau(\langle \pi_i \rangle) = \alpha_i$.
(2) Compute $\mathbf{t} \in \mathbb{Z}^n$ such that $\mathbf{t} \bmod p = m$ and $\mathbf{A} \cdot \mathbf{t} \bmod q = \alpha_i$.
(3) Output $\sigma \leftarrow \mathsf{SamplePre}(\Lambda_1 \cap \Lambda_2, \mathbf{T}, \mathbf{t}, v) \in (\Lambda_1 \cap \Lambda_2) + \mathbf{t}$.

$\mathsf{Verify}(\mathsf{pk}, \tau, m, \sigma, f)$
    (pk– a public key,
    τ– a tag in $\{0,1\}^n$,
    m– a message in $\mathbb{F}_p^n$,
    σ– a signature in $\mathbb{Z}^n$,
    f– a function in $\mathcal{F}$)

(1) If all of the following conditions hold, output 1 (accept); otherwise output 0 (reject):
    (a) $\|\sigma\| \leq k \cdot \frac{p}{2} \cdot v\sqrt{n}$.
    (b) $\sigma \bmod p = m$.
    (c) $\mathbf{A} \cdot \sigma \bmod q = \omega_\tau(\langle f \rangle)$.

$\mathsf{Evaluate}(\mathsf{pk}, \tau, f, \vec{\sigma})$.
    (pk– a public key,
    τ– a tag in $\{0,1\}^n$,
    f– a function in $\mathcal{F}$ encoded as $\langle f \rangle = (c_1, \ldots, c_k) \in \mathbb{Z}^k$ with $c_i \in (-p/2, p/2]$,
    $\vec{\sigma}$– a tuple of signatures $\sigma_1, \ldots, \sigma_k$ in $\mathbb{Z}^n$)

(1) Output $\sigma := \Sigma_{i=1}^k c_i \sigma_i$.

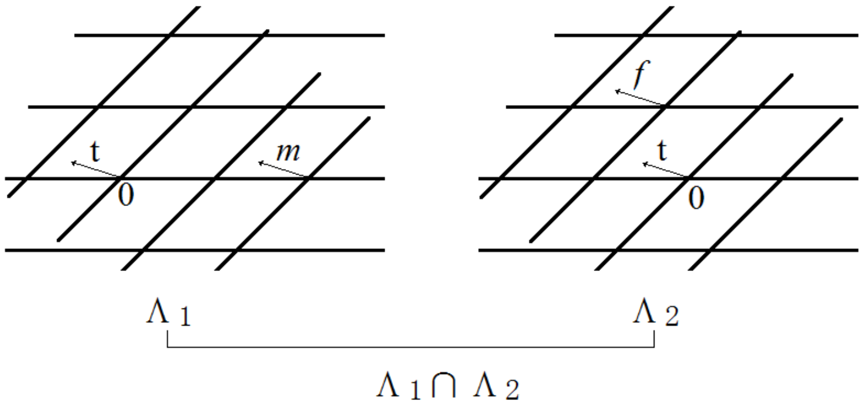


FIGURE 4.1.1. An example of the intersection of two 2-d lattices

This is a complicated homomorphic signature scheme for linear functions. Setup is an algorithm like KeyGen in all other signature schemes. Its purpose is to generate public key pk and secret key sk as well as define system parameters. Public key pk consists of two lattices, k, v and a hash function H. The reason why there are two lattices in public key is that one needs to verify a GPV signature which is a signature on both a message and a hash of an encoding of a function $\langle f \rangle$ to authenticate both the message and the function as well as bind them together. Putting message m in $\mathbb{Z}^n/\Lambda_1$ and hashed encoding function $\omega_\tau(\langle f \rangle)$ in $\mathbb{Z}^n/\Lambda_2$ gives a opportunity to apply Chinese remainder theorem that would define a coset of $\Lambda_1 \cap \Lambda_2$ in $\mathbb{Z}^n$. Then $\mathsf{Sign}(\mathsf{sk}, \tau, m, i)$ algorithm signs the message m and a projection function that projects the ith message out of a message set $m_1, \ldots, m_k$, and output a short vector to locate the message and the function in the coset defined by a short basis of $\Lambda_1 \cap \Lambda_2$. Evaluate algorithm is analogical as Sign with acceptable functions well hashed. Eventually, Verify checks a signature's length and its property– $\sigma \bmod \Lambda_1 = m$ and $\sigma \bmod \Lambda_2 = \omega_\tau(\langle f \rangle)$ to accept the signature or not.

Now the requirement of *correctness* of the linearly homomorphic signature scheme define above is calculated.

First, we check the three verification conditions for the signature on a message related to a projection function π_i defined by $\pi_i(m_1, \ldots, m_k) = m_i$ and encoded as $\langle \pi_i \rangle = \mathbf{e}_i$. Let $\tau \in \{0,1\}^n$, $m \in \mathbb{F}_p^n$, and $i \in \{1, \ldots, k\}$. Then by algorithm Sign in the scheme, it can be obtained that

$$\sigma \leftarrow \mathsf{Sign}(\text{sk}, \tau, m, i).$$

(1) From the definition of algorithm TrapGen in Sign, it can be gained $\|\tilde{\mathbf{T}}\| \leq O(p \cdot \sqrt{l \log q})$, and therefore $v \geq \|\tilde{\mathbf{T}}\| \cdot \omega(\sqrt{\log l})$[3, Theoren 3.2]. Then it can be concluded that $\|\sigma\| \leq v\sqrt{n}$ exists with overwhelming probability by Lemma 4.4 in [25].

(2) By correctness of $\mathsf{SamplePre}$ defined in [15], it can be gained $\sigma \in (\Lambda_1 \cap \Lambda_2) + \mathbf{t}$, where $\sigma \bmod p = \mathbf{t}$. By definition of $\mathbf{t}$ in algorithm Sign, $\mathbf{t} \bmod p = m$.

(3) As above, we have $\sigma \in (\Lambda_1 \cap \Lambda_2) + \mathbf{t}$. Since Λ_2 is defined by a matrix $\mathbf{A}$ in $\mathbb{Z}_q$, $\mathbf{A} \cdot \sigma \bmod q = \alpha_i$ is obtained, where α_i is defined as $\omega_\tau(\langle \pi_i \rangle)$ in Sign.

Then, we discuss the situation of the signature on the result of applying a function on messages. Here the function applied on messages can be any of acceptable functions in $\mathcal{F}$, and it is encoded as $\langle f \rangle = (c_1, \ldots, c_k) \in \mathbb{Z}^k$, where $|c_i| \le p/2 \in \mathbb{Z}$. Let $\tau \in \{0,1\}^n$, $\vec{m} = (m_1, \ldots, m_k) \in (\mathbb{F}_p^n)^k$, and $\vec{\sigma} = (\sigma_1, \ldots, \sigma_k)$ with $\sigma_i \leftarrow \mathsf{Sigh}(\mathrm{sk}, \tau, m_i, i)$. Then by algorithm $\mathsf{Evaluate}$ in the scheme, we have

$$\sigma' \leftarrow \mathsf{Evaluate}(\mathrm{pk}, \tau, f, \vec{\sigma}) = \Sigma_i c_i \sigma_i.$$

Next, check the three verification conditions:

(1) Since $|c_i| < p/2$ for all i, it can be seen that

$$\|\sigma'\| = \Sigma_{i=1}^k c_i \|\sigma_i\| \le k \cdot \frac{p}{2} \cdot \max\{\|\sigma_i\| : \sigma_i \in \vec{\sigma}\}.$$

By Lemma 4.4 in [25], it can be achieved that

$$\|\sigma'\| \le k \cdot \frac{p}{2} \cdot v\sqrt{n}.$$

(2) By correctness of individual signatures, $\sigma_i \bmod p = m_i$ for $i \in \{i, \ldots, k\}$ is gained. Thus,

$$\sigma' \bmod p = \sum_i c_i \sigma_i \bmod p = \sum_i c_i m_i = f(\vec{m}).$$

(3) By correctness of individual signatures, $\mathbf{A} \cdot \sigma_i \bmod q = \alpha_i$ for $i \in \{1, \ldots, k\}$, which follows that

$$\begin{aligned} \mathbf{A} \cdot \sigma' \bmod q &= \sum_i \mathbf{A} \cdot \sigma_i \bmod q \\ &= \sum_i c_i \alpha_i \bmod q \\ &= \omega_\tau(\langle f \rangle). \end{aligned}$$

Unforgeability. Unforgeability of this linearly homomorphic scheme is based on that it is difficult to find a short nonzero vector in Λ_2[5].

It can be supposed that an adversary who forges a signature for the linearly homomorphic scheme can be quite able to compute a short vector in the lattice Λ_2 computed in Step 3 of Setup. By *Theorem 3* in [3], the distribution of matrices $\mathbf{A}$ which is used to define Λ_2 is statistically close to uniform over $\mathbb{F}_q^{l \times n}$. Therefore, the distribution of lattices Λ_2 is statistically close to the distribution of challenges for the $\mathsf{SIS}_{q,n,\beta}$ problem (for any β)[25, 5].

According to *Section 4.2 and Section 7.3* in [5], it is concluded that "if $\mathsf{SIS}_{q,m,\beta}$ is infeasible for $\beta = k \cdot p^2 \cdot n \log n \sqrt{\log q}$, then the linearly homomorphic signature cheme defined above is unforgeable".

Worst-case connections. To apply the conclusion mentioned above, restrictions should be satisfied. In worst case, the $\mathsf{SIS}_{q,m,\beta}$ problem is as hard as approximating the SIVP problem within $\beta \cdot \widetilde{O}(\sqrt{n})$, when $q \geq \beta \cdot \omega(\sqrt{n \log n})$[15]. Indeed, the requirement in Setup that $q \geq (nkp)^2$ ensures that q is large enough for the conclusion above to apply.

Privacy. Now whether the linearly homomorphic signature scheme is weakly context hiding should be discussed. To determine its privacy property, the privacy challenge game mentioned in Section 2.2 is needed. In the privacy game, an adversary gives the challenger two message set $\vec{m}_0^*, \vec{m}_1^*$ that contain k messages each and multiple acceptable functions $f_1, \ldots, f_s$. It is defined that all functions received by the challenger satisfy:

$$f(\vec{m}_1^*) = f(\vec{m}_1^*) \in \mathbb{F}_q^n, \quad \text{for all } i = 1, \ldots, s.$$

In the scheme, $\Lambda_1 \cap \Lambda_2$ and τ are used to answer the challenge. For $j = 1, \ldots, k$, let $\sigma_j^{(0)}$ and $\sigma_j^{(1)}$ be the challenger's signatures on messages in $\vec{m}_0^*$ and $\vec{m}_1^*$, respectively. For $i = 1, \ldots, s$, let $d_i^{(b)}$ be the derived signature on result $f_i(\vec{m}_b^*)$ by applying $\mathsf{Evaluate}$ to signature sets $\vec{\sigma}^{(b)} = \{\sigma_1^{(b)}, \ldots, \sigma_k^{(b)}\}$ and the function f_i, $b = \{0, 1\}$. Consider $\sigma_j^{(b)}, d_i^{(b)}$ and f_i as matrix $E^{(b)} \in \mathbb{Z}^n, D^{(b)} \in \mathbb{Z}^n$ and $F \in \mathbb{Z}^{s \times k}$ so that row j of $E^{(b)}$ is $(\sigma_j^{(b)})^T$;row i of $D^{(b)}$ is $(d_i^{(b)})^T$; and row i of F is $\omega_\tau(\langle f_i \rangle)$. Then, it can be seen that $D^{(b)} = FE^{(b)}$ for $b = 0, 1$ as $\sigma' = \sum c_i \sigma_i$ in $\mathsf{Evaluate}$ in Section 3.3.

According to [5], when b=0, every signature $\sigma_j^{(0)}$ is sampled from a distribution statistically close to $\mathcal{D}_{\Lambda + t_i, v}$, of which t_i is the result of Step (2) of the Sign algorithm. By *Theorem 4.14* in [4], the derived signatures $D^{(0)} = FE^{(0)}$ are statistically close to a certain distribution, which can be defined by $\Lambda, \sigma, F, F\vec{m}_0^*$. So will be $D^{(1)}$ and the certain distribution that $D^{(1)}$ is closed to is the same as the one $D^{(0)}$ closed to, since $F\vec{m}_0^* = F\vec{m}_1^*$. Therefore, $D^{(0)}$ and $D^{(1)}$ are statistically close, which consequently concludes the adversary cannot win the challenge game.

Length Efficiency. To decide whether or not the scheme introduced above is length efficient, we focus the up limit of the norm of signatures output by $\mathsf{Sign}(\mathsf{sk}, ., ., .)$ and $\mathsf{Evaluate}$ to check what their bit length depend on.

According to *Lemma 4.4* in [25], the norm of signatures output by Sign with overwhelming probability will not exceed $v\sqrt{n}$, which implies its bit length is at most $n\lg(v\sqrt{n})$.

In the definition of Evaluate, σ is defined as $\Sigma_{i=1}^{k} c_i \sigma_i$. Since $|c_i| \leq p/2$ and $\sigma_i \leq v\sqrt{n}$ (σ_i output from Sign), the norm of σ is at most $k \cdot \max\{c_i\} \cdot \max\{\|\sigma_i\|\} = k \cdot \frac{p}{2} \cdot v\sqrt{n}$. Therefore, the bit length of the derived signature σ will not exceed $n\lg k + n\lg\left(\frac{p}{2} \cdot v\sqrt{n}\right)$, which depends logarithmically on k. Now the length efficiency of the scheme can be concluded.

