8-2014

Homomorphic Encryption and the Approximate GCD Problem

Nathanael Black
Clemson University, nblack@g.clemson.edu

Homomorphic Encryption and the Approximate GCD Problem

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Mathematical Science

---

by
Nathanael David Black
August 2014

Accepted by:
Dr. Shuhong Gao, Committee Chair
Dr. Elena Dimitrova
Dr. Matthew Macauley
Dr. Gretchen Matthews

Abstract

With the advent of cloud computing, everyone from Fortune 500 businesses to personal consumers to the US government is storing massive amounts of sensitive data in service centers that may not be trustworthy. It is of vital importance to leverage the benefits of storing data in the cloud while simultaneously ensuring the privacy of the data. Homomorphic encryption allows one to securely delegate the processing of private data. As such, it has managed to hit the sweet spot of academic interest and industry demand. Though the concept was proposed in the 1970s, no cryptosystem realizing this goal existed until Craig Gentry published his PhD thesis in 2009.

In this thesis, we conduct a study of the two main methods for construction of homomorphic encryption schemes along with functional encryption and the hard problems upon which their security is based. These hard problems include the Approximate GCD problem (A-GCD), the Learning With Errors problem (LWE), and various lattice problems. In addition, we discuss many of the proposed and in some cases implemented practical applications of these cryptosystems.

Finally, we focus on the Approximate GCD problem (A-GCD). This problem forms the basis for the security of Gentry's original cryptosystem but has not yet been linked to more standard cryptographic primitives. After presenting several algorithms in the literature that attempt to solve the problem, we introduce some new algorithms to attack the problem.

Table of Contents

List of Tables

List of Figures

Notation

Symbol	Explanation
$\mathbb{F}$	Denotes any finite field
$\mathbb{F}_p$	Denotes the finite field with p elements
$\mathbb{N}$	Denotes the set of natural numbers
$\mathbb{R}$	Denotes the set of real numbers
$\mathbb{R}^n$	Denotes the set of n-dimensional vectors where each component is an element of $\mathbb{R}$
$\mathbb{Z}$	Denotes the ring of integers
$\mathbb{Z}_p$	Denotes the ring of integers modulo p
$\mathbb{Z}_p^n$	Denotes the set of n-dimensional vectors where each component is an element of $\mathbb{Z}_p$
$\mathbf{x}$	Denotes a vector
$\mathbf{x}_i$	Denotes the i-th vector in an indexed set
$\mathbf{x}_{[i]}$	Denotes the i-th component of the vector $\mathbf{x}$
$\|\mathbf{x}\|$	Denotes the norm of $\mathbf{x}$, assumed to be the standard Euclidean norm
$\|\mathbf{x}\|_p$	Denotes the p-norm of $\mathbf{x}$
$\langle \mathbf{x}, \mathbf{y} \rangle$	Denotes the inner product of $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, given by $\sum_{i=1}^{n} x_i y_i$
$\det(M)$	Denotes the determinant of the matrix M
$\lfloor a \rceil$	Denotes rounding a to the nearest integer
$a \mod p$	Denotes reducing a modulo p into the interval $(-p/2, p/2]$
$\|a\|_{\text{bin}}$	Denotes the binary size of a (i.e. the number of digits in the binary representation of a)

Symbol	**Explanation**
$a_{(i)}$	Denotes the i-th bit in the binary representation of a
$\log(a)$	Denotes the base 2 logarithm of a

Chapter 1

Introduction

In the last 40 years, the computer and the internet have fundamentally changed the way data is stored and transferred. During this time, many cryptosystems have been proposed to protect this digital data. Now, the rapid adoption and widespread use of cloud computing is posing a new challenge. Can data be stored and processed remotely without compromising privacy? A solution to this problem is homomorphic encryption.

A cryptosystem is a scheme for securing data. Unsecured data is called a plaintext or message, and secured data is called a ciphertext. An encryption function E and an encryption key K are used to secure data and a decryption function D and a decryption key k are used to retrieve data. The functions E and D are such that for the correct keys and a plaintext m.

$$D\big(E(m)\big) = m$$

When the key for encryption and decryption are the same (i.e. $K = k$), the cryptosystem is said to be a symmetric key system. If person A wants to send data to

person B in a secure fashion using this type of system, they must agree on the key in advance. In many cases, like transmission over the internet, this may be impossible. To accommodate these situations, two separate keys are used: a public key (K) for encrypting data and a private key (k) for decrypting data, and the system is called a public key cryptosystem.

In their 1978 paper [43], Rivest, Adleman, and Dertouzos proposed the idea of homomorphic cryptosystems. Intuitively, a cryptosystem with encryption function E and plaintext messages m_1 and m_2 is homomorphically correct for an operation $\star$ on the plaintext space and corresponding operation $\odot$ on the ciphertext space if

$$E(m_1) \odot E(m_2) = E(m_1 \star m_2). \tag{1.1}$$

Basically this means that applying the operation $\odot$ to the encrypted data is the same as applying the operation $\star$ to the original unencrypted data and then encrypting the result. Thus, in some sense, the order of application does not matter between applying the encryption function and applying the operation. One can encrypt the data and then apply $\odot$ or one can apply $\star$ to the data and then encrypt the result. Both ciphertexts will decrypt to $m_1 \star m_2$. Since operations may be performed after data is encrypted, a homomorphic system allows a user to hand off the processing of private data to an untrusted party. This is done by encrypting the data, sending it out for processing, and then decrypting the result that is returned. The untrusted party only deals with ciphertexts so the privacy of the data is maintained.

The homomorphic property (1.1) is appropriate for a deterministic cryptosystem, a system where each plaintext corresponds to exactly one ciphertext. However, in a cryptosystem where a plaintext message has many different encryptions due to randomness in the encryption process (i.e. a probabilistic system) we revise the

definition so that the cryptosystem is homomorphically correct if

$$D\big(E(m_1) \odot E(m_2)\big) = D\big(E(m_1 \star m_2)\big) = m_1 \star m_2. \tag{1.2}$$

This is due to the fact that $E(m_1) \odot E(m_2)$ is one of many encryptions of $m_1 \star m_2$ so we just insist that decrypting $E(m_1) \odot E(m_2)$ yields $m_1 \star m_2$. In fact, for a probabilistic system it is almost always the case that $E(m_1) \odot E(m_2) \neq E(m_1 \star m_2)$.

In a cloud computing application that utilizes homomorphic encryption, an untrusted party will be given access to the encryption of all the private data. If the encryption algorithm is deterministic they might be able to recognize patterns in the encrypted data which reveal information about the underlying plaintext data. To ensure that this does not occur, a probabilistic system should be used. A probabilistic cryptosystem is secure if the following properties hold.

1. Given a message m and a ciphertext c, it is computationally impossible to determine if c is an encryption of m.

2. Given two plaintext messages $m_1 \neq m_2$ and a ciphertext c that is the encryption of one of them, it is computationally impossible to determine whether c is an encryption of m_1 or m_2. (indistinguishability of chosen plaintexts)

A natural extension of a cryptosystem that is homomorphically correct for a single operation is one that is correct for the evaluation of a function (or a class of functions). Normally, one might compute

$$FUNCTION(parameters, data) \rightarrow results$$

but would like to do so homomorphically

$$FUNCTION\Big(E(parameters), E(data)\Big) \to E(results).$$

For example, if we have plaintext messages in $\mathbb{F}_2$ (i.e. bits) and ciphertexts which are integers, then the following diagram provides an overview of homomorphically evaluating a function f. The function $\hat{f}$ is a related function on the ciphertext space which depends on f and possibly the keys k and K, E_K is the encryption function, and D_k is the decryption function.

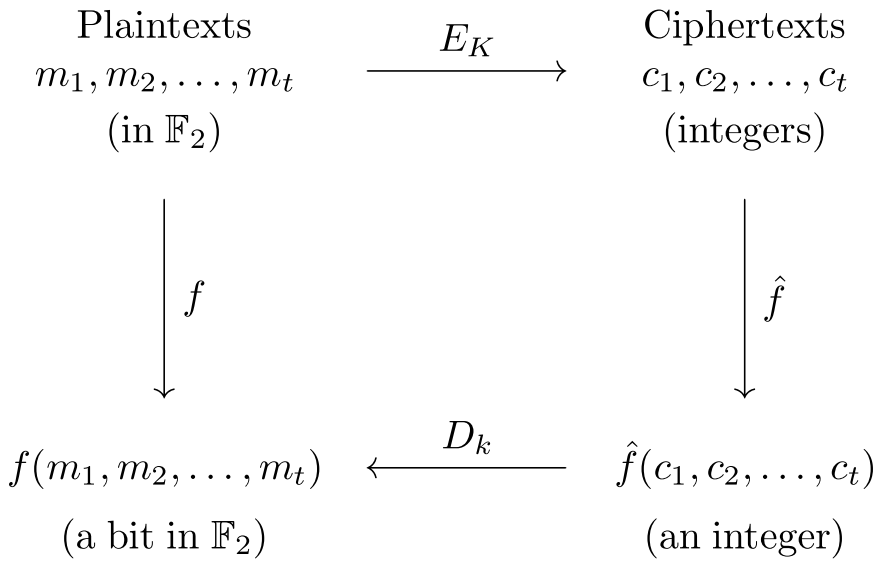

Figure 1.1: Homomorphic Encryption Overview

A cryptosystem that is homomorphically correct for every function on the plaintext space is said to be a fully homomorphic system (FHS), while systems that are correct for only some operations are known as partially homomorphic systems (PHS). If a cryptosystem is homomorphically correct for any type of operation, but can only handle a limited number of applications of an operation it is called a somewhat homomorphic system (SHS).

To obtain a fully homomorphic cryptosystem may seem like an unattainable goal at first glance, but note the following. Functions that can be evaluated on a computer can be represented as a Boolean circuit. Over a finite field, Boolean operations can be represented as an expression involving only addition and multiplication operations over that field. Consequently, any cryptosystem which is additively and multiplicatively homomorphic can homomorphically evaluate any function and will thus be a fully homomorphic system. Oftentimes a cryptosystem can provide homomorphic addition (Pallier cryptosystem) or homomorphic multiplication (ElGamal or RSA) but not both simultaneously.

The process of taking a function and creating a Boolean circuit that corresponds to it is an active research area. There are often several circuits which correspond to the same arithmetic expression. For example, consider the "majority of three" function, $MAJ(x_1, x_2, x_3)$, which takes as input three bits and returns 1 if two or more of the arguments are 1s and 0 otherwise. The circuits shown below, which correspond to $x_1x_2 + x_1x_3 + x_2x_3 = x_1(x_2 + x_3) + x_2x_3$ and $(x_1 + x_2)(x_1 + x_3) + x_1$ respectively are both valid circuits for the majority function, but the second one involves only a single multiplication instead of the two required for the first one. This example is taken from [6].

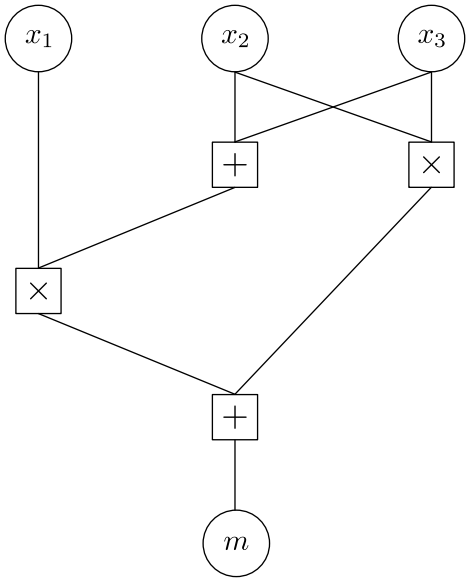

Figure 1.2: Majority Circuit 1

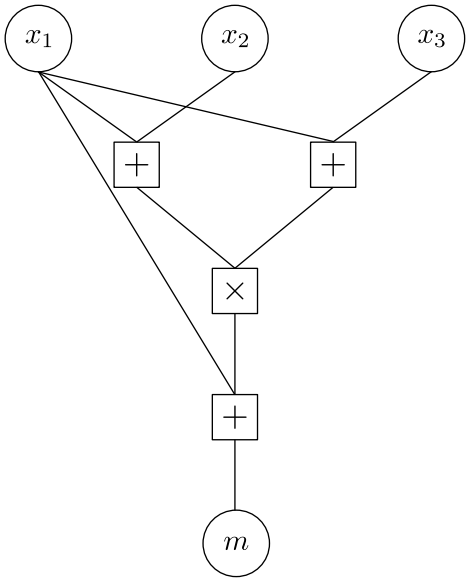

Figure 1.3: Majority Circuit 2

The "best" possible circuit representations of a function is dependent on the application (often the one with the fewest number of multiplications) and in some cases it is not even clear as to what the best representation is.

There are some minor qualifications on the premise that any function that can be run on a computer can be represented as a Boolean circuit. For example, the $WHILE$ construct is a common feature of many mainstream programming languages which executes its body until the test condition is satisfied. The exact number of times the body will be executed is unknown until runtime and thus would require a circuit of unbounded depth to represent it. So, to represent a function including this construct as a circuit one must provide an upper bound on the number of executions of the $WHILE$ loop. This is similar to the approach needed to run programs on a Turing machine that does not have random access. In fact, when designing a circuit to represent a function no optimizations at all may be made based on information about the data being processed since the point of homomorphic encryption is to process data while it is encrypted.

A fully homomorphic cryptosystem allows one to securely delegate the processing of private data. Typically, a user will first encrypt some private data, then hand the encrypted data over to some untrusted third party which will perform the desired computations on the encrypted data. To this third party, the result they obtain from this computation is nonsensical, but when the user receives it back they decrypt it to obtain the result of applying that same computation to the original unencrypted data. In a sense, the user has outsourced the computation of the data while maintaining the security of the data.

There are several desired properties of such a system. First, the privacy of the data should be maintained. The party doing the homomorphic computation should not be able to learn anything (or at least not anything that the user considers to be private) about the data they are processing. Second, since the point of using homomorphic encryption is to off-load the work involved in processing the data to someone else, the user should not do more work in recovering the result of the homomorphic evaluation than he would have otherwise incurred in performing the computations himself on the unencrypted data. This property is referred to as ciphertext compactness and basically requires that the size of the ciphertext does not grow with the complexity of the function being evaluated. Third, the scheme should be efficient in terms of space. Like any cryptosystem, the ciphertext expansion (i.e. the ratio of the size of the ciphertext to the size of the underlying plaintext) should be small.

A typical homomorphic encryption system consists of the four functions shown below.

Function	**Input**	**Output**
KeyGen	security parameter (λ)	secret key (k) and public key (K)
Encrypt	plaintext (m) and public key (K)	ciphertext (c)
Re-crypt	a noisy ciphertext c for m	a fresh ciphertext $\bar{c}$ for m
Decrypt	ciphertext (c) and secret key (k)	plaintext (m)

Table 1.1: Functions in a Homomorphic Cryptosystem

Creating a fully homomorphic system poses several challenges. First, designing a system that exhibits the homomorphic properties discussed above and can be proven to be secure is quite difficult. Second, any function that is to be homomorphically evaluated needs to be represented as a Boolean circuit, and finding a good circuit representation is often hard. Third, all of the homomorphic cryptosystems that have been proposed so far are impractical in terms of the time it takes to process data and the size of the ciphertexts and keys. In Chapter 2, we define the hard problems that are used as a basis for many homomorphic systems. Next, in Chapter 3, we examine the two main families of homomorphic cryptosystems as well as functional encryption. Then, in Chapter 4, we survey current and potential applications for homomorphic encryption. Finally, in Chapter 5, we consider the approximate GCD problem and propose some new algorithms to attack it.

Chapter 2

Hard Problems

The security of any cryptosystem is based on a problem that is hard to solve. These problems are known as security primitives. There are a couple of ways in which a problem may be hard to solve. First, it may be hard in an information-theoretic sense. This means that using the information provided to an attacker there is no way to distinguish between a correct or incorrect solution to the problem. Second, it may be hard in a computational sense. In this case, a solution is easily verified as being correct, but finding that solution even using advanced algorithms and significant amounts of computing power is challenging. While information-theoretic problems are the stronger of the two, there are very few of them and so many cryptosystems are based on computationally hard primitives.

An example of a problem that is hard to solve in the information-theoretic sense is recovering a plaintext message that has been encrypted with a one-time pad. In a one-time pad encryption system the plaintext, m, is a binary string and the key, k, is a binary string that is used only once to encrypt a plaintext. The encryption of m is simply the binary string $c = m \oplus k$. Given the ciphertext, c, the attacker can always find a key, $\bar{k}$, such that the ciphertext could be the encryption of any possible

message, $\bar{m}$, since if $\bar{k} = \bar{m} \oplus c$ then $\bar{m} \oplus \bar{k} = c$. Since every pair $(\bar{m}, \bar{k})$ is a valid plaintext-key pair producing the ciphertext, c, the attacker cannot determine m or k.

An example of a problem that is hard to solve in the computational sense is factoring a number, n, which is the product of two large primes. There are known algorithms for factoring, but none of them run in time that is polynomial in the number of bits of n. So by choosing the bit size of the prime factors to be large enough, the attacker will be unable to factor n in a reasonable amount of time using as much computational power as he has available at his disposal.

Computationally hard problems are susceptible to advances in technology. First, as advancements are made in computer hardware the size of these problem instances may need to be increased to maintain the same level of practical security. Second, improvements in algorithms may also necessitate an increase in the problem parameters. Many computational problems are assumed to be hard because no one has yet discovered an efficient way to solve them, but that does not rule out the possibility of someone coming along and finding an efficient way to do so. Third, many computationally hard problems are assumed to be hard in the classical sense. This means that a practical implementation of a quantum computer would make these problems no longer hard. An example of this is Peter Shor's algorithm for factoring numbers using a quantum computer. Problems, that are not susceptible to algorithms on a quantum computer are said to be post-quantum primitives.

The complexity of algorithms that attempt to solve these hard problems are measured in terms of the total number of bit operations as a function of the size of the input parameters. Algorithms are said to run in polynomial time if the number of bit operations needed to solve a problem is a polynomial function of the size of the input parameters. Though the degree of the polynomial may be large and the coefficients may be huge, these algorithms are considered to have efficiently solved

the problem. An algorithm is said to run in exponential time if the number of bit operations is an exponential function of the size of the input parameters. Though the exponent may be tiny and the coefficients may be quite small, these algorithms are not considered to have efficiently solved the problem since as the parameter sizes increase the running time will eventually pass that of any polynomial time algorithm.

We now describe some of the hard problems on which the security of cryptography schemes are based. For each problem, the best known algorithm for solving it is exponential in the problem size.

2.1 The Approximate GCD Problem

Problem 1 (Approximate GCD Problem (A-GCD))
Given a set of m integers of the form $x_i = q_i p + r_i$ where $q_i, p, r_i \in \mathbb{Z}$ and q_i and r_i are chosen randomly from some distribution (possibly different distributions), find p.

The size (in binary) of r_i is smaller than that of p and the size of p is much smaller than the size of q_i. Often, q_i is chosen uniformly at random from the set of all integers of a fixed size in binary and r_i is chosen to be non-zero from a Gaussian distribution centered at zero with a maximum size in binary. If two or more of the r_i's are 0, then by computing the GCD of all m^2 pairs (x_i, x_j) with $i \neq j$ one can recover p. For this reason, the extra stipulation that $r_i \neq 0$ is added.

Currently, there are no algorithms in the literature which can solve this problem in time that is polynomial in the binary size of q_i, p, and r_i. Oftentimes, a slightly relaxed version called the Partial Approximate GCD Problem (pA-GCD) is used. In this variant, $x_0 = q_0 p$ is a distinguished clean multiple of p. Though it seems to be easier, none of the known algorithms can solve even this variant in polynomial time. In Chapter 5, we will discuss some new algorithms to attack this problem.

2.2 Lattice Problems

Definition 1 (Lattice)

A lattice L of dimension n is a set of points in $\mathbb{R}^n$ that is closed under addition. This set of points is generated by taking all possible integer combinations of the vectors in some linearly independent set $B = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$ called a basis of the lattice, namely,

$$L = \left\{ \sum_{i=1}^{n} \alpha_i \mathbf{v}_i | \alpha_i \in \mathbb{Z} \right\}.$$

A lattice has many bases. For example, L can be generated by $\{\mathbf{b}_1, \mathbf{b}_2\}$ or $\{\mathbf{d}_1, \mathbf{d}_2\}$.

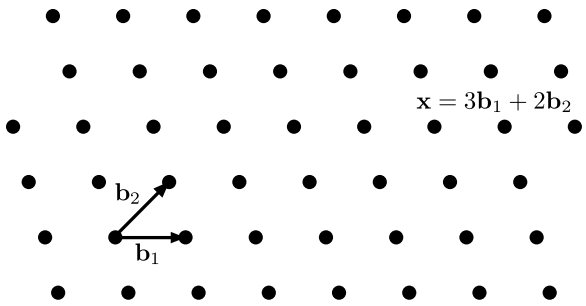

Figure 2.1: Lattice Basis $B = \{\mathbf{b_1}, \mathbf{b_2}\}$ for L

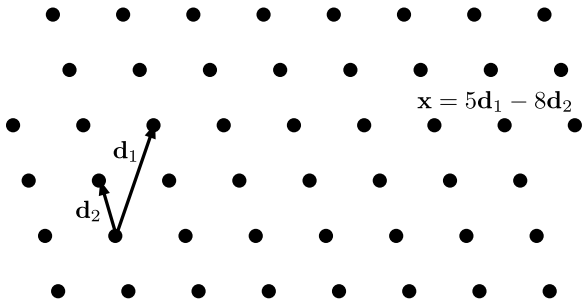

Figure 2.2: Lattice Basis $D = \{\mathbf{d_1}, \mathbf{d_2}\}$ for L

Definition 2 (Lattice Volume)

Let $B = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$ be a basis for a lattice L, and define each entry of a matrix M as $M_{i,j} = \langle \mathbf{v}_i, \mathbf{v}_j \rangle$. The volume of L is $\mathrm{vol}(L) = |\det(M)|$, and will be the same for any basis of L.

Since a lattice is obtained by taking integer combinations of finitely many vectors, there is at least one non-zero vector of minimum size. In the basis D above, it appears that $\mathbf{d}_2$ is a short vector, but it may not be a shortest vector if $\mathbf{b}_1$ or $\mathbf{b}_2$ (or some other vector in L) is shorter.

Definition 3 ($\lambda_n(L)$)
The length of a set of vectors in L is the size of the largest vector in the set. For a lattice L, $\lambda_n(L)$ is the minimum length among all sets of n linearly independent vectors in L. Thus, $\lambda_1(L)$ is the size of a smallest nonzero vector in L.

