Optimizing Fully Homomorphic Encryption

By Kevin C. King
B.S., C.S. M.I.T., 2015

September 30, 2016

Submitted to the Department of Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Author: ____________________
Kevin C. King, Department of Electrical Engineering and Computer Science
September 30, 2016

Certified By: ____________________
Prof. Vinod Vaikuntanathan, Thesis Supervisor
September 30, 2016

Accepted By: ____________________
Christopher Terman, Chairman, Masters of Engineering Thesis Committee

Optimizing Fully Homomorphic Encryption

By Kevin C. King
Under the Supervision of Vinod Vaikuntanathan

Submitted to the Department of Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements for
the Degree of Master of Engineering in Electrical Engineering and Computer Science

September 30, 2016

Abstract

Fully homomorphic encryption (FHE) presents the possibility of removing the need to trust cloud providers with plaintext data. We present two new FHE scheme variants of BGV'12, both of which remove the need for key switching after a ciphertext multiplication, overall halving the runtime of bootstrapping. We also present multiple implementations of 32-bit integer addition evaluation, the fastest of which spends 16 seconds computing the addition circuit and 278 seconds bootstrapping. We find that bootstrapping consumes approximately 90% of the computation time for integer addition and secure parameter settings are currently bottlenecked by the memory size of commodity hardware.

Contents

Chapter 1

Introduction

1.1 Background

Fully Homomorphic Encryption (FHE) enables a server to store, and compute on, encrypted data without being able to recover the plaintext. The problem of creating ciphertexts that may be computed on was proposed by Rivest et al. [RAD78] in 1978. The first theoretical construction came about in 2009 in Gentry's PhD. Thesis [Gen09a]. In 2011, Brakerski and Vaikuntanathan constructed FHE with security based on the well-studied hardness assumption of Ring-LWE [BV11]. Soon after, the BGV'12 construction [BGV12] developed new noise management techniques and brought parameters closer to the reach of practice. Making modifications to the BGV'12 scheme, the authors of [HS13] developed HElib, an open-source implementation of FHE. With HElib as the current bleeding edge of BGV-esque FHE, we explore theoretical and practical methods to bring efficiency closer to being feasible for modern computation.

1.2 Motivation

A widespread trend in modern computing is to rent computational resources in datacenters hosted by large cloud providers such as Amazon AWS, Microsoft Azure, and Google Cloud. These cloud providers experience large economies-of-scale, lowering the costs their customers endure compared to purchasing and maintaining personal hardware. However, these customers must trust any data they host on rented servers, such as customer credit card numbers, to these cloud providers. Furthermore, machines hosted in these datacenters alongside other arbitrary customers are prone to side-channel attacks as shown in [IGI+15].

One solution is to use symmetric or asymmetric encryption and only store ciphertexts on servers out of one's custody. However, in these cases any computation must be done on a client or self-hosted machine, reversing much of the utility of using a cloud provider in the first place. Practical FHE would allow one to store an entire database encrypted in a remote datacenter, have the provider run some sort of processing such as MapReduce on the data using commodity hardware, and output the encrypted result to the customer. While FHE has been constructed and implemented, these constructions and implementations are not yet efficient or fully-featured enough to perform non-trivial computations, thus motivating our work to explore optimizations to both the underlying schemes and use of the constructions.

1.3 BGV Cryptosystems

The BGV'12-like FHE schemes described in this work operate over polynomial rings. At a high level, ciphertexts contain two elements, (a, b), each a polynomial in the ring. a is generated uniformly at random. A message $m \in \{0, 1\}$ is encoded in b as $b = a\mathfrak{s} + 2e + m$, where e is a small error and $\mathfrak{s}$ is the secret key. $\mathfrak{s}$ can be used to recover $2e+m$ from b. Finally, m may be recovered from $2e + m$ by computing the remainder modulo 2. Not only do these schemes accomplish asymmetric encryption, but they also enable additively-homomorphic operations (trivially) and multiplicatively-homomorphic operations (non-trivially).

1.4 Results

We present two variants of the BGV'12 [BGV12] FHE scheme that preserve linearity in the secret key after a multiplication of two ciphertexts without key switching. Both variants were developed in discussions about FHE implementations [BV16]. The first variant, called SS (for s-squared), involves sampling secret keys $\mathfrak{s}$ such that $\mathfrak{s} = \mathfrak{s}^2$. While we conjecture it to be secure, this scheme cannot use modulus switching for noise management because of the large coefficient representation magnitude of its secret keys. The second BGV'12 variant we present, called QE (for quadratic equation), involves publishing a random quadratic equation of $\mathfrak{s}$ publicly. After a multiplication, the evaluator uses substitution, made possible by the quadratic equation, to essentially convert the $\mathfrak{s}^2$ term back into sub-quadratic terms.

Next, we present implementations of both SS and QE in HElib. While multiplication using SS is faster than the key-switching of HElib, its incompatibility with modulus switching limits its utility. Multiplication circuits are evaluated two-to-five times faster when using QE, and more complex operations like bootstrapping and large integer addition are around two times faster.