4.2. A Polynomial Homomorphic Signature Scheme

In this section, a construction, a signature scheme that authenticates polynomial functions on signed messages without the original authenticator, is described as well as its *correctness* requirement.

Generally speaking, the concept of the polynomial system is replacing the lattices Λ_1, Λ_2 by *ideals* and assuming the lattice $\mathbb{Z}^n$ has a ring structure[5]. After the replacement, ring homomorphisms are built within mappings in the system. With ring homomorphisms, adding or multiplying signatures becomes a manner of adding or multiplying the corresponding messages and functions.

Concretely, let $F(x) \in \mathbb{Z}[x]$ be a monic irreducible polynomial of degree n, and the number field F can be defined by $F(x)$ as $\mathbb{Q}[x]/(F(x))$. Then the lattice in $\mathbb{Q}^n$ corresponding to the ring of integers of k is denoted by $\mathcal{O}_K$, which leads us to $\mathbb{F}_p$ through $\mathcal{O}_K/\mathfrak{p}$ as well as $\mathbb{F}_q$ ($\mathfrak{p}$ and $\mathfrak{q}$ are prime ideals that are subsets of $\mathcal{O}_K$ of norm p, q.) When $\mathbb{F}_p$ and $\mathbb{F}_q$ are obtained, sign messages can be signed exactly as in the linearly homomorphic scheme.

In the linearly homomorphic scheme, the projection functions π_i are used as "original" functions to be a generating set for acceptable functions. the function $f = \Sigma c_i \pi_i$ is encoded by its coefficients $(c_1, \ldots, c_k)$ as a vector. To compare with the linearly homomorphic scheme, the projection functions π_i in a polynomial homomorphic scheme are exactly the linear monomials x_i, and polynomial functions can be obtained by adding and multiplying monomials. With the same approach, polynomial functions on $\mathbb{F}_p[x_i, \ldots, x_k]$ can be encoded as its vector of coefficients after an ordering on all monomials is built[5].

The hash function $\omega_\tau(\langle f)$ needs to be re-defined as well. For a function f in $\mathbb{F}_p[x_1, \ldots, x_k]$ that is encoded as $\langle f \rangle = (c_i, \ldots, c_l)$, a "good" hash function $\omega_\tau(\langle f)$ is defined as $\hat{f}(\alpha_1, \ldots, \alpha_k)$, where $\hat{f}$ is a polynomial function which can reduce to f mod p[5].

To evaluate polynomials on signatures, given a polynomial f and signatures $\sigma_1, \ldots, \sigma_k \in K$ on messages $m_1, \ldots, m_k \in \mathbb{F}_p$, the polynomial $\hat{f} \in \mathbb{Z}[x_1, \ldots, x_k]$ is applied on the signatures, which is $\hat{f}(\sigma_1, \ldots, \sigma_k)$.

Now a formal description of the scheme from [5] is introduced, which theoretically can compute on signed messages for polynomials with small coefficients and bounded degree:

The polynomially homomorphic signature scheme.

```
Setup($1^n, k$).
    (n– a security parameter,
     k– a data set size):

    (1) Choose a monic and irreducible polynomial $F(x) \in \mathbb{Z}[x]$ of degree $n$ with $\gamma_F = poly(n)$.
        Let $K := \mathbb{Q}[x]/(F(x))$ and $R = \mathbb{Z}^n$ be the lattice that corresponds $\mathbb{Z}[x]/(F(x)) \subset \mathcal{O}_K$.
    (2) Run the PrincGen algorithm[30] with inputs $F, n$ to produce distinct principal degree-one prime ideals $\mathfrak{p} = (p, x-a)$ and $\mathfrak{q} = (q, x-b)$ of $R$ with generators $g_{\mathfrak{p}}, g_{\mathfrak{q}}$, respectively.
    (3) Generate a basis $\mathbf{T}$ of $\mathfrak{p} \cdot \mathfrak{q}$.
    (4) Set $v := \gamma_F^2 \cdot n^3 \log n$.
    (5) Choose positive integers: the coefficient bound $y = poly(n)$ and $d = O(1)$.
    (6) Compute $l = \begin{pmatrix} k+d \\ d \end{pmatrix} - 1$, which defines $\mathbb{Z}^l$ where reduced polynomial $\hat{f}$ get encoded.
    (7) Let $H : \{0,1\}^* \to (\mathbb{F}_q^l)$ be a hash function.
    (8) Output the public key pk:= $(F, p, q, a, b, v, y, d, H)$ and the ssecret key sk=$\mathbf{T}$.

Sign(sk,$\tau, m, i$).
    (sk– a secret key,
     $\tau$– a tag in $\{0,1\}^n$,
     $m$– a message in $\mathbb{F}_p^n$,
     $i$– an index):

    (1) Compute $\alpha_i := H(\tau \| i) \in \mathbb{F}_q^l$. $\omega_\tau(\langle \pi_i \rangle) = \alpha_i$.
    (2) Compute $h = h(x) \in R$ such that $h(a) \bmod p = m$ and $h(b) \bmod q = \alpha_i$.
    (3) Output $\sigma \leftarrow$ SamplePre($\mathfrak{p} \cdot \mathfrak{q}, \mathbf{T}, h, v$) $\in (\mathfrak{p} \cdot \mathfrak{q}) + h$.
```

```
Verify(pk,$\tau, m, \sigma, f$)
    (pk– a public key,
     $\tau$– a tag in $\{0,1\}^n$,
     $m$– a message in $\mathbb{F}_p^n$,
     $\sigma$– a signature in $\mathbb{Z}^n$,
     $f$– a function in $\mathcal{F}$)

     (1) If all of the following conditions hold, output 1 (accept); otherwise output 0 (reject):
         (a) $\|\sigma\| \leq l \cdot y \cdot \gamma_F^{d-1} \cdot (v\sqrt{n})^d$.
         (b) $\sigma(a) \bmod p = m$.
         (c) $\sigma(b) \bmod q = \omega_\tau(\langle f \rangle)$.
```

```
Evaluate(pk,$\tau, f, \vec{\sigma}$).
    (pk– a public key,
     $\tau$– a tag in $\{0,1\}^n$,
     $f$– a function in $\mathcal{F}$ encoded as $\langle f \rangle = (c_1, \dots, c_l) \in \mathbb{Z}^l$
         with $c_j \in (-y, y]$,
     $\vec{\sigma}$– a tuple of signatures $\sigma_1, \dots, \sigma_k$ in $\mathbb{Z}^n$)

     (1) Set $\hat{f} := \Sigma_{j=1}^{l} c_j Y_j(x_1, \dots, x_k)$. $Y_j$ is the set of all non-constant monomials $x_1^{e_1} \dots x_k^{e_k}$ ordered.
     (2) Output $\hat{f}(\sigma_1, \dots, \sigma_k)$.
```

Correctness of this Polynomially Homomorphic Signature Scheme. First, we check the three verification conditions for individual signatures relative to a projection function π_i defined by monomial x_i and encoded as $\langle \pi_i \rangle = \mathbf{e}_i$. Let $\tau \in \{0,1\}^n$, $m \in \mathbb{F}_p$, and $i \in \{1, \dots, k\}$. Then by algorithm Sign in the secheme, it can be seen that

$$\sigma \leftarrow \mathsf{Sign}(\mathrm{sk}, \tau, m, i).$$

(1) From the definition of algorithm PrincGen[30, Section 3.1] in Sign, the norm of the generators $g_\mathfrak{p}$ and $g_\mathfrak{q}$ is at most $n^{1.5}$, by which it can be obtained that

$$\|g_\mathfrak{p} g_q x^i\| \leq \gamma_F^2 \cdot n^3 \ (\text{for} i = 0, \dots, n-1).$$

It follows that the basis $\mathbf{T}$ has $\|\tilde{\mathbf{T}}\| \leq \gamma_F^2 \cdot n^3$, and therefore $v \geq \|\tilde{\mathbf{T}}\| \cdot \omega(\sqrt{\log n})$[3, Theoren 3.2]. Then it can be concluded that $\|\sigma\| \leq v\sqrt{n}$ exists with overwhelming probability by Lemma 4.4 in [25].

(2) By correctness of SamplePre defined in [15], $\sigma \in (\mathfrak{p} \cdot \mathfrak{q}) + h(x)$, where $\sigma(a) \bmod p = h(a) \bmod p$. By definition of $h(x)$ in algorithm Sign, it can be achieved that $h(x) \bmod p = m$.

(3) As above, we have $\sigma \in (\mathfrak{p} \cdot \mathfrak{q}) + h(x)$, and therefore $\sigma(b) \bmod q = h(b) \bmod q = \alpha_i$.

Then, we discuss the situation of the signature on the result of applying a function on messages. Here the function applied on messages can be any of acceptable functions in $\mathcal{F}$, and it is encoded as $\langle f \rangle = (c_1, \ldots, c_l) \in \mathbb{Z}^l$, where $|c_j| \leq y \in \mathbb{Z}$. Let $\tau \in \{0,1\}^n$, $\vec{m} = (m_1, \ldots, m_k) \in \mathbb{F}_p^k$, and $\vec{\sigma} = (\sigma_1, \ldots, \sigma_k)$ with $\sigma_i \leftarrow \mathsf{Sigh}(\mathrm{sk}, \tau, m_i, i)$. Then by algorithm $\mathsf{Evaluate}$ in the scheme, it can be gained that

$$\sigma' \leftarrow \mathsf{Evaluate}(\mathrm{pk}, \tau, f, \vec{\sigma}) = \Sigma_j c_j Y_j(\vec{\sigma}).$$

Next, we check the three verification conditions:

(1) Since f is acceptable $|c_j|$ is less than y for all j. Thus it can be gained that

$$\|\sigma'\| = \Sigma_{j=1}^{l} c_j Y_j(\vec{\sigma}) \leq y \cdot \Sigma_{j=1}^{l} Y_j(\vec{\sigma}).$$

Next we discuss $Y_j(\vec{\sigma})$ in details. Since $Y_j(\sigma)$ consists of multiplying at most d of the σ_i, it can be obtained

$$\|Y_j(\vec{\sigma})\| \leq \gamma_F^{d-1} \cdot (max\{\|\sigma_i\| : \sigma_i \in \vec{\sigma}\}).$$

By Lemma 4.4 in [25],

$$\|Y_j(\vec{\sigma})\| \leq \gamma_F^{d-1} \cdot (v\sqrt{n})^d.$$

Therefore,

$$\|\sigma'\| \leq l \cdot y \cdot \gamma_F^{d-1} \cdot (v\sqrt{n})^d.$$

(2) Since a ring homomorphism built in, each monomial Y_j can be split:

$$\Big(Y_j(\vec{\sigma})\Big)(a) = Y_j\Big(\sigma_1(a), \ldots, \sigma_k(a)\Big) \bmod p.$$

By correctness of individual signatures, it can be seen that $\sigma_i(a) \bmod p = m_i$ for $i \in \{i, \ldots, k\}$. Thus, we have

$$\begin{aligned}\sigma'(a) \bmod p &= \sum_j c_j \Big(Y_j(\vec{\sigma})\Big) \bmod p \\ &= \sum_j c_i Y_j\Big(\sigma_1(a), \ldots, \sigma_k(a)\Big) \bmod p \\ &= \sum_j c_j Y_j(m_1, \ldots, m_k) \\ &= f(\vec{m}).\end{aligned}$$

(3) By correctness of individual signatures, it can be gained that $\sigma_i(b) \bmod q = \alpha_i$ for $i \in \{1, \ldots, k\}$. As above, we have

$$\begin{aligned}\sigma'(b) \bmod q &= \sum_j c_j Y_j(\alpha_1, \ldots, \alpha_k) \bmod q \\ &= \omega_\tau(\langle f \rangle).\end{aligned}$$

CHAPTER 5

A Homomorphic Signature Scheme: Implementation

In this chapter, a fully discussion of the implementation of our homomorphic signature schemes is given. We construct four independent programs, corresponding Setup, Sign, Verify, and Evaluate, to carry out our linearly homomorphic signature scheme. The polynomial homomorphic signature scheme replaces the lattice generating module in Setup of the linearly one, and the rest part is analogous and can be built in almost the same route.

5.1. Number Theory Library

Firstly, an introduction of a library for doing number theory is given. Since most of computations in our schemes are related to the calculation of matrices and vectors, it would be very fussy and unwise to built those basic operations and environments line by line.

NTL is a high-performance, portable C++ library[29] that provides algorithms for arbitrary length integers and for vectors, matrices and polynomials over the integers and over finite fields as well as some useful data structures.

NTL's Programming Interface. Here some important APIs from NTL that are used in our code are listed with short descriptions of their function and role.

ZZ: The class *ZZ* is used to represent signed, arbitrary length integers. The large prime q and relatively small prime q are initialized in this class.

GenPrime(ZZ& p, long len, long err = 80): *GenPrime* generates a random prime p of length l with a probability bounded by 2^{-err} that p is composite.

ZZ_p: The class *ZZ_p* is used to represent integers mod a modulus. This class found the fields of our vectors and matrices.

random_ZZ_p(): *random_ZZ_p* generates a random integer in modulus *ZZ_p::modulus*. This function is used to help to generate the random q-ary lattice Λ_2.

mat_ZZ, mat_ZZ_p, mat_zz_p: These classes are used to represent matrices with different bounds. Most of the matrices in our schemes are initialized in these classes.

vec_ZZ, vec_ZZ_p: These classes are used to represent vectors with different bounds. Most of the vectors in our schemes are initialized in these classes.

kernel(mat_ZZ_p& X, const mat_ZZ_p& A): *kernel* computes a basis for the kernel of the map $x \to x * \mathbf{A}$, where x is a row vector. This function is used to help calculate $\Lambda^{\perp}(\mathbf{A})$.

