A Proof of Concept for Homomorphically Evaluating an Encrypted Assembly Language

A PROOF OF CONCEPT FOR HOMOMORPHICALLY EVALUATING AN ENCRYPTED ASSEMBLY LANGUAGE

BY

DRAGAN RAKAS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

Master of Science

Master of Science (2013) McMaster University

(Computing and Software) Hamilton, Ontario, Canada

TITLE:	A Proof of Concept for Homomorphically Evaluating an Encrypted Assembly Language
AUTHOR:	Dragan Rakas B. Eng. Mgt, Computer Engineering and Management
SUPERVISOR:	Dr. Michael Soltys
NUMBER OF PAGES:	ix, 58

To my parents and brother.

Abstract

Fully homomorphic encryption allows computations to be made on encrypted data without decryption, while preserving data integrity. This feature is desirable in a variety of applications such as banking, search engine and database querying, and some cloud computing services. Despite not knowing the plaintext content of the data, a remote server performing the computation would still be aware of the functions being applied to the data. To address the issue, this thesis proposes a method of encrypting circuits and executing encrypted instructions, by combining fully homomorphic encryption and digital logic theory. We use the classic RISC Archtecture as a foundation of our work, and the result of our algorithm is essentially an encrypted programming language, where a remote server is capable of executing program code that was written and encrypted by a local client.

Acknowledgements

I want to thank Dr. Michael Soltys for the great feedback throughout the months of writing and working on this thesis. I would also like to thank the FRAISE Group at McMaster, which gave me the opportunity to present my work twice, and provided me with valuable feedback during this time. Special thanks to Dr. Sanzheng Qiao, Dr. Ridha Khedri, Jason, Ariel, Mohamed, Adam, Sceuchin, and everyone else in the FRAISE Group for all of their comments and advice. Lastly, a big thank you to my parents and brother for their ongoing support.

Notation and abbreviations

Since some equations can get lengthy, we use tilde above variable names to denote a set of variables. For instance, $\tilde{s_i}$ is a set of variables s with subscripts with an index range i. Suppose $0 \leq i \leq N$, then $\tilde{s_i}$ is short for $s_0, s_1, \ldots, s_{N-1}, s_N$.

In diagrams and equations, multiplexer functions are denoted by $f_n(\cdots)$, where n is short for an n:1 multiplexer with n inputs. In the event that multiple functions are used in the same context or diagram, then each boolean function that the multiplexer implements is indexed, i.e., f_{n_i} denotes multiplexer i with n inputs and 1 output.

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem Statement

Currently, we know that fully homomorphic encryption algorithms allow us to compute any boolean function on encrypted data. We would like to extend this idea to not only being able to compute boolean functions over encrypted data, but also homomorphically interpret encrypted program code. To clarify, if a client writes a program, and then encrypts it, our goal is to provide a proof of concept for a method that allows a remote server to run the encrypted program without ever decrypting it. This criteria introduces many challenges, in which we explore the feasibility of handling basic arithmetic operations, simulating memory, handling program loops, recursion and File I/O. The requirement is that our program written by the client needs to be encrypted using fully homomorphic encryption, then our goal is to construct a circuit that a server can use to homomorphically evaluate and run the client's program.

1.2 A Brief Overview and History

1.2.1 History of Homomorphic Encryption

In many applications, such as internet banking, search engines, or some cloud computing services, it is often required to store user data and user requests encrypted, as opposed to plain unencrypted text. Whenever the stored data is needed for processing, current systems need to decrypt the data, and re-encrypt it after finishing their work. Homomorphic encryption allows us to skip this step entirely, and perform the desired operations on data while they are encrypted. In 1978, the idea of homomorphic encryption was introduced and discussed in [RAD78], which used RSA as a model to compute a limited amount of additive and multiplicative homomorphic operations. However, this scheme was vulnerable to attacks, and the article posed an open question of whether a useful homomorphic system existed in practice.

Given plaintext bits m_1 and m_2, and their corresponding ciphertexts c_1 and c_2, an operation is homomorphic if the operation applied to m_1 and m_2 yields the consistent results for some operation applied to c_1 and c_2. For instance, the original RSA encryption scheme is multiplicatively homomorphic because if for some public key (n, e), we let $c_1 = m_1^e \mod n$ and $c_2 = m_2^e \mod n$, then we note that $c_1 c_2 = (m_1^e m_2^e) \mod n$, which is just the encrypted product of $m_1 m_2$. Consequently, if we do not want to know what bits m_1 and m_2 are yet still use their product, we can compute their ciphertext product $c_1 c_2$ and then decrypt the result.

The operations on ciphertexts and their resulting plaintexts do not need to be identical. The Goldwasser-Micali (GM) cryptosystem soon followed the published works of [RAD78], presented a probabilistic encryption algorithm in [GM82] with interesting

homomorphic properties. In the GM cryptosystem, the encryption function uses $c_1 = y^2 x^{m_1} \mod N$ and $c_2 = y^2 x^{m_2} \mod N$. In this case, we obtain $c_1 c_2 = y^4 x^{(m_1+m_2)} \mod N$, which is actually $m_1 \oplus m_2$ when decrypted, unlike RSA. The GM cryptosystem is therefore additively homomorphic. The Paillier cryptosystem described in [Pai99] also has a similar property, where $c_1 c_2 = (g_1^m r_1^n)(g_2^m r_2^n) = g^{(m_1+m_2)}(r_1 r_2)^n \mod n^2$ yields an encryption of $(m_1 + m_2) \mod n$. There are many cryptosystems with homomoprhic properties. The interested reader is further directed to additional material on ElGamal in [Elg85], a lattice cryptosystem by Ajtai-Dwork in [AD97], and an additively homomorphic voting or election system proposed by Cohen and Fischer in [CF85].

1.2.2 The First Fully Homomorphic Scheme

Every encryption function presented thus far is either homomorphically additive or multiplicative, but none of them are both. The first ever fully homomorphic encryption (FHE) scheme was presented in 2009 by Craig Gentry in his Ph.D. thesis [Gen09]. Loosely defined, a system is *fully homomorphic* if it can support an arbitrary amount of operations on ciphertexts, or in other words, can evaluate any circuit consisting of AND and XOR gates. The reason why this definition is important is because Gentry describes an encryption function that adds "noise" to ciphertexts. With each additional operation (whether it be addition or multiplication), the ciphertext noise grows larger. Beyond some noise threshold, ciphertexts can no longer be decrypted. Gentry's most important idea was how to homomorphically remove noise, which was his idea of using a "bootstrappable" encryption scheme. Without the noise removal process, the cryptosystem is *semi-homomorphic* and can only compute limited sized

circuits on ciphertexts. In order to remove noise, we would need two procedures: Recrypt and Evaluate. The Recrypt procedure adds a second layer of encryption, and we use the Evaluate function to evaluate the decryption circuit. However, we would need a circuit for the decryption function, evaluate the circuit on the ciphertext homomorphically, and the resulting ciphertext will have less noise (subject to constraints discussed in Section 2.1). This procedure can be summarized in the following steps:

1. Encrypt bit m_1 such that $E_1(m_1) = c_1$ using Public Key 1.
2. Use encrypted c_1 for a few homomorphic operations (the exact amount of operations depends on security parameters and noise tolerance).
3. Use Public Key 2 to Recrypt encrypted bit c_1, obtaining $E_2(E_1(m_1))$.
4. Using *encrypted* Secret Key 1, evaluate decryption circuit homomorphically on $E_2(E_1(m_1))$.
5. The resulting ciphertext is $E_2(m_1)$, which now has a "fresh" noise associated with Public Key 2.

The main issue with this approach is that the decryption circuit may be too large. If the decryption circuit's depth is too large, then we may add too much noise before we can finish evaluating the decryption circuit. As a result, the bootstrappability condition requires the semi-homomorphic scheme to be able to compute circuits of larger depth than the decryption circuit. The decryption circuit can also be "squashed" (i.e., simplified such that its depth is lowered) by adding additional information to the public key. The squashing is done by adding a new set to the public key whose

secret subset sums to the original key. The bootstrappability concept was the key idea that made fully homomorphic encryption possible. Gentry's work was also followed by an implementation documented in [GH11]. For secure system parameters, the paper reported a public key size of 2.3 GB, 2.2 hours to generate public and private keys, 3 minutes to encrypt, and 30 minutes to refresh ciphertexts. Although the suggested solution was too slow to use in practice, Gentry's ideas were a great proof of concept for upcoming work in the field.

1.2.3 Fully Homomorphic Encryption over Integers

Following Gentry's breakthrough work, [vDGHV10] proposed a fully homomorphic encryption scheme using integers only (known as the DGHV scheme), as opposed to ideal lattices in [Gen09]. The idea was also described in Gentry's original work, but the authors in [vDGHV10] were able to obtain a bootstrappable scheme with "secure" parameters. A simplified symmetric version of their construction uses a large odd private integer p (which serves as the key), with pseudo-randomly generated integers q_i and r_i. To encrypt a bit m_i, evaluate:

$$c_i = pq_i + 2r_i + m_i \tag{1.1}$$

To decrypt, compute:

$$m_i = (c_i \mod p) \mod 2 \tag{1.2}$$

Such a simple setup allows homomorphic computations over addition and multiplication modulo 2. For example, computing the sum of two encrypted bits $c_i + c_j$

yields:

$$c_i + c_j = p(q_i + q_j) + 2(r_i + r_j) + (m_i + m_j) \mod 2 \tag{1.3}$$

The product of encrypted bits $c_i c_j$ is:

$$\begin{aligned} c_i c_j =& (pq_i + 2r_i + m_i) * (pq_j + 2r_j + m_j) \mod 2 \\ =& p(q_i q_j + 2q_i r_j + q_i m_j + 2q_j r_i + q_j m_i) \\ &+ 2(2r_i r_j + r_i m_j + r_j m_i) \\ &+ m_i m_j \mod 2 \end{aligned} \tag{1.4}$$

In both cases, the sum and product modulo 2 contain exactly one multiple of p and one multiple of 2, with $m_i + m_j$ or $m_i m_j$ as remaining terms. In this simple setup, the r_i values introduce noise, and in order for decryption to work, $|r_i| < |\frac{p}{2}|$. We can also see that for addition, noise grows by a factor of 2 ($r_i + r_j$), and for multiplication, the noise grows by a squared factor ($r_i r_j$). Note that the noise is actually ($c_i \mod p$), a step in the decryption process, which is then removed using a modulo 2 operation. Without the noise, the adversary's task would be to just solve for factors p and q_i using the GCD algorithm with two c_i bits as input, but adding r_i forces the ciphertext's integer value to be close to a multiple of p, but not exactly. Taking an encrypted bit $c_i = pq_i + 2r_i + m_i$, we can see that we are at a distance of $2r_i + m_i$ from the exact multiple pq_i. The security of this symmetric scheme relies on the Approximate GCD problem, which asks how one can recover p given many near-multiples of p. In an implementation of the DGHV scheme by the same authors of [vDGHV10], and in [NCMT11], the public key size has been reduced to 802MB for large parameters. This construction is extended and used as a public-key homomorphic cryptosystem,

which is discussed in detail in Section 2.1.