A classic result known as Minkowski's Theorem [37] provides a bound on the smallest size of a non-zero vector in a lattice.

Theorem 1 (Minkowski's Theorem)
For a lattice L with dimension n, $\lambda_1(L) < 2^n \cdot \mathrm{vol}(L)$.

Generally, a basis composed of short and mostly orthogonal vectors is desired as many computations on lattices are easier for such a basis. To obtain such a basis, one can utilize the celebrated LLL algorithm by Lenstra, Lenstra, and Lovász [30]. This algorithm uses the Gram-Schmidt orthogonalization process and rounding to transform a basis for a lattice into a shorter, more orthogonal basis.

The two main hard problems for lattices are as follows.

Problem 2 (Shortest Vector Problem (SVP))
Given a basis for a lattice L of dimension n, output a nonzero vector $\mathbf{v} \in L$ with length $\lambda_1(L)$ (i.e. find a shortest nonzero vector).

This problem was shown to be NP-hard in [50]. From the introduction in [27], this problem can be solved with a deterministic algorithm having complexity $n^{n/2+o(n)}$ see [28] or using a probabilistic algorithm with complexity $2^{O(n)}$ see [2].

Problem 3 (Closest Vector Problem (CVP))

Given a basis for a lattice L of dimension n and a vector $\mathbf{t} \in \mathbb{R}^n$, output a vector $\mathbf{v} \in L$ such that $\|\mathbf{t} - \mathbf{v}\| \leq \|\mathbf{t} - \mathbf{u}\|$ for all $u \in L$.

This problem was shown to be as hard as solving SVP in [24], so it is also NP-hard. It can be solved with an algorithm having complexity $\tilde{O}(2^{2n})$ see [36].

There is no known algorithm for solving either of these problems in time that is polynomial in the dimension of the lattice. Most algorithms, such as the LLL algorithm, instead guarantee that they can find a solution which is within some small approximating factor of the true solution. The following lattice problems are such approximating problems, where the approximating factor $\gamma > 1$ is a function of n, the dimension of the lattice.

Problem 4 (γ-Shortest Vector Problem (SVP))

Given a basis for a lattice L of dimension n, output a nonzero vector $\mathbf{v} \in L$ of length at most $\gamma \cdot \lambda_1(L)$.

In [34], this problem was shown to be NP-hard for $\gamma < 2^{1/p}$ for any p-norm.

Problem 5 (γ-Shortest Independent Vector Problem (SIVP))

Given a basis for a lattice L of dimension n, output a set of linearly independent vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n \in L$ such that $\max_{i \in \{1,2,\ldots,n\}} \|\mathbf{v}_i\| < \gamma \cdot \lambda_n(L)$.

In [4], it was shown that this problem is NP-hard for $\gamma < 2^{[\log(n)]^{1-\epsilon}}$ where $\epsilon > 0$ is arbitrary but fixed.

Problem 6 (γ-Gap Shortest Vector Problem (GapSVP))

Given a basis for a lattice L of dimension n and $d > 0 \in \mathbb{R}$, output YES if $\lambda_1(L) \leq d$ and NO if $\lambda_1(L) > \gamma \cdot d$. (In any other case, the output can be YES or NO).

In [35], this problem was shown to be NP-hard for $\gamma < \sqrt[p]{2}$ for any p-norm.

Problem 7 (γ-Closest Vector Problem (CVP))

Given a basis for a lattice L of dimension n and a vector $\mathbf{t} \in \mathbb{R}^n$, output a vector $\mathbf{v} \in L$ such that $\|\mathbf{t} - \mathbf{v}\| \leq \gamma \cdot dist(L, \mathbf{t})$.

In [18], this problem was shown to be NP-hard for $\gamma < 2^{(\log n)^{1-\epsilon}}$ where $\epsilon = (\log \log n)^{-c}$ for any constant $c < \frac{1}{2}$.

Problem 8 (γ-Closest Vector Problem with promise)

Given a basis for a lattice L of dimension n and a vector $\mathbf{t} \in \mathbb{R}^n$ with the promise that there is a unique vector $\mathbf{v} \in L$ such that $\|\mathbf{t} - \mathbf{v}\| \leq \gamma \cdot \lambda_1(L)$, find $\mathbf{v}$.

This problem is also known as the γ-Bounded Distance Decoding Problem (BDDP) in the literature such as in [32]. It was shown in [31] to be NP-hard for $\gamma > \frac{1}{\sqrt{2}}$.

2.3 The LWE Problem

While it is believed that the Approximate GCD problem is hard, there is no proof showing that this is the case. A problem that can be reduced to solving one of these hard lattice problems is the Learning with Errors Problem (LWE). The LWE problem was originally introduced by Regev in [42] and is described as follows.

Problem 9 (Learning With Errors Problem (LWE))

Given m samples of the form $(\mathbf{a}_i, b_i = \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i)$ where $q > 1$ and is dependent on

n, $\mathbf{a}_i \in \mathbb{Z}_q^n$ is chosen from a uniform random distribution, $\mathbf{s} \in \mathbb{Z}_q^n$ is chosen uniformly random but fixed for all pairs, and $b_i, e_i \in \mathbb{Z}_q$ determine the value of $\mathbf{s}$.

If $e_i = 0$, then given $m = n$ samples this problem can be solved in polynomial time. If $e_i \neq 0$, then for any m that is a polynomial in n this problem seems hard to solve. Normally, $|e_i|$ is much smaller than q.

Regev originally reduced the hardness of solving the LWE problem to solving GapSVP and SIVP, but part of his reduction involved the use of a quantum computer. In [40], Peikert was able to remove the quantum portion of the reduction and classically reduce the hardness of solving the LWE problem to the hardness of solving GapSVP on a general lattice. So, the LWE problem is NP-hard as well.

One additional problem is found in many of the existing homomorphic systems, and we provide its definition below.

Problem 10 (Subset Sum Problem (SSP))
Given a set S and x decide if there exists $T \subset S$ such that $\sum_{y \in T} y = x$.

This problem is NP-complete. Additional assumptions, such as the subset T is sparse, do not make the problem any easier.

Chapter 3

Homomorphic Encryption

3.1 Background

Recall that a deterministic cryptosystem is homomorphic for an operation $\star$ on the plaintext space and corresponding operation $\odot$ on the ciphertext space if

$$E(m_1) \odot E(m_2) = E(m_1 \star m_2). \tag{3.1}$$

A simple example of a deterministic cryptosystem that is partially homomorphic for multiplication given in [43] is an unpadded RSA cryptosystem.

RSA Cryptosystem	
Secret key:	d
Public key:	modulus n and exponent e
Plaintext:	$m \in \mathbb{Z}_n$
Encryption:	$c = E(m) = m^e \mod n$

The system is multiplicatively homomorphic since (3.1) holds. Here the operation on the plaintext space and the ciphertext space are the same: multiplication modulo n.

$$E(m_1) \cdot E(m_2) = m_1^e \cdot m_2^e \equiv (m_1 \cdot m_2)^e \mod n = E(m_1 \cdot m_2).$$

This system is only partially homomorphic since other operations, such as addition, do not satisfy the homomorphic property.

$$E(m_1) + E(m_2) = m_1^e + m_2^e \mod n \not\equiv (m_1 + m_2)^e \mod n = E(m_1 + m_2).$$

Also, recall that for a probabilistic cryptosystem we modified definition 3.1 as follows

$$D\big(E(m_1) \odot E(m_2)\big) = D\big(E(m_1 \star m_2)\big) = m_1 \star m_2. \tag{3.2}$$

A simple example of a probabilistic cryptosystem is the ElGamal cryptosystem, which was proposed in [19].

ElGamal Cryptosystem

Secret key:	$s \in \{1, 2, \ldots, q-1\}$ chosen randomly
Public key:	q
	a cyclic group G of order q
	α which is a generator for G
	$h = \alpha^s$
Plaintext:	$m \in G$
Encryption:	Pick a random $r \in \{1, 2, \ldots, q-1\}$
	Compute $g = \alpha^r$ and $y = h^r \cdot m$
	Ciphertext is $c = (g, y)$
Decryption:	Compute $y \cdot (g^s)^{-1} = (\alpha^{rs} \cdot m) \cdot (\alpha^{rs})^{-1} = m$

First, note that multiple encryptions of m are likely to be different since a different r will be chosen each time. Second, this system is homomorphically correct for multiplication. Since the system is probabilistic, we show that Property 3.2 holds for ciphertexts $c_1 = (\alpha^{r_1}, h^{r_1} \cdot m_1)$ and $c_2 = (\alpha^{r_2}, h^{r_2} \cdot m_2)$. Here the operation on the plaintext space is multiplication in G and the operation on the ciphertext space is

component-wise multiplication in G.

$$\begin{aligned} D\big(E(m_1)\cdot E(m_2)\big) &= D\big((\alpha^{r_1}, h^{r_1}\cdot m_1)\cdot(\alpha^{r_2}, h^{r_2}\cdot m_2)\big) \\ &= D\big((\alpha^{r_1+r_2}, h^{r_1+r_2}\cdot(m_1m_2))\big) \\ &= \big(h^{r_1+r_2}\cdot(m_1m_2)\big)\cdot\big((\alpha^{r_1+r_2})^s\big)^{-1} \\ &= \alpha^{s(r_1+r_2)}\cdot(m_1m_2)\cdot\big(\alpha^{s(r_1+r_2)}\big)^{-1} \\ &= m_1m_2 \end{aligned}$$

This cryptosystem is also only a partially homomorphic system since it cannot handle addition. Both of these examples are multiplicatively homomorphic, but in general most partially homomorphic systems can handle addition but not multiplication.

Though the concept of a fully homomorphic encryption system was proposed in 1978, little progress was made on developing such a system for the next 30 years. There were several attempts to create a fully homomorphic cryptosystem, and some systems that just happened to have homomorphic properties. A summary of some of these systems (based on [48]) is provided in Table 3.1.

Year	Citation	Author(s)	Capabilities
1978	[44]	Rivest, Shamir, and Adleman	unlimited multiplications
1984	[26]	Goldwasser and Micali	unlimited XOR of bits
1985	[19]	ElGamal	unlimited multiplications
1985	[14]	Cohen and Fischer	unlimited additions
1996	[1]	Ajtai and Dwork	lattice based system with probabilistically unlimited additions
1999	[39]	Paillier	unlimited additions (mod m)
1999	[46]	Sander, Young, and Yung	compute logical AND (homomorphic over a semigroup)
2005	[5]	Boneh, Goh, and Nissim	unlimited additions and a single multiplication
2008	[41]	Peikert and Waters	unlimited additions

Table 3.1: Early Homomorphic Encryption Schemes

In 2009, Stanford PhD student Craig Gentry proposed the first fully homomorphic encryption system in his thesis [20]. This seminal work also provided a basic outline for achieving fully homomorphic encryption. The outline is often referred to as "Gentry's blueprint" in the literature.

Though the system proposed by Gentry in his thesis and those that follow Gentry's blueprint are theoretically correct they are impractical. For example, in Gentry's original scheme, to obtain 2^λ security, each plaintext bit is encrypted as a ciphertext that has $\lambda^5 \cdot polylog(\lambda)$ bits and requires $\lambda^6 \cdot polylog(\lambda)$ amount of computation to decrypt [21]. Since the publication of his thesis, many improvements have been made to increase the efficiency of his scheme in terms of runtime and storage space. Though substantial progress has been made, the scheme and its variants are still unsuitable for practical implementation.

Another point of concern in Gentry's work is the theoretical underpinnings of his security reductions. The systems which follow Gentry's blueprint base their security on the assumed hardness of certain lattice problems over ideal lattices. These approaches often include other very strong assumptions such as the hardness of the sparse subset sum problem (SSSP). Since these problems have not been studied as extensively over ideal lattices as they have been for general lattices, there is a fair amount of skepticism in the literature as to whether these problems are truly hard.

In 2011, a second approach to creating a fully homomorphic system was proposed by Brakerski and Vaikuntanathan in [10]. This approach was the first to deviate from Gentry's blueprint in an attempt to base its security on hard problems over general lattices. In this approach, the Learning with Errors (LWE) problem was used as a cryptographic primitive, and its security was reduced to solving a variant of the Shortest Vector Problem (SVP) for general lattices. The original reduction involved a quantum component, but subsequent work by Peikert showed that the hardness of LWE could be classically reduced to another SVP variant.

The LWE approach is very promising since it is orders of magnitude more efficient. Each ciphertext is of size $\lambda \log(\lambda)$ and requires $\log(\lambda)$ amount of computation to decrypt.

The most recent approaches (as of August 2014), such as [25], focus on achieving homomorphic encryption not for any arbitrary function, but for certain classes of functions or for specific permitted functions. These approaches are known as functional encryption schemes and come with the additional feature that homomorphic evaluation results in a plaintext result as opposed to an encrypted result. This is done through a combination of garbled circuits, attribute based encryption (ABE), and standard homomorphic encryption. Similar efforts involve identity based encryption.

3.2 Gentry's Approach

3.2.1 Gentry's Blueprint

Gentry's main contribution was his blueprint for building a fully homomorphic system. A summary of this blueprint is provided below. We will now examine each of the these steps in detail.

Gentry's Blueprint

1. Select a somewhat homomorphic system (SHS).
2. Squash the decryption circuit for the system to reduce its complexity.
3. The squashed decryption circuit can now be homomorphically evaluated correctly by the somewhat homomorphic system in a process known as bootstrapping.
4. Begin evaluating the function you wish to compute on the encrypted data.
5. When operations can no longer be homomorphically computed because the error in the ciphertexts has gotten too large, homomorphically evaluate the squashed

decryption circuit to obtain a fresh ciphertext with small error. This is known as ciphertext refreshing.

6. Continue evaluating the function using the fresh ciphertexts.

7. Repeat the ciphertext refreshing step as needed until the function is fully evaluated.

There are three main components to the systems that follow Gentry's blueprint: an underlying somewhat homomorphic system (SHS), a squashing technique to reduce the complexity of the SHS's decryption circuit, and a bootstrapping method to turn the SHS into a fully homomorphic system (FHS). The following example deals with integers and is roughly the scheme developed by van Dijk, Gentry, Halevi, and Vaikuntanathan in [49] but it can be generalized to ideals in any ring and then extended to lattices, which is how Gentry originally posed it.

The user chooses a large odd integer p to be the secret key for the system. A plaintext is a single bit m. To encrypt a message m and obtain a ciphertext c, the user chooses $q, e \in \mathbb{Z}$ independently and randomly from some distribution for each encryption and computes

$$E(m, p) = qp + 2e + m.$$

The binary size of q is generally chosen to be greater than the binary size of p. The binary size of e, which is referred to as the error, is much smaller than the binary size of p. We will discuss the exact sizes for these parameters later on. The decryption function for this system is given by

$$D(c, p) = (c \mod p) \mod 2$$

To see that the decryption function is correct for a ciphertext $c = qp+2e+m$, observe

that

$$\begin{aligned} D(c,p) &= (qp + 2e + m \mod p) \mod 2 \\ &= (2e + m) \mod 2 \\ &= m \end{aligned}$$

Note that for this to work we need $2|e| + 1 < \frac{p}{2}$, which implies that $|e| < \frac{p}{4} - \frac{1}{2}$ so that $(2e + m \mod p) = 2e + m$.

Though it is very simple, this cryptosystem has two very useful properties. First, given two ciphertexts

$$\begin{aligned} c_1 &= q_1 p + 2e_1 + m_1 \\ c_2 &= q_2 p + 2e_2 + m_2 \end{aligned}$$

one can add them together to obtain

$$c_1 + c_2 = (q_1 + q_2)p + 2(e_1 + e_2) + (m_1 + m_2)$$

which is a valid encryption of $m_1 + m_2$ provided $|e_1 + e_2| < \frac{p}{4} - \frac{1}{2}$. So the cryptosystem is additively homomorphic. Second, for multiplication we have

$$c_1 c_2 = (q_1 q_2 p + 2q_1 e_2 + q_1 m_2 + 2q_2 e_1 + q_2 m_1)p + 2(2e_1 e_2 + e_1 m_2 + e_2 m_1) + (m_1 m_2)$$

which is a valid encryption of $m_1 m_2$ provided $|2e_1 e_2 + e_1 m_2 + e_2 m_1| < \frac{p}{4} - \frac{1}{2}$. So the cryptosystem is multiplicatively homomorphic.

The system as presented so far is a symmetric key cryptosystem since the secret key p is needed for encryption and decryption.

Algorithm	**Input**	**Output**	**Formula**
KeyGen	security parameter (λ)	secret key (p)	
Encrypt	plaintext (m) and p	ciphertext (c)	$E(m,p) = qp + 2e + m$
Decrypt	c and p	m	$D(c,p) = (c \mod p) \mod 2$

Table 3.2: Gentry Symmetric Key Cryptosystem

In the discussion that follows, it will be necessary to allow an untrusted user to encrypt data. Since we do not want to hand over the secret key p to allow them to do so (which would allow the untrusted user to decrypt ciphertexts), we need to turn this symmetric key system into a public key system. To do so, we make the following modifications. The KeyGen algorithm produces τ integers of the form $x_i = q_i p + r_i$, where q_i and r_i are drawn uniformly at random and a distinguished odd value x_0 with $x_0 > x_i$ and $x_0 = q_0 p$ (i.e. a clean multiple of p). Often the size of q and q_i are the same but the size of r_i is much less than the size of the e's used in the encryption algorithm. These x_i's are published as the public key. The Encrypt algorithm for a plaintext, m, is now

$$E(m) = 2\left(\sum_{i \in T} x_i + e\right) + m \mod x_0$$

for some random subset $T \subseteq \{1, 2, \ldots, \tau\}$. The Decrypt algorithm is unchanged and is still correct when $|e| < \frac{p}{4} - \frac{1}{2}$ (since the r_i's are so much smaller than e they do

not affect the correctness of the decryption function).

In this cryptosystem, each addition increases the error size by 1 bit, while each multiplication doubles the error size. Since this scheme can only handle homomorphic operations while the magnitude of the error is less than $\frac{p}{4} - \frac{1}{2}$ (i.e. the binary size of the error is less than the binary size of p) it is a somewhat homomorphic system and is the underlying SHS component in Gentry's blueprint. For a given function, as we evaluate the operations in the circuit representation of the function, the error in each intermediary ciphertext will grow. For this cryptosystem, a polynomial f can be homomorphically evaluated if

$$\deg(f) \leq \frac{|p|_{\text{bin}} - 4 - \log\left(\|f\|_1\right)}{|r_i|_{\text{bin}} + 2}, \tag{3.3}$$

where $\deg(f)$ is the degree of f and $\|f\|_1$ is the 1-norm of the coefficient vector of f [49].

Once the error becomes too big, $|e| \geq \frac{p}{4} - \frac{1}{2}$, we can no longer homomorphically evaluate the circuit. If, at this point, we could take the intermediary ciphertexts, decrypt them to obtain the corresponding plaintexts, and then re-encrypt the plaintexts we would have "fresh" ciphertexts with small error size again and could continue to homomorphically evaluate the circuit. This process is known as refreshing the ciphertexts. One obvious method to accomplish this is to give the secret key to the party doing the homomorphic evaluation. Then they could periodically decrypt everything and re-encrypt it while doing the homomorphic evaluation, but this would mean they could decrypt the original ciphertexts and we would no longer have privacy.

To allow for this refreshing of the ciphertexts while maintaining privacy, we make the following key observation. The decryption function of the SHS is itself a function. Specifically, it is a function which takes a ciphertext and the private key as

input and returns the original plaintext. If the number of multiplications and additions in the circuit that represents the decryption function is less than the maximum number that the SHS can handle then the SHS can homomorphically evaluate its own decryption circuit. Gentry summarized this concept in what is known as his bootstrapping theorem in the literature [20].

Theorem 2 (Bootstrapping Theorem)
If a somewhat homomorphic system (SHS) can evaluate the following two circuits

1. *A single addition operation followed by decryption.*

2. *A single multiplication operation followed by decryption.*

then it can be bootstrapped into a fully homomorphic system (FHS).

Normally for the secret key p, a plaintext m, and a ciphertext $c = E(m)$, with a large amount of error we have,

$$D(c, p) = D(E(m), p) \to m.$$

However, we can homomorphically evaluate the decryption circuit as follows

$$D\big(E(c), E(p)\big) = D\Big(E\big(E(m)\big), E(p)\Big) \to \overline{E(m)}.$$

$E(m)$ and $\overline{E(m)}$ are both valid encryptions of m, but $\overline{E(m)}$ is a "fresh" ciphertext with small error. So if we publish an encryption of the secret key, $E(p)$, the party doing the homomorphic evaluation can encrypt the intermediary ciphertexts (using the public key) and homomorphically evaluate the decryption circuit to obtain refreshed ciphertexts. Note that publishing the encryption of the secret key (under the public key) means that this cryptosystem needs to be circularly secure.

Recall, that to homomorphically evaluate a function, you represent it as a Boolean circuit. The circuit normally takes bits as input and produces bits as output. When evaluated homomorphically though, it will take a ciphertext for each bit and produce a ciphertext for each bit of output by performing the corresponding operations for the addition (XOR) and multiplication (AND) gates on the ciphertexts as opposed to bits. Decrypting each output bit should yield the result that would have been obtained by operating directly on the plaintext bits. In other words, instead of taking bits and producing bits it now takes encryptions of bits and produces encryptions of bits.

There are two things needed to make this concept of ciphertext refreshing work.

1. The number of multiplications in the decryption circuit must be less than the maximum number that the SHS can handle.

2. The public key should include an encryption of the secret key to allow for evaluating the decryption circuit homomorphically.

The decryption function is

$$D(c, p) = (c \mod p) \mod 2 \to m$$

which is the same thing as

$$D(c, p) = LSB(c) \oplus LSB(\lfloor c/p \rceil) \to m.$$

where $LSB(a)$ extracts the least significant bit from a. This can easily be seen because

$$c \mod p = c - \lfloor c/p \rceil \cdot p,$$

and since p is odd,

$$(c \mod p) \mod 2 = c - \lfloor c/p \rceil \mod 2.$$

When this function is represented as a Boolean circuit, the complicated part lies in computing $c/p = c \cdot 1/p$. When this calculation is done homomorphically over the binary representation of c and $1/p$ it takes roughly $(|p|_{\text{bin}})^2$ multiplications (and some additions) in order to accurately compute it. This will cause the error size of the ciphertexts to grow by this same factor. Thus, we would need $(|p|_{\text{bin}})^2|e|_{\text{bin}} < |p|_{\text{bin}}$ to be successful in homomorphically evaluating the decryption circuit, which is impossible.

