April 2013

The LTV Homomorphic Encryption Scheme and Implementation in Sage

Quanquan Ma
Worcester Polytechnic Institute

The LTV Homomorphic Encryption Scheme and Implementation in Sage

A Major Qualifying Project
submitted to the Faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements for
the Degree of Bachelor of Science

by

Quanquan Ma

April 25, 2013

Approved:

Professor William J. Martin, Project Advisor

Professor Berk Sunar, Project Advisor

Abstract

The purpose of this project is to study the Multi-key Fully Homomorphic Encryption (FHE) scheme developed by López-Alt, Tromer and Vaikuntanathan, which we abbreviate as the LTV scheme. An FHE scheme is a cryptosystem in which we can evaluate any circuit in an encrypted form and decrypt the result afterwards. The LTV scheme we studied is based on NTRU, a public-key cryptosystem using lattice-based cryptography, and it encrypts each single bit of data into one corresponding polynomial.

This report includes the background research on NTRU encryption scheme, the presentation of this FHE scheme in a single-key version, and the implementation of it in Sage, an open-source mathematics tool. The results of this project are a more accessible version of the original scheme with serious mathematical proofs and a Sage package that implements the basic scheme and some real-world applications such as an n-bit Adder. The Sage package is posted in Sage Interact Community website.

Contents

1 Introduction

In the 21st century, computing is not limited to being performed locally on end users' devices anymore. It is transforming to a model consisting of services that are provided by servers at remote locations and delivered in a way similar to traditional utilities such as water, electricity, and gas [1]. In such a model, end users with light-weight devices such as web browsers and mobile phones are able to access powerful computing services regardless of the location of the service host or the manner of delivery. *Cloud computing*, the most dominant computing paradigm to realize this utility computing vision, has drawn tremendous attention from everyone in the world. A survey of 1,650 IT and business executives in 2012 showed that, on average, more than a third of their current IT budgets are now allocated to cloud computing [2].

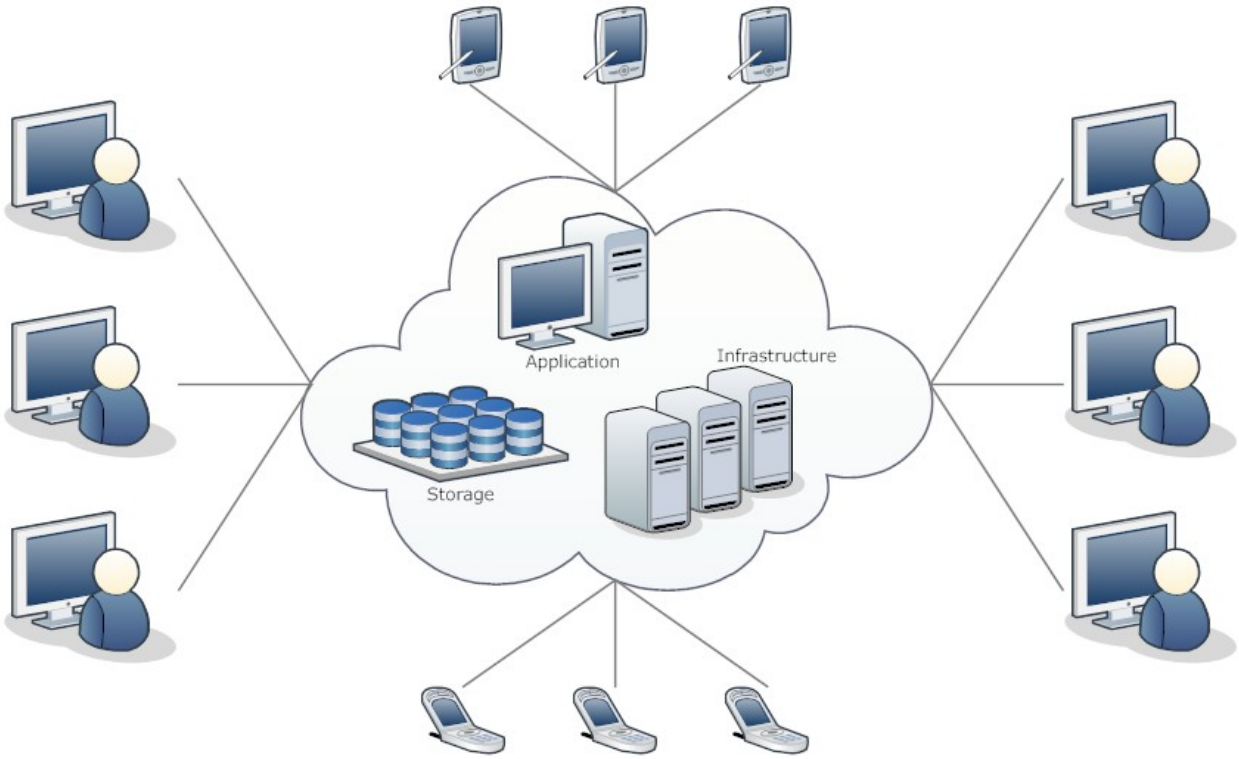

Figure 1: The cloud infrastructure [3]

Cloud computing allows businesses and users to access applications, infrastructure, platforms from anywhere in the world via any device as long as it is connected to the cloud – the powerful remote servers (shown in Figure 1). Although cloud computing offers numerous advantages in various ways, security issues have always been a problem of much debate, since data stored in the cloud could be vulnerable to unauthorized use by the cloud provider and even other cloud clients [4]. Therefore, encrypting data that contains sensitive information before sending it to the cloud is crucial to any cloud user.

However, imagine the case where we want to use tax preparation software from the cloud. We would like to perform computations from the cloud, but, certainly, we do not want to leak the financial numbers involved in computing, which means that we need to encrypt the data before throwing it in the cloud. Although traditional encryption schemes can make sure the data is well-secured, it is not hard to see that cloud-based software has little hope of performing meaningful computations on the data when it is encrypted. This no doubt limits the great advantage of cloud computing. Is it possible for the cloud to do computations on inputs in encrypted form while making sure that the result can be decrypted later (shown in Figure 2)?

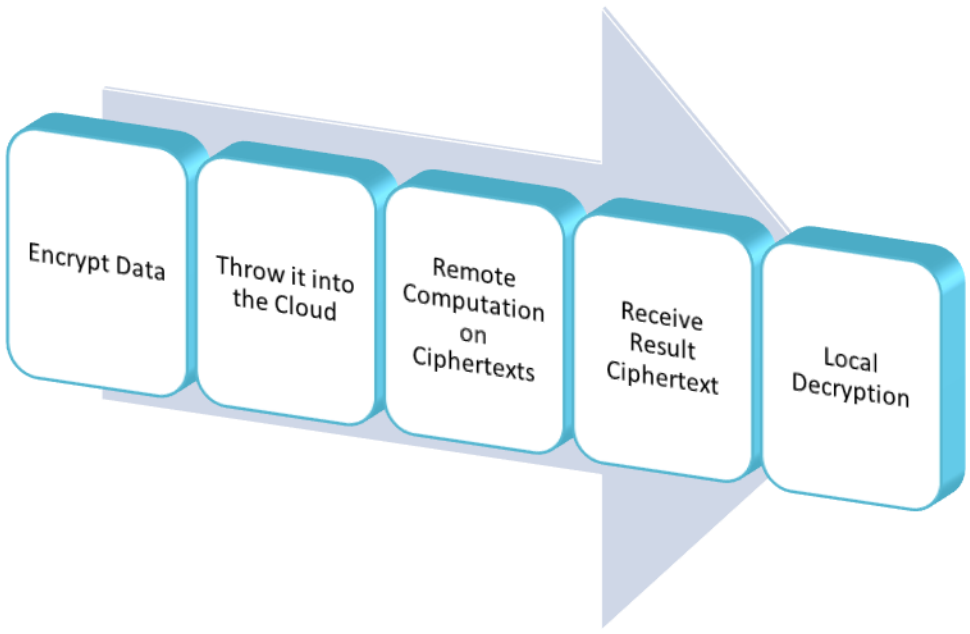

Figure 2: Ideal flow when working with the cloud

The answer is yes, with homomorphic encryption. In simple terms, a homomorphic encryption scheme is a scheme that allows us to perform arbitrary computations on ciphertexts and decrypt the result afterwards. Let's start from a very simple encryption scheme, the Caesar cipher. Here, given a secret key X, we shift every letter to X letters after it in the alphabet. In our case, we choose $X = 13$. (When $X = 13$, this scheme is also called ROT13.) This scheme is partially homomorphic with respect to the concatenation operation. Below is a

simple demonstration due to Craig Stuntz [5]:

$$\begin{aligned} c_1 &:= \mathsf{Encrypt}(\text{“HELLO”}, 13) = \text{“URYYB”} \\ c_2 &:= \mathsf{Encrypt}(\text{“WORLD”}, 13) = \text{“JBEYQ”} \\ c &:= \mathsf{Concat}(c_1, c_2) = \text{“URYYBJBEYQ”} \\ m &:= \mathsf{Decrypt}(c) = \text{“HELLOWORLD”} \end{aligned}$$

As we can see, we are able to perform concatenate operation on the ciphertexts with no need to decrypt them first. Figure 3 gives us a graphical demonstration of this example.

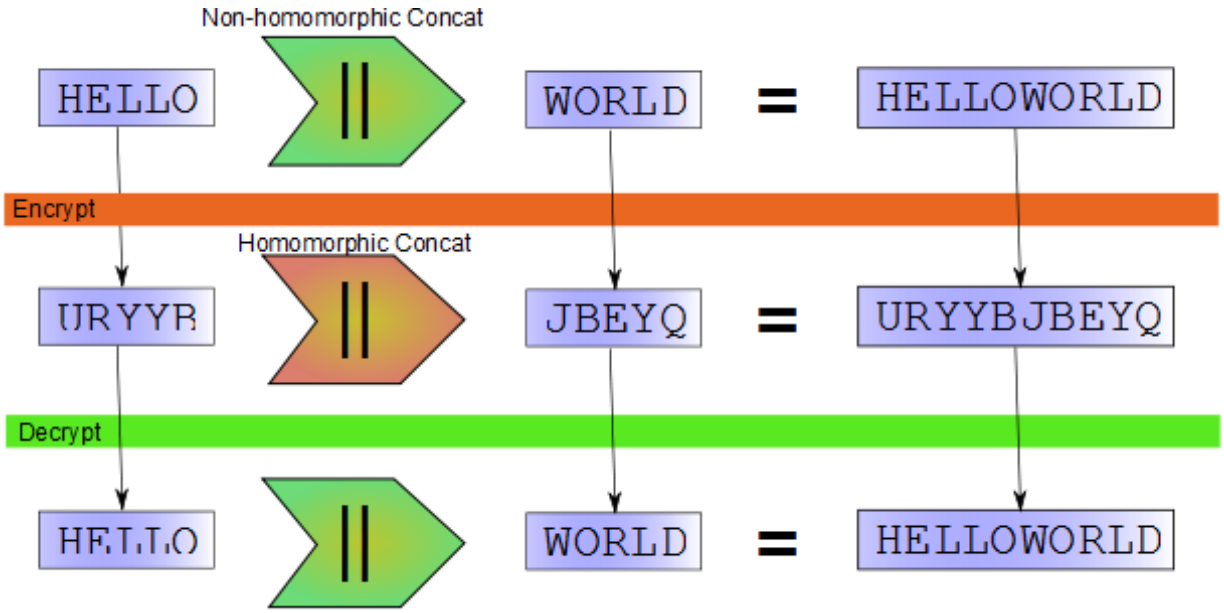

Figure 3: Homomorphic Concatenation [5]

Ignoring the obvious security concern, the Caesar cipher is a very good example of a *partially homomorphic encryption* scheme. Actually, many of the popular encryption schemes are partially homomorphic with respect to some specific operations. For example, RSA, the most popular cryptosystem in the world developed by Ron Rivest, Adi Shamir, and Leonard Adleman in 1977 [10], is homomorphic with respect to multiplication. Let's recall the RSA scheme. Given a public key (n, e) where n and e are integers, the encryption of a message m is very simple. The ciphertext c is defined as

$$c \equiv m^e \pmod{n} .$$

Observe that given two messages m_1 and m_2, we can generate the ciphertext of their product by multiplying their ciphertexts together:

$$(m_1)^e \cdot (m_2)^e = (m_1 \cdot m_2)^e .$$

Therefore, RSA is homomorphic with respect to multiplication.

Up to this point, one might question that it seems homomorphic encryption is very easy to realize. If we are satisfied with certain encryption scheme with some specific operations (i.e. partially homomorphic encryption), then it is true that there is not much work for us to do. However, it is obvious that partially homomorphic encryption schemes are far behind what is needed in cloud computing, because the reason we perform computations from the cloud is that those computations tend to be very heavy and complex. Therefore, we need to develop a homomorphic encryption scheme with universal homomorphic operations, which means we need to be able to perform any computations we want on the ciphertexts and can still decrypt the result correctly. Making a partially homomorphic encryption scheme fully homomorphic or building a brand new fully homomorphic encryption scheme is very hard.