In [CNT12], J-S Coron et al. describe a public-key compression technique to dramatically reduce the public key size to only 10MB. The idea was to use a seed, which is stored in the public key, for generating large pseudo-random integers (labelled χ_i) as large as each ciphertext. Then compute:

$$\delta_i = (\chi_i - 2r_i - m) \mod p \tag{1.5}$$

δ_i is then stored in the public key. Since δ_i is modulo p, it is significantly smaller than the original ciphertext, since $q_i \gg p$. To easier visualize the significance of this reduction, ciphertexts under "secure" parameters are approximately $2 \cdot 10^7$ bits, where as p is roughly 2700 bits. To recover the original ciphertext, we re-generate the large pseudo-random integers using the seed, which was included in the public key, and compute:

$$c_i = \chi_i - \delta \tag{1.6}$$

Although the public key sizes were compressed down to 10MB, the implementation by J-S Coron et al. still required a time consuming 11 minutes to compute one ciphertext refresh procedure.

1.3 Our Contribution

In all FHE schemes described, there is an issue of circuit privacy. The server that evaluates a circuit C using encrypted input bits is aware of what C is. In some cases, we would like the functions evaluated over encrypted data to be also unknown to the server. To prevent an outside party of identifying operations that a server computed

from C, Gentry proposed in [Gen09] to compute a new encryption of 0, ψ, and add it to the output ciphertext bit c_i. The new ψ needs to have a large noise associated with it to the point where it is indeterminate whether the noise originates from c_i or ψ. However, the server is still aware of the steps it took to compute ciphertext bit c_i.

In our work, we combine digital logic theory and concepts from circuit design to create a 'virtual processor', which receives an encrypted circuit C, and processes it. In short, it would be ideal to be able to execute foreign Assembly code and return results, without ever knowing what the program does, as shown from Figure 1.1.

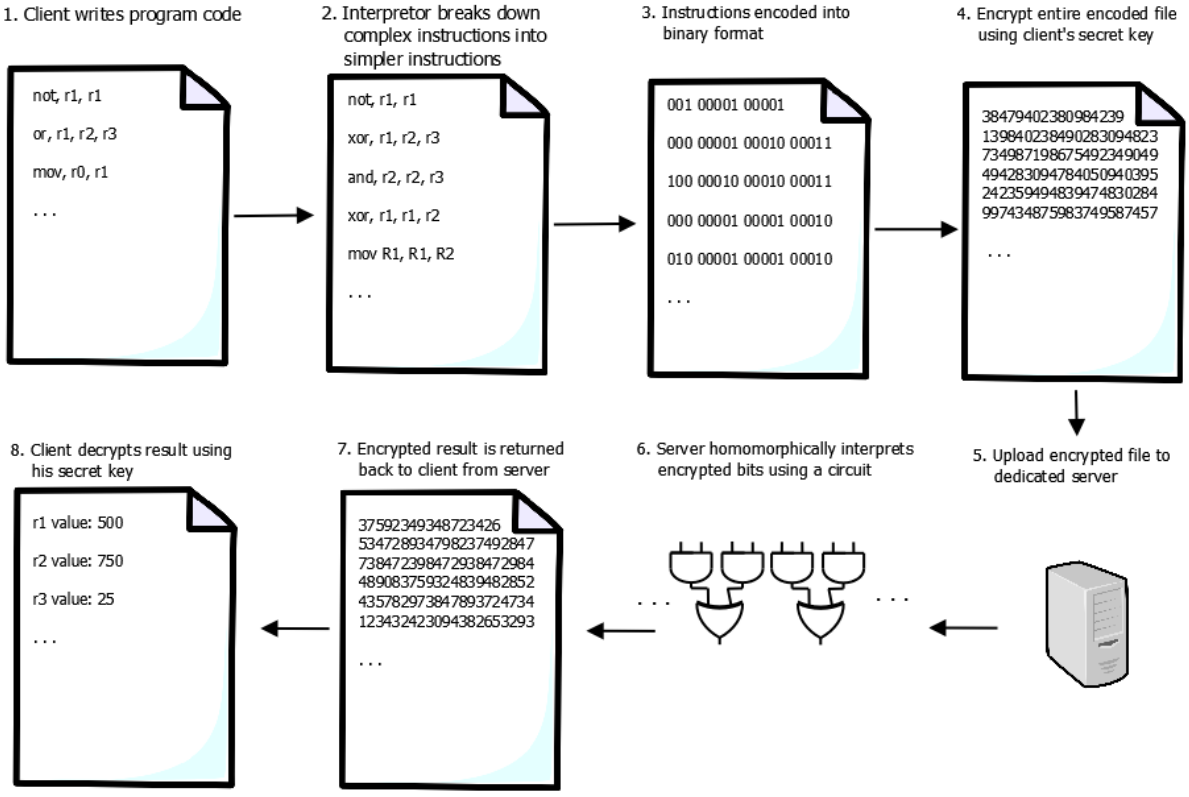

Figure 1.1: High level system overview

Suppose we do not have the resources or time to execute a long program, and would like to run it on a remote machine. However, we would prefer not to reveal

the details of our program, which is a problem. One suggestion to solve this problem is to have the remote server execute our program homomorphically. Therefore, the server has a fixed circuit, but the circuit is responsible for interpreting our program code. As one can imagine, such a circuit would be complex, and likely not feasible with current secure parameters for fully homomorphic encryption. In this thesis, we propose a structure in which a client-side interpreter can encode an Assembly program (construction based on the RISC architecture) into a binary circuit, encrypt it using FHE, and once sent to the server, the server can process it to return its output. This entire process is shown in Figure 1.1, where we propose solutions for steps 2,3, 6, and 8. Note that in this figure and in our solution, the resulting output is a memory dump of the server's registers, which is sent back to the client to decrypt.

Chapter 2

Background

2.1 Fully Homomorphic Encryption over Integers

2.1.1 Parameters

In this section, we provide a background on a fully homomorphic system over the integers, combining results presented in [vDGHV10] and [CNT12]. To be consistent with parameter symbols and notation, we use the following parameters to describe the FHE scheme:

- λ is the main security parameter, while other parameters are a function of λ
- x_i is an element in the public key, which is a set of many x_i elements
- γ is the bit length of x_i public key elements
- η is the bit length of the secret key, p
- τ is number of x_i elements in the public key

- ρ is the bit length of a noise element r_i used to generate x_i elements
- ρ' is the bit length of second noise parameter used in encryption

In the symmetric construction in Chapter 1, we used p for both encryption and decryption, where $c_i = pq_i + 2r_i + m_i$ and $m_i = (c_i \mod p) \mod 2$. This does not work because we cannot reveal what p is to a server, and we therefore need a public-key scheme. The general idea is to create a set of encryptions of zero (i.e., Encrypt(p, $\tilde{0}$) as the public key, and use only a subset of this key for encrypting m_i. The security of this therefore relies on an adversary not being able to guess which subset was used to encrypt m_i, also known as the Sparse Subset Sum (SSP) Problem.

2.1.2 Public Key Generation

The first step is a procedure called KeyGeneration(λ). From λ, we set up the values of the remaining security parameters described above. In [vDGHV10], the security parameter constraints were:

- $\rho = \omega \cdot (\log \lambda)$ to avoid brute force attacks on the noise
- $\eta \geq \rho \cdot \Theta \cdot (\lambda \log^2 \lambda)$ to allow a large enough modulus to support enough homomorphic operations when evaluating the decryption circuit
- $\gamma = \omega \cdot \eta^2 \cdot (\log \lambda)$ to prevent lattice-based attacks
- $\tau \geq \gamma \cdot \omega \cdot (\log \lambda)$ in order to reduce security problem to Approximate GCD (proof in [vDGHV10])
- $\rho' = \rho + \omega \cdot (\log \lambda) = 2\rho$ to obfuscate ρ values during encryption

The DGHV key generation algorithm is described fully in Algorithm 2.1.2. The procedure first computes τ encryptions of zero, given a random odd integer p, and generated integers q_i and r_i. These are not strictly encryptions of zero, because the noise is not guaranteed to be even (i.e., we output $pq_i + r$ and nor $pq_i + 2r$). This saves space and reduces public key size, and the factor of 2 for the noise gets added back during the encryption phase. The largest element in the public key is x_0 because any ciphertext $c_i \mod x_0$ will reduce the ciphertext by the largest amount. The second half of Algorithm 2.1.2 ensures that x_0 is odd and that $x_0 \mod p$ is even, in order for modular reduction to preserve the parity of encrypted bits m_i.

2.1.3 Encryption and Decryption

Once the public key is set up, encryption is straight forward. A function Encrypt is used that takes in the public key PK and a message bit m_i as input. Firstly, we need to obtain a random subset $S_{pk} \in \{1, 2, \ldots, \tau\}$ and a new noise offset r_i in the range $(-2^{\rho'}, 2^{\rho'})$. Then, we can compute the ciphertext bit c_i as:

$$c_i = \left(m + 2r_i + 2 \sum_{i \in S_{pk}} x_i \right) \mod x_0 \tag{2.1}$$

It is also not necessary to pre-generate vector S_{pk}. During the computation of c_i, for each public element PK_i, one can flip a coin and multiply PK_i by 0 or 1, which is equivalent to generating a random subset of PK_i on the fly.