To reduce the number of multiplications in the decryption circuit, a squashing technique must be employed. This is accomplished by publishing a hint about the private key of the SHS (or in this case a hint about $1/p$) in the public key. To this end, we modify the KeyGen algorithm to run as follows and add in the encryptions of the secret key as well. Note that for a rational number x, the expression $x \mod 2$ maps x into the interval $[0, 2)$. For example, $7.1317 \mod 2 \to 1.1317$.

1. Generate the secret key p and the public values x_i as before.

2. Generate rational (decimal) numbers $b_i \leftarrow [0, 2)$ with α bits of precision to form the set $B = \{b_1, b_2, \ldots, b_n\}$.

3. Find, by possibly adjusting one (or more) of the b_i values, a sparse subset $S \subset \{1, 2, \ldots, n\}$ of size r such that $\sum_{i \in S} b_i = 1/p - \epsilon \mod 2$, where $|\epsilon| < 2^{-\alpha}$.

4. Encode S as a vector $\mathbf{s}$ having $s_{[i]} = 1$ if $i \in S$ and $s_{[i]} = 0$ otherwise (i.e. an indicator vector for S).

5. The secret key is now $\mathbf{s}$.

6. The x_i's and B are published as the public key.

7. Also, publish $z_i = E(s_{[i]})$, the encryption of the bits of the secret key, either as part of the public key or as a separate evaluation key.

In addition, when a plaintext m is encrypted some additional information is appended to the ciphertext c. Compute $\bar{d}_i = c \cdot b_i \mod 2$ for $i = 1, 2, \ldots n$ so that the ciphertext for m is now $\bar{c} = \left(c, \left[\bar{d}_1, \bar{d}_2, \ldots, \bar{d}_n\right]\right)$. For each $\bar{d}_i$ only $l = \lceil \log(r) \rceil + 3$ bits of precision after the binary decimal point are stored.

The decryption function now becomes,

$$D(\bar{c}, \mathbf{s}) = \left(c - \left\lfloor \sum_{i=1}^{n} \mathbf{s}_{[i]} d_i \right\rceil \right) \mod 2$$

This enables us to replace the multiplications involved with computing $1/p$ with a small number of additions. We now describe the full details of the circuit used to compute the decryption function. To do so, we need a few preliminaries.

First, recall the definition of the elementary symmetric polynomials.

Definition 4 (Elementary Symmetric Polynomials)
Let $e_0(x_1, x_2, \ldots, x_n) = 1$*, and for any integer* $k > 0$ *define*

$$e_k(x_1, x_2, \ldots, x_n) = \sum_{|S|=k} \prod_{i \in S} x_i$$

where $S \subseteq \{1, 2, \ldots, n\}$*. Note,* e_k *is of degree* k *and oftentimes if* $\mathbf{x}$ *is a vector with components* $x_1, x_2, \ldots, x_n$ *we denote* $e_k(x_1, x_2, \ldots, x_n)$ *as* $e_k(\mathbf{x})$*.*

Second, we will need Lucas' Theorem which holds in general for any prime p, but for our purposes we only need the case when $p = 2$.

Theorem 3 (Lucas' Theorem for $p = 2$)

Let n and m be integers with $n \leq m$, where

$$m = m_0 + m_1 \cdot 2 + m_2 \cdot 2^2 + \dots$$
$$n = n_0 + n_1 \cdot 2 + n_2 \cdot 2^2 + \dots$$

are the binary expansions for m and n respectively. Then

$$\binom{m}{n} \equiv \binom{m_0}{n_0}\binom{m_1}{n_1}\binom{m_2}{n_2}\dots \mod 2.$$

A special case of interest, is that if $n = 2^i$ then

$$\binom{m}{2^i} \equiv \binom{m_i}{1} \equiv m_i \mod 2.$$

Third, a well known result (Lemma 11 in [6]) shows that for a binary vector $\mathbf{x}$ with Hamming weight w, the bits of w can be represented as a polynomial in the $\mathbf{x}_{[i]}$'s.

Theorem 4 (Bits of the Hamming Weight)

For a binary vector $\mathbf{x}$ with Hamming weight w, where $(w_n, w_{n-1}, \dots, w_1, w_0)$ is the binary representation of w with w_0 being the least significant bit. Then

$$w_i = e_{2^i}(\mathbf{x}) \mod 2.$$

The proof of this result is quite elegant. First, notice that evaluating e_{2^i} at $\mathbf{x}$

yields

$$\begin{aligned} e_{2^i}(\mathbf{x}) &= \binom{w}{2^i} \\ &\equiv \binom{w_i}{1} \mod 2 \\ &\equiv w_i \mod 2 \end{aligned}$$

where the first line (assuming e_{2^i} is evaluated over the integers) is due to exactly w of the $\mathbf{x}_{[i]}$'s being 1 so there are $\binom{w}{2^i}$ terms in $e_{2^i}(\mathbf{x})$ which are 1 and the second line holds due to Lucas' Theorem.

Fourth, we will need a trick that allows us to replace the sum of 3 numbers of a certain bit length with the sum of two numbers having a bit length of at most 1 bit longer than the longest one.

Definition 5 (Three-for-Two Trick)
Consider three t-bit numbers a, b, c having binary representations as follows

$$\begin{aligned} a &= (a_{t-1}, \ldots, a_1, a_0) \\ b &= (b_{t-1}, \ldots, b_1, b_0) \\ c &= (c_{t-1}, \ldots, c_1, c_0). \end{aligned}$$

Then define two $(t+1)$-bit numbers m, n based on the following bit-wise formulas

$$\begin{aligned} n_0 &= 0 \\ m_i &= a_i + b_i + c_i \mod 2 \\ n_{i+1} &= a_i b_i + a_i c_i + b_i c_i \mod 2 \\ m_t &= 0. \end{aligned}$$

In essence, the bits of n hold the carry bits from the addition. This can be computed by the following circuit which has a multiplicative depth of 2.

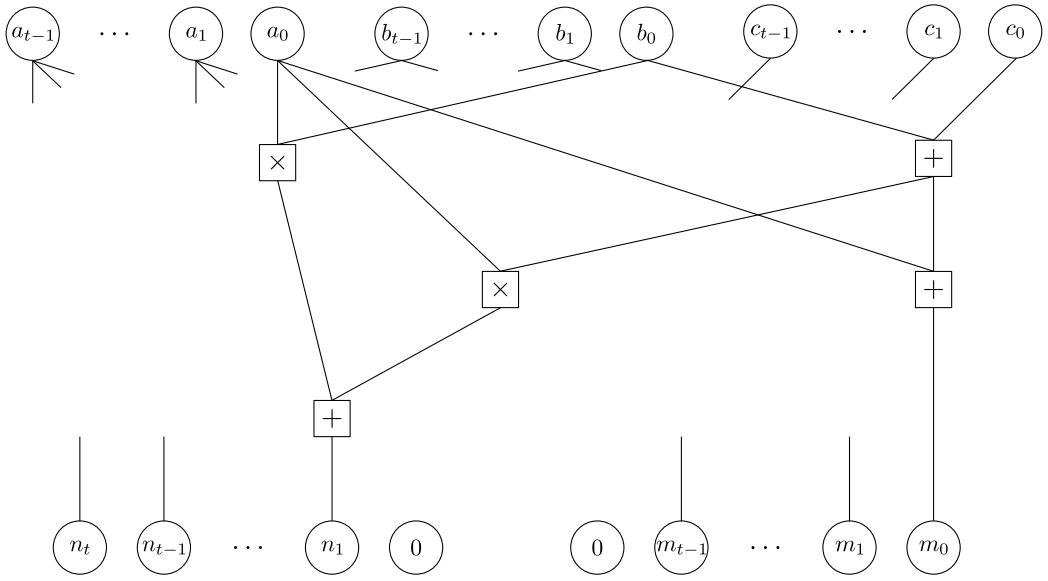

Figure 3.1: Three for Two Circuit

For the sum of k numbers, this three-for-two trick can be applied repeatedly to reduce the sum to the sum of two numbers. It will take at most $\lceil \log_{3/2}(k) \rceil + 2$ rounds to do so, which corresponds to a polynomial of degree at most $2^{\lceil \log_{3/2}(k) \rceil + 2}$.

Following the same lines as the corresponding discussion in [49] we break the evaluation of the decryption function up into 3 sections and describe the circuit needed to handle each section.

Section 1

Compute $a_i = \mathbf{s}_{[i]} \cdot d_i$ for $i = 1, 2, \ldots, n$. Since $\mathbf{s}_{[i]}$ are bits and d_i are rational numbers in $[0, 2)$ this can be done with the following circuit which has a multiplicative depth of 1.

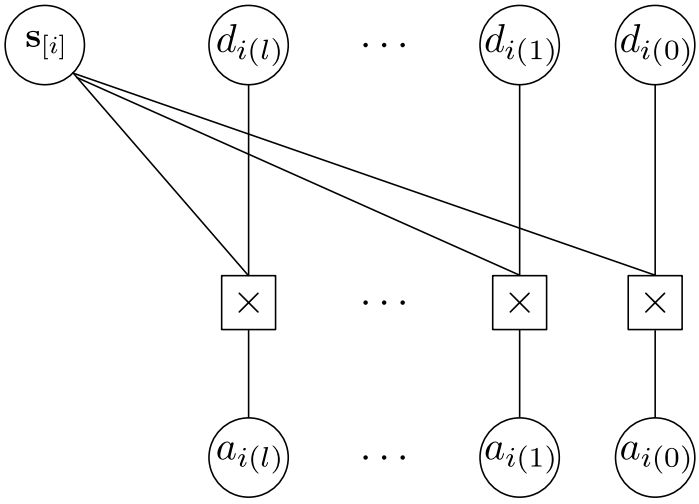

Figure 3.2: Section 1 Circuit

Thus each bit of the output is a degree 2 polynomial in the inputs: $a_{i(j)} = \mathbf{s}_{[i]} \cdot d_{i(j)}$.

Section 2

Next, compute the sum $\sum_{i=1}^{n} a_i$. This is done using what Gentry calls the "grade-school" arithmetic approach. Each addend a_i in this sum is represented in binary and entered as a row in the following table. Then the sum (as an integer) of the j-th column is stored in W_j (without any carrying to the next column). Note that this is just the Hamming weight of the column.

$$\begin{array}{cccccccc} & a_{1(l)} & \ldots & a_{1(j)} & \ldots & a_{1(1)} & a_{1(0)} \\ & a_{2(l)} & \ldots & a_{2(j)} & \ldots & a_{2(1)} & a_{2(0)} \\ & \vdots & & \vdots & & \vdots & \vdots \\ + & a_{n(l)} & \ldots & a_{n(j)} & \ldots & a_{n(1)} & a_{n(0)} \\ \hline & W_l & \ldots & W_j & \ldots & W_1 & W_0 \end{array}$$

Table 3.3: Grade School Arithmetic

Since only r of the s_i values are 1, only r of the values: $\{a_{1(j)}, a_{2(j)}, \ldots, a_{n(j)}\}$ are possibly non-zero for each column so $W_j \leq r$. This means, we can represent W_j in binary using at most $h = \lceil \log(r+1) \rceil$ binary digits. From the theorem above, each bit of W_j can be represented as a polynomial of degree 2^j in the bits $a_{1(j)}, a_{2(j)}, \ldots, a_{n(j)}$.

$$W_{j(k)} = e_{2^k}(a_{1(j)}, a_{2(j)}, \ldots, a_{n(j)}) \mod 2$$

The polynomial for the $(h-1)$-th bit will require the largest polynomial of degree $2^{h-1} = 2^{\lceil \log(r+1) \rceil - 1} < 2^{\log(r)} = r$ so the circuit that computes the bits of W_j will have a multiplicative depth of $\lceil \log(r) \rceil$.

Section 3

To compute the sum $\sum_{j=0}^{l+1} 2^{j-(l+1)} W_j$, which is equal to $\sum_{i=1}^{n} a_i$, let $w_j = 2^{j-(l+1)} W_j$ so the sum becomes $\sum_{j=0}^{l+1} w_j$. We now repeatedly use the three-for-two trick shown above to obtain two numbers y_1 and y_2 such that $y_1 + y_2 = \sum_{j=0}^{l+1} w_j$. Note, that since we are looking for the sum mod 2 the numbers never grow to have more than $l+1$ bits in total since we can discard the "carry bits" for the 2's position. As noted above, this corresponds to a polynomial of degree at most

$$2^{\lceil \log_{3/2}(l+1) \rceil + 2} < 2^{\log_{3/2}(l+1)+3} = 8 \cdot 2^{\frac{\log(l+1)}{\log(3/2)}} = 8(l+1)^{1/\log(3/2)}.$$

Now, to add the final two numbers y_1 and y_2 together, which are $(l+1)$-bit numbers we normally would use a circuit that is of size $\lceil \log(l+1) \rceil$. However, since we want $\lfloor y_1 + y_2 \rceil \mod 2$ they show that since $\sum_{i=1}^{n} \mathbf{s}_{[i]} d_i$ is within 1/4 of an integer that $\lfloor y_1 + y_2 \rceil \mod 2$ can be computed by a degree 4 polynomial.

Adding the final gates to compute $c - \lfloor y_1 + y_2 \rceil \mod 2$ does not introduce any multiplications so the circuit for this section corresponds to a polynomial of degree at most $32(l+1)^{1/\log(3/2)}$.

In total, we have that the circuit in Section 1 corresponds to a degree 2 polynomial, the circuit in Section 2 corresponds to an r degree polynomial, and the circuit in Section 3 corresponds to a $32(l+1)^{1/\log(3/2)}$ degree polynomial. Putting this together implies that the decryption circuit corresponds to a polynomial of degree $64r(l+1)^{1/\log(3/2)}$. Let f be the polynomial corresponding to the circuit composed of a single operation (multiplication or addition) followed by the decryption circuit. We now obtain the following bound on the degree of f.

$$\begin{aligned}
\deg(f) &\leq 2 \cdot 64r(l+1)^{1/\log(3/2)} \\
&\leq 128\lambda\big((\lceil \log(r) + 3 \rceil) + 1\big)^{1.71} \\
&\leq 128\lambda(\lceil \log(\lambda) + 4 \rceil)^{1.71} \\
&\leq 128\lambda \log^2(\lambda)
\end{aligned}$$

where the second inequality holds if we set $r = \lambda$. Recall from the bound (3.3) above that for a polynomial f we need

$$\deg(f) \leq \frac{|p|_{\text{bin}} - 4 - \log\left(\|f\|_1\right)}{|r_i|_{\text{bin}} + 2}.$$

to be able to homomorphically evaluate f. We set $|r_i|_{\text{bin}} = \lambda$ to avoid brute force attacks on the noise. We compute $\log(\|f\|_1)$ as follows. Since f is a polynomial in the n bits $\mathbf{s}_{[i]}$. Thus, there can be at most $\binom{n}{128\lambda \log^2(\lambda)} \leq n^{128\lambda \log^2(\lambda)}$ monomial terms. Since f is a polynomial over $\mathbb{F}_2$ each coefficient can be at most 1 and we have $\|f\|_1 \leq n^{128\lambda \log^2(\lambda)}$. Taking the logarithm of this expression (and noting that $n \leq \lambda^7$)

yields

$$\begin{aligned}\log(\|f\|_1) &\leq 128\lambda\log^2(\lambda)\log(n)\\ &\leq 128\lambda\log^2(\lambda)\cdot 7\log(\lambda)\\ &\leq 896\lambda\log^3(\lambda).\end{aligned}$$

Setting $|p|_{\text{bin}} \geq C\lambda^2\log^2(\lambda)$ for some constant $C > 128$ we have that

$$\begin{aligned}\frac{|p|_{\text{bin}} - 4 - \log(\|f\|_1)}{|r_i|_{\text{bin}} + 2} &\geq \frac{C\lambda^2\log^2(\lambda) - 4 - 896\lambda\log^3(\lambda)}{\lambda + 2}\\ &\geq C\lambda\log^2(\lambda)\\ &\geq 128\lambda\log^2(\lambda)\\ &\geq \deg(f)\end{aligned}$$

and so the system can evaluate f and by Gentry's Bootstrapping Theorem (Theorem 2 above) is a fully homomorphic system.

To summarize, the SHS will correctly homomorphically evaluate any circuit where $|e|_{\text{bin}} < |p|_{\text{bin}}$ in the final output. When starting with fresh ciphertexts, the output of performing a single operation and then evaluating the squashed decryption circuit will satisfy that requirement. The process for doing so (often called recrypt or refresh) is outlined below.

1. For a ciphertext $\bar{c} = \left(c, \left[\bar{d}_1, \bar{d}_2, \ldots, \bar{d}_n\right]\right)$ with large error we wish to obtain a fresh ciphertext with small error.

2. Let $(c_t, \ldots, c_1, c_0)$ be the binary representation for the integer c, and define $y_i = E(c_{(i)})$ for $i = 0, 1, \ldots, t$ (i.e. the encryption of the i-th bit of the integer c using the public key SHS system encryption function without the appended

list of extra info).

3. Let $(d_{i(l)}, \ldots, d_{i(1)}, d_{i(0)})$ be the binary representation for the decimal number d_i, and define $g_{i,j} = E(d_{i(j)})$ for $i = 1, 2, \ldots, n$ and $j = 0, 1, \ldots, l$ (again, just the integer part of the encryption). Note, that one can use the bits $d_{i(j)}$ directly as the integer ciphertexts, since technically they are valid encryptions when the chosen subset of the x_i's is empty and the error term is 0.

4. Evaluate the decryption circuit over the integers using the integer values y_i, $g_{i,j}$, and z_i (from the public key) in place of the bits $c_{(i)}$, $d_{i(j)}$, and $\mathbf{s}_{[i]}$ respectively in the circuit.

5. The output of this evaluation will be a single integer that is a fresh ciphertext with small error. The d_i values for this new encryption can be calculated using the b_i's in the public key.

This process of homomorphically evaluating the decryption circuit (or some squashed version of it) is referred to as bootstrapping since the cryptosystem uses its existing somewhat homomorphic properties to transform into a fully homomorphic cryptosystem.

We summarize the completed fully homomorphic system below.

Alg.	Input	Output	Formula/Comments
KeyGen	security parameter (λ)	secret key: ($\mathbf{s}$) public key: (x_i's, b_i's, z_i's)	$\mathbf{s}$ is an indicator vector $x_i = q_i p + r_i$ $b_i \in [0, 2)$ $z_i = E(\mathbf{s}_{[i]})$
Encrypt	plaintext (m)	ciphertext ($\bar{c}$)	$E(m) = 2\left(\sum_{i \in T} x_i + e\right) + m \mod x_0$
Decrypt	$\bar{c}$ and $\mathbf{s}$	m	$D(\bar{c}, \mathbf{s}) = \left(c - \left\lfloor \sum_{i=1}^{n} \mathbf{s}_{[i]} d_i \right\rceil\right) \mod 2$

Table 3.4: Gentry Public Key Cryptosystem

3.2.2 Improvements to Gentry Systems

The cryptosystem presented in Gentry's thesis [20] used ideal lattices instead of integers. Basically, the secret key p and errors that are multiples of 2 are replaced with the basis for a secret ideal I and a basis for another (public) ideal J that the errors come from. One of the disadvantages of this approach is that generating out the bases for these ideals is quite computationally intensive.

One of the first attempts to improve Gentry's system was by Smart and Vercauteren [47]. They reduced the public and private key sizes by using integers instead of ideal lattice bases. To do so, they used a principal ideal lattice, which could be represented by two large integers. Unfortunately, the complexity of the KeyGen procedure was such that not only did it take a while (several hours in their experiments)

to generate out the keys for lattices of dimension $< 2^{11}$ (having associated security parameter $\lambda = 54$) it was unable to generate out keys for bigger lattices. In order to bootstrap the somewhat homomorphic system into a fully homomorphic system, lattices of dimension 2^{27} were needed and so while in theory the system was fully homomorphic it was never so in practice.

In late 2010, Gentry and Halevi [22] revisited the Smart and Vercauteren approach and were able to achieve bootstrapping leading to a fully homomorphic system. They did so by eliminating the requirement that the resultant of a couple of polynomials used in the KeyGen process be prime and squashing the decryption circuit from a polynomial in several hundred degrees to a polynomial of degree 15.

The van Dijk integer [49] version is the example presented above. This scheme deviated from the previous ones by using lattices for security assumptions, but using simple arithmetic of integers in implementation.

In 2011, Coron improved on this integer version [16] and reduced the public key size from $O(\lambda^{10})$ down to $O(\lambda^7)$. The main contributions were as follows. First, in the system described in detail above the public key contains τ integers. However, these integers could be represented by only $2\sqrt{\tau}$ integers if they were formed as the product of integers $x_{i,j} = x_{i,0} \cdot x_{j,1} \mod x_0$ for $1 \leq i, j \leq \lceil\sqrt{\tau}\rceil$. Thus a quadratic, as opposed to a linear representation, of the public key elements results in significant space savings since instead of $O(\lambda^5)$ numbers in the public key there are now $O(\lambda^{2.5})$. Second, there are $O(\lambda^8)$ hints, b_i, in the public key, but instead of storing all of them one can provide a seed for a public random number generator (RNG) and use the RNG to generate the b_i's on the fly. This trade off of space in the public key for a fixed amount of computation by the party doing the homomorphic computation is quite reasonable. Often the party doing the computation has access to massive amounts of computation and conceivably the network and the storage space on the

users end are the main bottlenecks in the system.

Another interesting approach is provided by Coron in [17]. He observes that if one is willing to sacrifice the notion of fully homomorphic encryption to allow for the evaluation of functions having a fixed maximum number of levels, L, in their circuit representation then a very efficient cryptosystem can be achieved using the somewhat homomorphic system and a technique called modulus switching. In short, modulus switching which Coron adapts from [10] allows a ciphertext in $\mathbb{Z}_{q_1}$ with a large amount of error to be turned into a ciphertext in $\mathbb{Z}_{q_2}$, where $q_2 < q_1$, with a small amount of error. This process is invoked after each level of circuit computation to "clean up" the ciphertexts. This obviates the need for homomorphic evaluation of the decryption circuit so a squashed decryption circuit is no longer needed and bootstrapping is also not required (though it can be used as an optimization feature). The sequence, or ladder, of decreasing moduli $q_1, q_2, \ldots, q_L$ with $q_{i+1} < q_i$ must be included in the public key. However, since we do not have to provide a "hint" about the secret key anymore the size of the public key decreases drastically. The KeyGen algorithm is also much faster, but the Recrypt process will take longer since the homomorphic evaluation will have to "walk" down the ladder of descending moduli for every single operation.

Note that the ratio of the error size to the size of the ciphertext remains unchanged. This cryptosystem is an example of a leveled scheme in that the limit on the multiplications is not on the number of operations, but on the number of levels in the circuit representation.