HNF(mat_ZZ& W, const mat_ZZ& A, const ZZ& D): *HNF* computes the Hermite Normal Form of **A**. This function is implemented using the algorithm of [10], and we will discuss another high performance algorithm to compute HNF in later section.

mat_RR: *mat_RR* is used to represent matrices whose entries are in $\mathbb{R}$. This class is used to help sample discrete gaussians.

vec_RR: *vec_RR* is used to represent vectors whose entries are in $\mathbb{R}$. Some coefficients are stored in vectors in this class, when sampling discrete gaussians.

CRT(mat_ZZ& a, ZZ& prod, const mat_zz_p& A): This function implements Chinese Remainder Theory. This function is used for "dual role" signing.

5.2. Implementing Setup of Linearly Homomorphic Signature Scheme

Overview of Setup. According to the description, the main purpose of Setup is to generate the public key and secret key, and it has been implemented in *setup.cc*. It can be seen that Setup has been divided into five steps in the flow chart Figure 5.2.

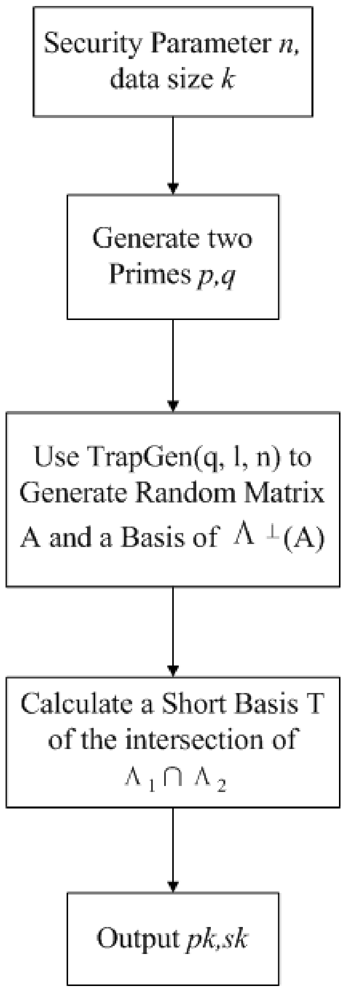

Security Parameter *n*, data size *k*
Generate two Primes *p*,*q*
Use TrapGen(q, l, n) to Generate Random Matrix A and a Basis of $\Lambda^{\perp}$(A)
Calculate a Short Basis T of the intersection of $\Lambda_1 \cap \Lambda_2$
Output *pk*,*sk*

FIGURE 5.2.1. the flow chart of the algorithm Setup

The Input of Setup. First, a security parameter n and a data set size k are put into Setup. Although there is no experiment to tell how large a security parameter could give us a reasonable computational security, it can still be assumed that n is in range of a *long*.

Generating Two Primes. Then, two primes p and q need to be generated. Since the minimal length requirement of *GenPrime* in NTL and the bound of q ($q \geq (nkp)^2$)required by security property, p and q are generated as following:

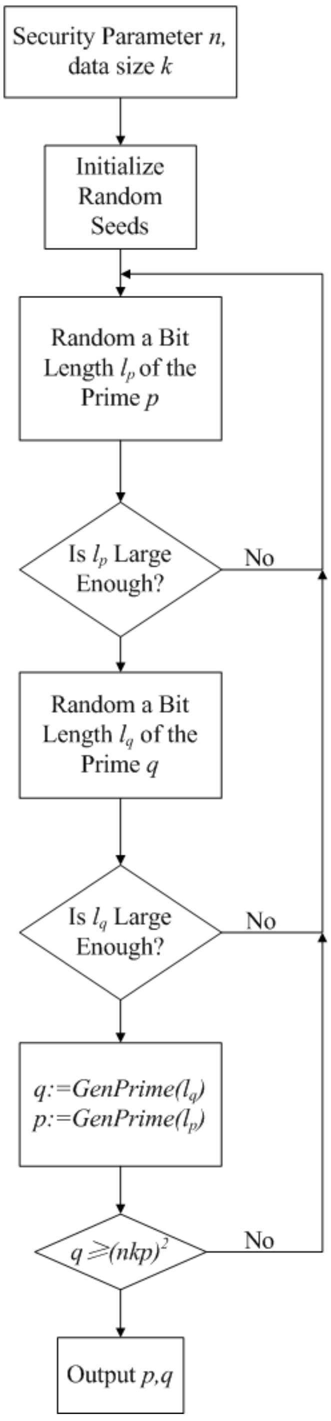


FIGURE 5.2.2. the flow chart of generating two primes

As shown in Figure 5.2.2, primes p and q are obtained by using *GenPrime* in NTL with their length picked up by C++ standard random function. It generates the primes repeatedly untill p and q are in the good range mentioned above.

Generating Two Lattices. After two primes generated, two lattices are about to be generated. The first lattice Λ_1 is defined as $p\mathbb{Z}^n$, which indicates a matrix with p in its diagonal. The other one is a lattice associated with a matrix $\mathbf{A}$, which TrapGen is used to construct following the flow chart in Figure 5.2.3.

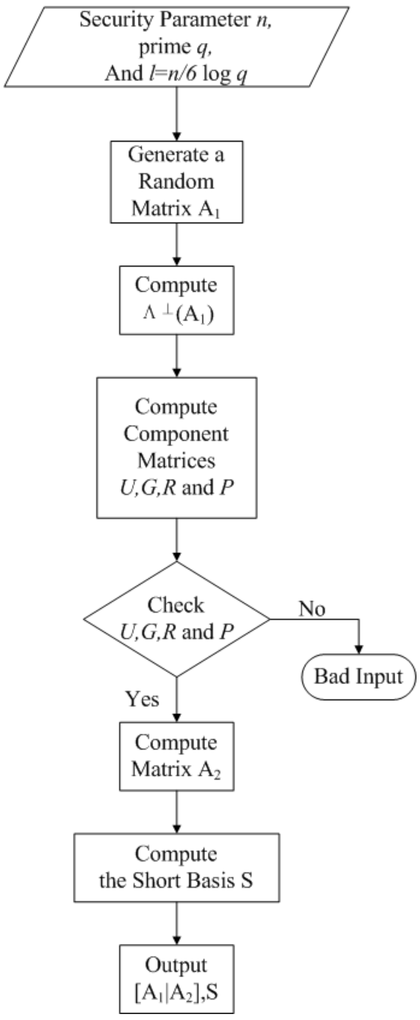

Security Parameter n, prime q, And $l=n/6 \log q$
Generate a Random Matrix A_1
Compute $\Lambda^{\perp}(A_1)$
Compute Component Matrices U,G,R and P
Check U,G,R and P
No
Bad Input
Yes
Compute Matrix A_2
Compute the Short Basis S
Output $[A_1|A_2]$,S

FIGURE 5.2.3. the flow chart of generating lattice Λ_2

TrapGen. Basic algorithm in TrapGen as described in [3]:

```
**Algorithm***  Framework for constructing $\mathbf{A} \in \mathbb{Z}_q^{l\times n}$
                and basis $\mathbf{S}$ of $\Lambda^{\perp}(A)$.
**Input:** $\mathbf{A}_1 \in \mathbb{Z}_q^{l\times n_1}$ and dimension $n_1$ (in unary).
**Output:**
    $\mathbf{A}_2 \in \mathbb{Z}_q^{l\times n_2}$;
    a basis $\mathbf{S}$ of $\Lambda^{\perp}(\mathbf{A})$, where $\mathbf{A} = [\mathbf{A}_1|\mathbf{A}_2] \in \mathbb{Z}_q^{l\times n}$
                                        for $m = m_1 + m_2$.

(1) Generate component matrices $\mathbf{U} \in \mathbb{Z}^{m_2\times m_2}$;
    $\mathbf{G},\mathbf{R} \in \mathbb{Z}^{m_1\times m_2}$; $\mathbf{P} \in \mathbb{Z}^{m_2\times m_1}$;
    and $\mathbf{C} \in \mathbb{Z}^{m_1\times m_1}$ such that $\mathbf{U}$ is nonsingular and
    $(\mathbf{GP}+\mathbf{C}) \subset \Lambda^{\perp}(\mathbf{A}_1)$
(2) Let $\mathbf{A}_2 = -\mathbf{A}_1 \cdot (\mathbf{R}+\mathbf{G}) \in \mathbb{Z}_q^{n\times m_2}$.
(3) Let $\mathbf{S} = \begin{pmatrix} (\mathbf{G}+\mathbf{R})\mathbf{U} & \mathbf{RP}\text{-}\mathbf{C} \\ \mathbf{U} & \mathbf{P} \end{pmatrix} \in \mathbb{Z}^{m\times m}$.
(4) **return** $\mathbf{A}_2$ and $\mathbf{S}$.
```

This algorithm extends the given input random matrix $\mathbf{A}_1 \in \mathbb{Z}_q^{n\times m_1}$ to $A = [\mathbf{A}_1|\mathbf{A}_2] \in \mathbb{Z}_q^{n\times m}$ by equation:

$$n\{[\underbrace{A_1}_{m_1} | \underbrace{A_2}_{m_2}] \begin{bmatrix} (G+R)\cdot U & R\cdot P - C \\ \underbrace{U}_{m_2} & \underbrace{P}_{m_1} \end{bmatrix} \begin{matrix} \}m_1 \\ \}m_2 \end{matrix} = 0 \in \mathbb{Z}_q^{n\times m}$$

*Construction of Algorithm**

In Algorithm*, the basic idea is that sub-matrix $\mathbf{G}$ contains the m_1 columns of $\mathbf{H}' = \mathbf{H} - \mathbf{I}$[1], and $\mathbf{P}$ simply selects those columns to yield $\mathbf{GP} = \mathbf{H}'$[3]. Moreover, appended columns in $\mathbf{G}$ are included to ensure both $\mathbf{U}$ and $\mathbf{GU}$ short, which gives the reason why there exists an extra factor ($l = \log_r q$) in the restrictions.

Then definitions of parts of outputs of Algorithm* can be given[3]:

Definition of G

$$\mathbf{G} = \left[\mathbf{G}^{(1)}|\ldots|\mathbf{G}^{(m_1)}|\mathbf{M}|\mathbf{0}\right] \in \mathbb{Z}^{m_1\times m_2}$$

(1) For $i \in [m_1]$, width of $\mathbf{G}^{(i)}$: $w_i = \lceil \lg h_{i,i} \rceil$

(2) For $j \in w_i$, the j-th column of $\mathbf{G}^{(i)}$: $g_j^{(i)} = 2^{j-1} \cdot \mathbf{e}_i \in \mathbb{Z}^{m_1}$

(3) The number of total columns of all $\mathbf{G}^{(i)}$ =

$$\sum_{i\in[m_1]} w_i \leq n \lg q + \sum_{i\in[m_1]} \lg h_{i,i} \leq 2n \lg q$$

[1]Given $\mathbf{A}_1$, let $\mathbf{H} \in \mathbb{Z}^{m_1\times m_1}$ be the Hermite normal form of $\Lambda^{\perp}(\mathbf{A}_1)$

(4) If q is prime, there are at most n values of $h_{i,i} > 1$(all are q), which results in that the number of columns is at most $n\lceil \lg q \rceil$ (because $\lceil \lg 1 \rceil = 0$)

(5) The matrix $\mathbf{M}$ is all zero but first d rows, and is a some kind of complicated matrix that is only needed for analysis of $\|\tilde{\mathbf{S}}\|$

Definition of P

$$\mathbf{P} = \left[\mathbf{P}^{(1)}; \ldots; \mathbf{P}^{(m_1)}; \mathbf{0}; \mathbf{0}\right] \in \mathbb{Z}^{m_2 \times m_1}$$

, which mirroring the structure of $\mathbf{G}$.

(1) For each $i, j \in [m_1]$, $\mathbf{P}^{(i)}$ contains entries $\mathbf{p}_{k,j}^{(i)} \in \{0, 1\}$ such that

$$h'_{i.j} = \sum_{k \in [w_i]} \mathbf{p}_{k,j}^{(i)} \cdot 2^{k-1},$$

which actually turns $h'_{i,j}$ to binary and fill $\mathbf{p}_j^{(i)}$ with them

(2) $\mathbf{GP} = \mathbf{H}'$

Definition of U

$$\mathbf{U} = \mathrm{diag}(\mathbf{T}_{w_1}, \ldots, \mathbf{T}_{w_{m_1}}, \mathbf{I}) \in \mathbb{Z}^{m_2 \times m_2},$$

in which

$$\mathbf{T}_w = \begin{pmatrix} 1 & -2 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & -2 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & -2 \\ 0 & \cdots & \cdots & \cdots & 0 & 1 \end{pmatrix} \in \mathbb{Z}^{w \times w}$$

Definition of R

(1) Each entry in the top d rows of $\mathbf{R}$ is an independent $\{0, \pm 1\}$-valued random variable that is 0 with probability $\frac{1}{2}$, 1 with probability $\frac{1}{4}$, -1 with probability $\frac{1}{4}$

(2) The remaining entries are all 0

Fix the Input and Output of TrapGen*.* In the linearly homomorphic signature scheme, the input of TrapGen is (q, l, n), where q is a n-digit prime that satisfied $q \geq (nkq)^2$, and l is a fixed number that equals $\lfloor n/6 \log q \rfloor$. After given an appropriate input, TrapGen produces a output of a matrix $\mathbf{A} \in \mathbb{F}_q^{l \times n}$ along with a short basis $\mathbf{T}_q$ of $\Lambda_q^{\perp}(\mathbf{A})$.