To decrypt ciphertext c_i, we used the same function as mentioned in our template earlier, we let $m_i = (c_i \mod p) \mod 2$. An important observation here is that since p is odd, we can simplify the decryption expression into:

$$
\begin{aligned}
m_i &= (c_i \mod p) \mod 2 \\
m_i &= (c_i - p\lfloor\frac{c}{p}\rceil \mod 2) \\
m_i &= (c_i \mod 2) \oplus (\lfloor\frac{c}{p}\rceil \mod 2) \\
m_i &= LSB(c_i) \oplus LSB(\lfloor\frac{c}{p}\rceil)
\end{aligned}
\tag{2.2}
$$

2.1.4 Squashing the Decryption Circuit

This simplification is important because we still need to "squash" the decryption circuit, which essentially means that it needs to be simplified further to add minimal noise when computed homomorphically. In this case, the largest overhead is the $\lfloor\frac{c}{p}\rceil$ division term. To solve this problem, it would be convenient to have $\frac{1}{p}$ pre-computed and available to the server, but at the same time unknown to the server.

We can use a similar construction as the public key, but this time we need a "private key hint," which provides an easy way of computing $\frac{1}{p}$ and therefore avoiding the division from the $\lfloor\frac{c}{p}\rceil$ term. To do this, construct a set $\vec{y} = \{y_0, y_1, \ldots, y_{\Theta-1}\}$ such that $\sum_{i \in S} y_i \approx \frac{1}{p}$, where S is a sparse subset of $\vec{y}$ containing θ elements. θ is a new parameter which is originally set to λ, the security parameter of the scheme. Since $\vec{y}$ contains rational numbers, it would be convenient to normalize the elements in $\vec{y}$ to be in range $[0, 2)$. Therefore, in addition to the algorithm for key generation described earlier, we need to generate set $\vec{y}$:

1. Set $x_p \leftarrow \frac{2^\kappa}{p}$, where $\kappa = \frac{\gamma\eta}{\rho'}$.

2. Generate binary set $\vec{s} = \{s_0, s_1, \ldots, s_{\Theta-1}\}$ such that $s_i \in \{0, 1\}$ and the Hamming weight of $\vec{s}$ is θ (i.e., $\vec{s}$ contains θ 1's and $(\Theta - \theta)$ 0's).

3. Generate set $\vec{u} = \{u_0, u_1, \ldots, u_{\Theta-1}\}$ such that $u_i \in \mathbb{Z} \cap [0, 2^{\kappa+1})$ and $\sum_{i=0}^{\Theta-1} s_i u_i = x_p \mod 2^{\kappa+1}$.

4. Create a set $\vec{y} = \{y_0, y_1, \ldots, y_{\Theta-1}\}$ such that $y_i = \frac{u_i}{2^\kappa}$. Note that $\vec{y}$ can be represented as rational with numerators u_i and a common denominator 2^κ, which is just a power of 2 (requires only a bit-shift to compute).

5. Add $\vec{y}$ to the public key, in addition to S_{pk} generated before

Since we computed $y_i = \frac{u_i}{2^\kappa}$ in Step 4, we removed the original factor that was introduced from computing $\frac{2^\kappa}{p}$ in Step 1. The end result is that some subset of $\vec{y}$ adds up to approximately $\frac{1}{p}$. The elements of that subset are defined in $\vec{s}$, such that $\sum_{i=0}^{\Theta-1} s_i y_i \approx \frac{1}{p}$. Consequently, p is no longer needed as the private key since we can now use $\vec{s}$, and the new decryption function to recover plain text bit m from ciphertext c becomes:

$$m = c - \lfloor \sum_{i=0}^{\Theta-1} c s_i y_i \rceil \mod 2 \tag{2.3}$$

In the DGHV system, decryption can be simplified by storing ciphertexts as a pair $(c, \vec{z})$. This means that for every ciphertext computed in Evaluate, compute $z_i = c \cdot y_i \mod 2$ for all $0 \leq i \leq \Theta$, and store the pair $(c, \vec{z})$ as the ciphertext bit. For optimization purposes, the original authors in [vDGHV10] (Lemma 6.1, pg. 16) proved that it is enough to store only $n = \lceil \log \theta \rceil + 3$ bits of precision in z_i, while the remaining bits can be discarded. The final decryption function is therefore:

$$m = c - \lfloor \sum_{i=0}^{\Theta-1} s_i z_i \rceil \mod 2 \tag{2.4}$$

The security of the DGHV scheme described in this section relies on the Approximate GCD problem (AGCD). The Approximate GCD problem is to find p, given polynomially many near-multiples of p, i.e., polynomially many (in terms of λ) integers of the form $pq_i + r_i$. One way to do this is to brute-force the noise. Since we know that r_i is an integer in $(-2^\rho, 2^\rho)$, then for two ciphertexts c_1 and c_2, guess r_1 and r_2 and compute $p' = \gcd(c_1 - r_1, c_2 - r_2)$. If the number of bits in p' is η, then p' could be a solution. The overall number of guesses required is $2^{2\rho}$, which means that the noise needs to be large enough to withstand such attacks. Other attacks such as lattice attacks using the LLL reduction algorithm are also described in [vDGHV10], in Section 5.2 and in Appendix B.

2.2 Boolean Algebra and Circuits

Boolean algebra was first proposed by George Boole in 1854 and it is a form of algebra over base two numbers with domain $B \in \{0, 1\}$ [Boo03]. The fundamental binary arithmetic operators are conjunction ($\wedge$) or disjunction ($\vee$), and negation ($\neg$). Given a set S of boolean operators, the set is functionally complete if and only if all functions $f : B^n \to B$ can be expressed in terms of the operators in S. The standard set $S = \{\wedge, \vee, \neg\}$ contains the fundamental binary arithmetic operators, which as a set are also functionally complete. We also note that functions of zero arity ($f : B^0 \to B$) can be generated using constant inputs True or False to the negation function. The importance of functional completeness is that the primary

operators used in FHE schemes are exclusive or ($\oplus$) and conjunction ($\wedge$), and if we want servers to be able to compute arbitrary functions over data, our basic operators need to be functionally complete.

Given two binary elements x and y, the binary operation done to operands x and y is defined by a truth table. A truth table contains an enumeration of all possible input values of the binary operands (in this case x and y) and their corresponding truth values. Truth tables for the basic and relevant derived operators are shown in Table 2.1.

Algorithm 1 Public Key Generation for DGHV Scheme

$p \leftarrow \mathbb{Z} \cap (-2^{\eta}, 2^{\eta})$

if $(p \mod 2 = 0$ and $P < 2^{\eta} - 2)$ **then**
  $p \leftarrow p + 1$
end if

$r_i Max \leftarrow -2^{\rho}$
$r_i MaxIndex \leftarrow 0$

for $i = 0 \rightarrow (\tau - 1)$ **do**
  $q_i \leftarrow \mathbb{Z} \cap [0, \frac{2^{\gamma}}{p})$
  $r_i \leftarrow \mathbb{Z} \cap [-2^{\rho}, 2^{\rho})$

  $PK_i \leftarrow q \cdot p + r$

  if $PK_i > r_i Max$ **then**
    $r_i Max \leftarrow PK_i$
    $r_i MaxIndex \leftarrow i$
  end if
end for

$\text{Swap}(PK_0, PK_{r_i MaxIndex})$

if $(PK_0 \mod p) \mod 2 = 1$ **then**
  if $PK_0 > (2^{\gamma} - p)$ **then**
    $PK_0 \leftarrow PK_i - p$
  else
    $PK_0 \leftarrow PK_i + p$
  end if
end if

if $(PK_0 \mod 2 = 1)$ **then**
  $PK_0 \leftarrow PK_0 + 1$
end if

Table 2.1: Truth tables for common operators

$\boldsymbol{x}$	$\neg\boldsymbol{x}$
0	1
1	0

$\boldsymbol{x}$	$\boldsymbol{y}$	$\boldsymbol{x} \wedge \boldsymbol{y}$	$\boldsymbol{x} \vee \boldsymbol{y}$	$\boldsymbol{x} \oplus \boldsymbol{y}$	$\boldsymbol{x} \rightarrow \boldsymbol{y}$
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	1

We can prove functional completeness of the set $R = \{\wedge, \oplus\}$ by using its operators to reconstruct an existing functionally complete set. From Table 2.1, by equating truth table values we obtain the following relationships: $\neg x = x \oplus 1$ and $(x \vee y) = (x \oplus y) \oplus (x \wedge y)$. Thus from $\{\wedge, \oplus\}$ we have a new set $V = \{\neg, \wedge, \oplus, \vee\}$. Since $S \subseteq V$, we can conclude that V is also functionally complete, which by transitivity implies that the original set R and the basic homomorphic operations are functionally complete.

In general, n inputs contain 2^n truth values because there are 2^n possible combinations of inputs. For any particular boolean function $f_i \in f : B^n \rightarrow B$ with inputs $x_0, x_1, \ldots, x_{n-1}$, a minterm is a conjunction of input literals where f_i evaluates to a $\tilde{1}$. Similarly, a maxterm is a disjunction of input literals where f_i evaluates to a $\tilde{0}$. It then follows that any boolean function can be expressed as a sum of minterms (often called sum of products) or a product of maxterms (product of sums). For example, if $f_i = x \oplus y$, from Table 2.1 we can see that the minterms for f_i are $(x \wedge \neg y)$ and $(\neg x \wedge y)$, and the maxterms are $(x \vee y)$ and $(\neg x \vee \neg y)$. As a result, taking the product of the two minterms yields the entire function: $(x \wedge \neg y) \vee (\neg x \wedge y)$, which we know is the exclusive or (XOR) function.

2.3 Digital Logic and Circuit Design

Digital systems serve as an alternative to analog systems. Analog systems use voltage levels represented using real numbers to implement functionality, whereas digital systems use two discrete states $\tilde{0}$ and $\tilde{1}$. The first logical operations required electrical switching circuits, and were proposed by Charles Sanders Pierce in 1886 [2]. It is convenient to use two discrete states because we are able to import and apply boolean algebra theory to create advanced functionality in digital systems.