Scheme	λ	**Public Key**	**Key Gen**	**Recrypt**
Smart-Vercauteren [22]	≈72	3.01 MB	3.2 min	N/A
Gentry-Halevi [22]	≈72	2.25 GB	2.2 hours	31 min
van Dijk Integer version [49]	72	λ^{10}	N/A	N/A
Coron Integer (quad) [16]	72	λ^7 (802 MB)	43 min	14 min 33 sec
Leveled [17]	72	λ^5 (18 MB)	6 min 18s	2 hour 27 min

Table 3.5: Summary of Gentry Approach Cryptosystems

1. Note that the Smart-Vercauteren system in not fully homomorphic.
2. Smart-Vercauteren timings done with a 64-bit quad core Intel Xeon at 3GHz and 24GB of RAM.
3. Integer timings done on a single core Intel Core2 Duo at about 3GHz.

3.2.3 Security of Gentry Systems

The security of Gentry's original system and those that follow his blueprint is based on the hardness of the A-GCD problem. In Gentry's PhD thesis he showed that the security of his cryptosystem is based on the Ideal Coset Problem (the A-GCD equivalent when dealing with ideals in a ring) or the decisional BDDP (the A-GCD equivalent for ideal lattices). One of the crucial components in his statement of the BDDP problem is the method of generating a sample from a set of lattice points. If this sampling technique has low minimum entropy then the BDDP should be hard.

While the problems themselves are assumed to be hard for general lattices, it is not clear in the research community whether they still remain hard for ideal lattices. Perhaps the special structure of an ideal lattice actually renders these problems easy to solve.

3.3 LWE Approach

The security assumptions of Gentry's original approach were based on the hardness of lattice problems over ideal lattices. The improvements discussed above did much to improve other elements of Gentry's original system (mainly in terms of space and the time it takes to evaluate a circuit), but their security still relied on the hardness of solving these problems over ideal lattices. In 2011, in an effort to base the security assumptions on more stable theoretical ground, Brakerski and Vaikuntanathan in [10] proposed a fully homomorphic system which used the LWE problem as a basis for their security. As discussed when originally introduced, the LWE problem can be reduced to solving gapSVP over general lattices so this avoids the extra assumption that the problem is hard for ideal lattices.

An additional feature in this paper is that they avoid the need to squash the decryption circuit by introducing a dimension reduction technique. The following is an overview of the system they proposed.

We first present a SHS that can handle up to D multiplications and then discuss how it can be developed into a FHS via bootstrapping. The user chooses a secret key $\mathbf{s} \in \mathbb{Z}_q^n$ to be the secret key for the system. A plaintext is a single bit m. To encrypt the message m and obtain a ciphertext c, the user chooses $\mathbf{a} \in \mathbb{Z}_q^n$ uniform randomly and $e \in Z_q$ according to a distribution (the error distribution). This sampling is done independently for each encryption and generates the ciphertext as

follows.

1. Compute $b = \langle \mathbf{a}, \mathbf{s} \rangle + 2e + m$
2. Output the ciphertext $c = (\mathbf{a}, b) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$

The random e in the encryption process is often referred to as the error or noise, and should be "small" relative to q. The error distribution is said to be a $\mathcal{B}$-bounded distribution meaning that $\Pr[|e| > \mathcal{B}]$ is vanishingly small.

The decryption procedure is outlined below.

1. Extract $\mathbf{a}$ and b from c.
2. Using the secret key $\mathbf{s}$, recover m as follows:

$$\begin{aligned}(b - \langle \mathbf{a}, \mathbf{s} \rangle \mod q) \mod 2 &= (2e + m) \mod 2 \\ &= m.\end{aligned}$$

Note that just like Gentry's approach we need $|e| < \frac{q}{4} - \frac{1}{2}$. Thus, $(2e + m \mod q) = 2e + m$.

In fact, instead of viewing each ciphertext as a tuple, we can think of the ciphertext as a linear multivariate polynomial in $\mathbb{Z}_q[x_1, x_2, \ldots, x_n]$ with $n + 1$ terms having $b, \mathbf{a}_{[1]}, \mathbf{a}_{[2]}, \ldots, \mathbf{a}_{[2]}$ as coefficients and $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ being the vector whose i-th component is the symbolic variable x_i.

$$\begin{aligned}c = (\mathbf{a}, b) \Leftrightarrow f[c](\mathbf{x}) &= b - \langle \mathbf{a}, \mathbf{x} \rangle \pmod q \\ &= b - \sum_{i=1}^{n} \mathbf{a}_{[i]} \cdot \mathbf{x}_{[i]} \in \mathbb{Z}_q[x_1, x_2, \ldots, x_n]\end{aligned}$$

In this approach, encryption is as follows for $\mathbf{a}$ and e chosen as before

$$\begin{aligned} E(m,\mathbf{s}) &= \big(\langle \mathbf{a},\mathbf{s}\rangle + 2e + m\big) - \sum_{i=1}^{n} \mathbf{a}_{[i]} \cdot \mathbf{x}_{[i]} \\ &= b - \sum_{i=1}^{n} \mathbf{a}_{[i]} \cdot \mathbf{x}_{[i]} \in \mathbb{Z}_q[x_1, x_2, \ldots, x_n] \\ &= f[c](\mathbf{x}) \end{aligned}$$

and decryption is done by evaluating the ciphertext polynomial at $\mathbf{s}$ and taking the result mod 2:

$$D(c,\mathbf{s}) = f[c](\mathbf{s}) \mod 2.$$

This simple system is clearly additively homomorphic. Given two ciphertext polynomials $f[c_1](\mathbf{x})$ and $f[c_2](\mathbf{x})$ one can add them together to obtain

$$\begin{aligned} f[c_1](\mathbf{x}) + f[c_2](\mathbf{x}) &= \left(b_1 - \sum_{i=1}^{n} \mathbf{a}_{1[i]} \cdot \mathbf{x}_{[i]}\right) + \left(b_2 - \sum_{i=1}^{n} \mathbf{a}_{2[i]} \cdot \mathbf{x}_{[i]}\right) \\ &= \left(\big(\langle \mathbf{a}_1,\mathbf{s}\rangle + 2e_1 + m_1\big) - \sum_{i=1}^{n} \mathbf{a}_{1[i]} \cdot \mathbf{x}_{[i]}\right) \\ &\quad + \left(\big(\langle \mathbf{a}_2,\mathbf{s}\rangle + 2e_2 + m_2\big) - \sum_{i=1}^{n} \mathbf{a}_{2[i]} \cdot \mathbf{x}_{[i]}\right) \\ &= \left(\big(\langle \mathbf{a}_1 + \mathbf{a}_2,\mathbf{s}\rangle + 2(e_1+e_2) + (m_1+m_2)\big) - \sum_{i=1}^{n} (\mathbf{a}_{1,i} + \mathbf{a}_{2,i}) \cdot \mathbf{x}_i\right) \\ &= (b_1 + b_2) - \sum_{i=1}^{n} (\mathbf{a}_{1[i]} + \mathbf{a}_{2[i]}) \cdot \mathbf{x}_{[i]} \\ &= f[c_1 + c_2](\mathbf{x}) \end{aligned}$$

which is a valid encryption of $m_1 + m_2$ provided $|e_1 + e_2| < \frac{q}{4} - \frac{1}{4}$. Taking a linear combination of t ciphertext polynomials with coefficients g_i (the coefficient vector is

$\mathbf{g} = (g_1, g_2, \ldots, g_l))$ we have that

$$\begin{aligned}\sum_{i=1}^{t} g_i \cdot f[c_i](\mathbf{x}) &= \sum_{i=1}^{t} g_i\big(\langle \mathbf{a}_i, \mathbf{s}\rangle + 2e_i + m_i\big) - \sum_{i=1}^{t} g_i \langle \mathbf{a}_i, \mathbf{x}\rangle \\ &= \sum_{i=1}^{t} g_i\big(2e_i + m_i\big) + \sum_{i=1}^{t} g_i \left(\langle \mathbf{a}_i, \mathbf{s}\rangle - \langle \mathbf{a}_i, \mathbf{x}\rangle\right).\end{aligned}$$

If we evaluate this expression at $\mathbf{s}$ we obtain

$$\begin{aligned}\sum_{i=1}^{t} g_i \cdot f[c_i](\mathbf{s}) &= \sum_{i=1}^{t} g_i\big(2e_i + m_i\big) \\ &= 2\left(\sum_{i=1}^{t} g_i e_i\right) + \sum_{i=1}^{t} g_i m_i \\ &= 2\left(\sum_{i=1}^{t} g_i e_i\right) + \left(\sum_{i=1}^{t} g_i m_i - \left(\sum_{i=1}^{t} g_i m_i \mod 2\right)\right) \\ &\quad + \left(\sum_{i=1}^{t} g_i m_i \mod 2\right) \\ &= \underbrace{2\left(\sum_{i=1}^{t} g_i e_i + \frac{1}{2}\left(\sum_{i=1}^{t} g_i m_i - \left(\sum_{i=1}^{t} g_i m_i \mod 2\right)\right)\right)}_{\text{noise}} \\ &\quad + \underbrace{\left(\sum_{i=1}^{t} g_i m_i \mod 2\right)}_{\text{ciphertext}}.\end{aligned}$$

Now, assuming that $|e_i| \leq \mathcal{E}$, the noise in the ciphertext for the sum is bounded by

$$\begin{aligned}
\text{sum noise } &= \sum_{i=1}^{t} g_i e_i + \frac{1}{2}\left(\sum_{i=1}^{t} g_i m_i - \left(\sum_{i=1}^{t} g_i m_i \mod 2\right)\right) \\
&\leq \mathcal{E}\sum_{i=1}^{t} g_i + \frac{1}{2}\left(\sum_{i=1}^{t} g_i\right) \\
&\leq \mathcal{E}\|\mathbf{g}\|_t + \frac{\|\mathbf{g}\|_1}{2} \\
&\leq \|\mathbf{g}\|_1\left(\mathcal{E} + \frac{1}{2}\right). \qquad (3.4)
\end{aligned}$$

The system is also multiplicatively homomorphic, but it is not quite as straightforward to see. Multiplying two ciphertext polynomials produces the following expression

$$\begin{aligned}
f[c_1](\mathbf{x}) \cdot f[c_2](\mathbf{x}) &= \left(b_1 - \sum_{i=1}^{n} \mathbf{a}_{1[i]} \cdot \mathbf{x}_{[i]}\right) \cdot \left(b_2 - \sum_{i=1}^{n} \mathbf{a}_{2[i]} \cdot \mathbf{x}_{[i]}\right) \\
&= \left(\left(\langle \mathbf{a}_1, \mathbf{s}\rangle + 2e_1 + m_1\right) - \sum_{i=1}^{n} \mathbf{a}_{1[i]} \cdot \mathbf{x}_{[i]}\right) \\
&\quad \cdot \left(\left(\langle \mathbf{a}_2, \mathbf{s}\rangle + 2e_2 + m_2\right) - \sum_{i=1}^{n} \mathbf{a}_{2[i]} \cdot \mathbf{x}_{[i]}\right) \\
&= \left(\left(\langle \mathbf{a}_1, \mathbf{s}\rangle + 2e_1 + m_1\right) \cdot \left(\langle \mathbf{a}_2, \mathbf{s}\rangle + 2e_2 + m_2\right)\right) \\
&\quad - \left(\langle \mathbf{a}_1, \mathbf{s}\rangle + 2e_1 + m_1\right) \cdot \sum_{i=1}^{n} \mathbf{a}_{2[i]} \cdot \mathbf{x}_{[i]} \\
&\quad - \left(\langle \mathbf{a}_2, \mathbf{s}\rangle + 2e_2 + m_2\right) \cdot \sum_{i=1}^{n} \mathbf{a}_{1[i]} \cdot \mathbf{x}_{[i]} \\
&\quad + \left(\sum_{i=1}^{n} \mathbf{a}_{1[i]} \cdot \mathbf{x}_{[i]}\right) \cdot \left(\sum_{i=1}^{n} \mathbf{a}_{2[i]} \cdot \mathbf{x}_{[i]}\right)
\end{aligned}$$

$$
\begin{aligned}
f[c_1](\mathbf{x}) \cdot f[c_2](\mathbf{x}) = \Big(& \langle \mathbf{a}_1, \mathbf{s} \rangle \cdot \langle \mathbf{a}_2, \mathbf{s} \rangle + 2e_2 \langle \mathbf{a}_1, \mathbf{s} \rangle + m_2 \langle \mathbf{a}_1, \mathbf{s} \rangle \\
& + 2e_1 \langle \mathbf{a}_2, \mathbf{s} \rangle + 4e_1 e_2 + 2e_1 m_2 \\
& + m_1 \langle \mathbf{a}_2, \mathbf{s} \rangle + 2e_2 m_1 + m_1 m_2 \Big) \\
& - \langle \mathbf{a}_1, \mathbf{s} \rangle \cdot \langle \mathbf{a}_2, \mathbf{x} \rangle - 2e_1 \langle \mathbf{a}_2, \mathbf{x} \rangle - m_1 \langle \mathbf{a}_2, \mathbf{x} \rangle \\
& - \langle \mathbf{a}_2, \mathbf{s} \rangle \cdot \langle \mathbf{a}_1, \mathbf{x} \rangle - 2e_2 \langle \mathbf{a}_1, \mathbf{x} \rangle - m_2 \langle \mathbf{a}_1, \mathbf{x} \rangle \\
& + \langle \mathbf{a}_1, \mathbf{x} \rangle \cdot \langle \mathbf{a}_2, \mathbf{x} \rangle
\end{aligned}
$$

which is a valid encryption of $m_1 m_2$ provided $|4e_1e_2 + 2e_1m_2 + 2e_2m_1| < \frac{q}{4} - \frac{1}{2}$ since all the terms with inner products will cancel when the ciphertext polynomial is evaluated at $\mathbf{s}$. In a more concise fashion, we see that the above ciphertext polynomial for the product is now quadratic and can be written as

$$
f[c_1](\mathbf{x}) \cdot f[c_2](\mathbf{x}) = h_0 + \sum_{i=1}^{n} h_i \cdot \mathbf{x}_{[i]} + \sum_{i=1}^{n} \sum_{j>i}^{n} h_{i,j} \cdot \mathbf{x}_{[i]} \mathbf{x}_{[j]} \tag{3.5}
$$

for some coefficients, h_i and $h_{i,j}$, with $h_0 = b_1 b_2$. However, this new ciphertext polynomial has grown in degree and will continue to do so every time a multiplication operation is evaluated. This violates the desired property of compactness in the ciphertexts, which requires that a ciphertext which has been used in a homomorphic operation be the same size as a fresh ciphertext.

To alleviate this problem, the authors do something similar to what Gentry did in his original approach when he squashed the decryption circuit to reduce its complexity. (Though in this case, they are not squashing the decryption circuit, but rather modifying the ciphertext.) They include a "hint" about the secret key $\mathbf{s}$ in the public key. This hint is the quadratic terms $\mathbf{s}_{[i]}\mathbf{s}_{[j]}$, with $i < j$, and the linear terms $\mathbf{s}_{[i]}$ "pseudo-encrypted" under a new secret key $\mathbf{t}$. (They are pseudo-encryptions

since normally bits are encrypted, but here they "encrypt" integers.) The pseudo-encryptions of these hints look like the following:

$$\begin{aligned} E(\mathbf{s}_{[i]}, \mathbf{t}) &= \left(\langle \bar{\mathbf{a}}_i, \mathbf{t}\rangle + 2\bar{e}_i + \mathbf{s}_{[i]}\right) - \langle \bar{\mathbf{a}}_i, \mathbf{x}\rangle \\ &\Rightarrow \left(\langle \bar{\mathbf{a}}_i, \mathbf{t}\rangle + 2\bar{e}_i + \mathbf{s}_{[i]}\right) - \langle \bar{\mathbf{a}}_i, \mathbf{t}\rangle = \mathbf{s}_{[i]} + 2\bar{e}_i \\ E(\mathbf{s}_{[i]}\mathbf{s}_{[j]}, \mathbf{t}) &= \left(\langle \bar{\mathbf{a}}_{i,j}, \mathbf{t}\rangle + 2\bar{e}_{i,j} + \mathbf{s}_{[i]}\mathbf{s}_{[j]}\right) - \langle \bar{\mathbf{a}}_{i,j}, \mathbf{x}\rangle \\ &\Rightarrow \left(\langle \bar{\mathbf{a}}_{i,j}, \mathbf{t}\rangle + 2\bar{e}_{i,j} + \mathbf{s}_{[i]}\mathbf{s}_{[j]}\right) - \langle \bar{\mathbf{a}}_{i,j}, \mathbf{t}\rangle = \mathbf{s}_{[i]}\mathbf{s}_{[j]} + 2\bar{e}_{i,j}. \end{aligned}$$

Substituting the pseudo-encryptions of the secret key components in for $\mathbf{x}_{[i]}$ and $\mathbf{x}_{[i]}\mathbf{x}_{[j]}$ in (3.5) we obtain a new ciphertext polynomial for the product

$$\begin{aligned} f[\overline{c_1 c_2}](\mathbf{x}) = h_0 &+ \sum_{i=1}^{n} h_i \left(\left(\langle \bar{\mathbf{a}}_i, \mathbf{t}\rangle + 2\bar{e}_i + \mathbf{s}_{[i]}\right) - \langle \bar{\mathbf{a}}_i, \mathbf{x}\rangle \right) \\ &+ \sum_{i=1}^{n}\sum_{j>i}^{n} h_{i,j} \left(\left(\langle \bar{\mathbf{a}}_{i,j}, \mathbf{t}\rangle + 2\bar{e}_{i,j} + \mathbf{s}_{[i]}\mathbf{s}_{[j]}\right) - \langle \bar{\mathbf{a}}_{i,j}, \mathbf{x}\rangle \right). \end{aligned}$$

This expression is once again a linear function, but now it is an encryption of $m_1 m_2$ under $\mathbf{t}$. This process, not surprisingly, is called re-linearization. The encryption under $\mathbf{t}$ is slightly more noisy than the original encryption under $\mathbf{s}$,

$$\begin{aligned} f[\overline{c_1 c_2}](\mathbf{t}) &= f[c_1 c_2](\mathbf{s}) + \sum_{i=1}^{n} 2h_i\bar{e}_i + \sum_{i=1}^{n}\sum_{j>i}^{n} 2h_{i,j}\bar{e}_{i,j} \\ &= m_1 m_2 + 2\left(2e_1 e_2 + e_1 m_2 + e_2 m_1 + \sum_{i=1}^{n} h_i\bar{e}_i + \sum_{i=1}^{n}\sum_{j>i}^{n} h_{i,j}\bar{e}_{i,j}\right). \end{aligned}$$

The approximation will be a valid encryption of $m_1 m_2$ provided that the magnitude of the parenthesized term is less than $\frac{q}{4} - \frac{1}{2}$. One concern is that since $h_i \in \mathbb{Z}_q$, the noise due to $h_i\bar{e}_i$ is of binary size $|e_i|_{\text{bin}} + \lfloor \log(q) \rfloor$, which may be quite large (similar concerns exist for $h_{i,j}\bar{e}_{i,j}$). To avoid this trouble, instead of publishing pseudo-

encryptions of the linear and quadratic terms of $\mathbf{s}$ in an attempt to approximate $h_i\mathbf{s}_i$ and $h_{i,j}\mathbf{s}_i\mathbf{s}_j$ directly, they consider the binary representation of h_i and $h_{i,j}$

$$h_i = \sum_{r=0}^{\lfloor \log(q) \rfloor} 2^r h_{i,r}$$
$$h_{i,j} = \sum_{r=0}^{\lfloor \log(q) \rfloor} 2^r h_{i,j,r}$$

where $h_{i,r}, h_{i,j,r} \in \{0,1\}$. Then the products $h_i\mathbf{s}_i$ and $h_{i,j}\mathbf{s}_i\mathbf{s}_j$ can be written as

$$h_i\mathbf{s}_i = \sum_{r=0}^{\lfloor \log(q) \rfloor} 2^r h_{i,r}\mathbf{s}_i$$
$$h_{i,j}\mathbf{s}_i\mathbf{s}_j = \sum_{r=0}^{\lfloor \log(q) \rfloor} 2^r h_{i,j,r}\mathbf{s}_i\mathbf{s}_j.$$

This allows them to publish pseudo-encryptions of multiples of the linear and quadratic terms so that only a sum of the noise is introduced rather than the product of the noise with an element in $\mathbb{Z}_q$. (Basically, instead of having h_i multiplied by a small error term resulting in a potentially large total error term, we have a sum of small error terms, one for each bit of h_i, resulting in a much smaller total error term.) More precisely the following values will replace the pseudo-encryptions of the linear and quadratic terms in the public key for $1 \leq i < j \leq n$ and $0 \leq r \leq \lfloor \log(q) \rfloor$.

$$E\left(2^r\mathbf{s}_{[i]}, \mathbf{t}\right) = \left(\langle \bar{\mathbf{a}}_{\mathbf{i,r}}, \mathbf{t}\rangle + 2\bar{e}_{i,r} + 2^r\mathbf{s}_{[i]}\right) - \langle \bar{\mathbf{a}}_{\mathbf{i,r}}, \mathbf{x}\rangle$$
$$E\left(2^r\mathbf{s}_{[i]}\mathbf{s}_{[j]}, \mathbf{t}\right) = \left(\langle \bar{\mathbf{a}}_{\mathbf{i,j,r}}, \mathbf{t}\rangle + 2\bar{e}_{i,j,r} + 2^r\mathbf{s}_{[i]}\mathbf{s}_{[j]}\right) - \langle \bar{\mathbf{a}}_{\mathbf{i,j,r}}, \mathbf{x}\rangle$$

The ciphertext polynomial for the product is now

$$f[\overline{c_1c_2}](\mathbf{x}) = h_0 + \sum_{i=1}^{n}\sum_{r=0}^{\lfloor \log(q)\rfloor} h_{i,r}\Big(\big(\langle \bar{\mathbf{a}}_{i,r}, \mathbf{t}\rangle + 2\bar{e}_{i,r} + 2^r \mathbf{s}_{[i]}\big) - \langle \bar{\mathbf{a}}_{i,r}, \mathbf{x}\rangle\Big) + \sum_{i=1}^{n}\sum_{j>i}^{n}\sum_{r=0}^{\lfloor \log(q)\rfloor} h_{i,j,r}\Big(\big(\langle \bar{\mathbf{a}}_{i,j,r}, \mathbf{t}\rangle + 2\bar{e}_{i,j,r} + 2^r \mathbf{s}_{[i]}\mathbf{s}_{[j]}\big) - \langle \bar{\mathbf{a}}_{i,j,r}, \mathbf{x}\rangle\Big)$$

and the evaluation at $\mathbf{t}$ of the ciphertext polynomial for the product becomes

$$f[\overline{c_1c_2}](\mathbf{t}) = m_1m_2 + 2\Bigg(2e_1e_2 + e_1m_2 + e_2m_1 + \sum_{i=1}^{n}\sum_{r=0}^{\lfloor \log(q)\rfloor} h_{i,r}\bar{e}_{i,r} + \sum_{i=1}^{n}\sum_{j>i}^{n}\sum_{r=0}^{\lfloor \log(q)\rfloor} h_{i,j,r}\bar{e}_{i,j,r}\Bigg).$$

From this expression, we see that the binary size of the noise in the new ciphertext under $\mathbf{t}$ is on the order of $\max\left(|e_i|^2_{\text{bin}}, \frac{n^2}{2}\left(\lfloor \log(q)\rfloor + 1\right)|e_{i,r}|_{\text{bin}}\right)$.