However, the input of TrapGen does not match what Algorithm* wants. To make it work, a basic idea is that a matrix $\mathbf{A_1}_q^{l \times n_1}$ and $n2 = n - n1$ are put into Algorithm*. Then the output of Algorithm* is a matrix $\mathbf{A}_2 \in \mathbb{Z}^{l \times n_2}$ and a basis $\mathbf{S}$ of $\Lambda^{\perp}(\mathbf{A})$. By combining input matrix $\mathbf{A}_1$ and output matrix $\mathbf{A}_2$, it can be obtained that $\mathbf{A} = [\mathbf{A}_1 | \mathbf{A}_2] \in \mathbb{Z}_q^{l \times n}$for $n = n_1 + n_2$, which is exactly what TrapGen should output. Secondly the other desired output in linearly homomorphic signature scheme is a short basis. According to *Figure 1* in [3], it can be confirmed that $\mathbf{AS} = \mathbf{0} \in \mathbb{Z}_q^{l \times n}$, which means a basis of $\Lambda_q^{\perp}(\mathbf{A})$.

Algorithm* perfectly provides the expecting outputs, but with restrictions. Based on *Theorem 3.2* in [5] and *Lemma 3.5* in [3], Let $\delta = 1/3$, and $l = \lfloor n/6 \log q \rfloor$, n_1 should satisfy that

$$(1+\delta)l \log q \leq n_1 \leq n - (4+2\delta)l \log q,$$

which limits the random matrix $\mathbf{A}_1$. Thus, the random matrix $\mathbf{A}_1$ is created by following steps:

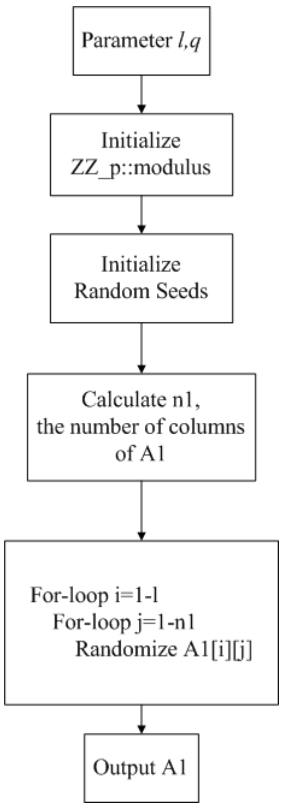


FIGURE 5.2.4. the flow chart of generating matrix $\mathbf{A}_1$

Computing Hermite Normal Form. One difficult procedure of TrapGen is that how *Hermite Normal Form* is computed.

It has been introduced that there is an API in NTL which computes the Hermite Normal Form of a matrix. However, the alogirthm used in NTL for HNF is very old and inefficient. Now we try to optimize the procedure by updating the algorithm. Foremost, a formally definition of *Hermite Normal Form* should be introduced:

Definition 5.2.1. *(Hermite normal form)*[27]. *For any $n \times m$ integer matrix A the Hermite normal form (HNF) of A is the unique matrix $H = (h_{i,j})$ such that there is a unimodular $n \times n$ matrix U with $UA = H$, and such that H satisfies the following two conditions:*

- *there exist a sequence of integers $j_1 < \cdots < j_n$ such that for all $0 \leq i \leq n$ $h_{i,j} = 0$ exists for all $j < j_i$ (row echelon structure),*
- *for $0 \leq k < i \leq n$ it can be gained $0 \leq h_{k,j_i} < h_{i,j_i}$(the pivot element is the greatest along its column and the coefficients above are nonnegative).*

For instance, an example can be

$$\begin{pmatrix} 5 & 3 & 1 & 4 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 19 & 16 \\ 0 & 0 & 0 & 3 \end{pmatrix}.$$

To compute Hermite normal form is quite like doing Gaussian Elimination without doing divisions, which causes it not the same as Gaussian Elimination. So it has to done by doing the row elimination which uses the extended Euclidean Algorithm.

There are numbers of algorithms for the computing HNF's, including [**28**, **11**, **6**, **26**]. Here, an algorithm by Clément Pernet and William Stein[**27**] is introduced and can be implemented, which is based on heuristically fast algorithm with several practical improvements.

Algorithm†: Hermite Normal Form[26]

Data Input: A: an $n \times n$ nonsingular matrix over $\mathbb{Z}$
Data Output: H: the Hermite normal form of A

```
1 begin
2 Write $A = \begin{bmatrix} B & b \\ c^T & a_{n-1,n} \\ d^T & a_{n,n} \end{bmatrix}$
3 Compute $d_1 = \det\left(\begin{bmatrix} B \\ c^T \end{bmatrix}\right)$
4 Compute $d_2 = \det\left(\begin{bmatrix} B \\ d^T \end{bmatrix}\right)$
5 Compute the extended gcd of $d_1$ and $d_2$: $g = sd_1 + td_2$
6 Let $C = \begin{bmatrix} B \\ sc^T + td^T \end{bmatrix}$
7 Compute $H_1$, the Hermite normal form of $C$, by working modulo $g$
8 Obtain from $H_1$ the Hermite form $H_2$ of $\begin{bmatrix} B & b \\ sc^T + td^T & sa_{n-1,n} + ta_{n,n} \end{bmatrix}$
9 Obtain from $H_2$ the Hermite form $H_3$ of $\begin{bmatrix} B & b \\ c^T & a_{n-1,n} \end{bmatrix}$
10 Obtain from $H1$ the Hermite form $H$ of $\begin{bmatrix} B & b \\ c^T & a_{n-1,n} \\ d^T & a_{n,n} \end{bmatrix}$
11 end
```

Algorithm† is an algorithm for computing HNF's on square matrices. The key ideas of Algorithm† go as following[27]:

(1) Every entry in the HNF **H** of a suqare matrix **A** is always less then the absolute value of the determinant det(**A**), so one can compute **H** be working modulo the determinant of **H**. This idea was first introduced and developed in [11];
(2) Computing **H** from **H′** which is the HNF computed of a small-determinant matrix constructed from **A**.

To improve Algorithm† for rectangular matrices, the algorithm need to be decomposed into steps and develop them[27]:

- The first bottleneck is in Steps 3 and 4 – Compute det. There are many algorithms for computing the determinant of an integer matrix A.

  The most common one involves computing the Hadamard bound on $\det(A)$, then computing the determinant modulo p for sufficiently many p using a Gaussian elimination algorithm, and finally using a Chinese remainder theorem reconstruction, which has bit complexity[13]

$$\mathcal{O}(n^4(\log n + \log \|A\|) + n^3 \log^2 \|A\|).$$

According to Abbott, Bronstein and Mulders[1], the average case complexity of the algorithm has been improved to $\mathcal{O}(n^3(\log^2 n + log\|A\|)^2)$. The two steps (Step 3 and 4) have been reduced into one solution – Doulbe determinant computation – because matrices in these steps are similar.

```
Algorithm‡: Double determinant computation
Data Input: $B$: an $(n-1) \times n$ matrix over $\mathbb{Z}$
Data Input: $c, d$: two vectors in $\mathbb{Z}^n$
Result: $(d1, d2) = (\det(\begin{bmatrix} B^T & c \end{bmatrix}), \det(\begin{bmatrix} B^T & d \end{bmatrix}))$
begin
1 Solve the system $\begin{bmatrix} B^T & c \end{bmatrix} x = d$ using Dixon's $p$-adic lifting[9]
2 $y_i = -\frac{x_i}{x_n}, y_n = \frac{1}{x_n}$ solves $\begin{bmatrix} B^T & d \end{bmatrix}]y = c$
3 $u_1 = \text{lcm}(\text{denominators}(x)$
4 $u_2 = \text{lcm}(\text{denominators}(y)$
5 Compute Hadamard's bounds $h_1$ and $h_2$
    on the determinants of $\begin{bmatrix} B^T & c \end{bmatrix}$ and $\begin{bmatrix} B^T & d \end{bmatrix}$
6 Select a set of primes $(p_i)$ s.t. $\Pi_i p_i > \max(\frac{h_1}{u_1}, \frac{h_2}{u_2})$
7 for each $p_i$ do
    compute $B^T = LUP$, the LUP decomposition of $B^T$ mod $p_i$
    $q = \Pi_{i=1}^{n-1} U_{i,i}$ mod $p_i$
    $x = L^{-1} cmodp_i$
    $y = L^{-1} dmodp_i$
    $v_1^{(i)} = qx_n modp_i$
    $v_2^{(i)} = qy_n modp_i$
8 reconstruct $v_1$ and $v_2$ using CRT
9 return $(/d_1, d_2) = (u_1 v_1, u_2 v_2)$
end
```

- The second problem is compute the HNF of C in Algorithm†. According to Pernet and Stein[27], this step has been improved by applying the standard row reduction Hermite normal form algorithm to C, always recuding all numbers modulo g, when the HNF of $\begin{bmatrix} C \\ gI \end{bmatrix}$ is $\begin{bmatrix} H \\ 0 \end{bmatrix}$ where H is the Hermite normal form of C.
- Then Step 8 of Algorithm† is to find a column vector e such that
$$\begin{bmatrix} H_1 & e \end{bmatrix} = U \begin{bmatrix} B & b \\ sc^T + td^T & a_{n-1,n} \end{bmatrix}$$
is in Hermite form, for a unimodular matrix U. In Micciancio and Warinschi's paper[26], the column e is computed using multi-modular computations and a tight bound on the size of the entries of e. Now this procedure has been improved by

using the p-adic lifting algorithm[2]. Then a column can be added as following:

```
Algorithm‡‡[27]: AddColumn
Data Input: $B = \begin{bmatrix} B_1 & b_2 \\ b_3^T & b_4 \end{bmatrix}$: an $n \times n$ matrix over $\mathbb{Z}$
Data Input: $\mathbf{H}_1$: the Hermite normal form of $\begin{bmatrix} B_1 \\ b_3^T \end{bmatrix}$
Data Output: $\mathbf{H}$: the Hermite normal form of $B$
begin
    Pick a random vector $u$ such that $|u_i| \leq \|B\|, \forall i$
    Solve $\begin{bmatrix} B_1 \\ u \end{bmatrix} y = \begin{bmatrix} b_2 \\ b_4 \end{bmatrix}$
    Compute a kernel basis vector $k$ of $B_1$
    $\alpha = \frac{b_4 - b_3^T \cdot y}{b_3^T \cdot k}$
    $x = y + \alpha k$
    $e = \mathbf{H}_1 x$
    return $[\mathbf{H}_1 e]$
end
```

Instead of multi-modular computations, Algorithm‡‡ tries to solve the problem by solving the system

$$\begin{bmatrix} B \\ sc^T + td^T \end{bmatrix} x = \begin{bmatrix} b \\ a_{n-1,n-1} \end{bmatrix}$$

by using p-adic lifting algorithm[9]. However, this way is not a shortcut as the row $sc^T + td^T$ that has much larger coefficients worsen the complexity of finding a solution. Therefore, the algorithm has been enhanced by an idea of metonymy and recovery. Algorithm‡‡ changes $sc^T + td^T$ to a random row u that has small entries, and solve the system

$$\begin{bmatrix} B \\ u \end{bmatrix} y = \begin{bmatrix} b \\ a_{n-1,n-1} \end{bmatrix}.$$

Then, Algorithm‡‡ tries to recover x with a kernel basis vector k of B in

$$x = y + \alpha k,$$

where α satisfies

$$\alpha = \frac{a_{n-1,n-1} - (sc^T + td^T) \cdot y}{(sc^T + td^T) \cdot k}.$$

- Finally, Steps 9 and 10 of Algorithm† consist of adding a new row to the current Hermite form and updating it to obtain a new matrix in Hermite form. The principle is to eliminate the new row with all existing pivots (Extended gcd based elimination) and update the already computed parts when necessary.

[2]The p-adic lifting algorithm solves mo$p$ and mo$p^2, \ldots,$ until mop^n problem

```
Algorithm‡‡‡[27]: AddRow
Data Input: $\mathbf{A}$:
Data Input: $b$:
Data Output: $\mathbf{H}$: the Hermite normal form of $\left[\begin{smallmatrix}\mathbf{A}\\ b\end{smallmatrix}\right]$
begin
    for all pivots $a_{i,j_i}$ of $\mathbf{A}$ do
        if $b_{j_i} = 0$ then continue
        if $\mathbf{A}_{i,j_i} | b_{j_i}$ then $b := b - b_{j_i}/\mathbf{A}_{i,j_i}\mathbf{A}_{i,1\ldots n}$
        else
            /* Extended gcd based dlimination */
            $(g, s, t) = XGCD(a_{i,j_i}, b_{j_i})$
            $\mathbf{A}_{i,1\ldots n} := s\mathbf{A}_{i,1\ldots n} + tb_{j_i}$
            $b := b_{j_i}/g\mathbf{A}_{i,1\ldots n} - \mathbf{A}_{i,j_i}/gb$
            for $k$=1 to $i-1$ do
                /* Reduce row $k$ with row $i$ */
                $\mathbf{A}_{k,1\ldots n} := \mathbf{A}_{k,1\ldots n} - \lfloor \mathbf{A}_{k,j_i}/\mathbf{A}_{i,j_i} \rfloor \mathbf{A}_{i,1\ldots n}$
    if $b \neq 0$ then
        let $j$ be the index of the first nonzero element of $b$
        insert $b^T$ between rows $i$ and $i+1$ such that $j_i < j < j_{i+1}$
    Return $\mathbf{H} = \left[\begin{smallmatrix}\mathbf{A}\\ b\end{smallmatrix}\right]$
end
```

Before proceeding, one thing has to be noted is that Algorithm†[26] is designed to apply to square matrices. In the case where the matrix A is rectangular, we have to try to calculate submatrix of A and use Algorithm‡‡ and Algorithm‡‡‡ to add additional columns and rows for multiple times until correct.