The goal of this project is to present a fully homomorphic encryption scheme developed by Adriana López-Alt, Eran Tromer and Vinod Vaikuntanathan [6] in a single-key version that is more accessible to non-experts and implement the scheme in Sage, an open-source mathematics software system based on Python. For simplicity, we denote this encryption scheme in this paper as the LTV Scheme where "LTV" is the abbreviation of the authors' names.

2 NTRU Background

In this section, we will look at NTRU encryption scheme, a public-key cryptosystem that uses ring-based cryptography. This section includes a description of how NTRU works and a discussion of how it is related to our problem.

2.1 Introduction to NTRU Scheme

The NTRU cryptosystem was developed by Jeffery Hoffstein, Jill Pipher, and Joseph H. Silverman in 1998 [8]. The most widely used public-key system at the time was RSA, which is based on the difficulty of factoring large numbers and was created by Rivest, Shamir and Adleman in 1977. Since we can factor large integers with Shor's Algorithm in polynomial time with a quantum computer, people have been concerned about the fact that RSA cryptosystem is quantum-machine vulnerable. The NTRU cryptosystem has many advantages over RSA including that it cannot be broken by quantum machines and it runs much faster. Table 1 compares the running time of different operations in RSA and NTRU on a virtual machine.

The NTRU scheme maps an N-bit binary message to a polynomial of degree $N-1$. The mapping is straightforward. For example, we will map the 8-bit message 00101101 to $x^5+x^3+x^2+1$. We will perform our operations on such polynomials, but using a special polynomial

Operation	NTRU (ms)	RSA (ms)
Key Generation	9,617	2,090,509
Encryption	515	1505
Decryption	1,132	35,102

Table 1: RSA-NTRU comparison [9]

ring. For better security, we must allow the coefficients of the polynomials to become very large after encryption. Therefore, each bit of data can be expanded to a large number of bits, which makes the space complexity of the NTRU scheme very inefficient. We can solve this problem with NTRU PKCS [8], but we will not discuss it further in this report since it is not our focus.

2.2 Description of NTRU algorithm

In this section, we will outline how NTRU works, including key generation, encryption and decryption. We will also study how decryption works with the manipulation of two different moduli.

2.2.1 Ring Definition

NTRU works over the ring $R = \mathbb{Z}[x]/\langle\phi(x)\rangle$ where $\phi(x) = x^n - 1$ for some positive integer n. We need two integers p and q which are co-prime to each other, and it is required that $p \ll q$. Note that p and q are not necessarily primes. The multiplication in this ring is defined as a cyclic convolution product. Suppose

$$f = \sum_{i=0}^{n-1} f_i x^i \in R \qquad \text{and} \qquad g = \sum_{i=0}^{n-1} g_i x^i \in R \ ,$$

then we have the product

$$h = f \cdot g = \sum_{k=0}^{n-1} h_k x^k \in R$$

where

$$h_k = \sum_{i+j \equiv k \pmod n} f_i g_j \ .$$

This is the same as multiplying f and g in $\mathbb{Z}[x]$, and then reduce the result modulo $\phi(x)$. For example, let

$$\phi(x) = x^4 - 1$$

and let

$$f = 2x^2 + 2x + 2 \qquad g = 2x^3 + x^2 \ .$$

Then we have

$$\begin{aligned} h &= f \cdot g \\ &= (2x^2 + 2x + 2)(2x^3 + x^2) \pmod{\phi(x)} \\ &= 4x^5 + 6x^4 + 6x^3 + 2x^2 \pmod{\phi(x)} \\ &= 6x^3 + 2x^2 + 4x + 6 \ . \end{aligned}$$

Also note that when we reduce a polynomial modulo q, we mean to reduce every coefficient of this polynomial modulo q. For example, for f and g as defined earlier, we have

$$\begin{aligned} h &= f \cdot g \pmod 5 \\ &= 6x^3 + 2x^2 + 4x + 6 \pmod 5 \\ &= x^3 + 2x^2 + 4x + 1 \ . \end{aligned}$$

Finally, let's define two kinds of norms used in this report.

Definition 2.1 (Infinity Norm)**.** *For any polynomial $f = \sum_{i=0}^{n} c_i x^i$, the infinity norm is defined as the maximum of the absolute values of its coefficients:*

$$\|f\|_\infty = \max\{|c_0|, |c_1|, \ldots, |c_n|\} \ .$$

Definition 2.2 (One Norm)**.** *For any polynomial $f = \sum_{i=0}^{n} c_i x^i$, the one norm is defined as the sum of the absolute values of its coefficients:*

$$\|f\| = \|f\|_1 = \sum_{i=0}^{n} |c_i| \ .$$

2.2.2 Key Generation

Let f and g be two random polynomials in the ring R. We need to make sure that f has inverses both modulo q and modulo p. We will denote these inverses by f_q and f_p respectively. Then we have

$$f_q \cdot f \equiv 1 \pmod q \qquad \text{and} \qquad f_p \cdot f \equiv 1 \pmod p \ .$$

Hoffstein, Phpher and Silverman state that for suitable parameter choices, these inverses of f exist for most choices of f [8]. We can find inverses using the Extended Euclidean Algorithm. For a review of the Extended Euclidean Algorithm, please refer to Section 5.1.2. Then we define our public key as

$$pk := h = f_q \cdot g \pmod q$$

and our secret key as

$$sk := f \ .$$

2.2.3 Encryption

Given a message $m \in R$ where every coefficient of m is either 0 or 1 representing an n-bit message, we generate a random polynomial s, and then compute

$$c = ps \cdot h + m \pmod{q}$$

where p is the small prime chosen above. We output c as the ciphertext.

2.2.4 Decryption

To decrypt a ciphertext c with a private key f, we first compute

$$\mu = f \cdot c \pmod{q}$$

where we reduce the coefficients of μ to lie in the range $(-\frac{q}{2}, \frac{q}{2}]$. Then we compute

$$m' = f_p \cdot \mu \pmod{p}$$

as our decrypted message.

2.2.5 Why Decryption Works

The polynomial μ satisfies

$$\begin{aligned}
\mu &= f \cdot c \pmod{q} \\
&= f \cdot (ps \cdot h + m) \pmod{q} \\
&= pf \cdot s \cdot h + f \cdot m \pmod{q} \\
&= pf \cdot s \cdot (f_q \cdot g) + f \cdot m \pmod{q} \\
&= pg \cdot s \cdot (f_q \cdot f) + f \cdot m \pmod{q} \\
&= pg \cdot s \cdot 1 + f \cdot m \pmod{q} \\
&= pg \cdot s + f \cdot m \pmod{q} .
\end{aligned}$$

If we can guarantee that every coefficient of the polynomial $pg \cdot s + f \cdot m$ lies within $(-\frac{q}{2}, \frac{q}{2}]$, then it is equal to its image modulo q. It is possible to control the size of $\|pg \cdot s + f \cdot m\|_\infty$ because every term in this expression is controllable, in that it is composed of terms that we sample from a well-chosen distributions. By contrast, f_p and f_q are not controllable because they are computed rather than sampled. Therefore, we have

$$\mu = pg \cdot s + f \cdot m \in R .$$

Reducing this polynomial modulo p will get rid of $(pg \cdot s)$ and leave us with

$$\mu \bmod p = f \cdot m \pmod{p} .$$

Multiplying it by f_p gives us the original message m because

$$f_p \cdot f \equiv 1 \pmod{p} .$$

Keep in mind that we are assuming every coefficient of the polynomial $pg \cdot s + f \cdot m$ lies within $[-\frac{q}{2}, \frac{q}{2})$. If this condition does not hold, we are having what we call a *wrap-arround error* which leads to failure in decryption. Since this project is not focused on NTRU cryptosystem, we will not discuss wrap-arround error in detail here. The problem of wrap-around error will appear again later in this paper in the LTV scheme, and we will analyze it in detail then.

2.3 Homomorphic Properties

Now let's look at the homomorphic properties that exist in the NTRU scheme. Is NTRU scheme homomorphic with respect to any operation? Suppose we have two ciphertexts c_1 and c_2 as encryptions of the original messages m_1 and m_2 with the same private key f. Let n, R, p and q be as defined earlier. Let's consider two operations here: addition and multiplication.

2.3.1 Addition

First, we add the two ciphertexts:

$$c_{add} = c_1 + c_2 .$$

Then we try to decrypt this sum by computing

$$\begin{aligned} \mu &= f \cdot c_{add} \pmod{q} \\ &= f \cdot (c_1 + c_2) \pmod{q} \\ &= f \cdot c_1 + f \cdot c_2 \pmod{q} \\ &= ps_1 \cdot g + f \cdot m_1 + ps_2 \cdot g + f \cdot m_2 \pmod{q} . \end{aligned}$$

With appropriate parameter choices, we can make sure that

$$\|ps_1 \cdot g + f \cdot m_1 + ps_2 \cdot g + f \cdot m_2\|_\infty \leq \frac{q}{2} ,$$

which means we recover exactly the same polynomial after reduction modulo q. Then we can compute

$$\mu \bmod p = f \cdot m_1 + f \cdot m_2 \pmod{p} .$$

Multiplying it by f_p gives us

$$\begin{aligned} m_{add} &= f_p \cdot f \cdot m_1 + f_p \cdot f \cdot m_2 \pmod{p} \\ &= m_1 + m_2 \ . \end{aligned}$$

Thus, we know that NTRU scheme is additively homomorphic assuming that the parameter choices are appropriate, i.e. without wrap-around error. Considering that each message represents a binary sequence, this operation gives us a bitwise XOR if we reduce the result modulo 2. However, the hardship to avoid wrap-around error grows linearly; thus, it is impossible for us to guarantee decryption works after an arbitrary number of additions. Such a scheme is denoted as *somewhat homomorphic*.

2.3.2 Multiplication

First, we multiply the two ciphertexts:

$$c_{mult} = c_1 \cdot c_2 \ .$$

Then we expand c_{mult} as

$$\begin{aligned} c_{mult} &= c_1 \cdot c_2 \\ &= (ps_1 \cdot h + m_1) \cdot (ps_2 \cdot h + m_2) \\ &= (ps_1 \cdot f_q \cdot g + m_1) \cdot (ps_2 \cdot f_q \cdot g + m_2) \ . \end{aligned}$$

In order to eliminate the uncontrollable term f_q, we need to multiply it by f twice, which gives us

$$\begin{aligned} f^2 \cdot c_{mult} &= (ps_1 \cdot g + f \cdot m_1) \cdot (ps_2 \cdot g + f \cdot m_2) \pmod{q} \\ &= p^2 s_1 \cdot s_2 \cdot g^2 + ps_1 \cdot g \cdot f \cdot m_2 + ps_2 \cdot g \cdot f \cdot m_1 + f^2 \cdot m_1 \cdot m_2 \end{aligned}$$

Again, if we can avoid wrap-around error, then we can decrypt the result by first reducing it modulo p and then multiplying it by f_p^2. However, this homomorphic multiplication (cyclic convolution product) does not mean anything to us since the operands are encryptions of binary sequences. Also, notice that the hardship to avoid wrap-around error grows exponentially, and we have to keep track of how many multiplications by f and f_p we need to perform.

Besides these disadvantages, consider the case when we need to decrypt a combination of addition and multiplication of the ciphertexts. For example, if we want to decrypt $c_1 + c_2 \cdot c_3$, multiplying it by either f or f^2 will give us a wrong result. Therefore, NTRU is far from a fully homomorphic encryption system but is close to a somewhat homomorphic encryption system. What can we do to improve it?

3 Somewhat Homomorphic Encryption Based on NTRU

In this section, we will study the LTV cryptosystem as a somewhat homomorphic encryption scheme based on NTRU and discuss the homomorphic properties of two operations in this scheme: addition and multiplication. We will outline a transition from a somewhat homomorphic scheme to a fully homomorphic one in Section 4.

3.1 The Scheme

This section introduces the basic operations of a somewhat homomorphic version of the LTV scheme including key generation, encryption and decryption. We will also discuss wrap-around error in detail and how this version of the scheme is somewhat homomorphic.

3.1.1 Key Generation

Let $\phi(x) = x^n + 1$ for some $n \in \mathbb{N}$ and q be a prime number. Let $R = \mathbb{Z}[x]/\langle\phi(x)\rangle$ and $R_q = R/qR$, the quotient ring obtained by reducing modulo the ideal qR of all polynomials whose coefficients are multiples of q. Converting a polynomial in R to a polynomial in R_q works in the same way as reducing every coefficient of the polynomial modulo q which is shown in the discussion of NTRU. Note that $R_q \equiv \mathbb{Z}_q[x]/\langle\phi(x)\rangle$, where ϕ is now viewed as an element of $\mathbb{Z}_q[x]$.