More rigorously, if we assume that $|e_i| \leq \mathcal{E}$ for $i = 1, 2$ (input noise) and that $|e_{i,r}| < \mathcal{B}$ and $|e_{i,j,r}| < \mathcal{B}$ (fresh noise), then the noise in the product ciphertext is bounded by

$$\begin{aligned}
\text{2-product noise} &\leq 2\mathcal{E}^2 + \mathcal{E} + \mathcal{E} + n\left(\lfloor \log(q)\rfloor + 1\right)\mathcal{B} + \frac{n^2}{2}\left(\lfloor \log(q)\rfloor + 1\right)\mathcal{B} \\
&\leq 2(\mathcal{E}^2 + \mathcal{E}) + \left(n + \frac{n^2}{2}\right)\left(\lfloor \log(q)\rfloor + 1\right)\mathcal{B} \\
&\leq 3\mathcal{E}^2 + n^2\left(\lfloor \log(q)\rfloor + 1\right)\mathcal{B} \\
&\leq 3\mathcal{E}^2 n^2\left(\lfloor \log(q)\rfloor + 1\right)\mathcal{B} \qquad (3.6)
\end{aligned}$$

where the second to last line holds assuming that $\mathcal{E} \geq 2$ and $n \geq 2$.

For a system that will handle L levels of multiplication, we need a new key for each level to enable us to re-linearize the ciphertext polynomial for any product. The

KeyGen algorithm will produce a sequence of keys: $\mathbf{s}_0, \mathbf{s}_1, \ldots, \mathbf{s}_L$ that form the secret key chain, and will publish the encryption of the bits of the linear and quadratic terms of each $\mathbf{s}_i$ encrypted under $\mathbf{s}_{i+1}$ as the public key. In addition, each ciphertext will be tagged with a label from 0 to L indicating what key it is encrypted under. The error growth due to multiplications is the dominant factor that governs how large L can be before the ciphertexts become too noisy for the re-linearization technique to work. It turns out that it is possible to obtain $L = n^\epsilon$ for any $\epsilon < 1$.

To compute the product of D ciphertexts, we need a circuit with $L = \lceil \log(D) \rceil$ levels. Assuming that $|e_i| \leq \mathcal{E}$ for each ciphertext and using the bound 3.6 recursively, let $\mathcal{E}_i$ be the noise after level i so that we have the following:

$$\begin{aligned}
\mathcal{E}_1 &\leq 3\mathcal{E}^2 n^2 \left(\lfloor \log(q) \rfloor + 1\right) \mathcal{B} \\
\mathcal{E}_2 &\leq 3\mathcal{E}_1^2 n^2 \left(\lfloor \log(q) \rfloor + 1\right) \mathcal{B} \\
&\leq 3\Big(3\mathcal{E}^2 n^2 \left(\lfloor \log(q) \rfloor + 1\right) \mathcal{B}\Big)^2 n^2 \left(\lfloor \log(q) \rfloor + 1\right) \mathcal{B} \\
&\leq 3^3 \mathcal{E}^4 n^6 \left(\lfloor \log(q) \rfloor + 1\right)^3 \mathcal{B}^3 \\
&\vdots \\
\mathcal{E}_L &\leq \mathcal{E}^{2^L} \Big(3n^2 \left(\lfloor \log(q) \rfloor + 1\right) \mathcal{B}\Big)^{2^L - 1}.
\end{aligned}$$

When computing the product of D ciphertexts (i.e. $L = \lceil \log(D) \rceil$) we have the following bound on the noise in the ciphertext polynomial for the product

$$\begin{aligned}
\text{D-product noise} &\leq \mathcal{E}^{2^{\lceil \log(D) \rceil}} \Big(3n^2 \left(\lfloor \log(q) \rfloor + 1\right) \mathcal{B}\Big)^{2^{\lceil \log(D) \rceil} - 1} \\
&\leq \mathcal{E}^{2D} \Big(3n^2 \left(\log(q) + 1\right) \mathcal{B}\Big)^{2D-1}. \qquad (3.7)
\end{aligned}$$

The system as described so far is a SHS and is summarized below.

Function	**Input**	**Output**	**Formula**
KeyGen	security parameter (λ)	secret key chain: $\{\mathbf{s}_i\}_{i=0}^{L}$ with $\mathbf{s}_i \in \mathbb{Z}_q^n$ re-linearization hints: linear: $\{(b_{i,r}, \bar{\mathbf{a}}_{i,r})\}$ quadratic: $\{(b_{i,j,r}, \bar{\mathbf{a}}_{i,j,r})\}$	$\mathbf{s}_i$ is chosen uniform randomly
Encrypt	plaintext (m) and $\mathbf{s}_0$	ciphertext $\big(f[c](\mathbf{x}) \in \mathbb{Z}_q[\mathbf{x}]\big)$	$b = \big(\langle \mathbf{a}, \mathbf{s}_0 \rangle + 2e + m\big)$ $f[c](\mathbf{x}) = b - \sum_{i=1}^{n} \mathbf{a}_{[i]} \cdot \mathbf{x}_{[i]}$
Decrypt	c and $\mathbf{s}_l$ for some level l	m	$f[c](\mathbf{s}_l) \mod 2$

Table 3.6: LWE Symmetric Key SHS

To homomorphically evaluate a function on encrypted data using this SHS, one breaks the circuit representation of the function up into levels where the bits of the intermediate results are involved in at most a single multiplication of the input bits. After each level, all these intermediate results (including those involved only in additions) are re-linearized and evaluation proceeds with the next level. This is done so that each level starts with inputs that are encrypted under the same key.

When using results from a higher level than the previous one, it may be necessary to perform "blank homomorphic operations" to bring the key level of the ciphertext polynomial to the correct level. This is done by performing the re-linearization technique on the ciphertext polynomial (several times possibly), though of course

there will not be any quadratic terms in a valid ciphertext polynomial.

Given a ciphertext polynomial $f[c](\mathbf{x})$ under a key $\mathbf{s}_l$ we can transform it into a ciphertext polynomial under $\mathbf{s}_{l+1}$ as follows:

$$f[c](\mathbf{x}) = (\langle \mathbf{a}, \mathbf{s}_l \rangle + 2e + m) - \sum_{i=1}^{n} \sum_{r=0}^{\lfloor \log(q) \rfloor} 2^r \mathbf{a}_{[i](r)} \mathbf{x}_{[i]}$$

$$f[\bar{c}](\mathbf{x}) = (\langle \mathbf{a}, \mathbf{s}_l \rangle + 2e + m) - \sum_{i=1}^{n} \sum_{r=0}^{\lfloor \log(q) \rfloor} \mathbf{a}_{[i](r)} \left(\left(\langle \bar{a}_{i,r}, \mathbf{s}_{l+1} \rangle + 2\bar{e}_{i,r} + 2^r \mathbf{s}_{l[i]} \right) - \langle \bar{a}_{i,r}, \mathbf{x} \rangle \right).$$

When this new ciphertext is evaluated at $\mathbf{s}_{l+1}$, we have

$$\begin{aligned}
f[\bar{c}](\mathbf{s}_{l+1}) &= (\langle \mathbf{a}, \mathbf{s}_l \rangle + 2e + m) \\
&\quad - \sum_{i=1}^{n} \sum_{r=0}^{\lfloor \log(q) \rfloor} \mathbf{a}_{[i](r)} \left(\left(\langle \bar{a}_{i,r}, \mathbf{s}_{l+1} \rangle + 2\bar{e}_{i,r} + 2^r \mathbf{s}_{l[i]} \right) - \langle \bar{a}_{i,r}, \mathbf{s}_{l+1} \rangle \right) \\
&= (\langle \mathbf{a}, \mathbf{s}_l \rangle + 2e + m) - \sum_{i=1}^{n} \sum_{r=0}^{\lfloor \log(q) \rfloor} \mathbf{a}_{[i](r)} \left(2\bar{e}_{i,r} + 2^r \mathbf{s}_{l[i]} \right) \\
&= (\langle \mathbf{a}, \mathbf{s}_l \rangle + 2e + m) - \sum_{i=1}^{n} \sum_{r=0}^{\lfloor \log(q) \rfloor} \mathbf{a}_{[i](r)} 2\bar{e}_{i,r} - \sum_{i=1}^{n} \sum_{r=0}^{\lfloor \log(q) \rfloor} \mathbf{a}_{[i](r)} 2^r \mathbf{s}_{l[i]} \\
&= (\langle \mathbf{a}, \mathbf{s}_l \rangle + 2e + m) - \sum_{i=1}^{n} \sum_{r=0}^{\lfloor \log(q) \rfloor} \mathbf{a}_{[i](r)} 2\bar{e}_{i,r} - \langle \mathbf{a}, \mathbf{s}_l \rangle \\
&= m + 2 \left(e - \sum_{i=1}^{n} \sum_{r=0}^{\lfloor \log(q) \rfloor} \mathbf{a}_{[i](r)} \bar{e}_{i,r} \right).
\end{aligned}$$

From this error term, we obtain the following bound on the increase in noise when performing a single blank homomorphic operation. Note that $|\bar{e}_{i,r}| \leq \mathcal{B}$ since they are fresh pseudo-encryptions in the public key

$$\text{blank noise increase} \leq n \left(\lfloor \log(q) \rfloor + 1 \right) \mathcal{B}. \tag{3.8}$$

Since we can always perform these blank homomorphic operations on a ciphertext, the decryption circuit will take as input a ciphertext with label L, the maximum level that the system can handle.

To turn the system into a FHS we need to be able to encrypt bits without knowing the secret key. To this end we modify the symmetric key SHS from above to obtain a public key SHS. First, the KeyGen algorithm now includes the following additional steps.

1. Choose a matrix $\mathbf{A} \in \mathbb{Z}_q^{d \times n}$ uniform randomly.
2. Choose a vector $\mathbf{e} \in \mathbb{Z}_q^d$ according to the error distribution.
3. Compute $\mathbf{b} = \mathbf{A}\mathbf{s}_0 + 2\mathbf{e} \in \mathbb{Z}_q^d$.
4. Publish $\mathbf{A}$ and $\mathbf{b}$ as the public keys for encryption.

Second, the Encryption algorithm is modified as follows to encrypt a bit m.

1. Choose $\mathbf{r} \in \mathbb{Z}_2^d$ uniform randomly.
2. Compute $\mathbf{a} = \mathbf{A}^\mathsf{T}\mathbf{r}$.
3. Compute $b = \mathbf{b}^\mathsf{T}\mathbf{r} + m$.
4. Output the ciphertext polynomial $f[c](\mathbf{x}) = b - \sum_{i=1}^{n} \mathbf{a}_{[i]}\mathbf{x}_{[i]}$.

The revised system is summarized below.

Algorithm	**Input**	**Output**	**Formula**
KeyGen	security parameter (λ)	secret key chain: $\{\mathbf{s}_i\}_{i=0}^{L}$ with $\mathbf{s}_i \in \mathbb{Z}_q^n$ public key: $\mathbf{A} \in Z_q^{d \times n}$ $\mathbf{b} \in \mathbb{Z}_q^d$ re-linearization hints: linear: $\{(b_{i,r}, \bar{\mathbf{a}}_{i,r})\}$ quadratic: $\{(b_{i,j,r}, \bar{\mathbf{a}}_{i,j,r})\}$	$\mathbf{s}_i$ and $\mathbf{A}$ are chosen uniform randomly $\mathbf{e}$ is chosen from the error distribution $\mathbf{b} = \mathbf{A}\mathbf{s}_0 + 2\mathbf{e}$
Encrypt	plaintext (m)	ciphertext $\big(f[c](\mathbf{x}) \in \mathbb{Z}_q[\mathbf{x}]\big)$	$\mathbf{r}$ is chosen uniform randomly $\mathbf{a} = \mathbf{A}^\mathsf{T}\mathbf{r}$ $b = \mathbf{b}^\mathsf{T}\mathbf{r} + m$ $f[c](\mathbf{x}) = b - \sum_{i=1}^{n} \mathbf{a}_{[i]} \cdot \mathbf{x}_{[i]}$
Decrypt	c and $\mathbf{s}_L$	m	$f[c](\mathbf{s}_L) \mod 2$

Table 3.7: LWE Public Key SHS

Note, the following points about this system. First, we can assume that the Decrypt function always has inputs which are ciphertexts under $\mathbf{S}_L$, since if they are not, one can always perform blank homomorphic operations to bring them to level L. Second,

a fresh ciphertext polynomial has noise bounded by

$$\begin{aligned} \text{level 0 noise} &= \mathbf{e}^{\mathsf{T}}\mathbf{r} \\ &\leq d\mathcal{B}. \end{aligned} \tag{3.9}$$

Now we derive a bound on when we can homomorphically evaluate a polynomial $g(x_1, x_2, \ldots, x_l) \in \mathbb{Z}_2[x_1, x_2, \ldots, x_l]$ with degree D.

1. Assume $|e_i| \leq \mathcal{E}$ for each input to g.

2. Since the degree of g is D, the error in each monomial in g is bounded by

$$\text{monomial noise} = \hat{\mathcal{E}} \leq \mathcal{E}^{2D}\Big(3n^2\left(\log\left(q\right)+1\right)\mathcal{B}\Big)^{2D-1}.$$

3. Since g is a function on l inputs and has degree D, there can be at most $\binom{l}{D} \leq l^D$ monomial terms. Also, since g is a function over $\mathbb{F}_2$ each coefficient can be at most 1 so $\|g\|_1 \leq l^D$. This means the error in the sum of the monomial terms is bounded by

$$\begin{aligned} \text{term sum noise} &\leq \|g\|_1\left(\hat{\mathcal{E}}+\frac{1}{2}\right) \\ &\leq l^D\left(\mathcal{E}^{2D}\Big(3n^2\left(\log\left(q\right)+1\right)\mathcal{B}\Big)^{2D-1}+\frac{1}{2}\right). \end{aligned}$$

4. Now to decrypt this ciphertext we may need to perform several blank homomorphic operations to bring the level to L. Using the bound 3.8 at most L times we have

$$\text{blank noise increase for } g \leq Ln\left(\lfloor\log\left(q\right)\rfloor+1\right)\mathcal{B}.$$

5. Combining the above together and using the fact that $\mathcal{E} \leq d\mathcal{B}$ for the public key system, we see that

$$\begin{aligned}
\text{noise in } g &\leq l^D \left(\mathcal{E}^{2D} \Big(3n^2 \left(\log\left(q\right) + 1 \right) \mathcal{B} \Big)^{2D-1} + \frac{1}{2} \right) + Ln \left(\lfloor \log\left(q\right) \rfloor + 1 \right) \mathcal{B} \\
&\leq l^D \left((d\mathcal{B})^{2D} \Big(3n^2 \left(\log\left(q\right) + 1 \right) \mathcal{B} \Big)^{2D-1} + \frac{1}{2} \right) + Ln \left(\lfloor \log\left(q\right) \rfloor + 1 \right) \mathcal{B} \\
&\leq l^D \left((d\mathcal{B})^{2D} \Big(3n^2 \left(\log\left(q\right) + 1 \right) \mathcal{B} \Big)^{2D-1} + Ln \left(\log\left(q\right) + 1 \right) \mathcal{B} \right) \\
&\leq l^D L \mathcal{B}^{4D-1} \Big(3dn^2 \left(\log\left(q\right) + 1 \right) \Big)^{2D}
\end{aligned}$$

where the third inequality holds if $l > 1$ for dropping $\frac{1}{2}$.

For the SHS to correctly evaluate the polynomial g homomorphically, we need to have

$$l^D L \mathcal{B}^{4D-1} \Big(3dn^2 \left(\log\left(q\right) + 1 \right) \Big)^{2D} < \frac{q}{4} - \frac{1}{2}. \tag{3.10}$$

For any constant $C_1 \in \mathbb{N}$ and $\epsilon \in (0, 1)$, we can set the parameters as follows to satisfy (3.10).

$$\begin{aligned}
n &\geq \lambda & L &= n^{\epsilon} \\
q &= 2^L & D &= \frac{CL}{\log(n)} \\
l &= n^{C_1} & \mathcal{B} &= n \\
C &= \frac{1}{3(C_1 + 10 + 2\log(12) + 4\epsilon)} & d &= n\log(q) + 2\lambda
\end{aligned}$$

Plugging these values into (3.10) we verify that

$$
\begin{aligned}
l^D L\mathcal{B}^{4D-1}\Big(3dn^2\left(\log\left(q\right)+1\right)\Big)^{2D} &= n^{C_1 D} L n^{4D-1}\Big(3dn^2\left(\log\left(q\right)+1\right)\Big)^{2D} \\
&\leq L n^{C_1 D+4D-1}\Big(6(n\log(q)+2\lambda)n^2\log\left(q\right)\Big)^{2D} \\
&\leq L n^{C_1 D+4D-1}(6(nL+2n)n^2 L)^{2D} \\
&\leq \frac{L}{n} n^{(C_1+10)D}(6(L+2)L)^{2D} \\
&\leq n^{(C_1+10)D}(12L^2)^{2D} \\
&\leq n^{(C_1+10)D}(12n^{2\epsilon})^{2D} \\
&\leq n^{(C_1+10+4\epsilon)D}(12)^{2D} \\
&\leq n^{(C_1+10+4\epsilon+2\log(12))D} \\
&\leq n^{(C_1+10+2\log(12)+4\epsilon)\frac{CL}{\log(n)}} \\
&\leq n^{\frac{L}{3\log(n)}} \\
&\leq 2^{\frac{L}{3}} \\
&< \frac{q}{4}-\frac{1}{2}
\end{aligned}
$$

where the fifth line holds if $L \geq 2$, the eighth line holds since $n > 2$, and the final lines holds if $L \geq 4$. Note, that for circuits with $L < 4$ it can be shown directly that they satisfy the bound, but due to our overestimating throughout the derivation we cannot conclude that from this result. However, once we do that, we conclude that this SHS can evaluate any polynomial of degree D on l variables in $\mathbb{Z}_2$.

To obtain a FHS, we need to be able to evaluate the decryption function with less than L levels of multiplication in order to do bootstrapping. Unfortunately, the decryption function requires at least $\max\left(n, \log q\right)$ levels of multiplication (due to the computation of $\langle \mathbf{a}, \mathbf{s} \rangle$) which will be greater than L since $L = n^{\epsilon} < n$. To handle this

difficulty they introduce a dimension reduction technique. First, notice that in the previous discussion on re-linearization, the second secret key $\mathbf{t}$ could have had dimension smaller than n say k. Second, for some $p < q$ we can approximate elements in $\mathbb{Z}_q$ with elements in $\mathbb{Z}_p$ by scaling the elements down. Using these two notions, they are able to reduce the complexity of the decryption function to have about $\max(k, \log p)$ levels of multiplication which is less than L. Once the decryption function can be processed by the SHS, bootstrapping can be applied in the same manner as described in Gentry's approach above and they obtain a FHS. (The encryptions of the bits of $\mathbf{s}_L$ under $\mathbf{s}_0$ are added to the public key so that homomorphic evaluation of the decryption circuit can produce "fresh" ciphertexts under $\mathbf{s}_0$.) One advantage of this approach is that the bits of one of the secret keys are never encrypted under the same secret key. Rather they are encrypted under a new secret key (the next one in the chain) so they need only assume that the system is weakly circular secure.

In the paper, the authors note that the bootstrapping step can be accomplished without squashing the decryption circuit, but in essence they have just repurposed the hint in the public key from helping decrease the complexity of the decryption circuit (squashing in Gentry based approaches) to preventing the expansion of the ciphertexts during multiplication operations (re-linearization in this approach) and shortening the ciphertexts at the end (dimension reduction) to make the decryption circuit smaller.

3.3.1 Improvements to LWE Systems

Following up quickly on his work earlier in the year, Brakerski in [11] demonstrates a cryptosystem that uses the Ring-LWE problem, or polynomial LWE (PLWE) as they refer to it, as a security basis. The Ring-LWE problem is the natural analog

of the standard LWE problem but phrased in terms of polynomial rings. It was first posed in [33] and is defined as follows.

Problem 11 (Ring Learning With Errors Problem (Ring-LWE))
Given poly(n) samples of the form $\Big(a(x), b(x) = a(x) \cdot s(x) + e(x)\Big)$ *where* $a(x), s(x) \in \mathbb{Z}_q[x]/(x^n+1)$ *with* $a(x)$ *being chosen from a uniform random distribution and* $s(x)$ *being fixed for all pairs and* $b(x), e(x) \in \mathbb{Z}[x]/(x^n+1)$ *with* $e(x)$ *chosen from a Gaussian distribution centered about* 0*, determine the value of* $s(x)$*.*

In their paper they actually have $s(x)$ taken from the error distribution, but they show that this is basically equivalent to selecting $s(x)$ uniformly at random. The purpose of the Gaussian distribution centered about 0 is to ensure that the error polynomials, $e(x)$, have "small" coefficients. The main contribution of this work is creating a system that is provably circularly secure. Previous works based on Gentry's original approach all assumed circular security (since they could not prove it) when they encrypted a hint about the secret key in the public key. They show that their system is secure when an attacker has access to an encryption of any polynomial function of the secret key.

In another 2011 paper [8], basically identical to [9], Brakerski showed how to construct a leveled fully homomorphic system based on Ring-LWE that doesn't use bootstrapping. To do so, this paper was the first to introduce the technique of modulus switching. (This is the technique that Coron adapted in [17] to improve a Gentry type system.) The modulus switching technique follows from the dimension reduction idea in [10] that by scaling and rounding carefully, for $p < q$ elements in $\mathbb{Z}_q$ can be approximated by elements in $\mathbb{Z}_p$. Here the notation $[x]_m$ denotes modulus reduction of x by m into the range $\left(\frac{-m}{2}, \frac{m}{2}\right]$.