Compute the Short Basis of the Intersection of Λ_1 and Λ_2. After a deep discussion about generating two lattices, we focus back on remaining parts of Setup.

It can be seen that the short basis $\mathbf{T}$ of $\Lambda_1 \cap \Lambda_2$ is calculated after two lattices constructed. To compute this short basis, two short bases of lattices Λ_1 and Λ_2 are needed.

On one hand, the problem of a short basis $\mathbf{T}_p$ of Λ_1 is relatively simple. Since Λ_1 is defined by $p\mathbb{Z}^n$, the short basis of the coset of Λ_1 and any other lattice Λ_* can be computed by multiplying p to a short basis $\mathbf{T}_*$ of Λ_*.

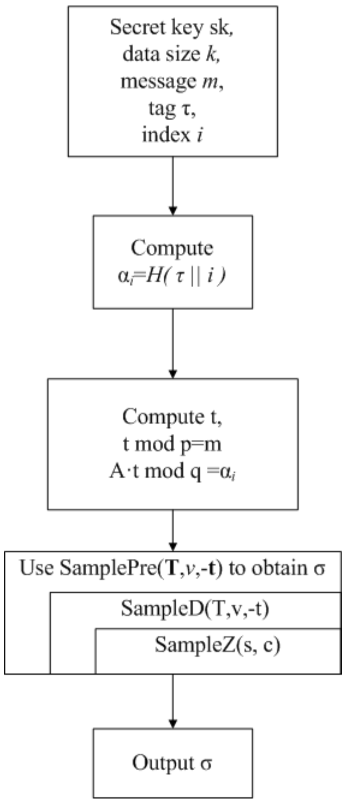
On the other hand, by using TrapGen a short basis of the lattice associated with a matrix $\Lambda^\perp(\mathbf{A})$ has been already created. a short basis $\mathbf{T}_q$ of Λ_2 can be obtained directly from the output of TrapGen.

Then the short basis output by TrapGen is multiplied by p to generated a new short basis $\mathbf{T}$ which can represent the intersection.

Arrange the Output of Setup. Finally, it is about to arrange what has been computed into two "packages" called *public key* and *secret key*. The public key consists of Λ_1, which can be replaced by p, Λ_2, which should be computed by matrix $\mathbf{A}$ by algorithm introduced in section 3.1.2, the data set size k and a parameter $v = p \cdot \sqrt{n \log q}$, which is used for sampling discrete gaussians in Sign. The secret key only consists the short basis $\mathbf{T}$ of the intersection of two lattices.

5.3. Implementing Sign of Linearly Homomorphic Signature Scheme

Overview of Sign. According to the description, the main purpose of Sign is to generate a signature that is a vector in $\mathbb{Z}^n$, and it has been implemented in *sign.cc*. It can be seen that Setup has been divided into five steps in the flow chart Figure 5.3.1.



Secret key sk, data size k, message m, tag τ, index i
Compute $\alpha_i=H(\tau \,||\, i)$
Compute t, t mod p=m A·t mod q =α_i
Use SamplePre($\mathbf{T}$,v,-$\mathbf{t}$) to obtain σ
SampleD(T,v,-t)
SampleZ(s, c)
Output σ

FIGURE 5.3.1. the flow chart of the algorithm Sign

The Input of Sign. In order to start a process of signing, the program needs a bunch of valid input. The input includes:

Secret Key sk: A short basis $\mathbf{T}$ of $\Lambda_1 \cap \Lambda_2$, by which sigantures can be sampled.

Message m: A vector in $\mathbb{F}_p^n$, which could be in $\mathbb{Z}^n$. It needs to be mapped into our message space.

Tag τ: An n-bit string of $\{0,1\}$, which is initialized as a *ZZ* during the implementation.

index i: An integer, which locates m in the message set tagged by τ.

Matrix A: A matrix initialized as a *mat_ZZ*, which defines Λ_2.

Hashing the Projection Function π_i. The hash function used to hash functions that will be applied on messages has been modeled to a random oracle in [5].

In our signing process, the hash function works as following algorithm:

```
Algorithm_{Hash}:
Data Input: $x \in \{0,1\}^*$, a prime $q$, a fixed integer $l$
Data Output: a vector in $\mathbb{F}_q^l$
$\mathbf{y} \leftarrow \mathbf{0} \in \mathbb{F}_q^l$
For $i = 0 \ldots l-1$ do
    Let $s = x \| i$
    $t \leftarrow$ SHA-512$(s) \in \{0, \ldots, 2^{512}-1\}$
    $\mathbf{y}_i \leftarrow t \bmod q$
Output $\mathbf{y}$
```

Considering two main security properties of this hash function:

One-way property: If there is an oracle that can break the one-way property of the algorithm above, this oracle can be used to break the one-way property of SHA-512: The challenger sends a hash $y \in \mathbb{F}_q$ to the adversary to generate its corresponding message. To response, the adversary set $l = 1$ and send $\mathbf{y}' \leftarrow y \in \mathbb{F}_q^1$ to the oracle and get a message x'. Then $x' \| 0$ is the message to hash y. If the oracle cannot output a message, then the adversary set $l = 2$ and the second output x'' of the oracle, which is valid, can construct the message to hash y, which is $x'' \| 1$.

Collision resistant: Since the security given by SHA-512, it can be concluded that the collision resistant property of hash algorithm above is fine.

With the hash function described above, the hash of the projection function π_i can be computed easily – applying Algorithm$_{Hash}(\tau \| i)$.

Computing the "Dual-Role" Signing Vector. It can be seen that the third step in the flow chart of Sign is to compute a vector $\mathbf{t} \in \mathbb{Z}^n$ such that $t \bmod p = m$ and $\mathbf{A} \cdot \mathbf{t} \bmod q = \alpha_i$.

First, a vector m' in $\mathbb{F}_p^l$ that is defined as $\mathbf{A} \cdot m$ is computed. Then a new definition of $\mathbf{t}$ can be obtained such that $\mathbf{A} \cdot \mathbf{t} \bmod p = m'$ and $\mathbf{A} \cdot \mathbf{t} \bmod q = \alpha_i$. With this definition, a reconstruction problem of $\mathbf{A} \cdot \mathbf{t}$ is formed, and it can be solved by Chinese Remainder Theorem directly. After CRT applied, a vector $\mathbf{t}'$ in $\mathbb{Z}^n$ is generated. Finally, Gaussian-Elimination algorithm is used to solve $\mathbf{A} \cdot \mathbf{t} = \mathbf{t}'$ to get $\mathbf{t}$.

For example, let $q = 7$, $p = 3$, the matrix $\mathbf{A} = \begin{bmatrix} 2 & 0 & 1 & 0 & 1 \\ 4 & 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 \end{bmatrix}$, a message $m = \begin{bmatrix} 1 \\ 0 \\ 2 \\ 0 \\ 0 \end{bmatrix}$, and a hash $\alpha_i = \begin{bmatrix} 0 \\ 3 \\ 5 \end{bmatrix}$. Then we proceed:

(1) $m' = \begin{bmatrix} 2 & 0 & 1 & 0 & 1 \\ 4 & 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 2 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$.

(2) $\mathbf{A} \cdot \mathbf{t} = \mathsf{CRT}(p, q, m', \alpha_i)$

 (a) $\begin{bmatrix} 0 \\ 3 \\ 5 \end{bmatrix} + \begin{bmatrix} 7 \\ 7 \\ 7 \end{bmatrix} u = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} \bmod p$.

 (b) $u = \begin{bmatrix} 1 & 1 & 2 \end{bmatrix}$.

 (c) $\mathbf{A} \cdot \mathbf{t} = \begin{bmatrix} 7 \\ 10 \\ 5 \end{bmatrix}$

(3) Solve $\begin{bmatrix} 2 & 0 & 1 & 0 & 1 \\ 4 & 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 \end{bmatrix} \mathbf{t} = \begin{bmatrix} 7 \\ 10 \\ 5 \end{bmatrix}$

 (a) $\mathbf{t} \leftarrow \{t_1, t_2, t_3, t_4, t_5\}$.

 (b) Let $t_4 = 0$ and $t_5 = 0$, since matrix $\mathbf{A}$ is an 3×5 matrix.

 (c) Get a solution:

$$t_1 = 1 + t_4 + \frac{1}{2} t_5 = 1,$$

$$t_2 = 3 - 2t_4 + t_5 = 3,$$

$$t_3 = 5 - 2t_4 = 5.$$

(d) $\mathbf{t} = \begin{bmatrix} 1 \\ 3 \\ 5 \\ 0 \\ 0 \end{bmatrix}$

(4) Output $\mathbf{t}$.

Sampling Discrete Gaussians. Generally, a message is hashed to a point in some region of space, and its signature is essentially a nearby lattice point, which is found using a "good" secret basis. However, in *Trapdoor Functions*[15], given by probabilitistic polynomial-time algorithms(SampleZ, SampleD, SamplePre), there are two main differences: first, they are based on random lattices that enjoy worst-case hardness; second and more importantly, the signatures are generated by a randomized decoding algorithm whose output distribution is *oblivious* to the geometry of the secret basis.

Gaussians on Lattices[15]

For any $s > 0$ define the Gaussian function on $\mathbb{R}^n$ centered at $\mathbf{c}$ with parameter s:

$$\forall x \in \mathbb{R}^n, \rho_{s,\mathbf{c}}(\mathbf{x}) = \exp\left(-\pi \|\mathbf{x} - \mathbf{c}\|^2 / s^2\right)$$

.

For any $\mathbf{c} \in \mathbb{R}^n$, real $s > 0$, and n-dimensional lattice Λ, define the *discrete Gaussian distribution over* Λ as:

$$\forall \mathbf{x} \in \Lambda, D_{\Lambda,s,\mathbf{c}}(\mathbf{x}) = \frac{\rho_{s,c}(\mathbf{x})}{\rho_{s,c}(\Lambda)}$$

The Gram-Schmidt Norm of a Basis. Before discussing how to sample discrete gaussians, the Gram-Schimdt norm of a basis need to be introduced as well as its orthogonalization and how to compute them. In any inner product space, one can choose the basis in which to work. It often greatly simplifies calculations to work in an orthogonal basis.

Let $\mathbf{S}$ be a set of vectors $\mathbf{S} = \{\mathbf{s}_1, \ldots, \mathbf{s}_k\}$ in $\mathbb{R}^m$. The following notations are used:

- $\|\mathbf{S}\|$ denotes the length of the longest vector in $\mathbf{S}$, i.e., $\max_{1 \le i \le k} \|\mathbf{s}_i\|$.
- $\tilde{\mathbf{S}} := \{\tilde{\mathbf{s}}_1, \ldots, \tilde{\mathbf{s}})k\} \subset \mathbb{R}^m$ denotes the Gram-Schmidt orthogonalization of the vectors $\mathbf{s}_1, \ldots, \mathbf{s}_k$.
- $\|\tilde{\mathbf{S}}\|$ denotes the Gram-Schmidt norm of $\mathbf{S}$.

To compute the Gram-Schmidt orthogonalization, a simple algorithm is applied in our program:

Algorithm$_{Gram-Schmidt}$

Input:

——a basis $\mathbf{B}$ of an n-dimensional lattice $\Lambda = \mathcal{L}(\mathbf{B})$

Output:

——a matrix which stores the Gram-Schmidt orthogonalization $\tilde{\mathbf{B}}$

```
(1) Expand $\mathbf{B}$ to $\mathbb{R}^{n\times n}$.

(2) For $i = 0, \ldots, n-1$ do:
    | $\tilde{\mathbf{b}}_i \leftarrow \mathbf{b}_i$.
    | For $j = 0, \ldots, i-1$ do:
    |   | Compute inner product $c_0$ of $\mathbf{b}_i$ and $\tilde{\mathbf{b}}_j$.
    |   | Compute inner product $c_1$ of $\tilde{\mathbf{b}}_j$ and $\tilde{\mathbf{b}}_j$.
    |   | Compute $\mathrm{m}_{i,j} = c_0/c_1$.
    |   | $\tilde{\mathbf{b}}_i = \tilde{\mathbf{b}}_i - \mathrm{m}_{i,j} \cdot \tilde{\mathbf{b}}_j$
(3) Output $\tilde{\mathbf{B}}$
```

Geometrically, $\mathrm{m}_{i,j} \cdot \tilde{\mathbf{b}}_j$ is the orthogonal projection of $\mathbf{b}_i$ on the space spanned by $\tilde{\mathbf{b}}_j$, and $\tilde{\mathbf{b}}_i$ is constructed by removing all orthogonal projections of $\mathbf{b}_i$ on the space spanned by orthogonal vectors which have been already constructed.

A Sampling Algorithm. Since *continuous* can not be generally achieved in computer, if it needs to sample some from Gaussians, it can only be sampled from discrete Gaussians as shown in Figure 5.3.2.