Definition 3.1 (B-Bounded Polynomial [6])**.** *A polynomial $f \in R$ is called B-Bounded if $\|f\|_\infty \leq B$.*

Let $B \in \mathbb{Z}$ and let f', g be two B-bounded polynomials in R_q. Set $f = 2f' + 1$ so that

$$f \equiv 1 \pmod 2 \ .$$

If f is not invertible in R_q, resample f'. Then we can define our public key as

$$pk := h = 2gf^{-1} \in R_q$$

and our secret key as

$$sk := f \in R_q \ .$$

Of course, if f^{-1} is made public, an attacker can find f. The designers use g as a mask and multiply by two to make this mask "removable" even though h will not typically have all even coefficients.

3.1.2 Encryption

We encrypt the message bit by bit in the LTV scheme unlike encrypting a string of bits all together in the NTRU scheme. Given a single-bit message $m \in \{0, 1\}$. Let s, e be arbitrary B-bounded polynomials in R_q. We generate our ciphertext as

$$c := hs + 2e + m \in R_q$$

where h is the public key.

3.1.3 Decryption

Given a ciphertext c and a secret key f, we can decrypt the ciphertext by computing

$$\mu := fc \in R_q \ .$$

We then compute

$$m' := \mu \pmod 2 \ .$$

Finally, we output m' as the decrypted message.

3.2 Proof of Correctness

In this section, we will discuss why decryption works in the LTV scheme as well as details about wrap-around error, including how wrap-around error occurs and how to avoid it.

3.2.1 Observation

First, let's define a new variable

$$\bar{\mu} := 2gs + 2fe + fm \in R \ .$$

Then we can easily show

$$\begin{aligned} \bar{\mu} \bmod 2 &= fm \bmod 2 \\ &= 2f'm + m \bmod 2 \\ &= m \ . \end{aligned}$$

If we can prove that $\bar{\mu} = \mu \pmod q$ is always true, then if $\|\bar{\mu}\|_\infty \leq \frac{q}{2}$, we can always shift the coefficients of μ into the interval $(-\frac{q}{2}, \frac{q}{2})$ to get $\bar{\mu}$. Thus, it is guaranteed that we can recover m from $\bar{\mu}$. In fact, if we can make sure all coefficients of $\bar{\mu}$ stay in $[a,\ a + q)$ for some $a \in \mathbb{Z}$, we can recover $\bar{\mu}$ by shifting the coefficients of μ to that interval. Otherwise, we cannot be sure to decrypt the correct message, because we cannot guarantee the result from shifting the coefficients of μ is unique. We denote this as a *wrap-around error*.

3.2.2 Relations in the Rings

Lemma 3.2 (Order of Reductions)**.** *Given $\phi(x) = x^n + 1$ for some $n \in \mathbb{N}$, and some prime q, for any $f \in \mathbb{Z}[x]$, we have*

$$f \bmod \phi(x) \equiv (f \bmod q) \bmod \phi(x) \pmod q$$

Proof. Let

$$f := c_0 + c_1 x + c_2 x^2 + \ldots + c_n x^n + \ldots$$

be an arbitrary element in $\mathbb{Z}[x]$. Then let

$$f_q := f \bmod q = r_0 + r_1 x + r_2 x^2 + \ldots + r_n x^n + \ldots$$

where for all $i \in \{0\} \cup \mathbb{N}$, $r_i \equiv c_i \pmod q$.

Next, define

$$\begin{aligned}\tilde{f} := f \bmod \phi(x) = (c_0 - c_n + c_{2n} - c_{3n} + \ldots) + \\ (c_1 - c_{n+1} + c_{2n+1} - c_{3n+1} + \ldots)x + \\ (c_2 - c_{n+2} + c_{2n+2} - c_{3n+2} + \ldots)x^2 + \\ \ldots \\ (c_{n-1} - c_{2n-1} + c_{3n-1} - c_{4n-1} + \ldots)x^{n-1} \ ,\end{aligned}$$

and define

$$\begin{aligned}\tilde{f}_q := f_q \bmod \phi(x) = (r_0 - r_n + r_{2n} - r_{3n} + \ldots) + \\ (r_1 - r_{n+1} + r_{2n+1} - r_{3n+1} + \ldots)x + \\ (r_2 - r_{n+2} + r_{2n+2} - r_{3n+2} + \ldots)x^2 + \\ \ldots \\ (r_{n-1} - r_{2n-1} + r_{3n-1} - r_{4n-1} + \ldots)x^{n-1} \ .\end{aligned}$$

We need to show that

$$\tilde{f} \equiv \tilde{f}_q \pmod q \ .$$

We know that for all $i \in \{0\} \cup \mathbb{N}$,

$$r_i \equiv c_i \pmod q \ .$$

Since every coefficient of x^i in $\tilde{f}$ is a linear combination of $c_1, c_2, c_3 \ldots$, and every coefficient of x^i in $\tilde{f}_q$ is a linear combination of $r_1, r_2, r_3 \ldots$ with the same operations, we have

$$\tilde{f} \equiv \tilde{f}_q \pmod q \ .$$

□

With Lemma 3.2, it is not hard to see it is always true that $\bar{\mu} = \mu \pmod q$, which guarantees a successful decryption if there is no wrap-around error.

3.2.3 Variable Range

Now let's discuss how to choose variables to guarantee that there is no wrap-around error.

Lemma 3.3 (Product Bound [6]). *Let $n \in \mathbb{N}$, let $\phi(x) = x^n + 1$, and let $R = \mathbb{Z}[x]/\langle\phi(x)\rangle$. For any $s, t \in R$,*

$$\begin{aligned} \|s \cdot t \bmod \phi(x)\| &\leq \sqrt{n} \cdot \|s\| \cdot \|t\| \\ \|s \cdot t \bmod \phi(x)\|_\infty &\leq n \cdot \|s\|_\infty \cdot \|t\|_\infty \end{aligned}$$

Let every symbol be as defined in the basic scheme. Lemma 3.3 gives us

$$\begin{aligned} \|fe\|_\infty &\leq n(2B+1)B \ , \\ \|gs\|_\infty &\leq nB^2 \ , \\ \|\bar{\mu}\|_\infty &\leq 2n(2B+1)B + 2nB^2 + 2B + 1 \\ &= 6nB^2 + 2nB + 2B + 1 \ . \end{aligned}$$

Below are the situations with some specific values of B.

For $B = 1$,

$$\begin{aligned} \|\bar{\mu}\|_\infty &\leq 6n + 2n + 2 + 1 \\ &\leq 8n + 3 \end{aligned}$$

For $B = 2$,

$$\begin{aligned} \|\bar{\mu}\|_\infty &\leq 24n + 4n + 4 + 1 \\ &\leq 28n + 5 \end{aligned}$$

For B very large, ($B \geq 5$)

$$\begin{aligned} nB^2 &\geq 5nB \\ &\geq 2nB + 2B + 1 \\ \|\bar{\mu}\|_\infty &\leq 6nB^2 + nB^2 \\ &= 7nB^2 \end{aligned}$$

We need to choose the prime q large enough to guarantee that

$$\frac{q}{2} > \|\bar{\mu}\|_\infty \ .$$

From now on, we will denote $\|\bar{\mu}\|_\infty$ as the *wrap-around error term* since it is the value we need to control.

3.2.4 Simple Wrap-around Example

To more clearly explain how wrap-around error occurs, let's look at a simple example.

Choose variables

Let $n = 4$ such that $\phi(x) = x^4 + 1$.
Let $q = 7,\ B = 1$. (Notice that q is smaller than $16n + 6 = 70$, the minimum value of q that guarantees no wrap-around.)
Let $f' = x^2 - 1$ such that $f = 2f' + 1 = 2x^2 - 1$.
Let $g = x^2 + x$.

Key generation

First, let's find f^{-1} using Extended Euclidean Algorithm:

$$\begin{aligned}[x^4 + 1] &= [2x^2 - 1](4x^2 + 2) + 3 \\ [2x^2 + 1] &= [3](3x^2 + 2) + 0\end{aligned}$$

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

Therefore,

$$3 = [x^4 + 1] - (4x^2 + 2)[2x^2 - 1]\ .$$

Since

$$1 \equiv 3 \cdot 5 \pmod 7\ ,$$

we have

$$1 = 5[x^4 + 1] - (20x^2 + 10)[2x^2 - 1]\ .$$

Hence,

$$\begin{aligned}f^{-1} &= -20x^2 - 10 \\ &= x^2 - 3\ .\end{aligned}$$

Then let's find the public key by computing

$$\begin{aligned}h &= 2f^{-1}g \\ &= 2(x^2 - 3)(x^2 + x) \\ &= 2x^3 + x^2 + x - 2\ .\end{aligned}$$

The secret key is

$$f = 2x^2 - 1\ .$$

Encryption

Suppose $s = x^3 + x + 1$, $e = x^3$, $m = 1$. We can generate the ciphertext as

$$c = hs + 2e + m \ ,$$

where

$$\begin{aligned} hs &= (2x^3 + x^2 + x - 2)(x^3 + x + 1) \\ &= 2x^6 + x^5 + 3x^4 + x^3 + 2x^2 - x - 2 \\ &= x^3 - 2x + 2 \ , \end{aligned}$$

and

$$2e = 2x^3 \ .$$

Thus,

$$c = 3x^3 - 2x + 3$$

Decryption

Now we try to decrypt by computing

$$\begin{aligned} \mu = fc &= (2x^2 - 1)(3x^3 - 2x + 3) \\ &= 6x^5 + 7x^3 + 6x^2 + 2x - 3 \\ &= -x^2 + 3x - 3 \ . \end{aligned}$$

We then have

$$\mu \bmod 2 = x^2 + x + 1 \neq m \ .$$

Analysis

Let

$$\bar{\mu} = 2gs + 2fe + fm \in R \ ,$$

then we can easily show

$$\bar{\mu} \bmod 2 = m \ .$$

In this case,

$$\begin{aligned} gs &= (x^2 + x)(x^3 + x + 1) \\ &= x^5 + x^4 + x^3 + 2x^2 + x \\ &= x^3 + 2x^2 - 1 \ , \end{aligned}$$

$$\begin{aligned} fe &= (2x^2 - 1)x^3 \\ &= 2x^5 - x^3 \\ &= -x^3 - 2x \ . \end{aligned}$$

Thus,

$$\begin{aligned} \bar{\mu} &= 2x^3 + 4x^2 - 2 + 2x^3 - 4x + 2x^2 - 1 \\ &= 6x^2 - 4x - 3 \end{aligned}$$

and

$$\begin{aligned} \bar{\mu} \bmod 2 &= 1 \\ &= m \ . \end{aligned}$$

Notice that

$$\bar{\mu} = 6x^2 - 4x - 3 \qquad \text{and} \qquad \mu = -x^2 + 3x - 3 \ ,$$

so

$$\bar{\mu} \equiv \mu \pmod 7 \ .$$

We cannot recover $\bar{\mu}$ from μ because a wrap-around error has occurred as both $6x^2$ and $-4x$ in $\bar{\mu}$ are out of bound.

3.3 Homomorphic Properties

Now let's see why this scheme is somewhat homomorphic. Let c_1 and c_2 be two ciphertexts, being encryptions of messages m_1 and m_2 with the same secret key f.

3.3.1 Addition

Define the sum of two ciphertexts as

$$c_{add} = c_1 + c_2 \ .$$

Then, as long as there is no wrap-around error, we have

$$\begin{aligned} f \cdot c_{add} \bmod 2 &= f \cdot (c_1 + c_2) \pmod 2 \\ &= f \cdot c_1 + f \cdot c_2 \pmod 2 \\ &= \mu_1 + \mu_2 \pmod 2 \\ &= m_1 + m_2 \pmod 2 \ . \end{aligned}$$

As we have seen above, we are able to decrypt the sum of the two ciphertexts to recover the sum of the two messages. Note that the modulo 2 sum of the two bits is the XOR gate in circuit design.

3.3.2 Multiplication

Define the product of two ciphertexts as

$$c_{mult} = c_1 \cdot c_2 \ .$$

Again, as long as there is no wrap-around error, we have

$$\begin{aligned} f^2 \cdot c_{mult} \bmod 2 &= f^2 \cdot (c_1 \cdot c_2) \pmod 2 \\ &= (f \cdot c_1) \cdot (f \cdot c_2) \pmod 2 \\ &= \mu_1 \cdot \mu_2 \pmod 2 \\ &= m_1 \cdot m_2 \pmod 2 \\ &= m_1 \cdot m_2 \ . \end{aligned}$$

As we have seen above, we are able to decrypt the product of the two ciphertext to recover the product of the two messages. Note that the product of the two bits is the AND gate in circuit design.

3.3.3 Combination of Addition and Multiplication

Recall that in the NTRU scheme, we are not able to decrypt the ciphertext after a combination of addition and multiplication. However, in the LTV scheme, since

$$f \equiv 1 \pmod 2 \ ,$$

we know that as long as there is no wrap-around error, it is true that

$$f^k m \bmod 2 = m$$

where f is the secret key, m is the message and k is any positive integer. This gives us the power to decrypt a combination of additions and multiplications via multiplying the result ciphertext by f^k where f is the secret key and k is the depth of the longest chain of multiplications. This is one of the most significant improvements from the NTRU scheme to the LTV scheme, which makes the LTV scheme somewhat homomorphic.