Fundamentally, digital logic can be subdivided into combinatorial logic and sequential logic. The key distinction between the two is that combinatorial logic allows the outputs to be immediately expressed as a boolean function in terms of the inputs, whereas sequential logic incorporates time dependencies. A good example would be the difference between the sum of two 4-bit numbers and a 4-bit digital counter. The sum of two numbers requires a combinatorial circuit whose output solely depends on a two input binary sequence, whereas the digital counter is continuously ticking, depending on its previous 4-bit sequence. Even though the digital counter is technically a sum of the previous input and 0001, it will continue to update on its own, because there is a direct connection between the output and the input of the circuit. Since the goal of FHE schemes is to apply a boolean function to encrypted data, combinatorial logic is the emphasis of this section.

In circuit design, the Institute of Electronic Engineers (IEEE) uses standardized symbols to denote logic gates, shown in Figure 2.1.

One of the most fundamental circuits used in circuit design is a multiplexer. A multiplexer in its simplest form is a sequential circuit that implements the boolean function $f_2(s, x_0, x_1) = (x_0 \wedge \neg s) \vee (x_1 \wedge s)$. Suppose s is assigned a value of $\tilde{0}$ as

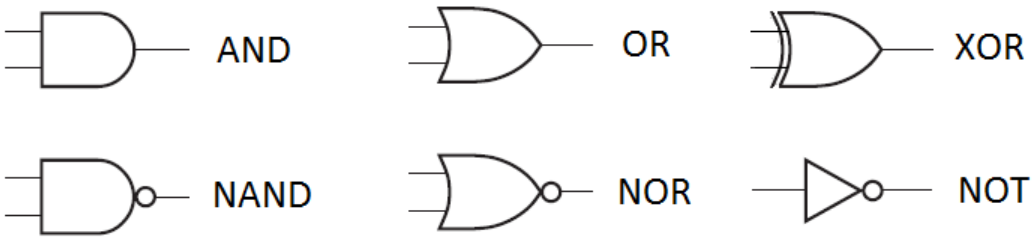

Figure 2.1: Standard Logic Gate Symbols used in Digital Circuit Design

input, then $(x_0 \wedge \neg s)$ evaluates to x_0 and $(x_1 \wedge s)$ evaluates to $\tilde{0}$. Similarly, if s is assigned a $\tilde{1}$, we can see by inspection that $f_2(s_0, x_0, x_1)$ simplifies to just x_1. Since the value of s simplifies $f_2(s, x_0, x_1)$ to x_0 or x_1, a multiplexer can be interpreted as a hardware equivalent of an *if else* statement. This multiplexer is commonly referred to as a 2:1 multiplexer because it has two inputs and one output.

There are many forms of multiplexers, often expressed in powers of two. Higher staged powers of two multiplexers can be built from lower staged ones recursively. For example, a 4:1 multiplexer implements the boolean function $f_4(s_0, s_1, x_0, x_1, x_2, x_3) = f_2(s_1, f_2(s_0, x_0, x_1), f_2(s_0, x_2, x_3))$. In general, the recursive relation to build an n:1 multiplexer is shown in Equation 2.5, where n is a power of 2, $m = \lceil \log n \rceil$, $0 \leq i \leq (m-1)$, $0 \leq j \leq (\frac{n}{2} - 1)$, and $\frac{n}{2} \leq k \leq n$.

$$f_n(s_0, \ldots, s_m, x_0, \ldots, x_n) = \begin{cases} (x_0 \wedge \neg s_0) \vee (x_1 \wedge s_0) & \text{for } n = 1 \\ f_2(s_0, f_{(n-1)}(\tilde{s_i}, \tilde{x_j}), f_{(n-1)}(\tilde{s_i}, \tilde{x_k})) & \text{for } n \geq 2 \end{cases} \tag{2.5}$$

Since multiplexers are a fundamental building block in implementing conditional logic, they also their own standard circuit symbol, which is shown in Figure 2.2. The diagram also provides a visual to clarify the recursive relation from Equation 2.5,

where a 4:1 multiplexer is built using three 2:1 multiplexers.

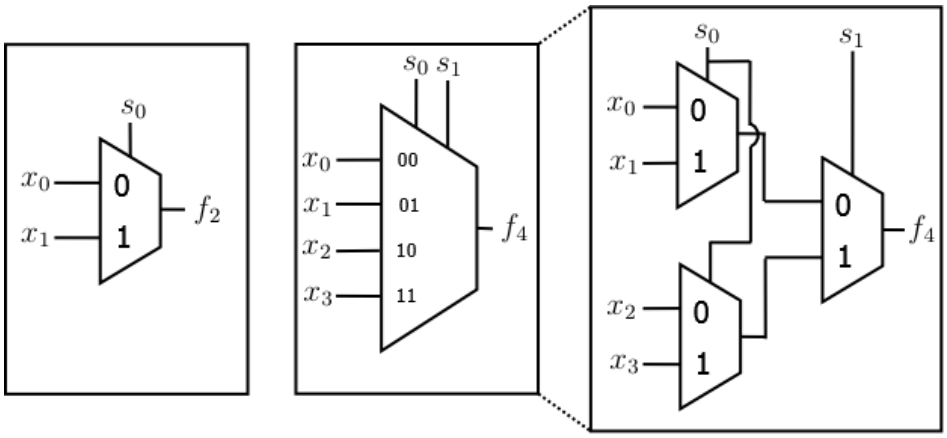

Figure 2.2: Standard circuit symbols for 2:1 and 4:1 multiplexers

2.4 RISC Architecture

2.4.1 RISC I Instruction Set

The RISC (Reduced Instruction Set Computing) architecture is a design approach for CPUs, originally coined by David Patterson, who had also contributed towards the design of RISC. The RISC project began in 1980, whose proposed architecture was published by David Patterson and Carlo H. Sequin in [PS81].

The RISC design approach is to use simple and optimized instructions, as opposed to the CISC architecture (Complex Instruction Set Computing), which supports more complex instructions that perform multiple primitive instructions at a time. One main advantage that RISC systems have over CISC architectures is that because the RISC instruction set contains less instructions, less dedicated hardware is required to execute their instructions. This design feature reduces cost and increases execution

speeds, at the expense of instruction complexity. Since current homomorphic systems have major performance issues, we focus on the instruction sets outlined in the RISC architecture, in order to minimize the number of homomorphic operations that need to be done per instruction.

The original RISC architecture contained 26 assembly instructions, shown in Figure 2.3 and described in [PS81]. All instructions, data, registers, and addresses were 32 bits long, although the data contained within registers was actually byte-addressable, and can be represented by 8, 16, or 32 bits. Each instruction contains one to three operands, depending on the type of instruction. Arithmetic instructions contain 3 operands: two source registers and one destination register, whereas load and store instructions only require a value (the value could be a literal value or a register value) and a target register. Lastly, jumps and branches also required two operands, a condition that needs to be true in order to jump and an offset to denote an address that we jump to. Unconditional jumps do not require a condition, and therefore use only one operand, which is the address of the instruction to jump to.

In the RISC architecture, op codes (operation codes) are used to determine what type of instruction we are executing. Since all instructions are converted into binary, each instruction has a corresponding op code, before processing the remaining bits of the instruction. The RISC instruction format is as follows:

OPCODE[0:6]	SCC[7]	DEST[8:12]	SRC1[13:17]	IMM[18]	SRC2[19:31]

Instruction bits $b_0, \ldots, b_6$ are dedicated for the opcode, which means that the RISC I architecture supports upto $2^7 = 128$ possible instructions. At first it seems illogical because we only have 26 instructions, but since we only needed 26 bits to store the instruction operands, the extra bits were added to the op code. Furthermore, more

Instruction	Operands	Comments	
ADD	S1,S2,Rd	Rd ← S1 + S2	integer add
ADDC	S1,S2,Rd	Rd ← S1 + S2 + carry	add with carry
SUB	S1,S2,Rd	Rd ← S1 - S2	integer subtract
SUBC	S1,S2,Rd	Rd ← S1 - S2 - carry	subtract with carry
AND	S1,S2,Rd	Rd ← S1 & S2	logical AND
OR	S1,S2,Rd	Rd ← S1 | S2	logical OR
XOR	S1,S2,Rd	Rd ← S1 xor S2	logical EXCLUSIVE OR
SLA	S1,S2,Rd	Rd ← S1 shifted by S2	shift left arithmetic
SRA	S1,S2,Rd	Rd ← S1 shifted by S2	shift right arithmetic
SLL	S1,S2,Rd	Rd ← S1 shifted by S2	shift left logical
SRL	S1,S2,Rd	Rd ← S1 shifted by S2	shift right logical
LDL	(Rx)X,Rd	Rd ← M[Rx+X]	load long
LDSU	(Rx)X,Rd	Rd ← M[Rx+X]	load short unsigned
LDSS	(Rx)X,Rd	Rd ← M[Rx+X]	load short signed
LDBU	(Rx)X,Rd	Rd ← M[Rx+X]	load byte unsigned
LDBS	(Rx)X,Rd	Rd ← M[Rx+X]	load byte signed
STL	Rm,(Rx)X	M[Rx+X] ← Rm	store long
STS	Rm,(Rx)X	M[Rx+X] ← Rm	store short
STB	Rm,(Rx)X	M[Rx+X] ← Rm	store byte
JMP	COND,X(Rm)	pc ← X+Rm	conditional jump
JMPR	COND,Y	pc ← pc + Y	conditional relative
CALL	Rm,X(Rn)	Rm ← pc, next pc ← X+Rn, CWP−	
CALLR	Rm,Y	Rm ← pc, next pc ← pc + Y, CWP−	
RET	Rm,X	pc ← Rm+X, CWP++	
GTLPC	Rm	Rm ← last pc	get last pc
GTIN	Rm	Rm ← INR	get interrupt number

Figure 2.3: RISC I Instruction Set

op code bits for fewer instructions allows for optimization for instruction decoding, since only 26 of 128 possible bit combinations are used. The SCC bit, b_7 is used as a flag for condition codes in conditional branch instructions, and the IMM bit, b_{18} is another flag that determines whether the instruction uses an immediate value or a register value. If $b_{18} = 1$, then the bits for SRC2, $b_{19}, \ldots, b_{31}$ represent a sign extended 13-bit literal value. Otherwise, if $b_{18} = 0$, then retrieve only the first 5 bits of SRC2 $(b_{19}, \ldots, b_{24})$ as a register address. One implicit feature of this instruction format is that any arithmetic instruction type is capable of taking an immediate value as an operand.