Definition 6 (Modulus Switching Technique)

For odd integers p, q with $p < q$ and integer vector $\mathbf{c}$ let $\bar{\mathbf{c}}$ be the integer vector closest to $\frac{p}{q} \cdot \mathbf{c}$ such that $\bar{\mathbf{c}} \equiv \mathbf{c} \mod 2$. Then, for any $\mathbf{s}$ with $|[\langle \mathbf{c}, \mathbf{s} \rangle]_q| < \frac{q}{2} - \frac{q}{p} \cdot \|\mathbf{s}\|_1$, we have the following:

1. $[\langle \bar{\mathbf{c}}, \mathbf{s} \rangle]_p \equiv [\langle \mathbf{c}, \mathbf{s} \rangle]_q \mod 2$
2. $|[\langle \bar{\mathbf{c}}, \mathbf{s} \rangle]_p| < \frac{p}{q} \cdot |[\langle \mathbf{c}, \mathbf{s} \rangle]_q| + \|\mathbf{s}\|_1$

As discussed before, this scaling maintains the ratio of the noise size to ciphertext size. However, the authors here note that by keeping the noise size down the rate of increase in the error size is reduced. So this technique is applied after each ciphertext multiplication in a tradeoff between the modulus size and the error size.

The main focus of this approach is to reduce the per-gate computation. They note that Gentry based systems require $poly(\lambda)$ computation, and previous LWE approaches required $O(\lambda^{3.5})$ computation, but this approach requires $O(\lambda \cdot L^3)$ computation where L is the number of levels in the circuit. Furthermore, using bootstrapping as an optimization they can actually achieve $O(\lambda^2)$ computation per-gate.

One interesting practical result of this paper is that Shai Halevi has created a low level C++ implementation of the cryptosystem described in the paper. The source code is available at `https://github.com/shaih/HElib`. Unfortunately, as of August 2014, it does not include the bootstrapping portion yet as the programming to do so is quite involved.

In 2012, Brakerski proposed a scale invariant homomorphic encryption scheme in [7] where the noise growth when multiplying ciphertexts grows linearly. He did so without using modulus switching and the security was classically reduced from the worst-case hardness of the GapSVP problem.

In 2013, Gentry, Sahai, and Waters proposed an improved version of the LWE based system that avoided the re-linearization step [23]. In this system, homomor-

phic operations are carried out with matrix operations leading to a faster and more simplistic system. Using this scheme, they created the first identity-based fully homomorphic encryption scheme and an attribute-based fully homomorphic encryption scheme.

3.4 Functional Encryption

As research efforts continued on reducing the size of the ciphertexts and the amount of computation involved in processing each gate of a circuit, it became clear that without a major breakthrough fully homomorphic encryption was not feasible for most situations. This realization prompted the research community to turn its attention to systems that would be homomorphic for only certain classes of functions which were presumably functions of interest. For example, the IARPA SPAR project (solicitation number IARPA-BAA-11-01) was a three and half year project started in 2011 that asked for researchers to come up with systems that could homomorphically evaluate some basic SQL commands such as SELECT, INSERT, UPDATE, and DELETE on a 10 TB database containing 100 million records.

In addition, researchers recognized challenges that could not be solved by homomorphic encryption alone. One such example is an oblivious spam filter. In this situation, a user of a cloud based email system may wish to have the cloud filter their emails for spam without revealing the contents of their emails to the cloud. At first glance, this may seem like yet another situation where Homomorphic Encryption saves the day. The user can publish a public key so that all emails sent to them are encrypted. The cloud can then homomorphically evaluate their spam identification function on the encrypted emails. However, the result of this computation will be the encryption of whether the email is deemed to be spam or not. The user is then faced

with a problem, either they must interact with the cloud for each email to decrypt this result and inform the cloud as to whether the email is spam or not (clearly this will not work) or they can give the cloud their private key and the emails are no longer private since the cloud can decrypt all the emails as well as the result of their computation. In this situation, the result of the homomorphic evaluation should be the result of the function and not an encryption of the result to allow the spam filter to decide (obliviously) as to whether the email is spam or not.

In the summer of 2013, Shaffi Goldwasser and colleagues [25] proposed a radically new approach to Homomorphic Encryption which allowed only specific functions, identified in advance by the holder of the secret key, to be evaluated homomorphically over any ciphertexts. Basically, the holder of the secret key can generate an access key for a specific function (or in some cases a family of functions). This access key allows any user to obtain the plaintext result of applying the function to the encrypted data, but does not allow the user to obtain the plaintext result of applying any other function to the encrypted data. The function may be applied to several sets of encrypted data and the plaintext result will be obtained for each set. This approach is known as functional encryption, since the access keys are function specific.

The functional encryption scheme proposed by Goldwasser utilizes three key components: garbled circuits, attribute based encryption, and fully homomorphic encryption. We now examine the first two components before presenting an overview of this approach.

3.4.1 Components

Garbled circuits were first introduced by Yao in [51] though he did not use the specific term. The idea is very similar to homomorphic encryption. For a function f

and an input x if one is given a garbled version of the function $\bar{f}$ and a garbled version of the input $\bar{x}$ one should be able to compute $f(x)$. This differs from homomorphic encryption in that one cannot compute any function on any encrypted data. Rather for the specific function - data pair (f, x) one can work with the garbled versions to obtain $f(x)$.

An important component in a garbling scheme is a Double-Key Cipher which we define below.

Definition 7 (Double-Key Cipher)

A Double-Key Cipher is an encryption scheme that requires 2 k-bit keys to encrypt and decrypt k-bit strings. For keys a and b and a string m, there is an encryption function, $E_{a,b}(m) = c$, and a decryption function, $D_{a,b}(c) = m$ as usual. Note that the order of the keys is significant as $E_{a,b}(m) \neq E_{b,a}(m)$ and $D_{a,b}(c) \neq D_{b,a}(c)$.

First, we show how to garble a single gate of a circuit. A typical gate in a circuit looks something like the following (we will use this AND gate to illustrate the process for our discussion below).

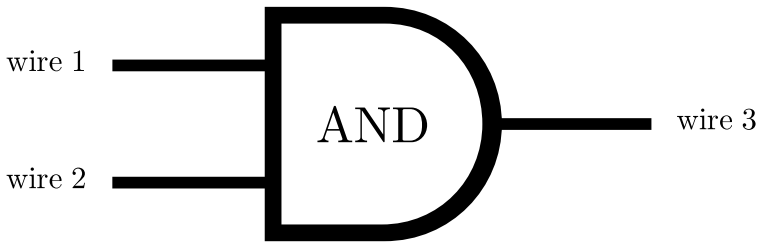

Figure 3.3: AND Gate

Each gate has two incoming wires and an outgoing wire. The initial incoming wires for the first level of gates are called input wires and the final outgoing wires for the last level of gates are called output wires. Each wire carries a token, which

is a k-bit binary string. The tokens encode a one-bit type by having the final bit of the token be the type. The type of a token is the apparent Boolean value that it represents and will be visible to the one who is evaluating the garbled circuit.

1. For each wire (incoming and outgoing) select 2 tokens: one of each type.

Wire 1	Wire 2	Wire 3
01100	11010	11010
10101	10001	01111

Table 3.8: Tokens for wires

2. Assign the semantics of 0 to one token and the semantics of 1 to the other token (the assigned semantics may not correspond to the type). The semantics of a token is the true Boolean value that it represents and will be hidden from the one who is evaluating the garbled circuit until the output of the very last level of gates. We denote a token for wire i with semantics of b as X_i^b.

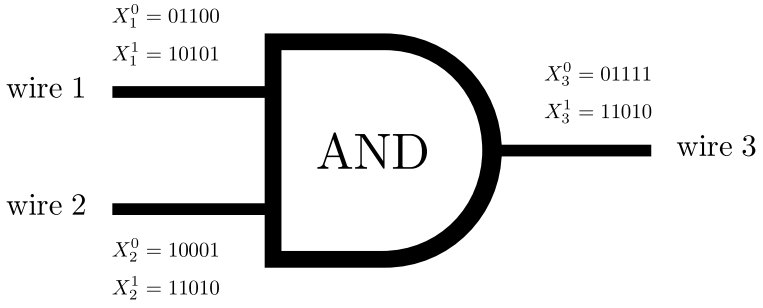

Figure 3.4: AND Gate with Tokens

3. Compute a garbled truth table for the gate where the output is determined

by the types of the tokens on the top and bottom incoming wires and is the encryption under the double key cipher (using the top and bottom tokens as keys) of the token on the outgoing wire having semantics that correspond to applying the gate to the (hidden) semantic values of the tokens on the incoming wires.

semantics			types		
top	bottom	output	top	bottom	output
0	0	0	0	1	$E_{X_1^0,X_2^0}(X_3^0)$
0	1	0	0	0	$E_{X_1^0,X_2^1}(X_3^0)$
1	0	0	1	1	$E_{X_1^1,X_2^0}(X_3^0)$
1	1	1	1	0	$E_{X_1^1,X_2^1}(X_3^1)$

AND truth table

Garbled truth table (variables)

top	bottom	output
0	1	$E_{01100,10001}(01111)$
0	0	$E_{01100,11010}(01111)$
1	1	$E_{10101,10001}(01111)$
1	0	$E_{10101,11010}(11010)$

Garbled truth table (values)

top	bottom	output
0	1	01110
0	0	10100
1	1	00011
1	0	01110

Garbled truth table (published)

Note: in the published truth table, the values in the output column when decrypted using the tokens corresponding to the types in the top and bottom column provide the corresponding token for the outgoing wire (i.e. $D_{01100,10001}(01110) = 01111 = X_3^0$, though the evaluator will not know the semantic value)

To garble a function f represent it as a Boolean circuit C. Garble each gate in C as described above with the exception that for the output wires (i.e. the outgoing wires in the last level of gates), the type and semantics of each token should match.

To garble an n-bit binary string input m let $\bar{m} = (X_1^{m_1}, X_2^{m_2}, \ldots X_n^{m_n})$, where m_i is the i-th bit of m. Here, we assume that the first n wires correspond to the n bits in the input.

The garbled input, the garbled truth tables for each gate, and the structure of C are given to the evaluator. The evaluator proceeds gate by gate using the tokens he knows to recover the token for each outgoing wire. When he reaches the last level of gates, the type of the tokens on the output wires match their semantics, so the bit values of $f(x)$ may be readily obtained from the last bits of each token.

Note that as presented above, a garbled circuit is a single use primitive since knowing another garbled input that differs in even a single bit will leak information about the tokens that were not revealed during the evaluation of the first garbled input. The above presentation is adapted from [3].

Attribute-based encryption (ABE), a term coined in [45], is a scheme in which successful decryption of a ciphertext is dependent on a public attribute associated with the ciphertext. The following is an overview of the components of such a system which will be used in the discussion that follows regarding functional encryption. Technically, this variant is known as a two-outcome attribute-based encryption scheme. A predicate is a function having a single bit as output.

Algorithm	**Input**	**Output**
Setup	security parameter (λ)	master public key (PK) and master secret key (sk)
KeyGen	predicate (P) and sk	secret key for predicate (sk_P)
Encrypt	PK, an attribute (a), and two messages (m_0 and m_1)	ciphertext (c)
Decrypt	ciphertext and sk_P	returns m_0 if $P(a) = 0$ and m_1 if $P(a) = 1$

Table 3.9: Functions in an Attribute-Based Encryption System

Note that the predicate function is a secret.

3.4.2 Goldwasser et al. Approach

Now we are equipped to survey the functional encryption scheme that Goldwasser and her colleagues proposed. First, we begin with a fully homomorphic system. The data $c = E(m)$ is encrypted and a function f is homomorphically evaluated to obtain the result $r = E(f(m))$. Second, to recover this result without using the secret key of the FHS, one could provide a garbled circuit of the decryption circuit with the secret key hard-coded in. However, to use the garbled circuit, a garbled version of r, namely the input tokens for the bits of r, are needed and there is no way to know what the bits of r will be in advance. (Also, note that using this garbled circuit more

than once will be a problem as discussed above). To solve this conundrum, they use the third component, attribute-based encryption, as follows. Let P_i be the predicate that takes c and returns the i-th bit of r. Then, when doing the ABE encrypting, the attribute is c and $m_0 = X_i^0$ and $m_1 = X_i^1$ (i.e. the tokens with semantics corresponding to 0 and 1 respectively). Thus, running the ABE decryption algorithm will return the correct token for the i-bit of r (and only that token). A summary of the procedure is provided below.

1. The data is encrypted under a fully homomorphic scheme to obtain the ciphertext c.

2. A function f is selected to be evaluated homomorphically.

3. A garbled version of the FHS decryption circuit (with the fully homomorphic secret key hardcoded in) is created.

4. The ABE system is initialized and the KeyGen algorithm is run for each of the n bits that will appear in the output of f using P_i (the predicate that extracts the i-th bit from $r = f(c)$) to obtain sk_P_i for each one.

5. The ABE encryption function is run for each of the n bits using c as the attribute and the tokens from the circuit garbling: X_i^0 and X_i^1 as m_0 and m_1 respectively to produce the ciphertext b_i.

6. The end user (think oblivious spam filter) is given c, sk_P_i and b_i for all i, and the garbled decryption circuit.

7. The end user evaluates the ABE decryption function on b_i and sk_P_i for each i to obtain the tokens: $X_1^{r_1}, X_2^{r_2}, \ldots, X_n^{r_n}$ (though he will not know the semantics r_i of the tokens).

8. Using these tokens, the end user evaluates the garbled decryption circuit and obtains the (plaintext) result.

This system requires that the function f be processed by the holder of the private data before any computation can take place. Each time a function is processed, a new garbled circuit will be generated for the decryption circuit so the garbled circuit is still a one-time use primitive. However, in their paper Goldwasser et al. showed how to extend their results to produce the first reusable garbled circuit scheme.

Chapter 4

Applications

Homomorphic encryption has many potential applications since it allows one to securely delegate the processing of sensitive data to an untrusted party. With the advent of cloud computing, the need for such methods is increasing rapidly. The following examples provide an insight into the wide range of possible applications. Some applications have already been implemented in practice but most are still infeasible due to the inefficiencies of homomorphic encryption. In many of the situations a FHS could be replaced by a SHS if there is an apriori bound on the complexity of the desired computations.

4.1 Private Information Retrieval

The most common situation where homomorphic encryption can be used is in a Private Information Retrieval (PIR) system. Suppose you have some sensitive business information stored in a database that you do not want to allow others to view. However, you would like to have the database stored and managed by some other company that specializes in data management. If an efficient homomorphic scheme

was available, you could encrypt your data and send it to the data management company. They could then host your database and even run queries on it for you. From their perspective, the data would remain encrypted at all times and the results they obtain from running a query for you would be encrypted as well. In essence, they would receive gibberish from you (your encrypted data), they would run a query on the gibberish, which would produce some gibberish results, and they would then send you these results. You would then decrypt these results to obtain the result of running the same query on your unencrypted data. The Defense Advanced Research Projects Agency (DARPA) in the U.S. recently announced plans to spend $20 million USD over a five year period to develop such methods to store and manipulate encrypted data.

A PIR instance is formally defined as follows.

Definition 8 (Private Information Retrieval (PIR))
A two party protocol which allows a user to retrieve a subset of the records held by a server without revealing to the server which of the records were retrieved.

4.1.1 Private Human Genome Sequencing

Another PIR application is providing privacy for distributed computation performed by computers using their spare clock cycles. There are several fairly well known projects such as the Great Internet Mersenne Prime Search (GIMPS) and Search for Extraterrestrial Intelligence (SETI@home) that utilize the spare clock cycles of their user base to perform massive amounts of computation. Now that scientists have mapped out the human genome it is becoming increasingly popular and inexpensive to have genomes sequenced for the general public. To bring the cost down even more, an open source effort similar to those mentioned above could be created

to map any persons DNA. While privacy may not matter too much when searching for large prime numbers or aliens, it is of great concern when dealing with something as personal as your own DNA. The process would look something like the following.

1. A user encrypts their raw DNA data under a homomorphic system.
2. This data is distributed to the many computers on the network and each one homomorphically evaluates the function to map out the genome data. The resulting information is combined together and returned to the user.
3. The user decrypts the result to obtain their sequenced genome.

The most computationally expensive part of DNA sequencing is the error detection and fragment matching performed at the end of the process. This is the part that can be distributed as described above.

4.1.2 Delegation of Computationally Intensive Processing

PIR applications are also very useful for mobile phones or other low powered devices that might want to run powerful applications but lack the resources to do so. Using homomorphic encryption they can hand off any computationally intensive processing to some server and then display the results on the local device. All the while being assured that they have not compromised the security or privacy of their information. For example, one such use case involves an unmanned aerial vehicle (UAV). The smallest UAV is a backpack sized object used by the military to provide reconnaissance during operations. It is launched into the air by hand and can remain airborne for a couple of hours. Due to its size and power constraints the UAV's computer uses most of its clock cycles to keep the vehicle in the air and operate the radio and other sensors. The UAV may be capturing real time video and need to make

decisions based on information obtained from it. However, the algorithms to do image detection on the video and other sophisticated processing for object identification are too complex to be handled by the onboard processor. Additionally, the UAV may not always be able to connect to the data network of its source country, due to being too far away or a network going down during battle. So, the UAV would like to be able to outsource this video processing to any of the coalition partners of its source country. This can be done as follows.

1. The UAV encrypts all of the video data it collects using a fully homomorphic system.

2. The UAV broadcasts this information to any coalition partner network that is available to process the data.

3. The coalition partner runs the various algorithms on the encrypted video data and sends the (encrypted) result back to the UAV.

4. The UAV can then decrypt the data and act accordingly.

This approach has a couple of nice side benefits. First, the UAV can change its encryption key whenever it is concerned that it may have been compromised without doing a key exchange. Second, as it moves spatially over the field of operation it can hop from network to network (or even parallelize the processing by using multiple coalition parties) since it is not tied to using a single base station for processing.

4.1.3 Optimized Cloud Computation

In [38], several Microsoft researchers propose a novel scheme to offload computation to the cloud securely. As pointed out previously, one of the biggest drawbacks

to homomorphic encryption is the large size of the ciphertexts. The steps for a client to send data $M = \{m_1, m_2, \ldots, m_n\}$ to the cloud for processing are as follows.

1. The client encrypts all m_i under a FHS and sends these ciphertexts to the cloud.
2. For each query, the client encrypts the parameters under the FHS and sends them to the cloud.
3. The cloud homomorphically evaluates the query using the encrypted parameters and database that it received from the client.
4. The result of the evaluation is a FHS ciphertext that is sent back to the client.
5. The client decrypts the FHS ciphertext to obtain the plaintext result.

In this situation, all of the transmission across the network is in the form of large FHS ciphertexts. In some cases, the transmission of the data may take longer than the processing of the data. To reduce the amount of network traffic needed to offload data processing to the cloud, the authors propose the following scheme.

1. The client selects a key K_{AES} for use in an AES system, and a key K_{FHS} for a FHS.
2. The client encrypts K_{AES} under the FHS to obtain $\bar{K} = E_{\text{FHS}}(K_{\text{AES}})$.
3. The database is encrypted with AES using K_{AES}.
4. $\bar{K}$ and the encrypted database are sent to the cloud in a one-time setup phase.
5. When the cloud receives the database, it evaluates

$$D_{\text{AES}}\Big(E_{\text{FHS}}\big(E_{\text{AES}}(m_i)\big), E_{\text{FHS}}\big(K_{\text{AES}}\big)\Big)$$

for each database entry m_i. This results in the database being encrypted under the FHS key K_{FHS}. To see that this is the case, note that the AES circuit normally works as follows.

$$D_{\text{AES}}\big(\text{ciphertext}, \text{key}\big) = \text{plaintext}$$
$$D_{\text{AES}}\big(E_{\text{AES}}(m), K_{\text{AES}}\big) = m$$

When evaluated homomorphically though it produces $E_{\text{FHS}}(m)$.

$$D_{\text{AES}}\Big(E_{\text{FHS}}\big(E_{\text{AES}}(m)\big), E_{\text{FHS}}\big(K_{\text{AES}}\big)\Big) = E_{\text{FHS}}(m)$$

6. For each query, the client encrypts the parameters under the AES key and sends them to the cloud.

7. The cloud "upgrades" the AES encrypted parameters to be encrypted under the FHS (just like how the database was upgraded in step 5) and then homomorphically evaluates the query.

8. The result of the evaluation is a FHS ciphertext. This could be sent back to the client, but the goal is to avoid transmitting lengthy FHS ciphertexts across the network if possible. In the paper, the authors propose using an LWE based system as the FHS. In this case, by applying the dimension reduction technique to the FHS ciphertext they can reduce the size drastically and send the shorter ciphertext to the client. This ciphertext is decryptable, but cannot have any more operations performed homomorphically. This is not a problem since the ciphertext is the final output and will only be decrypted.

9. The client receives either the full FHS ciphertext or some shortened version and

decrypts it to obtain the plaintext result.

In this scenario, there is only one FHS ciphertext sent across the network (during the setup phase). All other transmissions are ciphertexts encrypted under AES which are considerably smaller than if they were FHS ciphertexts. This approach trades off a considerable amount of network transmission at the cost of several homomorphic evaluations on the server side. Once again, the server is likely to be quite powerful and this tradeoff is probably an improvement over the original scheme.

4.2 Privacy Preserving Computations

4.2.1 Anonymous Voting

Another useful application is anonymous voting. First solved by David Chaum in [12], this problem involves a group of people who desire to vote on a decision. However, they do not want to reveal the vote of any individual. Using homomorphic encryption, each person encrypts their vote using the public key for a system. The encrypted votes are then added together, and the result is decrypted using the secret key that is held by a small trusted group of talliers to determine the outcome of the vote without needing to expose the actual votes of any of the participants. Other uses could be for anonymous surveys on sensitive issues such as past drug use or criminal records, where the goal is to collect accurate statistics without knowing the individual sample values. In the voting example, only one type of operation needs to be handled, namely addition, so this problem has been solved using systems which are only additively homomorphic such as the Pallier cryptosystem. Other statistics however, would need a fully homomorphic system. In a 2011 paper [38], a group from Microsoft Research demonstrate several basic statistics that can be computed

homomorphically.

4.2.2 Oblivious Prescription Verification

One common specific application is for medical records. Medical records often contain sensitive, extremely personal information. However, to receive proper care or have a prescription filled it may be necessary to run a check on these records to avoid placing the patient in danger. Things to check for might include allergies to medications, a family history of certain types of problems, and other current serious health conditions. This can be handled as follows.

1. The patient encrypts all of their personal data and stores it in a public medical database.

2. When the patient places an order for a certain type of medication, the pharmacist runs a safety check algorithm on the patients data. This algorithm checks all the desired conditions and returns a verification code indicating whether the patient can take this medicine.

3. The pharmacist sends the encrypted output from this algorithm to the patients phone, where they decrypt the result to obtain the verification code.

4. The patient shows this verification code to the pharmacist who uses it to decide whether the patient can safely take the medicine.