3.3.4 Wrap-around Error Term Growth

First, define $\bar{\mu}_1$ and $\bar{\mu}_2$ as

$$\begin{aligned} \bar{\mu}_1 &= 2gs_1 + 2fe_1 + fm_1 \\ \bar{\mu}_2 &= 2gs_2 + 2fe_2 + fm_2 \ . \end{aligned}$$

For addition, we need to control the growth of $\|\bar{\mu}_1 + \bar{\mu}_2\|_\infty$ in order to recover $\bar{\mu}_1 + \bar{\mu}_2$ from $\mu_1 + \mu_2$. Therefore, the wrap-around error term grows linearly.

For multiplication, we need to control the growth of $\|\bar{\mu}_1 \cdot \bar{\mu}_2\|_\infty$ in order to recover $\bar{\mu}_1 \cdot \bar{\mu}_2$ from $\mu_1 \cdot \mu_2$. Therefore, the wrap-around error term grows exponentially.

4 From Somewhat to Fully Homomorphic Encryption

In this section, we will talk about how to convert this somewhat homomorphic LTV scheme to a fully homomorphic one. We will apply Gentry's bootstrapping Theorem [6], but in order to do this, we must first apply two techniques: *Relinearization* and *Modulus Reduction*.

4.1 Relinearization

In the somewhat homomorphic LTV scheme as well as in the earlier NTRU scheme, we notice that in order to decrypt a product of k ciphertexts, we need to multiply the product ciphertext by the secret key k times. This requires us to keep track of the depth of the circuit we are evaluating, which tends to be very hard if the circuit is complex. Relinearization enables us to decrypt any combination of additions and multiplications of the ciphertexts by multiplying the resulting ciphertext by the secret key for one single time. We can achieve this by introducing an evaluation key.

4.1.1 Key Generation

Let $\phi(x) = x^n + 1$ for some $n \in \mathbb{N}$ and q be a prime number. Let $R = \mathbb{Z}[x]/\langle\phi(x)\rangle$ and $R_q = R/qR$. Let $B \in \mathbb{Z}^+$ and let f', g be two B-bounded polynomials in R_q. Set $f = 2f' + 1$. If f is not invertible in R_q, resample f'. Then we can define our public key as

$$pk := h = 2gf^{-1} \in R_q \ .$$

Our secret key is defined as

$$sk := f \in R_q \ .$$

Let $\ell = \lfloor \log q \rfloor$. (Notice that this is a base-2 logarithm.) For all $\tau \in \{0, \ldots, \ell\}$, let s_τ, e_τ be arbitrary B-bounded polynomials in R_q, and compute

$$\gamma_\tau := hs_\tau + 2e_\tau + 2^\tau f \in R_q \ .$$

This can be viewed as an encryption of f, shifted by τ bits. Our evaluation key is

$$ek := (\gamma_0, \ldots, \gamma_\ell) \in R_q^{\ell+1} \ .$$

4.1.2 Encryption

To encrypt a single-bit message $m \in \{0, 1\}$, we sample arbitrary B-bounded polynomials s, e from R_q. We then generate our ciphertext as

$$c := hs + 2e + m \in R_q \ .$$

4.1.3 Decryption

Given a ciphertext c and a secret key f, we can decrypt the ciphertext by computing

$$\mu := fc \in R_q \ .$$

We then compute

$$m' := \mu \pmod 2 \ .$$

Finally, we output m' as the decrypted message.

4.1.4 Evaluation

We can add two ciphertexts normally because we can always decrypt the sum of two ciphertext by multiplying the secret key only once. See Section 3.3.1.

When we multiply two ciphertexts c_1 and c_2, let

$$c := c_1 \cdot c_2 \in R_q \ .$$

Then find the binary representation of c by generating $\tilde{c}_\tau$ such that

$$c = \sum_{\tau=0}^{\ell} \tilde{c}_\tau 2^\tau \ .$$

Finally, output ciphertext as

$$c_{mult} = \sum_{\tau=0}^{\ell} \tilde{c}_\tau \gamma_\tau$$

where γ_τ is from the evaluation key.

4.1.5 Binary Representation Example

To avoid confusion, let's look at a concrete example that shows how to find the binary representation of a ciphertext c.

Suppose $c = 5x^3 - 3x^2 + 4$ in the ring $R = \mathbb{Z}_{11}[x]/\langle x^4 + 1\rangle$. Then we know that

$$\ell = \lfloor \log 11 \rfloor = 3 \ .$$

Next, we need to convert every coefficient to its binary form as shown in Table 2.

Decimal Form	Binary Form
5	0101
8	1000
0	0000
4	0100

Table 2: Coefficients of c in Binary

Notice that here we convert negative coefficients to their corresponding positive values, e.g. $-3 \equiv 8 \pmod{11}$. The reason for this conversion is that computers process integers in their binary form. For positive integers, we are able to use their binary forms directly. However, we cannot use the binary forms of negative integers because most modern computers use two's complement to represent signed integers, which does not work mathematically here.

For each $\tau \in \{0, \ldots, \ell\}$, we generate a polynomial $\tilde{c}_\tau$ based on the bit at position τ of the binaries, where 0 corresponds to the least significant bit (LSB) and ℓ corresponds to the most significant bit (MSB). In our example here, for $\tau = 0$, $\tilde{c}_0 = x^3$ because the coefficient of x^3 has the LSB equal to 1, while the other coefficients have the LSB equal to 0. Below is the complete binary representation of our polynomial c.

$$\begin{aligned}
\tilde{c}_0 &= x^3 \\
\tilde{c}_1 &= 0 \\
\tilde{c}_2 &= x^3 + 1 \\
\tilde{c}_3 &= x^2
\end{aligned}$$

It easy to show that

$$c = \sum_{\tau=0}^{3} \tilde{c}_\tau 2^\tau \ .$$

4.1.6 How does it work

Recall that the purpose of relinearization is to decrypt a product ciphertext via multiplying it by the secret key only once.

Suppose c_{mult} is the ciphertext that we generate with evaluation process. Then

$$c_{mult} = \sum_{\tau=0}^{\ell} \tilde{c}_\tau \gamma_\tau$$

and

$$c = c_1 c_2 = \sum_{\tau=0}^{\ell} \tilde{c}_\tau 2^\tau \ .$$

Since $\gamma_\tau = hs_\tau + 2e_\tau + 2^\tau f$, we have

$$\begin{aligned} c_{mult} &= \sum_{\tau=0}^{\ell} \tilde{c_\tau}(hs_\tau + 2e_\tau + 2^\tau f) \\ &= h\sum_{\tau=0}^{\ell} \tilde{c_\tau} s_\tau + 2\sum_{\tau=0}^{\ell} \tilde{c_\tau} e_\tau + f\sum_{\tau=0}^{\ell} \tilde{c_\tau} 2^\tau \\ &= h\sum_{\tau=0}^{\ell} \tilde{c_\tau} s_\tau + 2\sum_{\tau=0}^{\ell} \tilde{c_\tau} e_\tau + fc \\ &= hs' + 2e' + fc_0 \end{aligned}$$

for the obvious choice of s' and e'.

Therefore, when we decrypt, we have

$$\begin{aligned} fc_{mult} &= fhs' + 2fe' + f^2 c \\ &= 2gs' + 2fe' + (fc_1)(fc_2) \\ &= m_1 m_2 \pmod 2 \ . \end{aligned}$$

Thus, as long as we can control the wrap-around error, we can decrypt the ciphertext with only one multiplication by the secret key.

4.1.7 Error Growth

Although relinearization significantly simplifies our decryption process, we are able to find that it however adds to the wrap-around error term. We have from above

$$\begin{aligned} fc_{mult} &= 2gs' + 2fe' + (fc_1)(fc_2) \\ &= 2g\sum_{\tau=0}^{\ell} \tilde{c_\tau} s_\tau + 2f\sum_{\tau=0}^{\ell} \tilde{c_\tau} e_\tau + (fc_1)(fc_2) \ . \end{aligned}$$

From Lemma 3.3, we know that

$$\begin{aligned}\|\tilde{c_\tau} s_\tau\|_\infty &\leq nB \\ \|\tilde{c_\tau} e_\tau\|_\infty &\leq nB \ .\end{aligned}$$

Thus, we have

$$\begin{aligned}\|2g\sum_{\tau=0}^{\ell}\tilde{c_\tau}s_\tau + 2f\sum_{\tau=0}^{\ell}\tilde{c_\tau}e_\tau\|_\infty &\leq 2nB(\ell+1)nB + 2n(2B+1)(\ell+1)nB \\ &= 6n^2B^2(\ell+1) + 2n^2B(\ell+1) \\ &\leq 8n^2B^2(\ell+1) \ .\end{aligned}$$

Therefore, we have shown that relinearization adds $8n^2B^2(\ell+1)$ to the original wrap-around error term. Assume B = 1, then relinearization adds $8n^2\ell + 8n^2$ to the original wrap-around error term which is $17n^3$.

4.2 Modulus Reduction

Modulus reduction is a noise-management technique which provides an exponential gain on the depth of the circuit that can be evaluated. It allows us to keep the wrap-around error term small by scaling the ciphertext after each operation. This section includes why modulus reduction benefits us, how modulus reduction changes the scheme and why modulus reduction works mathematically.

4.2.1 The Idea of Modulus Reduction

The process of modulus reduction is surprisingly simple. Let q_i and $q_{i+1} < q_i$ be primes and let $c(x)$ be a polynomial with small coefficients. We simply transform $c(x)$ to a polynomial $c'(x)$ which is a rounded version of $\frac{q_{i+1}}{q_i}c(x)$ whose coefficients have the same parity as those of $c(x)$. So

$$c'(x) \in R_{q_{i+1}} \qquad \text{when} \qquad c(x) \in R_{q_i} \ .$$

After this operation, the wrap-around error term of the ciphertext is scaled by the same factor, $\frac{q_j}{q_i}$. How can this benefit us?

For example, suppose we have a 100-bit prime number q_0 as the first modulus and a 90-bit prime number q_1 as the second modulus. That is

$$q_0 \approx 2^{100} \qquad \text{and} \qquad q_1 \approx 2^{90}$$

Suppose the wrap-around error term we have is 15-bit, then after modulus reduction, since we have decreased the moduli by a factor of 2^{10}, we also decrease the wrap-around error

term by the same factor, which means we are left with a roughly 5-bit wrap-around error term. Denote the wrap-around error term by e_0 and e_1. We have

$$e_0 \approx 2^{15} \qquad \text{and} \qquad e_1 \approx 2^5$$

Suppose we need to make sure that the wrap-around error term is at most a quarter of the modulus. If we have 7 polynomials in R_{q_0} with wrap-around error terms approximately equal to 2^{15}, multiplying them altogether will give us a new wrap-around error term of the order 2^{105}, which is certainly out of bound.

However, if we multiply the 7 polynomials after modulus reduction, the new wrap-around error term is only 2^{35}, far away from our upper limit of $2^{90}/4$.

From this simple example, we can see that modulus reduction is very beneficial. Apply it repeatedly will significantly improve the maximum depth of the circuit that can be evaluated. In the following sections, let's look at the improved scheme using modulus reduction.

4.2.2 Key Generation

Let $\phi(x) = x^n + 1$ for some $n \in \mathbb{N}$ and let $R = \mathbb{Z}[x]/\langle\phi(x)\rangle$. We need to sample a ladder of decreasing moduli $q_0, q_1, \ldots, q_{d_{dec}}$ where d_{dec} is the depth of the circuit we want to evaluate. Let $B \ll q_{d_{dec}} \in \mathbb{Z}$.

For every $i \in \{0, \ldots, d_{dec}\}$, sample $g^{(i)}, u^{(i)}$ as B-Bounded polynomials and set $f^{(i)} := 2u^{(i)} + 1$. If $f^{(i)}$ is not invertible in R_{q_i}, resample $u^{(i)}$. Let $h^{(i)} := 2g^{(i)} \left(f^{(i)}\right)^{-1} \in R_{q_{i-1}}$, and set

$$pk := h^{(0)} \in R_{q_0} \qquad \text{and} \qquad sk := f^{(d_{dec})} \in R_{q_{d_{dec}}} \ .$$

For all $i \in [d_{dec}] := \{1, 2, \ldots, d_{dec}\}$, and $\tau \in \{0, \ldots, \lfloor \log q_{i-1} \rfloor\}$, sample $s_\tau^{(i)}, e_\tau^{(i)}$ as B-bounded polynomials and compute

$$\begin{aligned} \gamma_\tau^{(i)} &:= h^{(i)} s_\tau^{(i)} + 2e_\tau^{(i)} + 2^\tau f^{(i-1)} \in R_{q_{i-1}} \\ \zeta_\tau^{(i)} &:= h^{(i)} s_\tau^{(i)} + 2e_\tau^{(i)} + 2^\tau \left(f^{(i-1)}\right)^2 \in R_{q_{i-1}} \ . \end{aligned}$$

Set

$$ek := \left\{\gamma_\tau^{(i)}, \zeta_\tau^{(i)}\right\}_{i \in [d_{dec}], \tau \in \{0, \ldots, \lfloor \log q_i \rfloor\}} \ .$$

4.2.3 Encryption

Given a single bit message $m \in \{0, 1\}$. Let s, e be arbitrary B-bounded polynomials. We can generate our ciphertext as

$$c := hs + 2e + m \in R_{q_0} \ .$$

4.2.4 Decryption

Given a ciphertext $c \in R_{q_{d_{dec}}}$ and a secret key $sk = f^{(d_{dec})}$, we can decrypt the ciphertext by computing

$$\mu := f^{(d_{dec})} c \in R_q \ .$$

We then compute

$$m' := \mu \pmod 2 \ .$$

Finally, we output m' as the decrypted message.