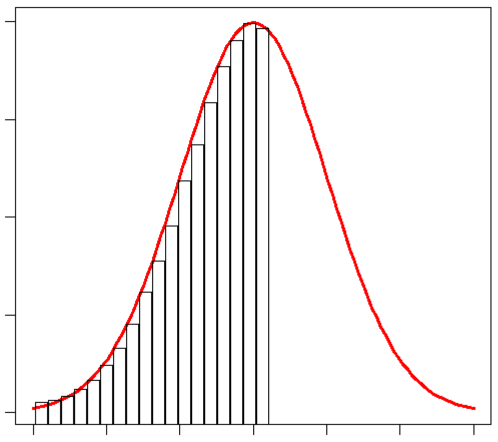


FIGURE 5.3.2. An abstract discrete Gaussians

Theorem 5.3.1. (Theorem 4.1 in [15]) *There is a probabilistic polynomial-time algorithm that, given a basis* $\boldsymbol{B}$ *of an* n*-dimensional lattice* $\Lambda =$

$\mathcal{L}(\mathbf{B})$, a parameter $s \geq \|\tilde{\mathbf{B}}\| \cdot \omega(\sqrt{\log n})$, and a center $\mathbf{c} \in \mathbb{R}^n$, outputs a sample from a distribution that is statistically close to $D_{\Lambda,s,\mathbf{c}}$.

```
SampleD
Input:
——a basis $\mathbf{B}$ of an $n$-dimensional lattice $\Lambda = \mathcal{L}(\mathbf{B})$
——a parameter $s \geq \|\tilde{\mathbf{B}}\| \cdot \omega(\sqrt{\log n})$
——a center $\mathbf{c} \in \mathbb{R}^n$
Output: a lattice vector
Step 1: Let $\mathbf{v}_n \leftarrow \mathbf{0}$ and $\mathbf{c}_n \leftarrow \mathbf{c}$. For $i \leftarrow n, \ldots, 1$, do:
  Let $c'_i = \langle \mathbf{c}, \tilde{\mathbf{b}}_i \rangle / \langle \tilde{\mathbf{b}}_i, \tilde{\mathbf{b}}_i \rangle \in \mathbb{R}$ and $s'_i = s/\|\tilde{\mathbf{b}}_i\|$
  Choose $z_i \sim D_{\mathbb{Z},s'_i,c'_i}$
    (1)Set a fixed function $t(n) \geq \omega(\sqrt{\log n})$
    (2)Set $s^* = s'_i, c^* = c'_i$
    (3)Choose an integer $z_i \leftarrow \mathbb{Z} \cap [c^* - s^* \cdot t(n), c^* + s^* \cdot t(n)]$
          (this has been coded in a procedure SampleZ as shown in the flow chart)
    (4)Get probability $\rho_s(x - c) \in (0, 1]$
    (5)According to $\rho_s(x - c) \in (0, 1]$, output $z_i$ or turn (3)
  Let $\mathbf{c}_{i-1} \leftarrow \mathbf{c}_i - z_i \mathbf{b}_i$ and let $\mathbf{v}_{i-1} \leftarrow \mathbf{v}_i + z_i \mathbf{b}_i$.
Step 2: Output $\mathbf{v}_0$
```

Pre-image Sampling with Trapdoor (SamplePre). According to [5], Algorithm SamplePre($\mathbf{A}$, $\mathbf{T}$, $\mathbf{t}$, v) simply calls SampleD($\mathbf{T}, v, -\mathbf{t}$) and adds $\mathbf{t}$ to the result. Thus SamplePre($\mathbf{A}$, $\mathbf{T}$, $\mathbf{t}$, v) is used to generate a valid signature σ.

5.4. Implementing Verify of Linearly Homomorphic Signature Scheme

Overview of Verify. According to the description, the main purpose of Sign is to check three verification conditions, and it has been implemented in *verify.cc*. It can be seen that Verify has been divided into three steps in the flow chart Figure 5.4.1.

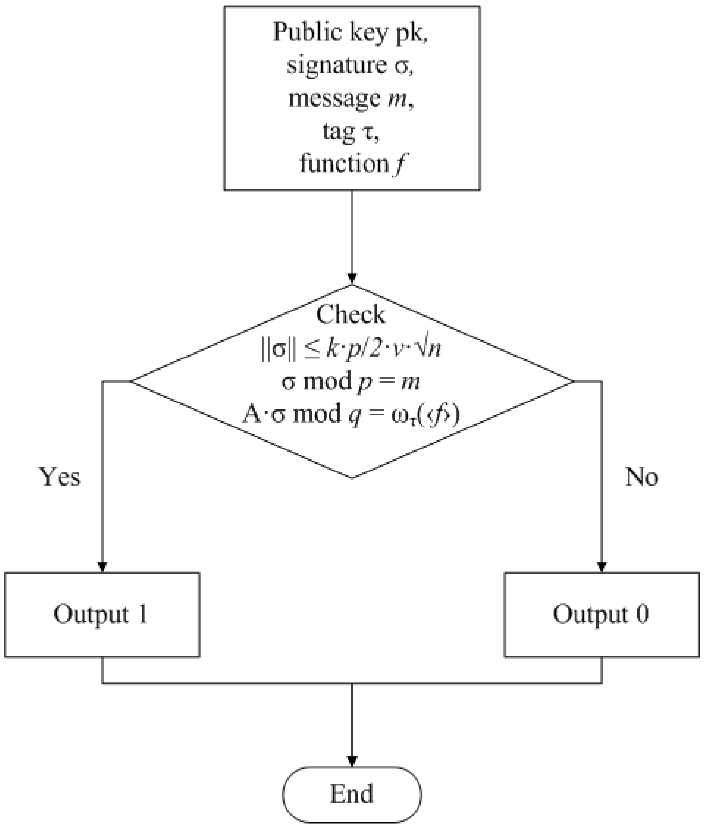


FIGURE 5.4.1. the flow chart of the algorithm Verify

The Input of Verify. In order to start a process of signing, the program needs a bunch of valid input. The input includes:

Public Key pk: An integer p that defines Λ_1, a matrix $\mathbf{A}$ that defines Λ_2, the data set size k and a parameter $v = p \cdot \sqrt{n \log q}$.

Message m: A vector in $\mathbb{F}_p^n$, which could be in $\mathbb{Z}^n$. It needs to be mapped into our message space.

Tag τ: An n-bit string of $\{0, 1\}$, which is initialized as a *ZZ* during the implementation.

Index i: An integer, which locates m in the message set tagged by τ.

Function f: a vector in $\mathbb{Z}^k$, which implies how function f is encoded as $\langle f \rangle$

Signature σ: A vector in $\mathbb{Z}^n$, which is supposed to be a signature of m on f.

Checking Three Verification Conditions. To determine whether a signature is valid, three verification conditions are required to check.

First, since the length of every valid signature is well bonded, a signature with inappropriate length should be culled.

Check the length of a signature

```
(1) Calculate $\|\sigma\| = \sqrt{\sigma \cdot \sigma}$.
(2) if $\|\sigma\| \geq k \cdot \frac{p}{2} \cdot v \cdot \sqrt{n}$
      Output 0
```

Then, we check whether signature σ is signed on m for f in $\Lambda_1 \cap \Lambda_2$.

Check the integrity of message m applied by function f

```
(1) Calculate $m' = \sigma \bmod p$.
(2) if $m' \neq m$
      Output 0
(3) Calculate $\omega_\tau(\langle f \rangle)$
      (1) For $i = 1, \ldots, k$ $\alpha_i \leftarrow H(\tau \| i)$.
      (2) $\omega_\tau(\langle f \rangle) = \Sigma_{i=1}^{k} c_i \alpha_i$.
(4) Calculate $h = \mathbf{A} \cdot \sigma \bmod q$
(5) If $h \neq \omega_\tau(\langle f \rangle)$
      Output 0
(6) Output 1
```

The Output of Verify. If the input signature σ get "1" from length check and the message m and function f get "1" from integrity check, Verify output "1" to indicate that message m with function f is authenticated by a valid signature σ.

5.5. Implementing Evaluate of Linearly Homomorphic Signature Scheme

According to the description, the main purpose of Evaluate is to generate a signature σ in $\mathbb{Z}^n$ for the result of applying function f on a message set $\vec{m}$, and it has been implemented in *evaluate.cc*.

One of its attractive advantages is that Evaluate generates signatures without contacting the original authenticator. Signatures generated by Evaluate are constructed from individual signatures on messages of a message set.

Evaluete has three steps as shown in the flow chart Figure 5.5.1.

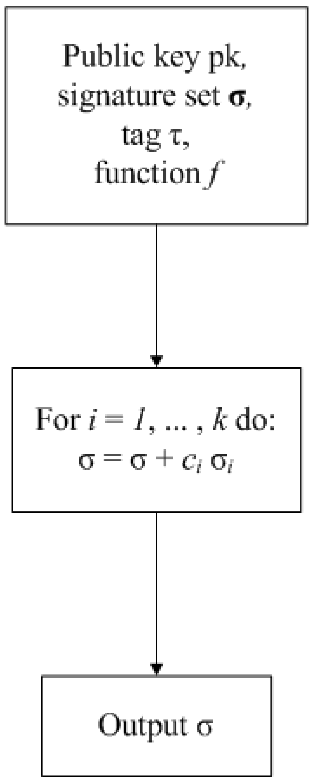

Public key pk, signature set σ, tag τ, function f
For i = 1, ... , k do: σ = σ + c_i σ_i
Output σ

Figure 5.5.1. the flow chart of the algorithm Evaluate

5.6. Summary

This chapter introduced the techniques we tried, described the process of implementing the linearly homomorphic scheme. We especially explained how four algorithms in the scheme are constructed, how specific functional algorithms connects in four algorithms, and detailed computations within them.

Core procedures in implemented programs have been tested to achieve better performance such as TrapGen and SampleD. The next chapter will illustrate the experiment results of their timing tests.

CHAPTER 6

Performance

The timing test measures the difference of time used to run procedures between specific procedures in our implementation and nested for loops. Time used by nested for-loops is used and rescaled as standard polynomial time to analysis the performance of procedures.

6.1. A Timing Test of Algorithm TrapGen

The first test performed is to investigate how fast a lattice associated with a matrix can be generated by TrapGen, including generating a random matrix. The algorithm TrapGen discussed in previous chapter is applied to several different security parameters and a fixed data size.

The test was performed running 10,000 times of these procedures with 256, 310, 450, 512, 740, 810 and 1024 as values for n. The data size k was set to 10. The average results are shown in table1. In order

Security Parameter n	TrapGen(n)	O(n^3)
256	92.45ms	39.57ms
320	151.8ms	77.01ms
450	353.7ms	218.7ms
512	479.6ms	316.6ms
740	1842.8ms	939.7ms
810	2988.9ms	1250.2ms
1024	7069.1ms	2520.4ms

TABLE 1. Average time used of timing test of TrapGen

to have an intuitive view, the average results in table 1 are fitted to curves. Since the theoretical time complexity of TrapGen is $O(n^3)$ as implemented, its curve has been fitted as a cubic curve. From the chart as shown in Figure 6.1, it can be seen that the increasing speed of TrapGen's curve is fairly faster than $O(n^3)$'s. One possible reason is that TrapGen consists of lots of inner computations which have time complex of $O(n^2)$, while cubic for-loops do not have. Another possible reason, which is more important, is that some functions from NTL that TrapGen used have time complexity of $O(n^4)$ but is fast or haven't been applied to n-scale size data. In a word, the time complexity of TrapGen is complicated but the average time it used is acceptable.

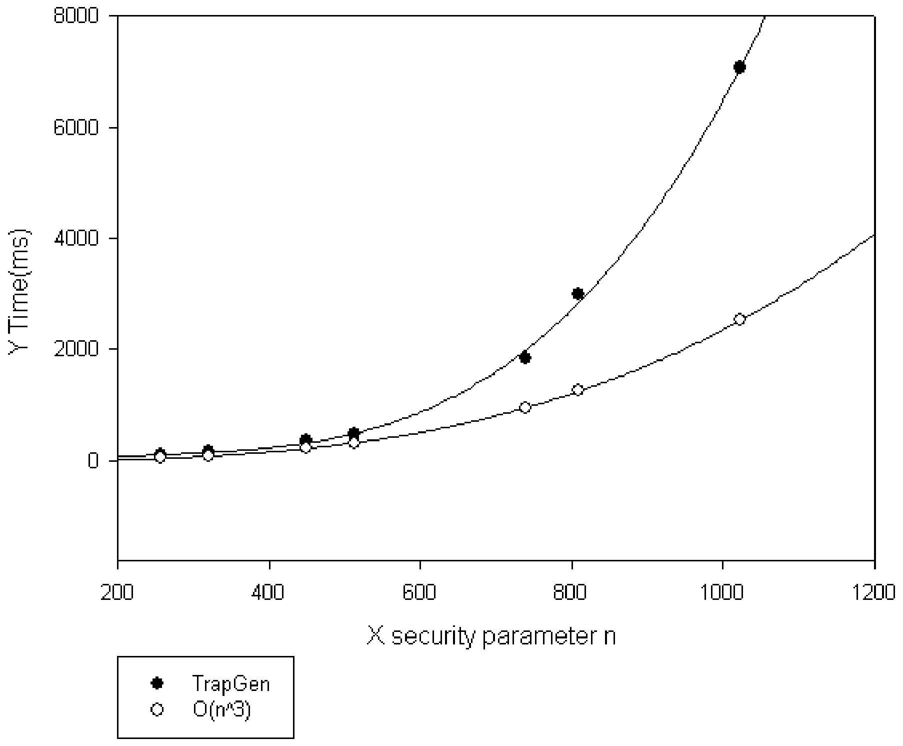


FIGURE 6.1.1. fitted curves of average time used of table 1

6.2. A Timing Test of Algorithm SampleD

Then another test is run to investigate the performance of algorithm SampleD, which samples a vector in a lattice, including generating a Gram-Schmidt orthogonalization. The algorithm SampleD discussed in previous chapter is applied to shor bases randomly generated by Setup with different security parameters and a fixed data size as well as corresponding primes.

The test was performed running 10,000 times of these procedures with 256, 310, 450, 512, 740, 810 and 1024 as values for n. The data size k was set to 10. The average results are shown in table2. Same as the analysis of TrapGen, the average results in table 1 are fitted to curves. Since the theoretical time complexity of SampleD is $O(n^3)$ as implemented, its curve has been also fitted as a cubic curve.