2.4.2 The Classic RISC Pipeline and MIPS Adaptation

The life cycle of one instruction goes through a five-stage pipelined, coined as the classic RISC pipeline. An example layout of a slightly modified architecture, built based upon the classic RISC architecture, is shown in Figure 2.4. During the first stage, Instruction Fetch (IF), a 32 bit instruction is loaded from the instruction cache, which is loaded with program instructions at the beginning of execution. During this stage, a program counter (PC) is also incremented by 4 in order to obtain the address of the next instruction. Since a 32 bit instruction contains 4 bytes, the PC is a byte offset from the base address of the first instruction in the cache. It is not always the case that PC is incremented by 4. For example, branch instructions can update the PC value to a new location in instruction memory. In more advanced pipeline implementations, the IF stage also contains "Branch Prediction" algorithms, where the algorithm predicts whether PC should be incremented by 4, or whether it should be added an off-set. We omit branch prediction algorithms and implementations in this section, but additional material on the topic for interested readers can be found in [YP91].

Once an instruction is loaded from memory and has entered the pipeline, the instruction needs to be interpreted, which is done in the Instruction Decode (ID) stage. The purpose of the ID stage is to prepare all data values for computational work. Register data is fetched from the register file, which is dedicated memory for run-time execution, consisting of 32 registers, each of size 32 bits in the RISC architecture. For instance, if instruction bits $b_{13}, \ldots, b_{17}$ were 00010, then the ID stage will query the register file, take the bits from register 2, and forward them to the next stage in the pipeline. This process is done for both source registers, if needed.

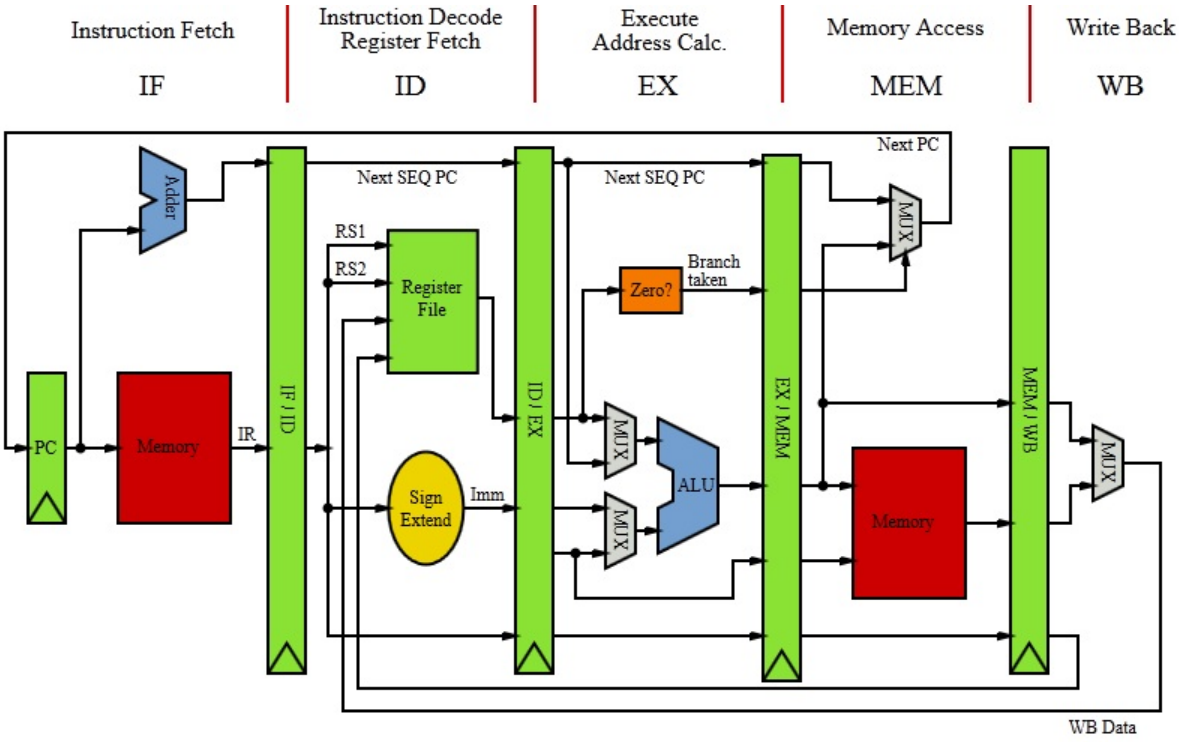

Figure 2.4: The Generic MIPS Pipeline

In the cases where the IMM bit is a 1 (i.e., b_{18}), the ID stage is also responsible for sign-extending the immediate operand $(b_{19}, \ldots, b_{31})$ so that the Arithmetic Logic Unit (ALU) in the next stage operates on 32-bit numbers. The type of instruction is also forwarded to the next stage, and is determined by the instruction's op code.

The Execution (EX) stage performs the main computations required for each instruction. Within the EX stage, the ALU unit is responsible for arithmetic operations, including computing the new address of the PC, if the current instruction is a branch instruction. Note that the ID stage previously determined whether the instruction is a branch instruction or an arithmetic instruction. The EX stages determines whether the branch is taken or not by doing a zero check on the bit received from the ID stage, and computing the address to be stored in PC at a later stage. If a branch is indeed

taken, the inputs to the ALU are the PC register, and the branch offset. Otherwise, the inputs to the ALU are retrieved register values or literal values for arithmetic instructions such as add, xor, etc. If the instruction is a load or store instruction, the ALU would compute the address for the next (MEM) stage.

The Memory (MEM) stage accesses memory only if the instruction wants to load data from memory to put it in a register, or if the instruction wants to store data from a register to memory. Otherwise, for arithmetic operations involving two registers, the MEM stage forwards the ALU results to the next stage.

Lastly, the final Write Back (WB) stage is responsible for modifying the register file. Two possible sources of data are stored in registers: data from arithmetic operations that were computed in the EX stage, or data fetched from memory (i.e., load instructions) obtained in the MEM stage. The register file is then modified and the five stages repeat for each instruction.

One of the purposes of the 5-stage pipeline is to allow for executing multiple instructions at the same time. For instance, if Instruction 1 (labelled I_1) is fetched in the IF stage, sent over to the ID stage, the IF stage is once again idle. Since each stage has dedicated hardware, then while I_1 is in the ID stage, the IF stage could continue to fetch a second instruction I_2 while ID is working on I_1. This effect is shown in Figure 2.5 for multiple instructions.

One major issue that the pipeline needs to account for are data and control hazards. Data hazards occur when a fetched instruction requires the register data of a currently executing instruction. For example, if $I_1 = add, R_1, R_2, R_3$ and $I_2 = add, R_5, R_1, R_6$, then I_2 requires the value for R_1 before instruction I_1 can finish. Control hazards are caused by branching instructions. For instance, if I_3 is a branch

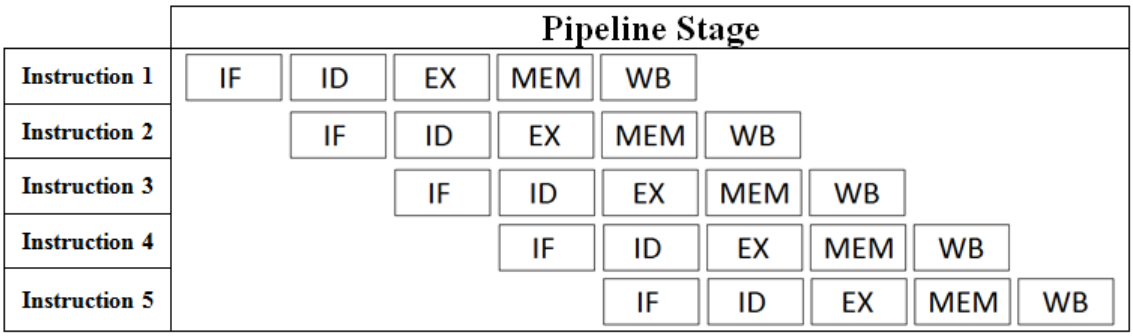

Figure 2.5: The RISC/MIPS Pipeline

instruction, then I_3 computes the branch address in the EX stage. Before I_3 reaches the EX stage, the IF stage has already retrieved the following instruction. However, this new instruction received by the IF stage is not the correct instruction because we would need to branch first before fetching the next instruction. Control hazards can also occur when branch prediction algorithms predict incorrectly. The most common data hazard resolution method is to forward data directly from the EX and MEM stages from I_1 to I_2 if I_2 needs register data before I_1 completes the WB stage, rather than making I_2 wait for I_1 to complete the WB stage. Similarly, when a control hazard occurs and an instruction gets mistakenly fetched, the instruction needs to get "flushed" out of the pipeline (i.e., ignored). For additional and more advanced methods for hazard detection and resolution, the interested reader is directed to [HP12].

Chapter 3

Contribution

3.1 Circuit Encoding Protocol

The purpose of a circuit encoding protocol is to provide a way of encoding circuits. In this case, we are limiting our scope to the binary encoding of circuits. A convenient encoding method would allow us to then encrypt an encoded circuit, send it to a server, and the server would have its own circuit to interpret and evaluate the encrypted and encoded circuit (described in 3.2). As a result, the only requirement from the encryption algorithm is that it must be fully homomorphic.

3.1.1 Brief Overview

One crucially important consideration for the protocol is determining operation priority. For example, suppose Alice wishes Bob to compute a simple function $f(A, B) = (A \wedge B) \vee (A \wedge \neg B)$. It is clear by inspection that negations and conjunctions are prioritized over disjunctions, but once the circuit is encrypted, Bob

does not know which operation to do first. This impacts the protocol because Bob can use a complex circuit for determining the order of operations in $f(A, B)$, but if we embed the order of operations as part of the protocol, then Bob would not require such a circuit.