4.2.5 Evaluation

We show how to evaluate a t-input circuit C where all inputs c_i are encrypted with the same public key/private key pair (pk, sk) having corresponding evaluation key ek, which is also publicly available. A cloud server needs not just pk but also ek to perform the computations. We assume without loss of generality that the circuit C is leveled; i.e., it is composed of alternating XOR and AND levels. We show how to homomorphically add and multiply two ciphertexts below.

Addition

Given two ciphertexts $c_1, c_2 \in R_{q_i}$, compute $c = c_1 + c_2$. For $\tau \in \{0, \ldots, \lfloor \log q_i \rfloor\}$, define $\tilde{c}_\tau$ so that

$$c = \sum_{\tau=0}^{\lfloor \log q_i \rfloor} 2^\tau \tilde{c}_\tau \ .$$

Then we define

$$\tilde{c} := \sum_{\tau=0}^{\lfloor \log q_i \rfloor} \tilde{c}_\tau \gamma_\tau^{(i)} \in R_{q_i} \ .$$

Finally, we reduce the modulus. Let c_{add} be the polynomial with integer coefficients which is closest to $\left(\frac{q_{i+1}}{q_i}\right) \tilde{c}$ subject to the condition that, coefficient-wise, $c_{add} \equiv \tilde{c} \pmod 2$. Output $c_{add} \in R_{q_{i+1}}$ as an encryption of the *sum* of the underlying messages.

Multiplication

Given two ciphertexts $c_1, c_2 \in R_{q_i}$, compute $c = c_1 \cdot c_2$. For $\tau \in \{0, \ldots, \lfloor \log q_i \rfloor\}$, define $\tilde{c}_\tau$ so that

$$c = \sum_{\tau=0}^{\lfloor \log q_i \rfloor} 2^\tau \tilde{c}_\tau \ .$$

Then we define

$$\tilde{c} := \sum_{\tau=0}^{\lfloor \log q_i \rfloor} \tilde{c}_\tau \zeta_\tau^{(i+1)} \in R_{q_i} \ .$$

Finally, we reduce the modulus. Let c_{mult} be the polynomial with integer coefficients closest to $\left(\frac{q_{i+1}}{q_i}\right)\tilde{c}$ such that $c_{mult} \equiv \tilde{c} \pmod 2$. Output $c_{mult} \in R_{q_{i+1}}$ as an encryption of the *product* of the underlying messages.

Observe: if our circuit is leveled, we only need $\gamma_\tau^{(i)}$ for i even and $\zeta_\tau^{(i)}$ for i odd.

4.2.6 Proof of Modulus Reduction

Lemma 4.1 (Modulus Reduction)**.** *Let q_0 and q_1 be two odd moduli and let $c \in \mathbb{Z}^n$. Let $c' \in \mathbb{Z}^n$ be the vector closest to $\frac{q_1}{q_0}c$ satisfying*

$$c'_i \equiv c_i \pmod 2 \qquad \textit{for all } i.$$

Then for any $f \in \mathbb{Z}^n$, with $|fc \bmod q_0| < \frac{q_0}{2} - \frac{q_0}{q_1}\|f\|_1$, we have

$$|fc' \bmod q_1| \equiv |fc \bmod q_0| \pmod 2 \ ,$$

and

$$|fc' \bmod q_1| < \frac{q_1}{q_0}|fc \bmod q_0| + \|f\|_1 \ .$$

Proof. We can write

$$fc \bmod q_0 = fc - kq_0$$

for some $k \in \mathbb{Z}$. Define $n_1 = fc' - kq_1$. We have

$$\begin{aligned} c &\equiv c' \pmod 2 \ , \\ q_o &\equiv q_1 \pmod 2 \ . \end{aligned}$$

Hence,

$$fc \bmod q_0 \equiv n_1 \pmod 2 \ .$$

Thus,

$$|fc \bmod q_0| \equiv |n_1| \pmod 2 \ .$$

We need to prove that

$$n_1 = fc' \bmod q_1 \ .$$

Since

$$fc = fc \bmod q_0 + kq_0 \ ,$$

we have

$$\frac{q_1}{q_0} fc = \frac{q_1}{q_0}(fc \bmod q_0) + kq_1 \ .$$

Therefore, we may write

$$\begin{aligned} n_1 &= fc' - kq_1 \\ &= \frac{q_1}{q_0}(fc \bmod q_0) + fc' - \frac{q_1}{q_0}(fc) \\ &= \frac{q_1}{q_0}(fc \bmod q_0) + fd \qquad (\text{for } d = c' - \frac{q_1}{q_0}c) \ . \end{aligned}$$

Since all entries of d lie in $[-1, 1]$, we know

$$|fd| < \|f\| \ .$$

Thus,

$$n_1 < \frac{q_1}{q_0}(fc \bmod q_0) + \|f\| \ .$$

By hypothesis,

$$\begin{aligned} |fc \bmod q_0| &< \frac{q_0}{2} - \frac{q_0}{q_1}\|f\| \\ \frac{q_1}{q_0}|fc \bmod q_0| &< \frac{q_1}{2} - \|f\| \\ |n_1| &< \frac{q_1}{2} \ . \end{aligned}$$

This shows that n_1 is closer to zero than any other integer in the form $fc' - lq_1$. So

$$n_1 = fc' \bmod q_1$$

and

$$|fc' \bmod q_1| < \frac{q_1}{q_0}|fc \bmod q_0| + \|f\|$$

□

Lemma 4.1 shows us that we can recover the same message in two different rings after modulus reduction.

5 Implementation of the Scheme

In this section, we will discuss the implementation of the LTV homomorphic encryption scheme in Sage and some advanced circuits built using this scheme.

Sage is a free open-source mathematics software system licensed under the GPL. It combines the power of many existing open-source packages into a common Python-based interface [7]. Sage supports on-line worksheets and can also be installed locally wrapped in a Linux virtual machine. Figure 4 is a screenshot of a Sage worksheet GUI.

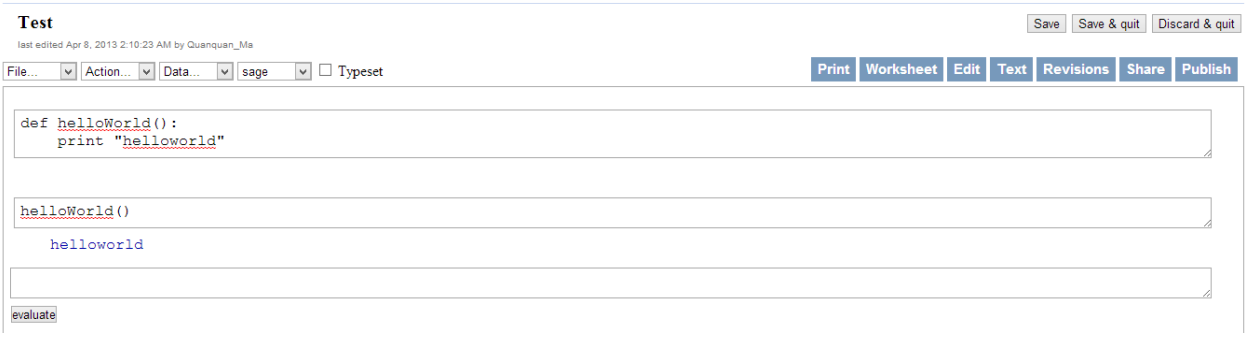

Figure 4: Sage worksheet

Since Sage is Python-based, it follows the syntax in Python and supports all the built-in data structures in Python such as lists and hashtables, which can be very useful. However, it is tricky that an integer in Sage is by default an integer in a Sage-specific package (sage.rings.integer.Integer) instead of a Python integer. Sometimes we need to perform conversions back and forth to avoid errors.

The Sage implementation developed in this project has been posted in Sage Interact Community website.

5.1 Inverse in Ring R_q

In order to generate the public key, we need to find the inverse of the secret key in the ring R_q. Recall that

$$R = \mathbb{Z}[x]/\langle \phi(x) \rangle \qquad \text{and} \qquad R_q = R/qR \ .$$

Since the ring R_q is not a commonly used ring, Sage does not support the environmental declaration of it. Although Sage supports the declaration of the ring R, it causes unexpected errors in later computations. Thus, we need to manually manipulate the additions and the multiplications in R_q. The manipulation of those operations is fairly straightforward, but it is not trivial to find the inverse of a polynomial, in our case, the inverse of the secret key.

5.1.1 Using System of Equations

A naive way to find the inverse is to generate a system of equations and solve it using Sage modulo solve – **solve_mod()**. Given the secret key

$$f = c_0 + c_1 x + c_2 x^2 + \ldots + c_{n-1} x^{n-1} \ ,$$

suppose

$$f^{-1} = s_0 + s_1 x + s_2 x^2 + \ldots + s_{n-1} x^{n-1} \ .$$

We have

$$f \cdot f^{-1} = 1 \in R_q \ .$$

This gives us

$$\begin{aligned}
c_0 s_0 - c_1 s_{n-1} - c_2 s_{n-2} - \ldots - c_{n-1} s_1 &\equiv 1 \pmod{q} \\
c_0 s_1 + c_1 s_0 - c_2 s_{n-1} - c_3 s_{n-2} - \ldots - c_{n-1} s_2 &\equiv 0 \pmod{q} \\
c_0 s_2 + c_1 s_1 + c_2 s_0 - c_3 s_{n-1} - c_4 s_{n-2} - \ldots - c_{n-1} s_3 &\equiv 0 \pmod{q} \\
&\ldots \\
c_0 s_{n-1} + c_1 s_{n-2} + \ldots + c_{n-1} s_0 &\equiv 0 \pmod{q} \ .
\end{aligned}$$

Solving this system of equations modulo q gives us f^{-1}. However, the time complexity of this approach is very inefficient. Although the equations can be generated almost instantly, it takes a very long time to solve the equations modulo q. A better approach is to use the Extended Euclidean Algorithm.

Also, it is worth mentioning that in order to declare n variables dynamically at execution time, I needed to use a Python hashtable. The key of the hashtable is an integer in $\{0, 1, \ldots, n-1\}$, while the value is a data type that represents actual variable. We can use **var()** to convert a string to a variable and put it in the hashtable for later usage.

5.1.2 Using the Extended Euclidean Algorithm

It is well known that Extended Euclidean Algorithm can be used to find the inverse of a number in $\mathbb{Z}_q$. For example, if we are trying to find the inverse of 11 in the ring $\mathbb{Z}_{29}$. First, we apply the Euclidean Algorithm.

$$\begin{aligned}
29 &= 11 \cdot 2 + 7 \\
11 &= 7 \cdot 1 + 4 \\
7 &= 4 \cdot 1 + 3 \\
4 &= 3 \cdot 1 + 1
\end{aligned}$$

Thus,

$$\gcd(29, 11) = 1 \ .$$

Now we apply the Extended Euclidean Algorithm.

$$\begin{aligned}
1 &= [4] - 1 \cdot [3] \\
&= [4] - 1 \cdot ([7] - 1 \cdot [4]) \\
&= 2 \cdot [4] - 1 \cdot [7] \\
&= 2 \cdot ([11] - 1 \cdot [7]) - 1 \cdot [7] \\
&= 2 \cdot [11] - 3 \cdot [7] \\
&= 2 \cdot [11] - 3 \cdot ([29] - 2 \cdot [11]) \\
&= 8 \cdot [11] - 3 \cdot [29]
\end{aligned}$$

Therefore, we know the inverse of 11 in $\mathbb{Z}_{29}$ is 8 because

$$8 \cdot 11 \equiv 1 \pmod{29} \ .$$

The same process can be applied to a polynomial in the ring R_q. For example, let

$$f = x^3 + 1 \qquad \phi(x) = x^4 + 1 \qquad q = 7 \ .$$

First, let's apply the Euclidean Algorithm.

$$\begin{aligned}
x^4 + 1 &= (x) \cdot (x^3 + 1) + (-x + 1) \\
x^3 + 1 &= (-x^2 - x - 1) \cdot (-x + 1) + 2
\end{aligned}$$

Note that when the reminder is any constant, not necessarily 1, we can stop the process. If the reminder is co-prime to q, then there is an inverse for f; otherwise, no inverse exists. Next, let's apply the Extended Euclidean Algorithm.

$$\begin{aligned}
2 &= [x^3 + 1] - (-x^2 - x - 1) \cdot [-x + 1] \\
&= [x^3 + 1] - (-x^2 - x - 1) \cdot ([x^4 + 1] - (x) \cdot [x^3 + 1]) \\
&= (-x^3 - x^2 - x + 1) \cdot [x^3 + 1] + (x^2 + x + 1) \cdot [x^4 + 1]
\end{aligned}$$

Since we know that

$$2 \cdot 4 \equiv 1 \pmod{7} \ ,$$

we have

$$2 \cdot 4 = (-4x^3 - 4x^2 - 4x + 4) \cdot [x^3 + 1] + (4x^2 + 4x + 4) \cdot [x^4 + 1] \ .$$

Thus,

$$(-4x^3 - 4x^2 - 4x + 4) \cdot [x^3 + 1] = 1 \in R_q \ .$$

Therefore, the inverse of f is

$$f^{-1} = -4x^3 - 4x^2 - 4x + 4 = 3x^3 + 3x^2 + 3x + 4 \in R_q \ .$$

The implementation of the Extended Euclidean Algorithm in Sage follows the same process as demonstrated. However, implementing it in Sage is not the same as working it out in paper. Since the only way we can handle the coefficients of the polynomials is the **coeffs()** function that returns a list of numbers, it is a little tricky to develop a Sage function. More details can be found in the Sage code package.