From the chart as shown in Figure 6.2, it can be seen that the increasing speed of SampleD's curve is a little bit faster than $O(n^3)$'s but much slower than $O(n^4)$'s. One possible reason is that sampleD consists of lots of inner computations which have time complex of $O(n^2)$, while cubic for-loops do not have. However, it is clear that SampleD's curve is "covered" by $O(n^4)$'s, and it is unlikely that a further inflection

Security Parameter n	sampleD(n)	$O(n^3)$	$O(n^4)$
256	5.473s	10.10	3.957s
320	10.718s	24.24s	7.701s
450	31.586s93.65s	21.44s	
512	46.182s	156.14s	31.66s
740	120.522s	599.23s	93.97s
810	233.017s	991.99s	125.02s
1024	414.981s	2456.92s	252.04s

TABLE 2. Average time used of timing test of SampleD

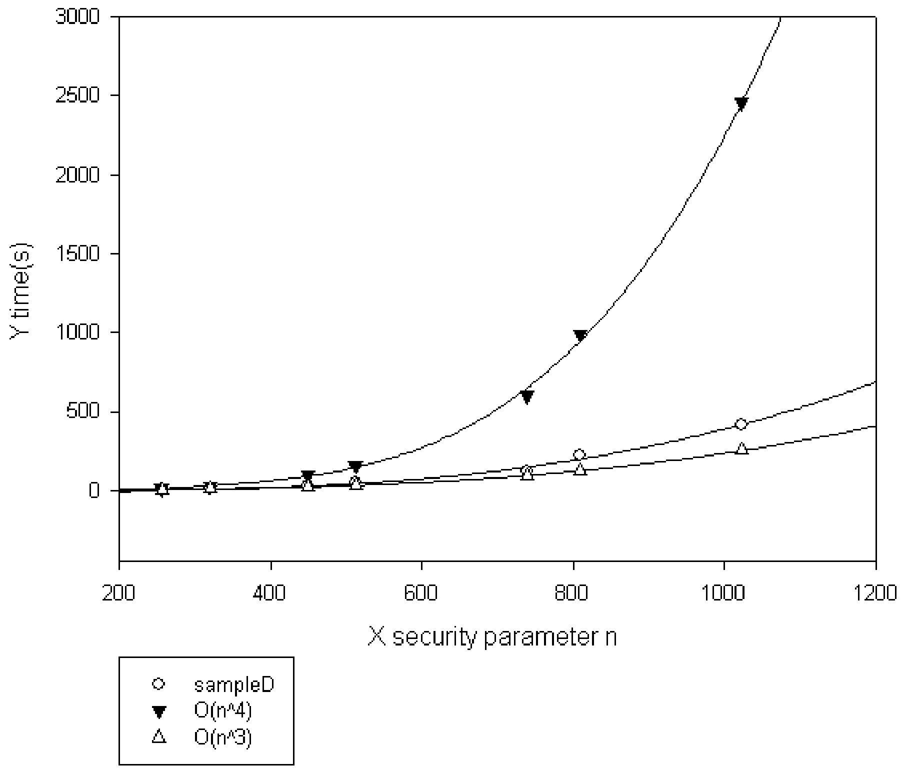


FIGURE 6.2.1. Performance of timing test of SampleD

point will rocket sampleD's curve above $O(n^4)$'s. Therefore, it can be concluded that the time complexity of sampleD is in polynomial time. However, in terms of average time used, SampleD takes much longer than TrapGen, by comparing table 1 and table 2. A probable answer is that the process of generating Gram-Schimdt orthogonalization takes too much time because many matrix computations within it. In a word, the time complexity of sampleD is quite good, but its performance still needs further optimization.

CHAPTER 7

Conclusion

This dissertation discussed homomorphic signature schemes proposed by Boneh and Freeman in depth. The objective was to implement a version of homomorphic signature that authenticates functions as well as results of applying them on messages. The functions mentioned above includes linear functions and polynomial functions of bounded degree. An implementation of linearly homomorphic signature has been achieved at the end of this dissertation, some critical tests included.

At the beginning, we introduced a question about computing on signed data and gave a brief overview of homomorphic signatures.

In order to implement a homomorphic signature scheme, first of all we need to know what it is. Chapter 2 presented many definitions related to homomorphic signatures. In this chapter, we showed how homomorphic signatures differs from conventional digital signatures. There are two major differences. The first one is that homomorphic signature not only signs data but also signs functions which doubles the work of Setup and Sign as well as Verify. The second one is homomorphic signature has a special algorithm to handle signature sets to generate other valid signatures.

Chapter 3 reviewed basic mathematical background of homomorphic signatures. Lattices and Ideals were explained with examples, and an approach to calculate a lattice associated with a matrix was introduced.

Chapter 4 discussed homomorphic signature schemes, by which the implementation can be carried out. The correctness of both two homomorphic signatures were proven, and security properties of linearly homomorphic were talked about.

After chapter 4, where a brief constructing flow was given, the process of implementation was described in chapter 5. In this chapter, details of building four algorithm of homomorphic signatures were presented seperately. Algorithms they used and techniques they based on were explained with examples. Some optimizations were given as well, along with their advantages and disadvantages.

Chapter 6 illustrated the results of timing tests of two core procedures. Time complexity of these two procedures were analyzed with consideration of real time used, and it concluded that their time complexity is acceptable.

7.1. Open Problems

There are many open problems that remain in homomorphic signatures. First, as explained in the introduction, derived signatures are desired not to leak information about the original data set. This privacy property still can not be solved for polynomial functions.

Second, hash function *SHA-512* is applied when computing hash of functions in the implemented homomorphic signature schemes. It is weak against a chosen message attack. To upgrade the security of our scheme, new techniques are needed to eliminate the random oracle.

Finally, the implementation in this dissertation can been as a step on the path to a *fully homomorphic signature scheme*, which could derive signatures of results of applying *any* function on signed data. Thus it would be a useful paralled to existing fully homomorphic encryption system.

Bibliography

[1] John Abbott, Manuel Bronstein, and Thom Mulders. Fast deterministic computation of determinants of dense matrices. In *Proceedings of the 1999 international symposium on Symbolic and algebraic computation*, ISSAC '99, pages 197–204, New York, NY, USA, 1999. ACM. 35

[2] Jae Hyun Ahn, Dan Boneh, Jan Camenisch, Susan Hohenberger, abhi shelat abhi shelat, and Brent Waters. Computing on authenticated data. Cryptology ePrint Archive, Report 2011/096, 2011. `http://eprint.iacr.org/`. 3

[3] Joël Alwen and Chris Peikert. Generating shorter bases for hard random lattices. *Theory of Computing Systems*, 48:535–553, 2011. 10.1007/s00224-010-9278-3. 10, 11, 18, 19, 23, 29, 30, 32

[4] Dan Boneh and David Freeman. Linearly homomorphic signatures over binary fields and new tools for lattice-based signatures. In Dario Catalano, Nelly Fazio, Rosario Gennaro, and Antonio Nicolosi, editors, *Public Key Cryptography – PKC 2011*, volume 6571 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-19379-8_1. 2, 4, 8, 20

[5] Dan Boneh and David Mandell Freeman. Homomorphic signatures for polynomial functions. Cryptology ePrint Archive, Report 2011/018, 2011. `http://eprint.iacr.org/`. i, 1, 2, 3, 4, 5, 6, 7, 8, 9, 13, 16, 19, 20, 21, 32, 39, 43

[6] R. Bradford. Hermite normal forms for integer matrices. In James Davenport, editor, *Eurocal '87*, volume 378 of *Lecture Notes in Computer Science*, pages 315–316. Springer Berlin / Heidelberg, 1989. 10.1007/3-540-51517-8_133. 33

[7] Christina Brzuska, Marc Fischlin, Tobias Freudenreich, Anja Lehmann, Marcus Page, Jakob Schelbert, Dominique Schröder, and Florian Volk. Security of sanitizable signatures revisited. In *Proceedings of the 12th International Conference on Practice and Theory in Public Key Cryptography: PKC '09*, Irvine, pages 317–336, Berlin, Heidelberg, 2009. Springer-Verlag. 3

[8] Ee-Chien Chang, Chee Liang Lim, and Jia Xu. Short redactable signatures using random trees, 2009. xujia comp.nus.edu.sg 14258 received 9 Jan 2009, last revised 13 Jan 2009. 3

[9] John D. Dixon. Exact solution of linear equations using&lt;i&gt;p&lt;/i&gt;-adic expansions. *Numerische Mathematik*, 40:137–141, 1982. 10.1007/BF01459082. 35, 36

[10] P. D. Domich, R. Kannan, and Jr. L. E. Trotter. Hermite normal form computation using modulo determinant arithmetic. *Math. Oper. Res.*, 12:50–59, February 1987. 26

[11] P. D. Domich, R. Kannan, and Jr. L. E. Trotter. Hermite normal form computation using modulo determinant arithmetic. *Math. Oper. Res.*, 12:50–59, February 1987. 33, 34

[12] U. Feige and A. Shamir. Witness indistinguishable and witness hiding protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, STOC '90, pages 416–426, New York, NY, USA, 1990. ACM. 8

[13] Joachim Von Zur Gathen and Jurgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2 edition, 2003. 34

[14] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford, CA, USA, 2009. AAI3382729. 2, 11, 13

[15] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, STOC '08, pages 197–206, New York, NY, USA, 2008. ACM. i, 2, 3, 16, 19, 20, 23, 41, 42

[16] Stuart Haber, Yasuo Hatano, Yoshinori Honda, William Horne, Kunihiko Miyazaki, Tomas Sander, Satoru Tezoku, and Danfeng Yao. Efficient signature schemes supporting redaction, pseudonymization, and data deidentification. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, ASIACCS '08, pages 353–362, New York, NY, USA, 2008. ACM. 3

[17] Robert Johnson, David Molnar, Dawn Xiaodong Song, and David Wagner. Homomorphic signature schemes. In *Proceedings of the The Cryptographer's Track at the RSA Conference on Topics in Cryptology*, CT-RSA '02, pages 244–262, London, UK, UK, 2002. Springer-Verlag. 1

[18] Robert Johnson, David Molnar, Dawn Xiaodong Song, and David Wagner. Homomorphic signature schemes. In *Proceedings of the The Cryptographer's Track at the RSA Conference on Topics in Cryptology*, CT-RSA '02, pages 244–262, London, UK, UK, 2002. Springer-Verlag. 3

[19] Joe Kilian. Improved efficient arguments. In *Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '95, pages 311–324, London, UK, 1995. Springer-Verlag. 3

[20] Vadim Lyubashevsky. Lattice-based identification schemes secure under active attacks. In Ronald Cramer, editor, *Public Key Cryptography – PKC 2008*, volume 4939 of *Lecture Notes in Computer Science*, pages 162–179. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-78440-1_10. 13

[21] Vadim Lyubashevsky and Daniele Micciancio. Asymptotically efficient lattice-based digital signatures. In *Proceedings of the 5th conference on Theory of cryptography*, TCC'08, pages 37–54, Berlin, Heidelberg, 2008. Springer-Verlag. 3

[22] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-13190-5_1. 2

[23] Silvio Micali. Computationally sound proofs. 30(4):1253–1298, 2000. 3

[24] Daniele Micciancio. Improving lattice based cryptosystems using the hermite normal form. In *Revised Papers from the International Conference on Cryptography and Lattices*, CaLC '01, pages 126–145, London, UK, 2001. Springer-Verlag. 11

[25] Daniele Micciancio and Oded Regev. Worst-case to average-case reductions based on gaussian measures. *SIAM J. Comput.*, 37:267–302, April 2007. 4, 18, 19, 20, 23, 24

[26] Daniele Micciancio and Bogdan Warinschi. A linear space algorithm for computing the hermite normal form. In *Proceedings of the 2001 international symposium on Symbolic and algebraic computation*, ISSAC '01, pages 231–236, New York, NY, USA, 2001. ACM. 33, 34, 35, 37

[27] Clément Pernet and William Stein. Fast computation of hermite normal forms of random integer matrices. *Journal of Number Theory*, 130(7):1675–1683, 2010. 32, 33, 34, 35, 36, 37

[28] KANNAN R. Polynomial algorithms for computing the smith and hermite normal forms of an integer matrix. *SIAM Journal on computing*, 8(4):499–507, 1979. 33

[29] Victor Shoup. Ntl: A library for doing number theory, Sept 2011. 25

[30] N. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In Phong Nguyen and David Pointcheval, editors, *Public Key Cryptography – PKC 2010*, volume 6056 of *Lecture Notes in Computer Science*, pages 420–443. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-13013-7_25. 22, 23

[31] Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. Efficient public key encryption based on ideal lattices. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 617–635. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-10366-7_36. 13

[32] Ron Steinfeld, Laurence Bull, and Yuliang Zheng. Content extraction signatures. In Kwangjo Kim, editor, *Information Security and Cryptology — ICISC 2001*, volume 2288 of *Lecture Notes in Computer Science*, pages 163–205. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-45861-1_22. 3

[33] Peter Stevenhagen. The arithmetic of number rings. In *Algorithmic number theory: Lattices, number fields, curves and cryptography*, volume 44 of *Mathematical Sciences Research Institute Publications*, pages 209–266. Cambridge Univ. Press, 2008. 13

[34] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *Proceedings of the 5th conference on Theory of cryptography*, TCC'08, pages 1–18, Berlin, Heidelberg, 2008. Springer-Verlag. 3

APPENDIX A

Source Code

setup.cc and sign.cc are appended.