To avoid a patient abusing the system and obtaining medicine that is dangerous for them, the verification code should be a function of the medicine being prescribed and some form of physical identification. For example, the verification code might be the hash of the patients drivers license number, the prescription number, and a random key value that the pharmacist used when evaluating the function.

If done correctly, this will prevent a patient from generating a valid code for medicine that they should not take.

4.2.3 Contextualized Advertisements

Using homomorphic encryption one can provide customized contextual advertisements to web browsers on mobile devices without revealing which advertisements are shown to which consumers or the consumers personal information. It works in the following manner.

1. Using a FHS, the consumer encrypts some information about the products they use, their current location, their browsing history, etc. and submits this to the website they are viewing.

2. Companies that buy advertisements from the website encrypt a description of the types of users that their advertisements target (keywords) and submit this information along with the encryption of their advertisements to the advertisement database of the website.

3. To decide which advertisements to show a user, the website uses the clients encrypted information to homomorphically run a matching query on the advertisement database containing the encrypted keywords and advertisements.

4. The result of this query is the set of advertisements for the consumer and these are sent to the consumer.

5. The consumer decrypts and displays the advertisements.

In this scheme the website learns nothing about the consumer's personal information since it is encrypted and is oblivious to which advertisements are sent to the consumer

since the matching is done homomorphically. One inherent downside to this approach is that the advertisement database must be encrypted differently for each user in order to run the matching query homomorphically. Presumably, the advertisers would store their keywords and advertisements in an unencrypted database and the website would encrypt the data on the fly for each visitor and then run the matching query homomorphically. Caching the encrypted versions for each visitor is an option that requires additional space, but saves on computation for future visits.

4.3 Functional Encryption Applications

4.3.1 Shared Patient Records

Another useful situation in the medical realm involves different doctors needing to share information about the same patient. A patient may visit several doctors: a general practitioner, a dentist, an eye doctor, an oncologist, etc. The system works as follows.

1. Each doctor uses the public key of the patient to encrypt the relevant medical records from their office and uploads it to the master record for the patient in the cloud.

2. When the patient is visiting one of his doctors and they need to query some information contained in one of the records from another doctor, the patient can use a functional encryption scheme to generate a key specific to this query.

3. With this key the doctor can check the patients medical records at any time to obtain the plaintext result of the query, but can learn nothing else about their records.

This allows the patient to control who has access to his records and avoids the problem of a doctor refusing to release records for a non-medical reason such as concern that the patient is going to see another doctor.

4.3.2 NSA Record Collection

In 2013, the NSA came under intense public scrutiny for the PRISM program which collects information from the internet and telecommunications companies on a massive scale. Under this program the NSA monitors and stores almost all the communication it can obtain. In some cases the NSA was given a direct connection to a company database and allowed to run queries at will. The idea was that when tracking down criminals or terrorists they could search through this trove of information to establish links between individuals in an attempt to thwart their nefarious activities. In this situation, there is a need to allow the NSA to access this information when it pertains to possible threats to national security, but at the same time there are legitimate privacy concerns if the NSA can eavesdrop on whomever it chooses (including US citizens who have their privacy protected under the Constitution).

It seems impossible to both allow the NSA to have access to all this data while at the same time preventing them from having indiscriminate access to any record they choose. The solution is to use a functional encryption system. The internet and telecommunications companies can encrypt their data under this system and the NSA can store all of the resulting encrypted data. When the NSA identifies individuals that pose a threat to the interests of the US, they can present their case for obtaining access to a judge. Having determined the legitimacy of their request, this judge will then generate them an access key which allows the NSA to obtain information about the specified individuals alone by homomorphically querying the encrypted database.

4.3.3 Private Airline Passenger List Checking

Air travel is tightly controlled by the government in an attempt to prevent terrorists from turning planes into weapons and to prevent criminals from evading law enforcement officials. One predominant means of control is by posting a "Do Not Fly" list (DNF list), which lists those individuals who are not allowed to fly on airlines in the country. Ideally, the airlines check the DNF list against the passenger list for each flight and notify the government if someone on it is attempting to take the flight. The government would like to ensure that the DNF list is checked for each flight and might want to check the passenger list themselves, since there have been cases where an airline did not check the DNF list or had an older version of the list and allowed someone to fly who should not have been allowed to do so. The airlines are understandably hesitant to turn over the passenger list for each flight to the government due to privacy concerns.

The following scheme using homomorphic encryption will allow the government to verify that the passenger list does not contain anyone on the DNF list without requiring the airline to provide the names of everyone on the flight.

1. The airline encrypts the passenger list under a functional encryption scheme.

2. For each person on the DNF list, the airline provides a function key to the government which allows them to search the passenger list and determine if that person is listed.

3. For each flight, the government takes the encrypted passenger list and uses the function keys to verify that no person from the DNF list is on the passenger list. If the list is clear of DNF list persons, then they give approval to the airline to proceed with the flight.

Chapter 5

Approximate GCD Algorithms

Recall from Chapter 2, that we defined the A-GCD problem as follows:

Problem 12 (Approximate GCD Problem (A-GCD))
Given a set of m integers of the form $x_i = q_i p + r_i$ where $q_i, p, r_i \in \mathbb{Z}$ and q_i and r_i are chosen randomly from some distribution (possibly different distributions), find p.

Also, we noted that oftentimes the problem is revised to include a clean multiple of p, denoted x_0.

Problem 13 (Partial Approximate GCD Problem (pA-GCD))
Given a set of m integers of the form $x_i = q_i p + r_i$ where $q_i, p, r_i \in \mathbb{Z}$ and q_i and r_i are chosen randomly from some distribution (possibly different distributions) and a clean multiple $x_0 = q_0 p$, find p.

Homomorphic cryptosystems that follow Gentry's blueprint often use the A-GCD problem as a basis for the security. Gentry proposes the following sizes for a security parameter of λ in the version presented in Chapter 3.

$$|p|_{\text{bin}} = \lambda^2 \qquad |q_i|_{\text{bin}} = \lambda^3 \qquad |r_i|_{\text{bin}} = \lambda$$

5.1 Direct Methods

5.1.1 Brute Force Error Search

The A-GCD problem can be solved by trying to find the error values with brute force. One clever algorithm to do so is provided by Chen and Nguyen in [13] and is summarized as follows.

1. For a fixed i, let $X = x_i(x_i + 1)(x_i - 1)\dots(x_i + r)(x_i - r)$ where $|r|_{\text{bin}}$ is the maximum size of the error.

2. Note that $X \equiv q_i p k \mod x_0$ for some k since one of the factors has r_i subtracted off.

3. Thus, $\gcd(x_0, X) = p$.

In this approach they use some clever tricks to parallelize the computation of the enormous product X and are able to reduce the runtime for the $\lambda = 72$ security level challenge problem posted online by Gentry from 569193 years to 2153 years on a 2.27 GHz machine with 72GB of memory.

5.1.2 Howgrave-Graham Method

In [29], Howgrave-Graham proposes a method to solve the A-GCD (or pA-GCD) problem when given only two numbers x_1 and x_2. The paper includes two algorithms to do so. One is based on computing the continued fractions approximation of $\frac{x_1}{x_2}$ and the other is a lattice based approach reminiscent of Coppersmith's Method [15] for factoring integers when the high bits of one factor are known. Both methods are successful when $(|q_i|_{\text{bin}} + |p|_{\text{bin}})|r_i|_{\text{bin}} < |p|^2_{\text{bin}}$.

5.1.3 Extended Euclidean Algorithm Approach

The Euclidean Algorithm is an efficient way to find the GCD of two integers x_1 and x_2. The Extended Euclidean Algorithm (EEA) keeps track of some extra information to produce coefficients $a_{i,1}$ and $a_{i,2}$ for the i-th step in the algorithm such that

$$a_{i,1}x_1 + a_{i,2}x_2 = k_i \gcd(x_1, x_2)$$

for some $k_i \in Z$. The positive term sequence $\{k_1, k_2, \ldots\}$ is strictly decreasing so that eventually the algorithm produces coefficients b_1 and b_2 such that

$$b_1x_1 + b_2x_2 = \gcd(x_1, x_2).$$

Given some "noisy" multiples of p: $x_1 = q_1p + r_0$ and $x_2 = q_2p + r_2$ where $|q_i|_{\text{bin}}$ is large, we make the crucial observation that the first several pairs in the sequence of coefficients produced by the EEA when run on the "clean" multiples q_1p and q_2p are exactly the same as the sequence of coefficients produced by the EEA when run on x_1 and x_2. This means that

$$\begin{aligned} a_{i,1}x_1 + a_{i,2}x_2 &= a_{i,1}(q_1p + r_1) + a_{i,2}(q_2p + r_2) \\ &= k_ip + (a_{i,1}r_1 + a_{i,2}r_2) \end{aligned}$$

and we have a noisy multiple of p that is smaller. Eventually, the size of the noise term will grow too large and this will not hold. Using the given clean multiple x_0 in place of one of the x_i's will help as well. The following algorithm is based on this observation.

1. Run the EEA (part way) on x_0 and any of the x_i's to obtain b_0 and b_i such that

the size of $b_0x_0 + b_ix_i$ is about the size of p (we want it to be a few more bits in size). If no clean multiple is available, just pick two random x_i's each time.

2. If the error hasn't grown too much then $b_0x_0 + b_ix_i$ should be a small multiple of p with some error.

3. Repeating this process on many of the x_i's one can obtain a good estimate of the higher bits of p.

This should work if $|q_i|_{\text{bin}} + |r_i|_{\text{bin}} \leq |p|_{\text{bin}}$. In most cases, $|q_i|_{\text{bin}} >> |p|_{\text{bin}}$ so this does not work for the parameter sizes proposed by Gentry.

5.2 Heuristic Methods

Using the EEA we seek to find a linear combination of x_1 and x_2 that is a small noisy multiple of p. However, the binary size of q_i forces the coefficients to be quite large and the error term grows too much. To improve on this idea, we seek to use more than just two numbers in the linear combination.

5.2.1 Binary Matrix Approach

Given the list of noisy multiples we would like to transform it into a list (possibly shorter) of small noisy multiples of p. To do so, we repeatedly add and subtract the noisy multiples in an attempt to produce smaller numbers. Looking at the binary representation of each x_i we can use other multiples with matching binary subsequences to reduce the size of x_i quickly. An algorithm developed from this concept is presented below.

1. Treat each x_i as a vector with components given by its binary representation.

2. Let M be the matrix whose rows are these vectors.

3. Row reduce M over $\mathbb{F}_2$.

4. While reducing M, if a row vector's size (i.e. the size of the binary representation of the corresponding integer) is about the size of p then remove it from the matrix and save the corresponding integer in a list of small near multiples of p.

5. Use the list of small near multiples of p and the EEA to try to recover p.

The idea behind this approach is to keep the "error bits" from growing, similar to the way noisy polynomial GCD algorithms work. We attempt to keep each bit in its own column so the error cannot overflow into the multiple of p. However, it does not appear to be guaranteed to work and when it does work, it may just be dumb luck.

5.2.2 Greedy List Reduction Approach

We do gain some intuition from the binary matrix approach. In order to reduce the size of the x_i's quickly we need to subtract pairs that are as close as possible in some sense. There are several ways to choose to do this, but the basic algorithm is presented below. The appendix includes a full implementation.

1. Let $I_0 = \{x_i\}$.

2. Form the set $U_j \subseteq I_j$, where $x \in U_j$ if $|x|_{\text{bin}} = \max(\{|y|_{\text{bin}} \mid y \in I_j\})$.

3. Order the elements in $U_j = \{s_0, s_1, \ldots, s_m\}$ from smallest (s_0) to largest (s_m).

4. Form the set $\bar{U}_j = \{s_{k+1} - s_k \mid k = 0, 1, \ldots, m-1\}$ (note that every element of $\bar{U}_j$ will have size strictly smaller than any of the elements in U_j).

5. Form the set $I_{j+1} = (I_j \setminus U_j) \cup \bar{U}_j$.

6. Run this same procedure (steps 2 - 5) on I_{j+1} until at least two (or some other specified number) of the elements in the set satisfy the bound for the EEA to work.

7. Run the EEA on random pairs of the elements obtained above which satisfy the bound. With high probability, the majority vote of the results will be the value of p.

The following table summarizes some computational results when this method is used. In the table on the left, $m = |p_i|_{\text{bin}} + |q_i|_{\text{bin}}$. In the table on the right, the value of m is increased and the other parameters are held constant. Notice, that after a certain point having more multiples actually hinders the process. This is possibly due to reducing each number with the next smallest number (see other possibilities below the table).

p size	q_i size	r_i size	m	break	p size	q_i size	r_i size	m	break
75	25	25	100	100%	75	150	25	225	0%
150	50	50	200	100%	75	150	25	250	0%
225	75	75	300	100%	75	150	25	275	0%
300	100	100	400	100%	75	150	25	300	2%
75	75	25	150	70%	75	150	25	325	2%
150	150	50	300	90%	75	150	25	350	12%
225	225	75	450	82%	75	150	25	375	34%
300	300	100	600	72%	75	150	25	400	66%
75	150	25	225	0%	75	150	25	425	94%
150	300	50	450	0%	75	150	25	450	90%
225	450	75	675	74%	75	150	25	475	86%
300	600	100	900	70%	75	150	25	500	74%
75	225	25	300	0%	75	150	25	525	76%
125	250	25	375	72%	75	150	25	550	60%
375	750	75	1125	62%	75	150	25	575	68%
125	500	25	625	0%	75	150	25	600	72%

Table 5.1: Greedy List Reduction Results

There are several options for reducing the numbers in U_j.

- Neighbor pairs (i.e. $s_{k+1} - s_k$)
- Largest (i.e. $s_m - s_k$)
- Smallest (i.e. $s_k - s_0$)

- Least "dirty": keep track of how many times a number has been involved in a subtraction and use the one that has been involved in the least number of operations to reduce the other numbers of that size.

- High bit similarity (this is tricky to analyze but seems to not work very well)

Our results seem to agree with the bound from Howgrave-Graham's method. However, we may do slightly better as the gap between $|p|_{\text{bin}}$ and $|r_i|_{\text{bin}}$ increases or as the number of multiples increases. In the third group in the left table, the Howgrave-Graham bound is exactly equal and we can solve it for bigger parameter sizes. Also, the bound is exactly satisfied for the right table and we are able to solve it as m increases.

5.3 Lattice Based Approaches

5.3.1 Standard LLL Reduction Approach

This idea of using many of the multiples to reduce the size of x_i can be improved on in at least two respects. First, if you are given a clean multiple x_0 subtracting x_j from x_i is not the only way to reduce the size of x_i. In some situations, you could add x_j to x_i and then reduce the result modulo x_0 and get a smaller multiple. Second, in essence we are looking for a linear combination of the x_i's with small integer coefficients (the coefficient on x_0 can be quite large if needed since it will not be multiplying any error). Recall from Chapter 2, that the LLL algorithm finds a short orthogonal basis for a lattice. The following algorithm utilizes these two improvements. It is called the h-star algorithm and a full implementation is included in the appendix.

1. Form the $(m+1) \times (m+1)$ basis matrix, L, for a lattice as follows

$$L = \begin{bmatrix} 0 & 0 & \dots & 0 & x_0 \\ 1 & 0 & \dots & 0 & x_1 \\ 0 & 1 & \dots & 0 & x_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & x_m \end{bmatrix}$$

   Intuitively, to obtain a short vector in this lattice you add and subtract the last m rows as needed to decrease the component in the last column of each row. The coefficients are tracked by the $m \times m$ identity matrix and the top row allows for reducing modulo x_0 with no increase in the coefficients.

2. Run the LLL algorithm on L to produce a reduced basis, B.

$$B = \begin{bmatrix} b_{0,0} & b_{0,1} & \dots & b_{0,m} \\ b_{1,0} & b_{1,1} & \dots & b_{1,m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m-1,0} & b_{m-1,1} & \dots & b_{m-1,m} \\ b_{m,0} & b_{m,1} & \dots & b_{m,m} \end{bmatrix}$$

3. The last row b_m in B often contains elements with large entries so we remove it to form the $m \times (m+1)$ matrix $\bar{B}$.

$$\bar{B} = \begin{bmatrix} b_{0,0} & b_{0,1} & \dots & b_{0,m} \\ b_{1,0} & b_{1,1} & \dots & b_{1,m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m-1,0} & b_{m-1,1} & \dots & b_{m-1,m} \end{bmatrix}$$

4. When we put $\bar{B}$ into echelon form (Hermit normal form) over the integers we obtain,

$$H = \begin{bmatrix} 1 & 0 & \dots & 0 & h_{0,m-1} & h_{0,m} \\ 0 & 1 & \dots & 0 & h_{1,m-1} & h_{1,m} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \\ 0 & 0 & \dots & 1 & h_{m-2,m-1} & h_{m-2,m} \\ 0 & 0 & \dots & 0 & h_{m-1,m-1} & h_{m-1,m} \end{bmatrix}$$

5. As it turns out, $h_{m-1,m-1}$ often has a common factor with the clean multiple $x_0 = q_0 p$. In fact, generally $h_{m-1,m-1} = q_0$ or $q_0 = k h_{m-1,m-1}$ for a small k.

6. To make the notation easier we will denote $h_{m-1,m-1}$ by h^*.

7. So we compute $\gcd(x_0, h^*)$ to find q_0 and then recover p from x_0.

The following examples illustrate the results one can obtain.

Example 1 With $|q_i|_{\text{bin}} = 30$, $|p|_{\text{bin}} = 20$, and $|r_i| < 2^5$ (i.e. r_i is at most 5 bits) we have:

$$L = \begin{bmatrix} 0 & 0 & 0 & 0 & 869449743762409 \\ 1 & 0 & 0 & 0 & 675478860334176 \\ 0 & 1 & 0 & 0 & 768514237144518 \\ 0 & 0 & 1 & 0 & 836955889912598 \\ 0 & 0 & 0 & 1 & 715256029681126 \end{bmatrix}$$

$$B = \begin{bmatrix} -78 & -313 & 107 & 38 & -61 \\ 2 & -50 & 148 & -63 & 386 \\ -115 & 54 & 266 & -364 & -299 \\ -455 & -69 & -386 & -130 & 207 \\ -10259 & 6498 & 7536 & 10182 & -396 \end{bmatrix}$$

$$\bar{B} = \begin{bmatrix} -78 & -313 & 107 & 38 & -61 \\ 2 & -50 & 148 & -63 & 386 \\ -115 & 54 & 266 & -364 & -299 \\ -455 & -69 & -386 & -130 & 207 \end{bmatrix}$$

$$H = \begin{bmatrix} 1 & 0 & 0 & 328736030 & 9204608811 \\ 0 & 1 & 0 & 734520646 & 20566578106 \\ 0 & 0 & 1 & 305312788 & 8548758085 \\ 0 & 0 & 0 & 1013081383 & 28366278724 \end{bmatrix}$$

So $h^* = 1013081383$ and in this case, the clean multiple x_0 factors as follows so we can easily find p.

$$x_0 = q_0 p = 1013081383 \times 858223$$

Example 2 With $|q_i|_{\text{bin}} = 30$, $|p|_{\text{bin}} = 20$, and $|r_i| < 2^8$ (i.e. r_i is at most 8 bits) we have:

$$L = \begin{bmatrix} 0 & 0 & 0 & 0 & 507427346501992 \\ 1 & 0 & 0 & 0 & 141803231186811 \\ 0 & 1 & 0 & 0 & 415217704063463 \\ 0 & 0 & 1 & 0 & 408155643521144 \\ 0 & 0 & 0 & 1 & 502527523593188 \end{bmatrix}$$

$$B = \begin{bmatrix} 54 & 147 & -299 & 194 & -33 \\ 720 & 216 & 366 & -67 & -116 \\ 116 & 534 & 239 & 375 & 498 \\ 650 & -335 & -385 & -327 & 321 \\ 697 & -1478 & 537 & 1585 & 157 \end{bmatrix}$$

$$\bar{B} = \begin{bmatrix} 54 & 147 & -299 & 194 & -33 \\ 720 & 216 & 366 & -67 & -116 \\ 116 & 534 & 239 & 375 & 498 \\ 650 & -335 & -385 & -327 & 321 \end{bmatrix}$$

$$H = \begin{bmatrix} 2 & 1 & 4 & 12850554 & 2724317581 \\ 0 & 4 & 3 & 522533 & 110776432 \\ 0 & 0 & 7 & 2052942 & 435224096 \\ 0 & 0 & 0 & 13555117 & 2873684804 \end{bmatrix}$$

So $h^* = 13555117$ and in this case, the clean multiple x_0 factors as follows so we can easily find p.

$$x_0 = q_0 p = 2^3 \times 7 \times 13555117 \times 668471$$

The following table summarizes some computational results for this method. To determine m for each set of parameters we ran the algorithm on 10 random instances. The first value of m such that decreasing it resulted in some instances not being solved is reported below.

	p size	q_i size	r_i size	m
Group 1	50	950	20	36
	50	950	21	37
	50	950	22	39
	50	950	23	40
	50	950	24	42
	50	950	25	44
	50	950	26	46
	50	950	27	49
	50	950	28	52
Group 2	100	950	20	13
	100	950	21	13
	100	950	22	13
	100	950	23	13
	100	950	24	14
	100	950	25	14
	100	950	26	14
	100	950	27	14
	100	950	28	14
Group 3	100	950	70	38
	100	950	71	39
	100	950	72	41
	100	950	73	42
	100	950	74	44
	100	950	75	47
	100	950	76	48
	100	950	77	51
	100	950	78	55
Group 4	50	450	20	17
	50	450	21	18
	50	450	22	19
	50	450	23	19
	50	450	24	20
	50	450	25	21
	50	450	26	22
	50	450	27	23
	50	450	28	24

Table 5.2: Standard LLL Reduction Results

Note the following observations about each group.

1. Group 1 and Group 3 are similar since they have the same number of bits in q_i, and the gap between the r size and the p size is the same.

2. For Group 2, the number of multiples needed is very sharp (i.e. $m = 13$ works every time $m = 12$ fails every time).

3. Group 4 requires about half as many multiples as Group 1 since the q size is about half the size of q in Group 1.

Unfortunately, this will still not work for the types of parameters that Gentry suggests. We attribute this to the fact that the LLL algorithm does not find a small enough vector unless we have many near multiples of p.