5.2 Encryption Scheme Implementation

The implementation of the basic scheme implementation exactly follows the process of the basic scheme. Below are the key functions implemented:

- $\mathsf{keygen}(N, B, q)$
  - INPUT: N as in $\phi(x) = x^N + 1$; B such that arbitrary polynomials are sampled as B-bounded; q prime as in R_q
  - OUTPUT: secret key, public key, evaluation key in this order
  - This function first randomly samples the variables needed, and applies Extended Euclidean Algorithm to find the inverse of the polynomial f. Finally, it outputs the keys generated, including the evaluation key, which is a list of polynomials.
- $\mathsf{enc}(N, B, q, h, m)$
  - INPUT: N as in $\phi(x) = x^N + 1$; B such that arbitrary polynomials are sampled as B-bounded; q prime as in R_q; h as the public key; m as a message in $\{0, 1\}$ to be encrypted
  - OUTPUT: the generated ciphertext
  - This function first randomly samples the variables needed, and then it computes and outputs the ciphertext.
- $\mathsf{encBits}(N, B, q, h, m)$
  - INPUT: N as in $\phi(x) = x^N + 1$; B such that arbitrary polynomials are sampled as B-bounded; q prime as in R_q; h as the public key; m as a list of bits to be encrypted
  - OUTPUT: the generated ciphertexts for each bit in a list
  - This function repeatedly calls $\mathsf{enc}()$ and append each result to a list. Then it outputs the list.
- $\mathsf{dec}(N, q, f, c)$
  - INPUT: N as in $\phi(x) = x^N + 1$; q prime as in R_q; f as the secret key; c as the ciphertext to be decrypted

- OUTPUT: the original message
- This function simply applies the decryption process and outputs the decrypted message.

- decBits(N, q, f, c)
  - INPUT: N as in $\phi(x) = x^N + 1$; q prime as in R_q; h as the public key; m as a list of ciphertexts to be decrypted
  - OUTPUT: the original messages for each ciphertext in a list
  - This function repeatedly calls dec() and append each result to a list. Then it outputs the list.

- eval(γ, c_1, c_2, q, N)
  - INPUT: γ as the evaluation key, which is a list of polynomials; c_1 and c_2 as two ciphertexts to be homomorphically multiplied; q prime as in R_q; N as in $\phi(x) = x^N + 1$;
  - OUTPUT: the ciphertext for the product of two ciphertexts
  - This function applies the relinearization process described earlier while multiplying two given ciphertexts, so that the generated ciphertext can be decrypted directly using dec().

- FHE_demo$(f', g, N, q, m, s, e, f_q)$
  - INPUT: f' and g as B-bounded polynomials required to generate the secret key and the public key; N as in $\phi(x) = x^N + 1$; q prime as in R_q; m as a message in $\{0, 1\}$ to be encrypted; s and e as two B-bounded polynomials required for the encryption process; f_q is an optional variable for the inverse of the secret key (if given, it will speed up the whole process.)
  - OUTPUT: the ciphertext generated
  - This function runs though the processes of key generation, encryption and decryption. It prints out every detailed step of each computation. It also monitors error growth.

- hMultiply(f, c_1, c_2, N, q)
  - INPUT: f as the secret key; c_1 and c_2 as two ciphertexts to be homomorphically multiplied; N as in $\phi(x) = x^N + 1$; q prime as in R_q;
  - OUTPUT: the ciphertext generated
  - This function first multiplies the two ciphertexts and then decrypts the result while printing out every detailed step of each computation. It also monitors error growth.

- $\mathsf{hAdd}(f, c_1, c_2, N, q)$
  - INPUT: f as the secret key; c_1 and c_2 as two ciphertexts to be homomorphically added; N as in $\phi(x) = x^N + 1$; q prime as in R_q;
  - OUTPUT: the ciphertext generated
  - This function first adds the two ciphertexts and then decrypts the result while printing out every detailed step of each computation. It also monitors error growth.

Figure 5 is a screen shot of the LTV scheme demonstration running in Sage.

5.3 Real-world Application

Besides the basic scheme, I also implemented several applications of the encryption scheme. Consider the following case. We have a database in the cloud with every bit encrypted with a secret key f. This database contains information for your recent transactions on a bank account and this database contains has two columns: the first column contains the categories for the transactions and the second column contains values of the transactions. Suppose you want to see the total money you spent on gas. How can we realize this using the LTV scheme?

5.3.1 Comparator

First, we need a comparator that takes two sequences of ciphertexts and returns a ciphertext indicating whether they are exactly the same. Let

$$c_X = X_n X_{n-1} \dots X_2 X_1 Y_0 \qquad c_Y = Y_n Y_{n-1} Y_0 \dots Y_2 Y_1 \ .$$

Then the ciphertext we need to compute is

$$c_{comp} = (X_n + Y_n + 1)(X_{n-1} + Y_{n-1} + 1) \dots (X_0 + Y_0 + 1) \ .$$

In order for c_{comp} to be 1, each of the $(X_i + Y_i + 1)$ must be 1, which means each $(X_i + Y_i)$ must be 0. This guarantees that $X_i = Y_i$. Below is the function implemented in Sage.

$\mathsf{compare}(a, b, N, q, \gamma)$

- INPUT: a and b as the two lists of ciphertexts to be compared; N as in $\phi(x) = x^N + 1$; q prime as in R_q; γ as the evaluation key

– OUTPUT: the ciphertext generated indicating whether a and b are exactly the same

– This function compares two sequences of ciphertexts and tells us if these two sequences are exactly the same. It uses relinearazation to simplify the decryption process.

5.3.2 Full Adder

The next thing we need is to add two integers represented in binary. To realize this, we need to implement a full adder using addition (XOR) and multiplication (AND). A full adder in this case takes in two ciphertexts a and b to be added and a carry-in ciphertext C_{in}; it generates a ciphertext s for the sum and another ciphertext C_{out} for carry-out. Figure 6 shows a 1-bit full adder module.

If we analyze the truth table of a full adder, we get the following equations using a Karnaugh map:

$$s = C_{in} + a + b$$

and

$$C_{out} = ab + bC_{in} + aC_{in} \ .$$

Note that addition here means XOR instead of OR.

However, here is another choice of the circuit for C_{out}.

$$\begin{aligned} C_{out} &= ab + bC_{in} + aC_{in} \\ &= (a + C_{in})(a + b) - a^2 \\ &= (a + C_{in})(a + b) + a^2 \\ &= (a + C_{in})(a + b) + a \end{aligned}$$

Although the second circuit takes fewer operations and fewer multiplications, it is difficult to decide which circuit to use. Let's assume that the wrap-around error term for every ciphertext is the same value E. Then we can make a rough approximation that the result error in the first circuit is $3E^2$ but the error in the second one is $4E^2 + E$. Both circuits seem to have their advantage. A comprehensive discussion on which circuit to use needs some future work.

Below is the implementation in Sage, which uses the first circuit.

$\mathsf{fullAdder}(a, b, C_{in}, N, q, \gamma)$

– INPUT: a and b as the two ciphertexts to be added; C_{in} as the carry-in ciphertext; N as in $\phi(x) = x^N + 1$; q prime as in R_q; γ as the evaluation key

– OUTPUT: two ciphertexts: one for the carry-out bit and the other for the sum bit

– This function implements 1-bit Full Adder. It uses relinearazation to simplify the decryption process.

The next thing we do is to repeatedly chain full adders to add tuples of bits. The implementation in sage is straightforward.

addBits(a, b, N, q, γ)

– INPUT: a and b as the two lists of ciphertexts to be added; N as in $\phi(x) = x^N + 1$; q prime as in R_q; γ as the evaluation key

– OUTPUT: a list of ciphertexts representing the sum of the two sequences of bits

– This function repeatedly uses fullAdder() to add two bit sequences. It uses relinearazation to simplify the decryption process. Note that the result has the same length as the inputs.

Figure 7 is a screen shot of the addBits() function running in Sage worksheet.

With compare() and addBits(), we can solve the database problem mentioned earlier. Suppose the first column has ciphertext sequences from A_1 to A_n indicating the categories, while the second column has ciphertext sequences from B_1 to B_n indicating the values. Given a specific category C, we can find the value we need by computing

$$S_i = \textbf{compare}(A_i, C) \cdot B_i$$

and adding them together

$$S = \textbf{addBits}(\textbf{addBits}(S_1, S_2), S_3) \ldots$$

5.3.3 Further Applications

Imagine a similar database where the second column stays the same but the first column becomes a column containing the time of the transactions. We want to see the total transaction value from the most recent month. What should we do?

This requires us to find the larger/smaller value of two ciphertext sequences. One option is to subtract two sequences of bits and look at the borrow-out ciphertext. If it's 1, then the minuend is smaller; otherwise, the minuend is bigger than or equal to the subtrahend. This will work on positive signed integers in two's complement and unsigned integers. Again, from the truth table, we can use a Karnaugh map to see that

$$s = B_{in} + a + b$$

and

$$B_{out} = (B_{in} + b)(1 + a) + bB_{in} \ .$$

From this, we have our Sage implementation:

subtracter$(a, b, B_{in}, N, q, \gamma)$

- INPUT: a as the ciphertext for the minuend and b as the ciphertext for the subtrahend; B_{in} as the borrow-in ciphertext; N as in $\phi(x) = x^N + 1$; q prime as in R_q; γ as the evaluation key
- OUTPUT: two ciphertexts: one for the borrow-out bit and the other for the difference bit
- This function implements the subtracter that works similar as full adder but for subtraction. It uses relinearazation to simplify the decryption process.

Next, if we apply subtracter() repeatedly, we can subtract two sequences of ciphertexts.

subBits(a, b, N, q, γ)

- INPUT: a as the ciphertext sequence for the minuend and b as the ciphertext sequence for the subtrahend; N as in $\phi(x) = x^N + 1$; q prime as in R_q; γ as the evaluation key
- OUTPUT: a list of ciphertexts representing the difference of the two sequences
- This function repeatedly uses subtracter() to subtract one bit sequence from another. It uses relinearazation to simplify the decryption process. Note that we can use this to find the smaller ciphertext sequence by checking B_{out}, and if this is the only purpose, we can eliminate the computation of the difference.

Now that we can find the smaller one of two given ciphertext sequences, we can find the transactions from the most recent month. We can apply similar operations as the previous case to compute the ciphertext sequence for the total value.

6 Conclusion and Future Work

This project presents a single-key version of the LTV scheme that is more accessible to non-experts. We provide serious mathematical proofs and detailed explanations on some implicit mathematical steps that were not addressed by the authors of the original scheme. We also trace back to the NTRU cryptosystem, which the LTV scheme was based on, to

see how it is related to our problem. In addition, this project includes a Sage package that implements the basic scheme and some real applications such as an n-bit Adder. The Sage code is attached in the appendix.

However, there are some topics that are interesting but we do not have enough time to cover. One of the topics is how to implement the database problem defined in Section 5.3 more efficiently. Right now, we need to multiply the result from comparator by the value in second column and add the product from each row altogether. This is obviously not satisfactory. Suppose we have 1,000,000 rows in a large database, but we only want the sum from approximately 100 certain rows. In the method described in this project, 99.99% of the work is unnecessary. But since the bit indicating whether we should add the value is masked, it is very hard to find a way around it. This is certainly very useful in practice.