Given the same $f(A, B)$ above, another consideration is the representation of the function. For example, $f(A, B)$ could also be transformed into a directed acyclic graph (DAG), encoded as an adjacency matrix, and Bob would receive an encrypted adjacency matrix, along with an encrypted vector whose elements represent the nodes (or operations). Another approach could be to describe the circuit in a programmable sequence. Then we could describe $f(A, B)$ in a similar manner as we would with most programming languages today:

1. $\langle$A$\rangle$
2. $\langle$B$\rangle$
3. $\langle$not, 2$\rangle$
4. $\langle$and, 1, 2$\rangle$
5. $\langle$and, 2, 3$\rangle$
6. $\langle$or, 4, 5$\rangle$

In this case, there is a significant resemblance with the Assembly programming language, where each line includes a $\langle g, i, j\rangle$ instruction, where a binary operation g is applied to lines i, j. A large drawback of this method is memory management, because for instance, line 1000 could potentially require line 1's output. The main attraction however, is that this approach allows us to break down complicated tasks

into simple bitwise operations. This method is also extendible to support subroutines to implement more complex instructions, provided that those instructions can be expressed in terms of our defined instructions.

3.1.2 Syntax and Semantics

From Alice's perspective, the most convenient method would be a programmable approach. If we could program a sequence of instructions, then simply encrypt and send the sequence to Bob, her work load would be minimized, as the server would do all of the processing. For this reason, a programming language such as Assembly could be simplified and applied here.

To explain our approach precisely, we will begin with single-bit binary numbers, and then extend our method to support and perform operations with multiple bit lengths (namely 16, 32, and 64, as these are most commonly used in practice today). The methods can be easily extended because the basic bitwise operators could be used as routines to support multiple-bit operations.

The basic instruction set is shown in Table 3.1. Every instruction and destination register is assigned a binary code, which is then encrypted and sent to the server for processing.

3.1.3 Instruction Encoding

Many of the instructions listed in Table 3.1 are straight forward to encode. In particular, source and destination registers can be the binary encoding of the register number. However, because the server is homomorphic, it would be extremely difficult to deal with variable bit lengths. For example, the address of R6 could be encoded as

Table 3.1: Basic instruction set for 1 bit instructions

Instruction	**Arguments**	**Example**	**Description**
li	$Dest, Data$	li, R1, 1	load data bit into destination register
mov	$Dest, Src1$	mov, R1, R2	copy contents from Src1 to Dest registers
not	$Dest, Src1$	not, R1	negate bit located in source register and store to destination register
xor	$Dest, Src1, Src2$	xor, R1, R2, R3	xor bits in Src1 and Src2 registers, store result in Dest register
and	$Dest, Src1, Src2$	and, R1, R2, R3	and bits in Src1 and Src2 regsters, store result in Dest register
or	$Dest, Src1, Src2$	or, R1, R2, R3	or bits in Src1 and Src2 registers, store result in Dest register

110, but R3's address would be 11. As in many Assembly-based systems, there are only a fixed amount of registers available. If the number of registers was 32, as it is in the MIPS instruction set, then the binary encoding of the address of R3 needs to be padded to become 00011.

Fully homomorphic cryptographic systems innately support modulo two addition (exclusive or) and multiplication (logical and). Due to the functional completeness of $\{\oplus, \wedge\}$, any other binary operation can be constructed using $\oplus$ and $\wedge$. During server-side computations, it would be much more convenient for the server to only be sent encrypted $\oplus$ and $\wedge$ operations. As a result, the interpreter for our miniature Assembly language needs to express all other binary operations in terms of $\oplus$ and $\wedge$, as shown in Table 3.2.

After converting all non XOR, AND instructions, each type of instruction needs

Table 3.2: Run-time Instruction Translations

Instruction	**Run-time Translation**
not, R1, R1	xor, R1, R1, 1
or, R1, R2, R3	xor, R1, R2, R3 and, R2, R2, R3 xor, R1, R1, R2

to be assigned an operation code for the compiler to understand. For simplicity, we will use a 3-bit op code, gs_0s_1, where g denotes whether an XOR or AND need to be computed, and s_0s_1 are used for determining if the current instruction is a one or two operand instruction, a load, or a negation.

Table 3.3: Instruction Op Codes

Instruction	**Op Code**
xor	000
and	100
not	001
mov	010
li	011

Combining Table 3.3 and using 5 bits to represent register addresses, we can convert supported instructions from Table 3.1 into binary. For example, we can encode a sample xor instruction $xor, R1, R2, R3$ into the binary sequence:

000 00001 00010 00011

However, the binary representation of the client's program still needs to be encrypted, and then sent to the server for processing.

3.2 Evaluating Encrypted Circuits

3.2.1 Interpreting Circuit Elements

At compile time, we have shown that we could replace any operation with exclusive OR and logical AND because the set $\{\oplus, \wedge\}$ is functionally complete. Since fully homomorphic cryptographic systems evaluate these operations only, we encrypt gate $g \in \{\oplus, \wedge\}$. The server is then able to evaluate the encrypted gate for one bit inputs using a simple circuit given in Figure 3.1. Given $f(A, B) = A \wedge B$, if the server needs to compute $E(f(A, B))$, because it does not know which gate $E(g)$ represents (although as a reader, we know it is $\wedge$), it needs to pre-compute both $E(A) \wedge E(B)$ and $E(A) \oplus E(B)$, and then select the appropriate result based on the value of $E(g)$. These homomorphic operations add noise to the encrypted bits and for more detailed noise analysis, please see Section 3.2.3.

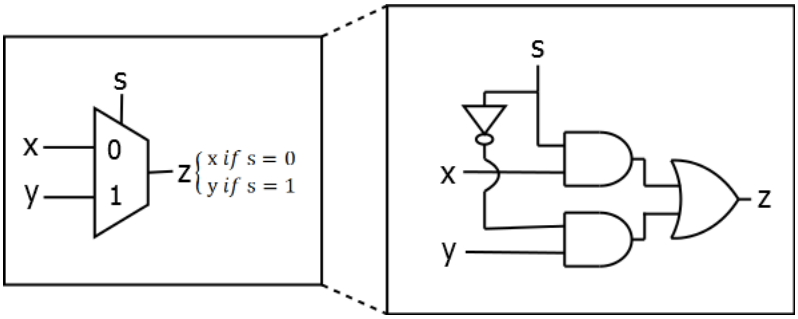

Figure 3.1: 2:1 Multiplexer used for evaluating $\oplus, \wedge$ operations

This method uses fully homomorphic encryption features by simulating a digital circuit at the bit level. A 2:1 multiplexer is used in this case so that the select line is the bit corresponding to the encrypted gate, i.e., $s = E(g)$ for $g \in \{0, 1\}$, and our two possibilities from above are $x = E(A) \oplus E(B)$ and $y = E(A) \wedge E(B)$.

3.2.2 Memory Management and Addressing

One of the challenges stated in Section 3.1.1 was that some of the initial methods proposed had no proper way of managing memory. However, we could query memory locations using a similar construction from Section 3.1. The routine can be re-used and cascaded to create a multiplexing network for memory addressing. A

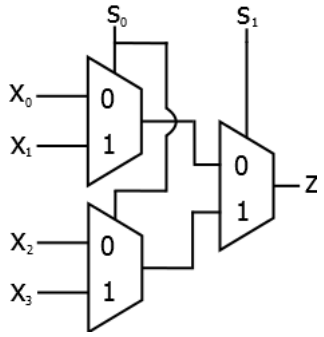

Figure 3.2: Sample 4:1 Multiplexer used for querying memory addresses

4:1 multiplexer shown in Figure 3.2 is enough to access 4 memory locations (namely x_0, x_1, x_2, x_3 in the diagram) given a two bit address (s_0 and s_1). This construction can be extended to create larger circuits that in general support 2^n memory locations given by $x_0, \ldots, x_{2^n}$, with an n bit address, $s_0, \ldots, s_n$.

Although we can access source register values and obtain their values fairly easily, writing values requires more work. Since the server computing the encrypted instructions does not know which register needs to be modified, it needs to make a decision for each register whether the register should maintain its old value, or obtain a new value. This problem is solved by cycling through all encodings of our registers, and creating additional logic to compare register values with the encrypted instruction's destination register. From the server's perspective, it is recomputing every register value, but it does not know whether the register value has changed.

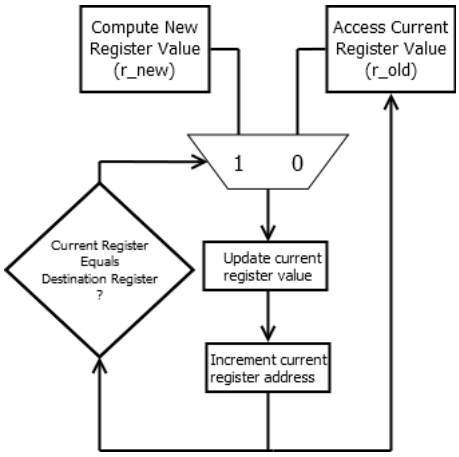

Figure 3.3: Writing to destination registers

Based on the schematic presented in Figure 3.3, we need an additional 5-bit adder circuit for incrementing register addresses (to account for all 32 registers), as well as a comparator, in order to compare the current register address with the destination register address. Accessing old register values to obtain r_{old} is done the same way as before.

However, r_{new} is not always the result of an XOR or AND instruction. In many cases, the programmer would like to load a fixed literal value into the register, or they would like to even copy the contents from one register to another. This can be accommodated by adding one additional multiplexer, which computes different results depending on the instruction op code bits, gs_0s_1. The bits gs_0s_1 serve as an op code and their purpose is to distinguish the type of instruction determined by the interpreter. As a result, Figure 3.4 shows how the op code bits are used with a 4:1 multiplexer by selecting one of the pre-computed r_{new} outputs.

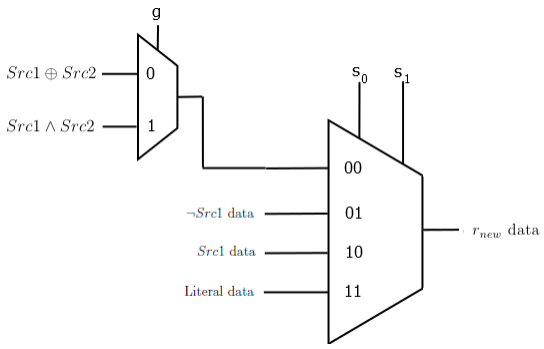

Figure 3.4: Computing destination register data (r_{new})

3.2.3 Supporting 32 Bit Instructions

Currently, our implementation supports only one bit register widths. Ideally, we would like our system to support at least 32 bit operations. In hardware, this can be done by layering multiplexers in parallel, but since we are simulating multiplexer functionality, we could achieve the same effect by sequentially looping our circuit evaluation over each bit, using an array of bits dedicated to each register.