setup.cc

```
/*
 * setup.cc
 *
 *  Created on: Aug 3, 2011
 *      Author: arthur
 */

#include<NTL/ZZ.h>
#include<NTL/ZZ_p.h>
#include<NTL/mat_ZZ.h>
#include<NTL/mat_ZZ_p.h>
#include<fstream>
#include<math.h>
#include<NTL/HNF.h>
NTL_CLIENT

ofstream fout;

mat_ZZ callHadamard(int d) {
 mat_ZZ mat, matT;
 mat.SetDims(d, d);
 if (d == 1)
  mat[0][0] = 1;
 else {
  matT = callHadamard(d / 2);
  int i, j;
  for (i = 0; i < d / 2; i++)
   for (j = 0; j < d / 2; j++) {
    mat[i][j] = matT[i][j];
    mat[i + d / 2][j] = matT[i][j];
    mat[i][j + d / 2] = matT[i][j];
    mat[i + d / 2][j + d / 2] = -matT[i][j];
   }
 }
 return mat;
}

```

```
void trapGen(ZZ q, long l, long n1, long n2, mat_ZZ_p* matA
    , mat_ZZ_p* matA1,
  mat_ZZ* basis, mat_ZZ_p* matA2) {

 mat_ZZ matU, matG, matR, matP, matHNF, matHp;
 matU.SetDims(n2, n2);
 matG.SetDims(n1, n2);
 matR.SetDims(n1, n2);
 matP.SetDims(n2, n1);
 int i, j, d = 4 * l * NumBits(q) / 3;

 /*
  * Calculate HNF of lattice_(A1)
  */

 mat_ZZ_p matL1;
 kernel(matL1, transpose(*matA1));
 matL1 = transpose(matL1);
 fout << "Kernel of A" << endl << matL1 << endl;

 mat_ZZ matA1Z;
 mat_ZZ_p matA1Zp;
 if (matL1.NumRows() == 0) {
  matA1Z.SetDims((*matA1).NumCols(), (*matA1).NumCols());
  for (i = 0; i < matA1Z.NumCols(); i++)
   matA1Z[i][i] = q;
 } else {

  matA1Z.SetDims(matL1.NumRows(), matL1.NumCols() + matL1.
      NumRows());
  matA1Zp.SetDims(matL1.NumRows(), matL1.NumCols());
  for (i = 0; i < matL1.NumRows(); i++)
   for (j = 0; j < matL1.NumCols(); j++) {
    matA1Z[i][j] = rep(matL1[i][j]);
    matA1Zp[i][j] = matL1[i][j];
   }
  for (i = matL1.NumCols(); i < matA1Z.NumRows(); i++) {
   matA1Z[i][i] = q;
  }
 }

 HNF(matHNF, transpose(matA1Z), power(q, (*matA1).NumRows()
     ));

 matHp.SetDims(matHNF.NumRows(), matHNF.NumCols());
 matHNF = transpose(matHNF);
 for (i = 0; i < matHNF.NumRows(); i++)
  for (j = 0; j < matHNF.NumCols(); j++)
```

```
      if (i != j)
       matHp[i][j] = matHNF[i][j];
      else
       matHp[i][j] = matHNF[i][j] - 1;

  /*
   * Calculate U
   */
  fout << "matU" << endl;
  int htU = 0;

  for (j = 0; j < n1; j++) {
   long tempLen = NumBits(matHNF[j][j]);
   if (tempLen != 1) {
    for (i = htU; i < htU + tempLen - 1; i++)
     matU[i][i + 1] = -2;
    htU += NumBits(matHNF[j][j]);
   }
  }
  for (j = 0; j < n2; j++)
   matU[j][j] = 1;
  fout << matU << endl;

  /*
   * Calculate G
   */
  int wdG = 0;
  for (i = 0; i < n1; i++) {
   long tempW = NumBits(matHNF[i][i]);
   if (tempW != 1) {
    for (j = wdG; j < wdG + tempW; j++)
     matG[i][j] = to_ZZ(pow(2, j - wdG));
    wdG += NumBits(matHNF[i][i]);
   }
  }
  int dG = 1;

  while (dG * 2 < n2 - 2 * l * NumBits(q))
   dG *= 2;
  mat_ZZ matHdm;
  matHdm = callHadamard(dG);

  for (i = 0; i < d; i++)
   for (j = 0; j < dG; j++)
    matG[i][j + wdG] = matHdm[i][j];

  /*
   * Calculate R
  */
 int dR, intr;
 dR = d;
 for (i = 0; i < dR; i++)
  for (j = 0; j < n2; j++) {
   intr = rand() % 4;
   switch (intr) {
   case 1:
    matR[i][j] = 1;
    break;
   case 2:
    matR[i][j] = -1;
    break;
   default:
    matR[i][j] = 0;
   }
  }
 fout << "matR" << endl << matR << endl;
 /*
  * Calculate P
  */
 ZZ hpZ;
 int htP = 0, k;
 for (i = 0; i < n1; i++) {
  for (j = 0; j < n1; j++) {
   hpZ = matHp[i][j];
   k = 0;
   while (hpZ > 0) {
    matP[k + htP][j] = hpZ % 2;
    hpZ /= 2;
    k++;
   }
  }
  htP += NumBits(matHNF[i][i]);
 }

 /*
  * A2
  */
 mat_ZZ_p matRp = to_mat_ZZ_p(matR), matGp = to_mat_ZZ_p(
     matG);
 (*matA2).SetDims(l, n2);
 (*matA2) = (-1) * (*matA1) * (matRp + matGp);

 (*basis).SetDims(n1 + n2, n1 + n2);
 mat_ZZ matT_1, matT_2;
 matT_1 = (matG + matR) * matU;
 matT_2 = matR * matP;
 for (i = 0; i < matT_2.NumRows(); i++)
  matT_2[i][i] -= 1;
 for (i = 0; i < n1; i++)
  for (j = 0; j < n2; j++)
   (*basis)[i][j] = matT_1[i][j];
 for (i = 0; i < n1; i++)
  for (j = n2; j < n1 + n2; j++)
   (*basis)[i][j] = matT_2[i][j - n2];
 for (i = n1; i < n1 + n2; i++)
  for (j = 0; j < n2; j++)
   (*basis)[i][j] = matU[i - n1][j];
 for (i = n1; i < n1 + n2; i++)
  for (j = n2; j < n1 + n2; j++)
   (*basis)[i][j] = matP[i - n1][j - n2];

 (*matA).SetDims(l, n1 + n2);
 clear(*matA);
 for (i = 0; i < l; i++)
  for (j = 0; j < n1; j++)
   (*matA)[i][j] = (*matA1)[i][j];
 for (i = 0; i < l; i++)
  for (j = n1; j < n1 + n2; j++)
   (*matA)[i][j] = (*matA2)[i][j - n1];

}

int main() {
 long n, k, l;

 fout.open("output.txt");
 cin >> n >> k;
 ZZ p, q;
 long len = 0, lp;

 p = 0;
 q = -1;
 while (q < (n * p * k) * (n * p * k)) {
  srand(time(NULL));
  len = rand() % (n / 25) + n / 49 + 1 + rand() % (n / 10);

  GenPrime(q, len, 80);
  lp = rand() % (len / 3) + 2;
  GenPrime(p, lp, 80);
 }

 l = n / (6 * NumBits(q));

 ZZ_p::init(q);

 long n1 = (long) (len * l * (1.34f) + 1);

 mat_ZZ bsT;
 mat_ZZ_p matA1, matA2, matL;
 matA1.SetDims(l, n1);

 int i, j;
 SetSeed(to_ZZ(time(NULL)));
 for (j = 0; j < n1; j++)
  for (i = 0; i < l; i++) {
   ZZ_p r = random_ZZ_p();
   matA1[i][j] = r;
  }
 ZZ vv = p * NumBits(to_ZZ(n)) * sqrt(n * NumBits(q));

 mat_ZZ_p matA;
 trapGen(q, l, n1, n - n1, &matA, &matA1, &bsT, &matA2);
 bsT = bsT * p;
 fout << "p" << endl << p << endl;
 fout << "matA" << endl << matA << endl;
 fout << "v" << endl << v << endl;
 fout << "k" << endl << k << endl;
 fout << "bsT" << endl << bsT << endl;
 fout.close();
 return 0;
}
```

sign.cc

```
/*
 * sign.cc
 *
 *  Created on: Aug 20, 2011
 *      Author: arthur
 */
#include<NTL/ZZ.h>
#include<NTL/ZZ_p.h>
#include<NTL/vec_ZZ.h>
#include<NTL/vec_ZZ_p.h>
#include<NTL/vec_RR.h>
#include<NTL/mat_ZZ.h>
#include<NTL/mat_ZZ_p.h>
#include<NTL/mat_RR.h>
#include<NTL/RR.h>

NTL_CLIENT

ZZ sampleZ(ZZ s, ZZ c, long n) {
 ZZ_p x;
 ZZ m = ZZ_p::modulus();
 long tn = 0, nn = n;
 while (nn > 0) {
  nn /= 10;
  tn++;
 }
 tn++;
 ZZ_p::init(2 * s * tn);
 x = random_ZZ_p();
 ZZ_p::init(m);
 return rep(x) + c;
}
mat_RR Gs(mat_ZZ matA) {
 mat_ZZ matAt = transpose(matA);
 mat_RR gsA, matAr;
 gsA.SetDims(matAt.NumRows(), matAt.NumCols());
 matAr.SetDims(matAt.NumRows(), matAt.NumCols());
 int i, j;
 RR tempR;
 for (i = 0; i < matAt.NumRows(); i++)
  for (j = 0; j < matAt.NumCols(); j++) {
   conv(tempR, matAt[i][j]);
   matAr[i][j] = tempR;
  }
 RR mij, inner1, inner2;
 for (i = 0; i < matAt.NumRows(); i++) {
  gsA[i] = matAr[i];
  for (j = 0; j < i; j++) {
   InnerProduct(inner1, matAr[i], gsA[j]);
   InnerProduct(inner2, gsA[j], gsA[j]);
   mij = inner1 / inner2;
   gsA[i] -= mij * gsA[j];
  }
 }
 return gsA;
}
vec_ZZ sampleD(mat_ZZ bsT, ZZ s, vec_ZZ c, long n) {
 vec_ZZ v, cn = c;
 v.SetLength(n);
 mat_RR gT;
 mat_ZZ bsTt = transpose(bsT);
 gT = Gs(bsT);
 int i, j;
 for (i = 0; i < n; i++)
  v[i] = to_ZZ(0);
 RR inner1, inner2;
 ZZ zn, spz, cpz;
 RR sp, cp, root;
 vec_RR cnr;
 cnr.SetLength(cn.length());
 RR tempR;
 for (i = n - 1; i >= 0; i--) {
  for (j = 0; j < cn.length(); j++) {
   conv(tempR, cn[j]);
   cnr[j] = tempR;
  }
  InnerProduct(inner1, cnr, gT[i]);
  InnerProduct(inner2, gT[i], gT[i]);
  cp = inner1 / inner2;
  conv(sp, s);
  conv(root, sqrt(inner2));
  sp = sp / root;
  conv(spz, sp);
  conv(cpz, cp);
  zn = sampleZ(spz, cpz, n);
  cn = cn - zn * bsTt[i];
  v = v + zn * bsTt[i];
 }
 return v;
}

vec_ZZ samplePre(mat_ZZ bsT, ZZ s, vec_ZZ c, long n) {
 SetSeed(to_ZZ(time(NULL)));
 vec_ZZ v = sampleD(bsT, s, -c, n);
 v += c;
 return v;
}

vec_ZZ_p Hash(vec_ZZ) {
 //return NULL;
}

int main() {
 mat_ZZ bsT, matMp, matAt, matA, matAlpha, matM;

 ZZ tau;
 vec_ZZ message, t;
 vec_ZZ_p messageC;
 ZZ v;
 long index;
 ZZ p = to_ZZ(11), q = to_ZZ(31);
 cin >> sk >> matA >> tau >> message >> index >> p >> q >> v;

 ZZ_p::init(p);
 message = to_vec_ZZ(to_vec_ZZ_p(message));
```

```

/*
 * calculate alpha_i
 */

/*
 * calculate t
 *
 * p can not be too big
 */
int i, x, j;
matM.SetDims(message.length(), 1);
for (i = 0; i < message.length(); i++)
 matM[i][0] = message[i];
zz_p::init(to_long(p));
matMp = matA * matM;
mat_zz_p matMpp = to_mat_zz_p(matMp);
matAt = matAlpha;
x = CRT(matAt, q, matMpp);
mat_RR matAex;
matAex.SetDims(matA.NumCols(), matA.NumCols());

for (i = 0; i < matA.NumCols(); i++)
  for (j = 0; j < matA.NumCols(); j++) {
    cout << i << " " << j << endl;
    if (i < matA.NumRows())
      matAex[i][j] = to_RR(matA[i][j]);
    else if (i == j)
      matAex[i][j] = to_RR(1);
    //FIXME set diag to 1;

  }
cout << matAex << endl << determinant(matAex) << endl;
mat_RR matAtex;
matAtex.SetDims(matA.NumCols(), 1);
for (i = 0; i < matA.NumRows(); i++)
  if (i < matAt.NumRows())
    matAtex[i][0] = to_RR(matAt[i][0]);
  else
    matAtex[i][0] = to_RR(0);
cout << matAtex << endl;
matAtex = inv(matAex) * matAtex;
cout << matAtex << endl;
vec_ZZ vecTZ;
matTZ.SetDims(matAt.NumRows(), 1);
for (i = 0; i < matAt.NumRows(); i++)
  vecTZ[i] = to_ZZ(matAtex[i][0] + 0.1);
vec_ZZ vv = samplePre(bsT, v, vecTZ, n);
```

```
  cout << vv << endl;
}
```