Another topic will be to analyze the security of the LTV scheme. This project completely skips the security part; therefore, it will be nice to have a future project to fill this gap. Some more possible topics include an implementation of this scheme in a lower-level language such as Java or C/C++, a research into how exactly the LTV scheme utilizes Gentry's bootstrapping scheme, an discussion of how to design the most efficient circuits for the LTV scheme and an expansion of the current Sage package.

References

[1] R. Buyya, et al., Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility, Future Generation Computer Systems (2009)

[2] J. McKendrick, More Than One-Third of IT Budgets Now Spent on Cloud: Survey, http://www.forbes.com/sites/joemckendrick/2012/04/11/more-than-one-third-of-it-budgets-now-spent-on-cloud-survey/

[3] Picture retrieved from http://www.tutorialspoint.com/shorttutorials/cloud-computing-from-the-home/

[4] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage., Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds, ACM Conference on Computer and Communications Security, Pages 199212, 2009.

[5] C. Stuntz, What is Homomorphic Encryption, and Why Should I Care?, 2010, http://blogs.teamb.com/craigstuntz/2010/03/18/38566/

[6] A. López-Alt, E. Tromer and V. Vaikuntanathan, On-the-Fly Multiparty Computation on the Cloud via Multikey Fully Homomorphic Encryption In STOC '12 Proceedings of the 44th symposium on Theory of Computing, Pages 1219-1234, 2012

[7] Sage Official Introduction, http://www.sagemath.org/

[8] J. Hoffstein, J. Pipher and H. Silverman, NTRU: A Ring-Based Public Key Cryptosystem, Algorithmic Number Theory (ANTS III), Portland, OR, June 1998, Lecture Notes in Computer Science 1423, 267-288

[9] Picture retrieved from http://uowdsummer2012-iact999-mobilesecurity.blogspot.com/

[10] R. Rivest, A. Shamir, L. Adleman, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, Communications of the ACM 21 (2): 120126, 1978

Appendix

```
########################################
# The LTV Homomorphic Encryption Scheme
#         Sage Implementation
########################################
# Developed for Major Qualifying Project
#              WJM5200
#                in
#    WORCESTER POLYTECHNIC INSTITITE
########################################
# Author:  Quanquan Ma
# Advisor: Professor William J. Martin
#          Professor Berk Sunar
########################################

from random import randint
R.<x> = QQ['x'];

#
# This encryption scheme is a single-key
# version of the cryptosystem by Lopez-Alt,
# Tromer and Vaikuntanathan.
#
# It includes basic operations
#     - keygen
#     - enc
#     - dec
#     - eval
# and more advanced circuits
#     - fullAdder
#     - n-bit adder
#     - subtractor
#     - comparator.
# It also includes demonstrations that shows
# detailed steps of the process.
#
# Variables are named based on the original scheme
#     - B for B-bounded polynomials
#     - n for phi = x^n + 1
#     - q for the modulus
#     - f for the secret key
```

```
#     - h for the public key
#     - gamma for the evaluation key
#

#
# Key generation
#     returns (sk, pk, ek)
# where ek is a list of polynomials
#
def keygen(N, B, q):
    fprime = sample(N, B)
    g = sample(N, B)
    f = 2 * fprime + 1
    # find inverse of f mod q
    fq = fastInverse(f, N, q)
    if fq == 0:
        print "f is not invertible mod q. Please pick another f'."
        return

    # computer public key h
    fqg = 2 * fq * g
    fqg = fqg.mod(x^N + 1)
    h = modCoeffs(fqg, q)

    # Compute the evaluation key
    logq = floor(log(q, 2))
    gamma_tau = []
    s_tau = []
    e_tau = []
    for tau in range(logq+1):
        s_tau.append(sample(N, B))
        e_tau.append(sample(N, B))
        res = h * s_tau[tau] + 2 * e_tau[tau] + f*2^tau
        gamma_tau.append(modCoeffs(res.mod(x^N+1), q))

    return f, h, gamma_tau

#
# Compare two lists of ciphertexts
#     returns a ciphertext indicating whether the
#              two inputs are the encryption of the same bit
#
```

```
def compare(a, b, N, p, gamma):
    result = 1
    for ai, bi in zip(a, b):
        result = eval(gamma, result, (ai + bi + 1), p, N)
    return result

#
# Full adder where each input and output is encryption of a bit
#      a, b for the two bits being added
#      cin for carry-in bit
#
#      returns (s, cout)
#                  where s is the sum and cout is the carry-out
#                    they are all ciphertexts
#
def fullAdder(a, b, cin, N, p, gamma):
    s = modCoeffs(a + b + cin, p)
    cout = eval(gamma, a, b, p, N) + eval(gamma, cin, b, p, N) + eval(gamma, a, cin, p,
    modCoeffs(cout, p)
    return s, cout

#
# Add to list of ciphertexts
#      in this order [LSB, ..., MSB]
#     returns a list of ciphertexts indicating the sum
#            notice that we get a n-bit integer when adding two n-bit integers
#                        the overflow is ignored
#
def addBits(a, b, N, p, gamma):
    result = []
    s = 0
    cin = 0
    for ai, bi in zip(a, b):
        s, cin = fullAdder(ai, bi, cin, N, p, gamma)
        result.append(s)
    return result

#
# Subtracter the works similar to a full adder but for subtraction
#      subtract b from a and Bin is borrow-in
#      returns (d, Bout) where d is the difference and Bout is borrow-out
#
```

```
def subtracter(a, b, Bin, N, p, gamma):
    d = modCoeffs(a + b + Bin, p)
    Bout= eval(gamma, modCoeffs(Bin+b, p), modCoeffs(1+a, p), p, N) + eval(gamma, Bin, b
    modCoeffs(Bout, p)
    return d, Bout

#
# Subtract a list of ciphertexts from another list of ciphertexts
#     a, b are the two lists of ciphertexts
#     returns (result, Bin) where result is the difference of the two inputs and
#                          Bin can be use to find which of the two inputs is larger
#
def subBits(a, b, N, p, gamma):
    result = []
    s = 0
    Bin = 0
    for ai, bi in zip(a, b):
        s, Bin = subtracter(ai, bi, Bin, N, p, gamma)
        result.append(s)
    return result, Bin

#
# multiply two polynomials in the ring R_q
#
def multiply(f, g, N, p):
    return modCoeffs(cProduct(f, g, N), p)

#
# multiply two polynomials in the ring R
#
def cProduct(f, g, N):
    return (f * g).mod(x^N + 1)

#
# Reduce the coefficients of a polynomial to modulo p
#       notice that we reduce it into [-p/2, p/2]
#
def modCoeffs(f, p):
    i = 0
    f_mod_p = 0
    for c in f.coeffs():
        c = c % p
```

```
        if c > p / 2:
            c -= p
        f_mod_p += c*x^i
        i += 1
    return f_mod_p

#
# Sage does not support degree() on integers
# This is a way around it
#
def degree(a):
    if a == 0:
        return 0
    a = a * x
    return a.degree() - 1

#
# Reduce the coefficients of f modulo p regularly
#    (into [0, p-1])
#
def modCoeffsPositive(f, p):
    i = 0
    f_mod_p = 0
    for c in f.coeffs():
        c = c % p
        f_mod_p += c*x^i
        i += 1
    return f_mod_p

#
# Find the inverse of a polynomial using Extended Euclidean Algorithm
#
def fastInverse(f, n, p):
    a, b = euclid(x^n+1, f, p)
    return modCoeffsPositive(b, p)

#
# Extended Euclidean Algorithm in the ring R_q
#
def euclid(f, g, p):
    q = []
    r = []
```

```
    condition = True
    while condition:
        qi, ri = wellDivide(f, g, p)
        f = g
        g = ri
        q.append(qi)
        r.append(ri)
        condition = not (degree(g) == 0)

    # f     = q_0 * g     + r_0
    # g     = q_1 * r_0   + r_1
    # r_n-2 = q_n * r_n-1 + r_n
    #
    # r_n = r_n-2 - q_n * r_n-1

    n = len(q) - 1
    # r_n   = a * r_n-2 + b * r_n-1
    # r_n-1 = r_n-3 - q_n-1 * r_n-2
    # so
    # r_n   = b * r_n-3 + (a - b * q_n-1) * r_n-2
    a = 1
    b = -q[n]
    while n >= 1:
        tmp = b
        b = a - b * q[n-1]
        a = tmp
        n -= 1
    var('t');
    k = int(solve_mod(ri*t==1, p, True)[0][t])
    return a*k, b*k
#
# Devide two polynomials
#   used for Extended Euclidean Algorithm
#     returns a quotient and a reminder
#          where f = g*res +rem
def wellDivide(f, g, p):
    res, rem = divide(f, g, p)
    curRem = rem
    while degree(curRem) >= degree(g):
        res2, curRem = divide(curRem, g, p)
        res += res2
        rem = curRem
```

```
    return res, rem

#
# A function that divides two polynomials but does not guarantee
#    that the remainder is smaller than g
#    used for wellDivide
#      returns res as a polynomial x^n where n is the difference between
#               f and g; rem as a polynomial the makes sure f = g*res +rem
#
def divide(f, g, p):
    # assume f >= g
    quo = x^(degree(f) - degree(g))
    a = f.coeffs()[degree(f)]
    b = g.coeffs()[degree(g)]
    var('t');
    sol = solve_mod(b*t==a, p, True)[0][t]
    res = int(sol) * quo
    rem = modCoeffsPositive(f - g * res, p)
    return res, rem

#
# Find a inverse of a polynomial via solving a system of equations
#    Very slow as the variables get large
#    Better way is to use fastInverse()
#
def inverse(f, N, p):
    # dynamically declare vars
    a = {}
    for i in range(N):
        variable = "a" + str(i)
        a[i] = var(variable)

    # coeff list c0 ... cn
    c = f.coeffs()

    # generate a list of eqns to solve
    eqns = []
    for k in range(N):
        left = 0
        for i in range(N):
            j = 0
            for cj in c:
```

```
                if (i + j) % N == k:
                    if (i + j) < N:
                        left += cj * a[i]
                    else:
                        left -= cj * a[i]
                j += 1
        if k == 0:
            eqn = left == 1
        else:
            eqn = left == 0
        eqns.append(eqn)

    print "Equations generated..."
    print eqns
    # modulo solve mod p
    sln = solve_mod(eqns, p)
    print "Solved..."

    # the inverse might not exist, return 0 because 0 can never a an inverse
    if len(sln) == 0:
        return 0
    else:
        fp = 0
        i = 0
        for c in sln[0]:
            fp += int(c)*x^i
            i += 1
        return fp

#
# Generate a random B-Bounded polynomial using random numbers from sage
#    if we want to define a distribution chi, here is where to modify
#
def sample(N, B):
    s = 0
    for i in range(N):
        r = randint(-B, B)
        s += r * x^i
    return s

#
# Encryption
```

```
#    returns a ciphertext as an encryption of m
#
def enc(N, B, q, h, m):
    s = sample(N, B)
    e = sample(N, B)
    c = cProduct(h, s, N) + 2 * e + m
    c = modCoeffs(c, q)
    return c

#
# Encrypt a list of messages
#     returns a list of ciphertext
#
def encBits(N, B, q, h, m):
    result = []
    for mi in m:
        s = sample(N, B)
        e = sample(N, B)
        c = cProduct(h, s, N) + 2 * e + mi
        c = modCoeffs(c, q)
        result.append(c)
    return result

#
# Decryption
#     returns a message which is actually a polynomial
#            But if the decryption is correct, it returns a single bit because
#                 the other terms are all zero
#
def dec(N, B, q, f, c):
    miu = multiply(f, c, N, q)
    m2 = modCoeffs(miu, 2)
    return m2

#
# Decrypt a list of bits
#
def decBits(N, B, q, f, c):
    result = []
    for ci in c:
        m2 = dec(N, B, q, f, ci)
        result.append(m2)
```

```
    return result

#
# Evaluation Process
#    This is used to multiply c1 and c2 using a eval key gamma.
#       returns a evaluated ciphertext as the result
#
def eval(gamma, c1, c2, q, N):
    logq = floor(log(q, 2))
    ctil = multiply(c1, c2, N, q)
    c0tau = []
    for tau in range(logq+1):
        c0tau.append(0)
    for tau in range(logq+1):
        i = 0
        if degree(ctil) != 0:
            for c in ctil.coeffs():
                if (c >= 0):
                    c0tau[tau] += (c % 2) * x^i
                    ctil -= (c % 2) * x^i
                    i += 1
                else:
                    c0tau[tau] -= (c % 2) * x^i
                    ctil += (c % 2) * x^i
                    i += 1
            ctil = ctil / 2
    c1til = 0
    for tau in range(logq+1):
        c1til += gamma[tau] * c0tau[tau]
    return modCoeffs(c1til.mod(x^N+1), q)

#
# A demo for modulus reduction
#
def FHE_modred_demo(ddec, u, g, N, q, B, m, s, e):
    print "####################"
    print "# Key generation:"
    print "####################"
    f = []
    fq = []
    h = []
    for i in range(ddec):
```

```
    # compute secret key f
    print "Iteration " + str(i) + ":"
    f.append(2*u[i]+1)
    print "Secret key: f_" + str(i) + " = " + str(f)

    # find inverse of f_i mod q_i
    fqi = fastInverse(f[i], N, q[i])
    if fqi == 0:
        print "f is not invertible mod q. Please pick another u."
        return
    else:
        print "Inverse of f_" + str(i) + " is " + str(fqi)
        fq.append(fqi)

    # computer public key h
    hi = 2 * g[i] * fq[i]
    hi = hi.mod(x^N+1)
    hi = modCoeffs(hi, q[i])
    h.append(hi)
    print "Public key: h = " + str(hi)

gamma = []
zeta = []
gamma.append([])
zeta.append([])
for fakei in range(ddec-1):
    i = fakei + 1
    # Compute the evaluation key
    print "\nEvaluation Key:\n"
    logqi = floor(log(q[i-1], 2))
    gamma_tau = []
    zeta_tau = []
    s_tau = []
    e_tau = []
    for tau in range(logqi+1):
        s_tau.append(sample(N, B))
        print "s_" + str(tau) + " = " + str(s_tau[tau])
        e_tau.append(sample(N, B))
        print "e_" + str(tau) + " = " + str(e_tau[tau])
        g_res = h[i] * s_tau[tau] + 2 * e_tau[tau] + f[i-1]*(2^tau)
        z_res = h[i] * s_tau[tau] + 2 * e_tau[tau] + (f[i-1]^2)*(2^tau)
        gamma_tau.append(modCoeffs(g_res.mod(x^N+1), q[i-1]))
```

```
            zeta_tau.append(modCoeffs(z_res.mod(x^N+1), q[i-1]))
            print "gamma_" + str(tau) + " = " + str(gamma_tau[tau])
            print "zeta_" + str(tau) + " = " + str(zeta_tau[tau])
        gamma.append(gamma_tau)
        zeta.append(zeta_tau)

    print ""
    print "###############"
    print "# Encryption:"
    print "###############"
    print "Encrypted message: c = h[0] * s + 2 * e + m"
    c = cProduct(h[0], s, N) + 2 * e + m
    print "                     = " + str(c) + " (mod q0)"
    c = modCoeffs(c, q[0])
    print "                     = " + str(c)

    print ""
    print "###############"
    print "# Decryption:"
    print "###############"
    print "First compute: miu = f * c (mod q)"
    miu = f[0] * c
    print "                   = " + str(miu) + " (mod x^N+1) (mod q)"
    miu = miu.mod(x^N+1)
    print "                   = " + str(miu) + " (mod q)"
    miu = modCoeffs(miu, q[0])
    print "                   = " + str(miu)

    print "Recover message: m' = miu (mod 2)"
    m2 = modCoeffs(miu, 2)
    print "                    = " + str(m2)
    return c, gamma, zeta

#
# A demo for addtion in modulus reduction
#
def eval_add_demo(gamma, c1, c2, q, j ,N):
    logq = floor(log(q[j], 2))
    ctil = modCoeffs((c1+c2).mod(x^N+1), q[j])
    print "c0~ = " + str(ctil)
    c0tau = []
    for tau in range(logq+1):
```

```
        c0tau.append(0)
    print "Binary Representation:"
    for tau in range(logq+1):
        i = 0
        for c in ctil.coeffs():
            if (c >= 0):
                #print str(c) + " pos=> " + str(c % 2)
                c0tau[tau] += (c % 2) * x^i
                ctil -= (c % 2) * x^i
                i += 1
            else:
                #print str(c) + " neg=> " + str(c % 2)
                c0tau[tau] -= (c % 2) * x^i
                ctil += (c % 2) * x^i
                i += 1
        ctil = ctil / 2
        print "c~(0," + str(tau) + ") = " + str(c0tau[tau])
    c1til = 0
    for tau in range(logq+1):
        c1til += gamma[j+1][tau] * c0tau[tau]
    c1til = modCoeffs(c1til.mod(x^N+1), q[j])
    print "c1~ = " + str(c1til)
    cadd = 0
    i = 0
    for c in c1til.coeffs():
        cnew = c*q[j+1]/q[j]
        cround = round(cnew)
        # preserve the parity
        if (cround + c) % 2 == 1:
            if (cround - cnew) > 0:
                cround -= 1
            else :
                cround += 1
        cadd += cround * x^i
        i += 1
    cadd = modCoeffs(cadd.mod(x^N+1), q[j+1])
    print "c_add = " + str(cadd)
    return cadd

#
# A demo for multiplication in modulus reduction
#
```

```
def eval_mul_demo(zeta, c1, c2, q, j ,N):
    logq = floor(log(q[j], 2))
    ctil = modCoeffs((c1*c2).mod(x^N+1), q[j])
    print "c0~ = " + str(ctil)
    c0tau = []
    for tau in range(logq+1):
        c0tau.append(0)
    print "Binary Representation:"
    for tau in range(logq+1):
        i = 0
        for c in ctil.coeffs():
            if (c >= 0):
                #print str(c) + " pos=> " + str(c % 2)
                c0tau[tau] += (c % 2) * x^i
                ctil -= (c % 2) * x^i
                i += 1
            else:
                #print str(c) + " neg=> " + str(c % 2)
                c0tau[tau] -= (c % 2) * x^i
                ctil += (c % 2) * x^i
                i += 1
        ctil = ctil / 2
        print "c~(0," + str(tau) + ") = " + str(c0tau[tau])
    c1til = 0
    for tau in range(logq+1):
        c1til += zeta[j][tau] * c0tau[tau]
    cadd = 0
    i = 0
    for c in c1til.coeffs():
        cnew = c*q[j+1]/q[j]
        cround = round(cnew)
        # preserve the parity
        if (cround + c) % 2 == 1:
            if (cround - cnew) > 0:
                cround -= 1
            else :
                cround += 1
        cadd += cround * x^i
        i += 1
    return modCoeffs(c1til.mod(x^N+1), q[j+1])

#
```

```
# A demo for the basic scheme
#
def FHE_demo(fprime, g, N, q, m, s, e, fq):
    print "####################"
    print "# Key generation:"
    print "####################"

    f = 2 * fprime + 1
    print "Secret key: f = " + str(f)

    # find inverse of f mod q
    if fq == 0:
        fq = inverse(f, N, q)
    if fq == 0:
        print "f is not invertible mod q. Please pick another f'."
        return
    else:
        print "Fq = " + str(fq)
        print "Verify: f * Fq = " + str(f * fq)
        ffq = cProduct(f, fq, N)
        print "   (mod x^N+1) = " + str(ffq)
        print "       (mod q) = " + str(modCoeffs(ffq, q))


    # computer public key h
    print "Public key: h = 2 * Fq * g (mod q)"
    fqg = 2 * fq * g
    print "              = " + str(fqg) + " (mod x^N-1) (mod q)"
    fqg = fqg.mod(x^N + 1)
    print "              = " + str(fqg) + " (mod q)"
    h = modCoeffs(fqg, q)
    print     "              = " + str(h)

    print ""
    print "###############"
    print "# Encryption:"
    print "###############"
    print "Encrypted message: c = h * s + 2 * e + m"
    c = cProduct(h, s, N) + 2 * e + m
    print "                     = " + str(c) + " (mod q)"
    c = modCoeffs(c, q)
    print "                     = " + str(c)
```

```
    print ""
    print "################"
    print "# Decryption:"
    print "################"
    print "First compute: miu = f * c (mod q)"
    miu = f * c
    print "                    = " + str(miu) + " (mod x^N+1) (mod q)"
    miu = miu.mod(x^N+1)
    print "                    = " + str(miu) + " (mod q)"
    miu = modCoeffs(miu, q)
    print "                    = " + str(miu)

    print "fc before mod q"
    gs = cProduct(g, s, N)
    print "gs = " + str(gs)
    fe = cProduct(f, e, N)
    print "fe = " + str(fe)
    fm = cProduct(f, m, N)
    print "fm = " + str(fm)
    fc = 2 * gs + 2 * fe + fm
    print "fc = " + str(fc)

    print "Recover message: m' = miu (mod 2)"
    m2 = modCoeffs(miu, 2)
    print "                     = " + str(m2)
    return c

#
# A demo for multiplying to ciphertexts in the basic scheme
#
def hMultiply(f, c1, c2, n, q):
    print "\n###################"
    print "# Multiplication"
    print "###################"
    ff = modCoeffs(cProduct(f, f, n), q)
    print "f^2 = " + str(ff)
    c1c2 = modCoeffs(cProduct(c1, c2, n), q)
    print "c1c2 = " + str(c1c2)
    ffc1c2 = modCoeffs(cProduct(ff, c1c2, n), q)
    print "ff(c1c2) = " + str(ffc1c2)
    result = modCoeffs(ffc1c2, 2)
```

```
    print "   (mod 2) = " + str(result)
    return result

#
# A demo for adding to ciphertexts in the basic scheme
#
def hAdd(f, c1, c2, n, q):
    print "\n#############"
    print "# Addition"
    print "#############"
    c1c2 = modCoeffs(c1 + c2, q)
    print "c1 + c2 = " + str(c1c2)
    fc1c2 = modCoeffs(cProduct(f, c1c2, n), q)
    print "f(c1 + c2) = " + str(fc1c2)
    result = modCoeffs(fc1c2, 2)
    print "      (mod 2) = " + str(result)
    return result
```

```
N = 4
q = 9973
B = 1
m = [0, 1, 1] # 110
n = [1, 1, 1] # 111
f, h, gamma = keygen(N, B, q)
print "Public key:    " + str(h)
me = encBits(N, B, q, h, m)
print ""
print "Ciphertext for m: "
print me
ne = encBits(N, B, q, h, n)
print ""
print "Ciphertext for n: "
print ne
s = addBits(me, ne, N, q, gamma)
print ""
print "Ciphertext for sum:"
print s
result = decBits(N, B, q, f, s)
print ""
print "Decryption of sum"
print result
# should be [1, 0, 1]
```

```
Public key:    1174*x^3 - 587*x^2 - 4692*x + 2346

Ciphertext for m:
[1759*x^3 + 4105*x^2 + 2931*x + 3518, -4698*x^3 + 2350*x^2 - 1176*x +
588, -3524*x^3 + 1757*x^2 + 4107*x + 2936]

Ciphertext for n:
[-2931*x^3 - 3522*x^2 + 1761*x + 4110, -2933*x^3 - 3522*x^2 + 1757*x +
4106, -1170*x^3 + 585*x^2 + 4696*x - 2343]

Ciphertext for sum:
[-1172*x^3 + 583*x^2 + 4692*x - 2345, -3554*x^3 + 1731*x^2 + 4073*x +
2939, 3027*x^3 + 3208*x^2 - 2419*x - 4540]

Decryption of sum
[1, 0, 1]
```