However, our implementation for bitwise instructions can be parallelized. The currently supported bitwise operators perform computations for each bit independently, and we can take advantage of multi-core CPUs to speed up the process. When obtaining register values from memory, querying for each individual bit is an independent operation, and can be done in parallel. This process is shown in Figure 3.5. In addition to querying, bitwise instructions such as and, xor, or mov are also parallelizable, and a similar scheme is used to implement their functionality.

$$\text{add, R3, R13, R7 } \underbrace{0111}_{\text{Op Code}}\ \underbrace{00011}_{\text{Dest}}\ \underbrace{\underbrace{01101}_{\text{Src1}}\ \underbrace{00111}_{\text{Src2}}\ 000000}_{\text{Immediate Bits}} \tag{3.1}$$

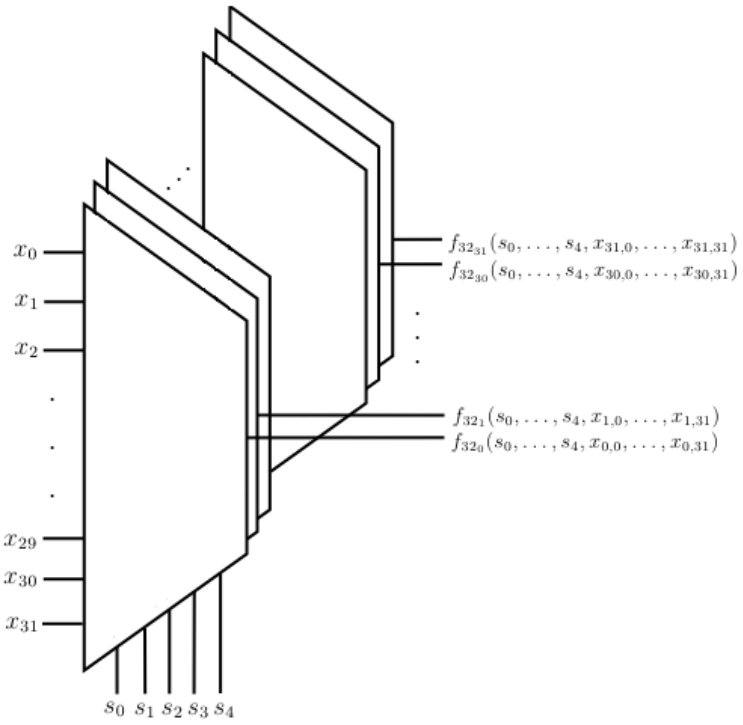

Figure 3.5: Parallel multiplexing for querying each register bit

Chapter 4

Conclusions and Future Work

Our original goal was to build a circuit capable of interpreting encrypted assembly language instructions. This circuit should be stored on the server, and a client would encrypt their program, and send it to the server for processing. Although the server is capable of simulating Assembly program code homomorphically, we have encountered several limitations, namely:

1. Executing branch instructions
2. Simulating dynamic looping behaviour (Related to branching)
3. Reading and Writing Files (File I/O)
4. These limitations imply that our implementation is not Turing complete (computationally universal)

In all of these cases, the root cause of the problem is an innate property of homomorphic operations: we do not know which instructions we are executing. At any given instruction, it is possible that it is a branch instruction, or a file I/O operation,

and trying to take these possibilities into account will create an extremely complex homomorphic circuit. In practice, even with currently optimized FHE cryptosystems, our implementation still remains as a proof of concept because our circuits are still relatively large, resulting in slow execution speeds.

To address branch instructions, a couple of solutions are possible. The naive approach would be that for every instruction in our program code, consider the possibility of a branch instruction. To accomodate this change, we propose implementing a program counter, and the program counter would update at every instruction. Consequently, if the instruction is a branch instruction, the program counter will still update accordingly. The difficult part is that once we update the program counter, we would need to re-execute the entire program code, searching for the appropriate line of code, just in case the current instruction was a branching instruction. In short, a possible modification is shown in Figure 4.1.

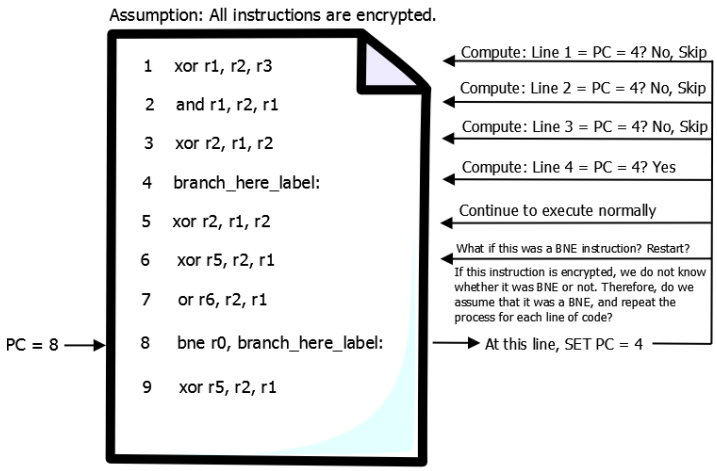

Figure 4.1: Using a program counter for branching instructions

There are several issues with this approach. Firstly, re-executing the entire program code for one instruction is extremely inefficient, and not scalable for large programs. Secondly, the original problem still remains unsolved, because if we always jump to the beginning of the program code, the entire program does not know when to terminate. Note that in Figure 4.1, "Skip" does not actually mean skip the instruction because the server does not know this. The end result of "Skip" is that the entire Register File remains unchanged, but to do this, we need to re-load all of the registers with their old values by following the same procedure shown in the previous section, in Figure 3.3. Although this idea illustrates the problem very well, it does not seem to be a possible solution.

A related issue is implementing looping behaviour. More specifically, conditional loops are an issue for the same reason as branching. When evaluating the loop condition, the server does not know when to stop looping, and where the beginning of the loop is, unless it repeats the same inefficient process shown in Figure 4.1. Once again, the server would still not know when to terminate the program.

A better solution is to support "static" loops. A loop is a static loop if and only if the number of iterations in the loop is determined before compile time. In this case, from the client's machine, during the instruction breakdown phase, we can perform a loop unrolling procedure shown in Figure 4.2. A side-effect of this method is that the server does not need to even consider loops, because the client adds additional instructions manually when interpreting program code.

Lastly, our instruction set has no access to memory. If such an instruction existed, the same problem persists: the server itself does not know whether a file needs to be created or read from memory. Even with knowledge to create a file in memory

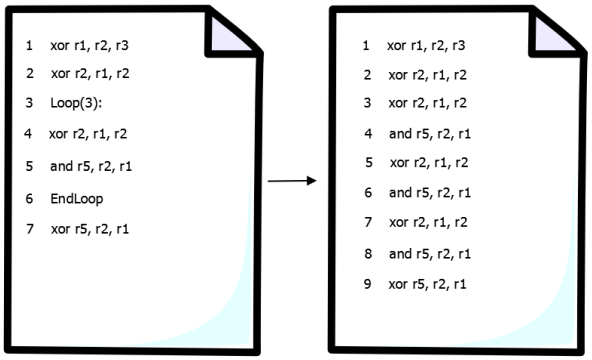

Figure 4.2: Static Loop Unrolling - Loops are not visible to the server

or not, a secondary problem of knowing when to write data to memory still exists. The main issue is that writing or reading contents to a file is not a homomorphic operation. Supporting such functionality would require a way of alternating between homomorphic operations, and regular operations such as memory reads and writes, which currently does not seem possible.

At this time, we have no solutions or suggestions for homomorphically executing dynamic loops (i.e., conditional branching) and File I/O, and leave these features as open problems for future work.