5.3.2 Linear Program Reduction Approach

The LLL algorithm produces an orthogonal basis for the lattice composed of "short vectors". In our case (and many other applications) the goal is to find just a single short vector (in the 1-norm sense) without regard of the size of the other vectors or whether they are orthogonal or not. To reduce a vector $\mathbf{b}$ by a set of vectors $V = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_t\} \subseteq \mathbb{Z}^n$ we need to solve the following minimization problem.

$$\begin{aligned} \min \quad & \|\mathbf{b} - (x_1\mathbf{v}_1 + x_1\mathbf{v}_1 + \ldots + x_t\mathbf{v}_t)\|_1 \\ \text{s.t.} \quad & x_i \in \mathbb{Z} \end{aligned}$$

This is equivalent to solving the following integer linear program.

$$
\begin{aligned}
IP: \quad \min \quad & \sum_{i=1}^{n} y_i & & \\
\text{s.t.} \quad & y_i \geq \mathbf{b}_i - \sum_{j=1}^{t} x_j \mathbf{v}_{ji} & \forall \;\; i = 1 \ldots n \\
& y_i \geq -\left(\mathbf{b}_i - \sum_{j=1}^{t} x_j \mathbf{v}_{ji} \right) & \forall \;\; i = 1 \ldots n \\
& x_i \in \mathbb{Z}
\end{aligned}
$$

Dropping the requirement that the coefficients be integers, we obtain the following linear program which we will refer to as LP.

$$
\begin{aligned}
LP: \quad \min \quad & \sum_{i=1}^{n} y_i & & \\
\text{s.t.} \quad & y_i \geq \mathbf{b}_i - \sum_{j=1}^{t} x_j \mathbf{v}_{ji} & \forall \;\; i = 1 \ldots n \\
& y_i \geq -\left(\mathbf{b}_i - \sum_{j=1}^{t} x_j \mathbf{v}_{ji} \right) & \forall \;\; i = 1 \ldots n
\end{aligned}
$$

Solving this relaxation of the original minimization problem is at the heart of the following subroutine called REDUCE, which takes a vector $\mathbf{b}$ and a set of vectors V and reduces the 1-norm of $\mathbf{b}$ as much as possible by the vectors in V.

function REDUCE($\mathbf{b}$, V)

1. solve LP
2. round each component of the solution from (1) to the nearest integer
3. using the rounded solution, $r = \mathbf{b} - \sum_{j=1}^{t} x_j \mathbf{v}_j$

if $\|r\|_1 < \|b\|_1$ **then**

return r

else

return b

end if

end function

This REDUCE subroutine is then used as part of the main reduction algorithm called POST_PROCESS. We sketch an outline of how it works below, and a full implementation is provided in the appendix.

Parameters

1. N is the number of "neighbor" vectors (vectors immediately adjacent in the list) to use initially in the REDUCE method
2. $TOL_{NEIGHBORS}$ is a tolerance for when we move from round 1 to round 2
3. $TOL_{OVERALL}$ is a tolerance for when we increase N

Procedure

▷ Start with a list of vectors.

▷ **Round 1**

1. For each vector in the list, use the next N vectors in the REDUCE subroutine to reduce the 1-norm of the vector.

2. After all vectors have been reduced, randomly shuffle the list.
3. Repeat round 1 until the difference in the average norm of the vectors from one iteration to the next is changing less than $TOL_{NEIGHBORS}$.

▷ Sort the list of vectors by increasing 1-norm.

▷ **Round 2**

1. For each vector in the list, use the next N vectors in the REDUCE subroutine to reduce the 1-norm of the vector.
2. After each reduction, bubble up the new vector to keep the list sorted and start reducing the list from the position of the new vector.

▷ If the average norm of the vectors in the list changed in rounds 1 and 2 more than $TOL_{OVERALL}$, then increase N by 1 and repeat rounds 1 and 2.

The following is an example of what we can achieve with this POST_PROCESS algorithm. The problem instance consisted of 100 vectors with $|p|_{\text{bin}} = 20$, $|q_i|_{\text{bin}} = 20$, $|r_i|_{\text{bin}} = 10$, and the algorithm was initialized with $N = 2$ (to start with).

	Smallest Norm	**Average Norm**
Original	525719146354	792105821828.48510742
LLL	8	25.58415842
POST_PROCESS	7	22.05940594

Running POST_PROCESS used 46 neighbors in the last iteration. It was terminated after 1 hour. The smaller vector with a 1-norm of 7 was found fairly quickly. The following table summarizes some computational results obtained from running this method.

num vectors	entry size	num neighbors	Original short	Original average	LLL short	LLL average	POST_ PROCESS short	POST_ PROCESS average	average decrease
20	20	5	352736	487818	4	7.52380952	4	6.85714286	8.9%
20	20	5	478828	743932.8095	4	7.80952381	4	6.95238095	11.0%
40	20	5	305847	442379.9756	2	6.51219512	2	5.43902439	16.5%
80	20	5	281885	420903.4321	2	5.65432099	2	4.49382716	20.5%
20	80	5	6.07E+23	8.73E+23	22	65733088.05	22	50829934.62	22.7%
40	80	10	4.24E+23	6.08E+23	17	23059502.76	17	19047560.02	17.4%*
40	80	10	5.56E+23	8.09E+23	18	30029364.49	16	26929374.93	10.3%*
40	80	20	3.47E+23	5.09E+23	17	19253819.88	17	15129039.98	21.4%
40	80	20	3.87E+23	5.86E+23	18	21174925	18	16483408.24	22.2%

Table 5.3: POST_PROCESS Results

From Table 5.3 on the previous page, note the following.

1. Rows marked with an * were terminated after an hour ($TOL_{OVERALL}$ should have been higher).

2. In row 7 we actually found a shorter vector.

Since the solving of LP really slows things down as N increases, we view the algorithm as a post processing step after running the LLL algorithm. From our experiments, the LLL algorithm does a really good job of finding a short vector (in both the 2-norm and 1-norm sense). So, while we can improve the average size of the vectors in the lattice basis, we are primarily making only the larger vectors smaller.

Appendix

The algorithms discussed in Chapter 5 were implemented in SAGE 6.2 and run on an 8 core 64-bit Intel Xeon processor at 2.4 GHz with 12 GB of RAM. The code for each one is provided below.

Extended Euclidean Algorithm Approach

```
def bin_size(num):
    """ returns the length of the binary representation of num"""

    num = abs(num)
    return len(bin(abs(num))[2:])
```

```
def EEA(g_0,g_1,size):
    """ returns a and b such that
        a*g_0 + b*g_1 = r
        where bin_size(r) <= size
    """

    reversed = false

    # ensure that |r_0| > |r_1|
    if abs(g_1) < abs(g_0):
        r_0 = g_0
        r_1 = g_1
    else:
        r_0 = g_1
        r_1 = g_0
        reversed = true
```

```
    r_i = r_1 # we just do this to "prime" the loop

    m = matrix(QQ,[[1,0],[0,1]])

    # note: r_i = m[1,0]*g_0 + m[1,1]*g_1
    while bin_size(r_i) > size:
        q_i = r_0 // r_1
        r_i = r_0 - q_i * r_1

        r_temp = r_0 - (q_i + 1) * r_1

        if abs(r_temp) < r_i:
            q_i += 1
            r_i = r_temp

        r_0 = r_1
        r_1 = r_i
        m = matrix(QQ,[[0,1],[1,-q_i]])*m

    # handle reversed inputs
    if reversed:
        return [m[1,1],m[1,0]]
    else:
        return [m[1,0],m[1,1]]
```

Greedy List Reduction Approach

```
def add_num(key_dict, num, err):
    """ adds (num, err) to the key_dict,
        which is keyed on the bin_size of nums
    """

    num_size = bin_size(num)

    if num_size not in key_dict.keys():
        key_dict[num_size] = []

    key_dict[num_size].append((num, err))
```

```
def get_sizes(key_dict):
    """ returns the sorted list of keys in key_dict"""

    sizes = key_dict.keys()
    return sorted(sizes, reverse=true)
```

```
def get_reduced_keys(key_list, q_size, r_size, p_size):
    """ neighbor method: subtract each number
        from the next biggest number (pairwise)
    """

    from operator import itemgetter
    from heapq import heapify, heappop, heappush

    key_dict = {}

    for num in key_list:
        add_num(key_dict, num, r_size)

    red_keys = []

    sizes = get_sizes(key_dict)
    cur_size = sizes[0]

    while cur_size > p_size:
        # form the set of the biggest keys
        cur_nums = key_dict.pop(cur_size)

        # check for |q_i| + |r_i| < |p| and |r_i| <= |q_i|
        for num_data in cur_nums:
            upper_bound_holds = True
            lower_bound_holds = True

            cur_q_size = cur_size - p_size
            cur_r_size = num_data[1]

            if cur_q_size + cur_r_size >= p_size:
                upper_bound_holds = False
                next

            if cur_r_size > cur_q_size:
                lower_bound_holds = False
                next

            if upper_bound_holds and lower_bound_holds:
                red_keys.append(num_data[0])

        if len(red_keys) > 0:
            return red_keys

        # now we actually reduce the numbers in the column
        cur_nums = sorted(cur_nums, reverse=true)
        num_1_data = cur_nums[0]

        for num_2_data in cur_nums[1:]:
            # calculate new number
```

```
            new_num = num_1_data[0] - num_2_data[0]

            # calculate new error estimate
            if num_1_data[1] == num_2_data[1]:
                new_err = num_1_data[1] + 1
            else:
                # possibly this should be + 1
                new_err = max(num_1_data[1], num_2_data[1])

            add_num(key_dict, new_num, new_err)

            num_1_data = num_2_data

        # get new biggest size
        sizes = get_sizes(key_dict)
        if len(sizes) > 0:
            cur_size = sizes[0]
        else:
            cur_size = 0

    # if we get to here we are returning an empty list
    return red_keys
```

```
def get_p_guesses(red_keys, p_size, num_of_guesses):
    """ run the EEA on random pairs of keys from
        red_keys to obtain guesses at multiples of p
        num_of_guesses is how many pairs to choose
    """

    p_guesses = {}
    p_bad_guesses = {}

    if len(red_keys) == 0:
        print("No reduced keys found")
    elif len(red_keys) == 1:
        print("Only one reduced key found:")
        p_guesses[red_keys[0]] = red_keys[0]
    else:
        for k in range(0, num_of_guesses):
            (r_0, r_1) = get_rand_keys(red_keys)

            output = EEA(r_0, r_1, p_size - 1)

            denom_1 = ZZ(abs(output[1]))
            if denom_1 == 0:
                # just assign unique negative values
                # so the estimates are not equal
                estimate_1 = -1
            else:
```

```
                estimate_1 = r_0//denom_1
                rem_1 = r_0 % denom_1
                if rem_1 > denom_1 / 2:
                    estimate_1 += 1

            denom_2 = ZZ(abs(output[0]))
            if denom_2 == 0:
                # just assign unique negative values
                # so the estimates are not equal
                estimate_2 = -2
            else:
                estimate_2 = r_1//denom_2
                rem_2 = r_1 % denom_2
                if rem_2 > denom_2 / 2:
                    estimate_2 += 1

            # only count the estimate if they are equal
            if estimate_1 == estimate_2:
                if p_guesses.has_key(estimate_2):
                    p_guesses[estimate_2] += 1
                else:
                    p_guesses[estimate_2] = 1
            else: # keep track of bad guesses just for fun
                if p_bad_guesses.has_key(estimate_1):
                    p_bad_guesses[estimate_1] += 1
                else:
                    p_bad_guesses[estimate_1] = 1

                if p_bad_guesses.has_key(estimate_2):
                    p_bad_guesses[estimate_2] += 1
                else:
                    p_bad_guesses[estimate_2] = 1

    return (p_guesses, p_bad_guesses)
```

Standard LLL Reduction Approach

```
def h_star_algorithm(multiples, clean_multiple):
    """ returns a multiple of p or 0 if none was found"""

    # the lattice
    L = []

    m = len(multiples)

    # add the clean multiple into the lattice
```

```
    L.append([0]*m + [clean_multiple])

    # add the noisy multiples into the lattice
    for i in range(0, m):
        L.append([0]*i + [1] + [0]*(m-i-1) + [multiples[i] %
            ↪ clean_multiple])

    L_matrix = Matrix(ZZ, L)

    reduced_basis = L_matrix.LLL()

    B = reduced_basis[0:m]
    H = B.echelon_form()

    # check the second to last position in the last row
    # for a possible factor of the clean multiple
    h_star = H[m-1][m-1]

    if h_star != 0 and clean_multiple % h_star == 0:
        p_mult = clean_multiple / h_star
        return p_mult
    else:
        return 0
```

Linear Program Reduction Approach

```
def reduce(b, V):
    """ find r = b - sum a_i v_i
        such that norm_1(r) <= norm_1(b)
    """

    prog = MixedIntegerLinearProgram(maximization = False)
    # the y's will be nonnegative, but that is forced by our constraints
    y = prog.new_variable(name = "y", nonnegative = False)
    x = prog.new_variable(name = "x", nonnegative = False)

    for i in range(0, len(V)):
        prog.set_min(x[i], None)

    prog.set_objective(sum(y[i] for i in range(0, len(b))))

    for i in range(0, len(b)):
        sum_term = sum(x[j] * V[j][i] for j in range(0, len(V)))

        prog.add_constraint(y[i] >= b[i] - sum_term)
        prog.add_constraint(y[i] >= -(b[i] - sum_term))
```

```
    prog.solve()

    red_vector = (vector(b) - sum(int(round(prog.get_values(x[j]))) *
        ↪ vector(V[j]) for j in range(0, len(V))))

    if red_vector.norm(1) <= b.norm(1):
        return red_vector
    else:
        return b
```

```
def sort_by_norm(L):
    """ sorts the vectors in L
        in increasing order by 1-norm
    """

    import heapq

    temp = []

    for row in L:
        temp.append((row.norm(1), row))

    heapq.heapify(temp)
    output = []

    while len(temp) > 0:
        pieces = heapq.heappop(temp)
        output.append(pieces[1])

    return output
```

```
def bubble_up(L, index):
    """ moves the vector at position
        index up in L so that L is
        sorted by increasing 1-norm
    """

    cur_row = L[index]
    cur_norm = cur_row.norm(1)

    cur_index = index - 1

    while cur_index >= 0:
        if cur_norm >= L[cur_index].norm(1):
            # remove the row and insert it in the correct place
            L.pop(index)
            L.insert(cur_index + 1, cur_row)
```

```
            return (L, cur_index + 1)
        else:
            cur_index -= 1

    # this row has a smaller norm than all the previous rows
    L.pop(index)
    L.insert(0, cur_row)

    return (L, 0)
```

```
def post_process_LLL(B, num_neighbors, tol_neighbors, tol_overall):
    """ reduces the 1-norm of all vectors in B"""

    import random

    basis = copy(B)

    old_avg_norm = 0
    # just to prime the loop
    cur_avg_norm = tol_overall + tol_neighbors

    while abs(cur_avg_norm - old_avg_norm) > tol_overall:

        # Round 1
        while abs(cur_avg_norm - old_avg_norm) > tol_neighbors:
            for i in range(0, len(basis)):
                cur_vector = basis[i]
                basis[i] = reduce(cur_vector, basis[max(i -
                    ↪ num_neighbors, 0):i])

            old_avg_norm = cur_avg_norm
            cur_avg_norm = float(sum([vec.norm(1) for vec in basis])) /
                ↪ len(basis)

            random.shuffle(basis)


        # Round 2
        red_basis = sort_by_norm(basis)

        cur_index = 1

        while cur_index < len(red_basis):
            cur_vector = red_basis[cur_index]
            index_bound = min(cur_index, num_neighbors)

            red_basis[cur_index] = reduce(cur_vector, red_basis[0:
                ↪ index_bound])
```

```
            (red_basis, cur_index) = bubble_up(red_basis, cur_index)

            cur_index += 1

        basis = copy(red_basis)

        old_avg_norm = cur_avg_norm
        cur_avg_norm = float(sum([vec.norm(1) for vec in red_basis])) /
            ↪ len(red_basis)

    return basis
```

Bibliography

[1] Mikls Ajtai and Cynthia Dwork. A public-key cryptosystem with worst-case/average-case equivalence. *Electronic Colloquium on Computational Complexity (ECCC)*, 3(65), 1996.

[2] Mikls Ajtai, Ravi Kumar, and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. In Jeffrey Scott Vitter, Paul G. Spirakis, and Mihalis Yannakakis, editors, *STOC*, pages 601–610. ACM, 2001.

[3] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM Conference on Computer and Communications Security*, pages 784–796. ACM, 2012.

[4] Johannes Blömer and Jean-Pierre Seifert. On the complexity of computing short linearly independent vectors and short bases in a lattice. In Jeffrey Scott Vitter, Lawrence L. Larmore, and Frank Thomson Leighton, editors, *STOC*, pages 711–720. ACM, 1999.

[5] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-dnf formulas on ciphertexts. In Joe Kilian, editor, *TCC*, volume 3378 of *Lecture Notes in Computer Science*, pages 325–341. Springer, 2005.

[6] Joan Boyar, Ren Peralta, and Denis Pochuev. On the multiplicative complexity of boolean functions over the basis (cap, +, 1). *Theor. Comput. Sci.*, 235(1):43–57, 2000.

[7] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. *IACR Cryptology ePrint Archive*, 2012:78, 2012. informal publication.

[8] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. *IACR Cryptology ePrint Archive*, 2011:277, 2011.

[9] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS*, pages 309–325. ACM, 2012.

[10] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. In Rafail Ostrovsky, editor, *FOCS*, pages 97–106. IEEE, 2011.

[11] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In *Proceedings of the 31st Annual Conference on Advances in Cryptology*, CRYPTO'11, pages 505–524, Berlin, Heidelberg, 2011. Springer-Verlag.

[12] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, February 1981.

[13] Yuanmi Chen and Phong Q. Nguyen. Faster algorithms for approximate common divisors: Breaking fully-homomorphic-encryption challenges over the integers. *IACR Cryptology ePrint Archive*, 2011:436, 2011.

[14] Josh D. Cohen and Michael J. Fischer. A robust and verifiable cryptographically secure election scheme (extended abstract). In *FOCS*, pages 372–382. IEEE Computer Society, 1985.

[15] Don Coppersmith. Finding a small root of a bivariate integer equation; factoring with high bits known. In Ueli M. Maurer, editor, *EUROCRYPT*, volume 1070 of *Lecture Notes in Computer Science*, pages 178–189. Springer, 1996.

[16] Jean-Sbastien Coron, Avradip Mandal, David Naccache, and Mehdi Tibouchi. Fully homomorphic encryption over the integers with shorter public keys. *IACR Cryptology ePrint Archive*, 2011:441, 2011.

[17] 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, *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 446–464. Springer, 2012.

[18] I. Dinur, G. Kindler, and S. Safra. Approximating-cvp to within almost-polynomial factors is np-hard. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, FOCS '98, pages 99–, Washington, DC, USA, 1998. IEEE Computer Society.

[19] Taher ElGamal. A Public Key Cryptosystem and a Signature Scheme based on Discrete Logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[20] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, Stanford, CA, USA, 2009. AAI3382729.

[21] Craig Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53(3):97–105, 2010.

[22] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. *IACR Cryptology ePrint Archive*, 2010:520, 2010.

[23] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. *IACR Cryptology ePrint Archive*, 2013:340, 2013.

[24] Oded Goldreich, Daniele Micciancio, Shmuel Safra, and Jean-Pierre Seifert. Approximating shortest lattice vectors is not harder than approximating closest lattice vectors. *Inf. Process. Lett.*, 71(2):55–61, 1999.

[25] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Succinct functional encryption and applications: Reusable garbled circuits and beyond. *IACR Cryptology ePrint Archive*, 2012:733, 2012.

[26] Shafi Goldwasser and Silvio Micali. Probabilistic Encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[27] Guillaume Hanrot and Damien Stehl. Improved analysis of kannan's shortest lattice vector algorithm. *CoRR*, abs/0705.0965, 2007.

[28] Bettina Helfrich. Algorithms to construct minkowski reduced and hermite reduced lattice bases. *Theor. Comput. Sci.*, 41(2-3):125–139, December 1985.

[29] Nick Howgrave-Graham. Approximate integer common divisors. In Joseph H. Silverman, editor, *CaLC*, volume 2146 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2001.

[30] Arjen K. Lenstra. Factorization of polynomials. *SIGSAM Bull.*, 18(2):16–18, May 1984.

[31] Yi-Kai Liu, Vadim Lyubashevsky, and Daniele Micciancio. On bounded distance decoding for general lattices. In Josep Daz, Klaus Jansen, Jos D. P. Rolim, and Uri Zwick, editors, *APPROX-RANDOM*, volume 4110 of *Lecture Notes in Computer Science*, pages 450–461. Springer, 2006.

[32] Vadim Lyubashevsky and Daniele Micciancio. On bounded distance decoding, unique shortest vectors, and the minimum distance problem. In Shai Halevi, editor, *CRYPTO*, volume 5677 of *Lecture Notes in Computer Science*, pages 577–594. Springer, 2009.

[33] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2010.

[34] Daniele Micciancio. The shortest vector problem is NP-hard to approximate to within some constant. *SIAM Journal on Computing*, 30(6):2008–2035, March 2001. Preliminary version in FOCS 1998.

[35] Daniele Micciancio and Shafi Goldwasser. *Complexity of Lattice Problems: a cryptographic perspective*, volume 671 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, 2002.

[36] Daniele Micciancio and Panagiotis Voulgaris. A deterministic single exponential time algorithm for most lattice problems based on voronoi cell computations. *SIAM J. Comput.*, 42(3):1364–1391, 2013.

[37] H. Minkowski. *Geometrie der Zahlen*. Teubner, Leipzig [reprinted: Chelsea, New York, 1953], 1896.

[38] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, pages 113–124, New York, NY, USA, 2011. ACM.

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

[40] Chris Peikert. Public-key cryptosystems from the worst-case shortest vector problem: extended abstract. In Michael Mitzenmacher, editor, *STOC*, pages 333–342. ACM, 2009.

[41] Chris Peikert and Brent Waters. Lossy trapdoor functions and their applications. In Cynthia Dwork, editor, *STOC*, pages 187–196. ACM, 2008.

[42] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *STOC*, pages 84–93. ACM, 2005.

[43] R L Rivest, L Adleman, and M L Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978.

[44] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[45] Amit Sahai and Brent Waters. Fuzzy identity based encryption. *IACR Cryptology ePrint Archive*, 2004:86, 2004.

[46] Tomas Sander, Adam L. Young, and Moti Yung. Non-interactive cryptocomputing for NC1. In *FOCS*, pages 554–567. IEEE Computer Society, 1999.

[47] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. *IACR Cryptology ePrint Archive*, 2009:571, 2009.

[48] Vinod Vaikuntanathan. Computing blindfolded: New developments in fully homomorphic encryption. In Rafail Ostrovsky, editor, *FOCS*, pages 5–16. IEEE, 2011.

[49] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. *IACR Cryptology ePrint Archive*, 2009:616, 2009.

[50] P. van Emde-Boas. *Another NP-complete partition problem and the complexity of computing short vectors in a lattice*. Report. Department of Mathematics. University of Amsterdam. Department, Univ., 1981.

[51] A. C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science*, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.