Figure 5: The LTV Scheme demonstration in Sage

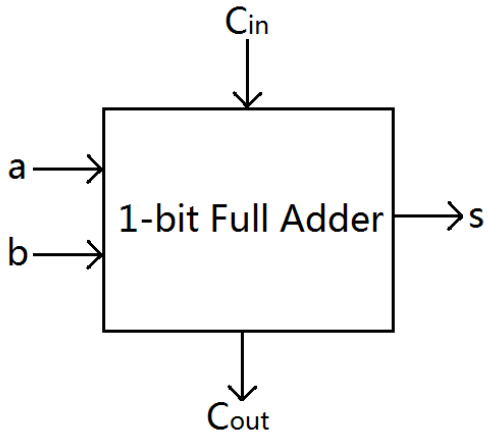

Figure 6: 1-bit full adder

```
N = 4
q = 9973
B = 1
m = [0, 1, 1] # 110
n = [1, 1, 1] # 111
f, h, gamma = keygen(N, B, q)
print "Public key:    " + str(h)
me = encBits(N, B, q, h, m)
print ""
print "Ciphertext for m: "
print me
ne = encBits(N, B, q, h, n)
print ""
print "Ciphertext for n: "
print ne
s = addBits(me, ne, N, q, gamma)
print ""
print "Ciphertext for sum:"
print s
result = decBits(N, B, q, f, s)
print ""
print "Decryption of sum"
print result
# should be [1, 0, 1]
```

```
Public key:    1174*x^3 - 587*x^2 - 4692*x + 2346

Ciphertext for m:
[1759*x^3 + 4105*x^2 + 2931*x + 3518, -4698*x^3 + 2350*x^2 - 1176*x +
588, -3524*x^3 + 1757*x^2 + 4107*x + 2936]

Ciphertext for n:
[-2931*x^3 - 3522*x^2 + 1761*x + 4110, -2933*x^3 - 3522*x^2 + 1757*x +
4106, -1170*x^3 + 585*x^2 + 4696*x - 2343]

Ciphertext for sum:
[-1172*x^3 + 583*x^2 + 4692*x - 2345, -3554*x^3 + 1731*x^2 + 4073*x +
2939, 3027*x^3 + 3208*x^2 - 2419*x - 4540]

Decryption of sum
[1, 0, 1]
```

Figure 7: 3-bit Adder in Sage