Appendix A

Selected Sample Code

A.1 Key Generation

```
/* Generate pseudo-random odd integer p */
public static void Private_KeyGen(RC4 RC4_NumberGen, out BigInteger key)
{
     //Generate pseudo-random key of size Eta bits
     key = RC4_NumberGen.GetRandWord(Eta);
     if(key % 2 == 0) key = key + 1;
}

/* Generate public key set S using odd integer p */
public static void Public_KeyGen(RC4 RC4_NumberGen,
     List<BigInteger> key_public, BigInteger key_private, int Tau)
{
     int index_max = 0;
```

```
BigInteger maxEncZero = 0;
BigInteger currentEncZero = 0;
BigInteger temp = 0;

for (int i = 0; i < Tau; i++)
{
    // Generate an encryption of zero
    currentEncZero = EncryptZero(RC4_NumberGen, key_private);

    // Add it to the public key
    key_public.Add(currentEncZero);

    // Keep record of maximum and it's location
    if(currentEncZero > maxEncZero)
       {
           maxEncZero = currentEncZero;
           index_max = i;
       }
}

// Swap first element with maximum element
temp = key_public[0];
key_public[0] = key_public[index_max];
key_public[index_max] = temp;
```

```
}


/* Compute 1/P Approximation to generate public key help list */
public static void Public_KeyHelp_Gen(RC4 RC4_NumberGen,
     List<BigInteger> key_decrypt_helper, BigInteger key_private,
     List<int> key_pub_indexMap, int Theta, int Gamma)
{
     // Generate pseudo-random new secret key vector subset_sum
     BigInteger subset_sum = RC4_NumberGen.GetRandWord(Theta);

     // Compute the number of 1's in the random vector
     int hamming_weight = 0; int temp;
     for (int i = 0; i < Theta; i++)
     {
           // Take current bit and compute hamming weight
           temp = (int)((subset_sum >> i) & 1);
           if (temp == 1) hamming_weight++;

           // Record the bit
           key_pub_indexMap.Add(temp);
     }

     // Normalize each element, by 2^40
```

```
 // if 40 = bit-length of the public key elements
 // e.g. #1 40 = (bitsize p (30) + bitsize q (8) + 2)
 BigInteger normalize_by = 1;

for (int i = 0; i < (Gamma + 2); i++) normalize_by = normalize_by << 1;

//Compute normalized p size e.g. 40bit#/30bit# ~ 10-11bit#
BigInteger normalized_p = normalize_by / key_private;
BigInteger norm_error_p = (normalize_by / normalized_p - key_private);

// Create a list of size hamming weight, so that sum(list) = normalized_p
List<BigInteger> subset_sum_p = new List<BigInteger>();
BigInteger sum_list = BigInteger.Zero;

// Compute how many bits we need to represent normalized_p
// & round down (MINUS 1 BIT)
// We round down because we want to guarantee that
// subset_sum <= normalized_p
int p_bits = (int)Math.Log((double)(normalized_p), 2);

// Compute max number from p_bits, and divide by # of elements required
// Round down again, because we do NOT want to overshoot random values
int max_per_element = (((int)Math.Pow(2, p_bits) - 1) / hamming_weight);
```

```
// Compute number of bits per element needed
// (round down again for same reason)
int bit_size_per_element = (int)Math.Log(max_per_element, 2);

// We also want to compute the bit size for generating non-1/p elements
// This is the average of all bit-sizes needed for 1/p elements + 1
// (for those >average)
int bit_size_non_p_elements = 0;

for (int i = 0; i < hamming_weight; i++)
    {
        // Last iteration of loop, add last element, and redistribute weights
        if (i == (hamming_weight - 1))
        {
            // Average EXCLUDING last element
            BigInteger average = sum_list / (hamming_weight - 1);

            // Make the last element the average of elements
            // 0 - hamming_weight-1
            subset_sum_p.Add(average);

            // Compute remaining amount to distribute
            BigInteger distribute_total = normalized_p - average - sum_list;
            BigInteger distribute = distribute_total / hamming_weight;
```

```
            BigInteger remainder = distribute_total % hamming_weight;

            // To each element, add amount distributed
            for (int j = 0; j < hamming_weight; j++)
            {
                subset_sum_p[j] = subset_sum_p[j] + distribute;

                // Add up bit sizes required to store 1/p elements
                bit_size_non_p_elements +=
                   ((int)Math.Log((double)subset_sum_p[j], 2) + 1);
            }

            // The remainder of the division goes to the last element
            subset_sum_p[hamming_weight - 1] += remainder;
            bit_size_non_p_elements +=
               (int)Math.Log((double)subset_sum_p[hamming_weight - 1], 2);

            // Compute average bit size required for non 1/p element
            bit_size_non_p_elements =
            (bit_size_non_p_elements / hamming_weight) + 1;
          }
        else
          {
            // Add a random element to the list
```

```
                subset_sum_p.Add(RC4_NumberGen.GetRandWord(bit_size_per_element));
                sum_list = sum_list + subset_sum_p[i];
            }
        }

    // Generate entire subset sum that (with and without 1/p elements)
    int index = 0;

    // Add the power of 2 used to normalize the key as the first element
    key_decrypt_helper.Add(normalize_by);

    for (int i = 0; i < Theta; i++)
    {
        // If this number contributes to 1/p, add it
        if (key_pub_indexMap[i] == 1)
        {
            // Add next element of subset_sum_p and increment its location
            key_decrypt_helper.Add(subset_sum_p[index++]);
        }

        // Otherwise, generate some new random number
        else
        {
            // Note: This number needs to have a MINIMUM
```

```
            // Otherwise, an attacker can eliminate low numbers
            // from subset immediately
            key_decrypt_helper.Add(RC4_NumberGen.GetRandWord(
                  bit_size_non_p_elements));
         }
     }
}
```

A.2 Encryption

```
/* Create an encryption of bit '0' */
public static BigInteger EncryptZero(RC4 RC4_NumberGen, BigInteger key,
     int Rho, int Gamma, int Eta)
{
     //Generate pseudo random value q
     BigInteger q = RC4_NumberGen.GetRandWord(Gamma - Eta - 1);

     //Generate pseudo random value r
     BigInteger r = RC4_NumberGen.GetRandWord(Rho);

     BigInteger bit_c = 2 * (key * q + r);
     return bit_c;
}

public static BigInteger EncryptBit(RC4 RC4_NumberGen,
```

```
     List<BigInteger> key_public, int bit_m)
{
     int index1, index2;
     BigInteger sum = BigInteger.Zero;

     //Find #bits needed to represent public key size
     int keysize_bits = (int)Math.Log(key_public.Count, 2);

     //Assumption: public key size can be a maximum of 32-bits
     index1 = (int)RC4_NumberGen.GetRandWord(keysize_bits);
     index2 = (int)RC4_NumberGen.GetRandWord(keysize_bits);

     //Make sure that index1 is always smaller
     if (index1 > index2)
     {
          int temp = index1;
          index1 = index2;
          index2 = temp;
     }

     //Add up elements from random index1 to random index2
     for (int i = index1; i < index2; i++)
          sum = sum + key_public[i];
```

```
    //Generate pseudo random value r
    BigInteger r = RC4_NumberGen.GetRandWord(5);

    BigInteger bit_c = (2*r + 2 * sum + bit_m) % key_public[0];
    return bit_c;
}
```

A.3 Decryption

```
/* Decryption of bit c using the private key p for testing*/
public static BigInteger DecryptBit(BigInteger key, BigInteger bit_c)
{
    return (bit_c % key) % 2;
}

/* Decryption of bit c using the private key s */
public static BigInteger DecryptBit(List<int> s_i, List<BigInteger[]> enc_bits,
    BigInteger key_private, List<BigInteger> key_decrypt_helper)
{
    // Local counters
    BigInteger si_zi = 0; BigInteger total_remainders = 0; int round_up = 0;

     */
       For each private key bit s_i {0,1}, multiply by encrypted data
       and remainders. Some data/remainders are useless, so only a
```

```
        "subset" of the data is what we want /*

     for (int i = 0; i < s_i.Count; i++)
        {
            si_zi += s_i[i] * enc_bits[i + 1][0];
            total_remainders += s_i[i] * enc_bits[i + 1][1];
        }

     // Determine if we need to round up or down our division
     if (total_remainders % key_decrypt_helper[0] >= (key_decrypt_helper[0] >> 1))
        round_up = 1;

     // Add all remainder terms to s_i * z_i
        si_zi = si_zi + (total_remainders / key_decrypt_helper[0] + round_up);

        return (enc_bits[0][0] - si_zi) % 2;
}
```

A.4 Sample Add Circuit

```
/* Code snippet of a binary addition circuit */

     //Convert encryption of zero from string to BigInteger
     BigInteger encrypt_Zero = BigInteger.Parse(encryptZero);
```

```
//Read operand bits from file
StreamReader inFile = new StreamReader(file);

//First line
total = inFile.ReadLine().Split('#');

List<String> currentLineList;

//Compute initial value of sum
for (int i = 0; i < total.Length; i++)
    sum.Add(new BigInteger(Convert.ToInt64(total[i])));

while ((currentLine = inFile.ReadLine()) != null)
{
    carry_in = 0;
    //currentLineNum = currentLine.Split('#');
    currentLineList = new List<String>(currentLine.Split('#'));
    List<BigInteger> LineNum = new List<BigInteger>();

    //Zero-extend currentLine until it's the same size as sum
    while (currentLineList.Count != sum.Count)
        currentLineList.Add(encryptZero);

    //Loop over each bit to compute next sum & carry bits
```

```
        for (int i = 0; i < currentLineList.Count; i++)
        {
            LineNum.Add(new BigInteger(Convert.ToInt64(currentLineList[i])));
            addTemp = (sum[i] + LineNum[i]);
            multTemp = (sum[i] * LineNum[i]);

            sum[i] = carry_in + addTemp;
            carry_in = (addTemp * carry_in + multTemp) +
                       (addTemp * carry_in * multTemp);

            //Reduce ciphertext bit sizes
            carry_in = carry_in % encrypt_Zero;
sum[i] = sum[i] % encrypt_Zero;
        }
       sum.Add(carry_in);
    }

   //Combine sum bits in one string
   for (int i = 0; i < sum.Count - 1; i++)
       {
           result += sum[i] + "#";
       }
   result += sum[sum.Count - 1].ToString();
```

```
    inFile.Close();
}
```

Bibliography

[AD97] Miklós Ajtai and Cynthia Dwork. A public-key cryptosystem with worst-case/average-case equivalence. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 284–293, New York, NY, USA, 1997. ACM.

[Boo03] George Boole. *An Investigation of the Laws of Thought*. Prometheus Books, 2003.

[CF85] Josh D. Cohen and Michael J. Fischer. A robust and verifiable cryptographically secure election scheme. In *Foundations of Computer Science, 1985., 26th Annual Symposium on*, pages 372–382, 1985.

[CNT12] Jean-Sbastien Coron, David Naccache, and Mehdi Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 446–464. Springer Berlin Heidelberg, 2012.

[Elg85] T. Elgamal. A public key cryptosystem and a signature scheme based

on discrete logarithms. *Information Theory, IEEE Transactions on*, 31(4):469–472, 1985.

[Gen09] Craig Gentry. A fully homomorphic encryption scheme. Ph.d. thesis, Stanford University, September 2009.

[GH11] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *EUROCRYPT*, pages 129–148, 2011.

[GM82] Shafi Goldwasser and Silvio Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, STOC '82, pages 365–377, New York, NY, USA, 1982. ACM.

[HP12] John Hennesy and David Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier Publishers (Morgan-Kaufmann), 2012.

[NCMT11] David Naccache, Jean-Sbastien Coron, Avradip Mandal, and Mehdi Tibouchi. Fully homomorphic encryption over the integers with shorter public keys. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference*, volume 6841 of *Lecture Notes in Computer Science*, page 483. Springer, 2011.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology EUROCRYPT 99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer Berlin Heidelberg, 1999.

[PS81] David A. Patterson and Carlo H. Séquin. Risc i: A reduced instruction set vlsi computer. In *ISCA*, pages 443–458, 1981.

[RAD78] R. Rivest, L. Adleman, and M. Dertouzos. On data banks and privacy homomorphisms. pages 169–177. Academic Press, 1978.

[vDGHV10] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. June 2010.

[YP91] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture*, MICRO 24, pages 51–61, New York, NY, USA, 1991. ACM.