Finally, we present multiple implementations of 32-bit integer arithmetic. We see large integer arithmetic as an important but generic metric for the practicality of FHE. Our fastest implementation uses a Kogge-Stone adder circuit based on the work of [KS73] and spends 16 seconds on the circuit computation and an additional 278 seconds on bootstrapping.

1.5 Preliminaries

We begin with the definition of FHE and the popular BGV'12 construction, as well as the computational problem from which its security is derived. We also redefine some variants on the computational problems that are used by HElib. Next, we introduce HElib, an open-source implementation of a BGV'12 variant. Finally, we introduce the adder circuit layouts used in the integer arithmetic implementations. All benchmarks are performed on a mid-2012 Macbook Pro Retina with 8GB of RAM and 2.3GHz Intel Core i7 processor. We keep parameters such that total memory consumption remains below 2GB to provide a more stable benchmarking environment free from swapping and memory compression.

1.5.1 Notation

$v[i]$ element i of vector v
$a \cdot b$ dot-product, $\sum_i a[i]b[i]$

$\mathbb{Z}$ the group of integers
$\mathbb{Z}_q$ the group of integers modulo q
$\mathbb{Z}^n$ vectors of n elements in $\mathbb{Z}$, $\mathbb{Z} \times \ldots \times \mathbb{Z}$
$\mathbb{Z}_q^*$ the group of non-negative integers less than and coprime to q
$\phi(\cdot)$ the Euler totient function
$\mathbb{Z}[x]$ polynomials with indeterminant x and coefficients in $\mathbb{Z}$
ξ_n a primitive n'th root of unity
$\Phi_m(x)$ the m'th cyclotomic polynomial: $\prod_{i \in \mathbb{Z}_m^*}(x - \xi_m^i)$
$\mathcal{R}_{m,q}$ the ring $\mathbb{Z}[x]_q/\Phi_m(x)$
n the ring-dimension of $\mathcal{R}_{m,q}$ such that $n = \phi(m)$

$\mathrm{QR}(\mathcal{R}_{m,q})$ the quadratic residues of $\mathcal{R}_{m,q}$
U the uniform distribution
χ an error distribution, typically Gaussian

$\mathfrak{s}$ a secret-key ring element in $\mathcal{R}_{m,q}$
$||\mathfrak{s}||_i$ l-i norm of the components or coefficients of $\mathfrak{s}$
$||\mathfrak{s}||$ l-2 norm of the components or coefficients of $\mathfrak{s}$
$[\mathfrak{s}]_q$ reduction of $\mathfrak{s}$ modulo q

PPT probabilistic polynomial time

`0..n` iteration from `0` to `n`, exclusive, $[0, \mathtt{n})$
`0...n` iteration from `0` to `n` inclusive, $[0, \mathtt{n}]$
`+` addition or vector concatenation

1.5.2 Fully Homomorphic Encryption Definition

We present a basic definition of homomorphic encryption similar to that in [BGV12].
A homomorphic encryption scheme $\mathcal{E}$ with security parameter λ and message space $\mathcal{R}_M$ is a tuple of the following functions:

1. $\mathsf{KeyGen}_{\mathcal{E}}$ takes the security parameter λ and outputs a secret key sk and public key pk.
2. $\mathsf{Encrypt}_{\mathcal{E}}$ takes a plaintext message $x \in \mathcal{R}_M$ and public key pk and outputs a ciphertext c.
3. $\mathsf{Decrypt}_{\mathcal{E}}$ takes a ciphertext c and secret key sk and outputs a plaintext message $x \in \mathcal{R}_M$.
4. $\mathsf{Evaluate}_{\mathcal{E}}$ takes ciphertexts $(c_1, \ldots, c_n)$, a public key pk, and an arithmetic circuit $f \in \mathcal{F}$ that operates over plaintexts in $\mathcal{R}_M$, and outputs ciphertexts $(c'_1, \ldots, c'_{n'})$.

Definition 1.5.1 (Correctness)**.** *A homomorphic encryption scheme $\mathcal{E}$ correctly evaluates a circuit family $\mathcal{F}$ if $\forall f \in \mathcal{F}$, $x_1, \ldots, x_n \in \mathcal{R}_M$, $(pk, sk) \leftarrow \mathsf{KeyGen}_{\mathcal{E}}(\lambda)$, $c_i = \mathsf{Encrypt}_{\mathcal{E}}(x_i)$, $c'_i = \mathsf{Evaluate}_{\mathcal{E}}(pk, f, c_1, \ldots, c_l)$, then*

$$\Pr[\mathsf{Decrypt}_{\mathcal{E}}(sk, c'_1), \ldots, \mathsf{Decrypt}_{\mathcal{E}}(sk, c'_l) \neq f(x_1, \ldots, x_l)] = \mathsf{negl}(\lambda)$$

over the randomness of the experiment.

Definition 1.5.2 (Correctness)**.** *A homomorphic encryption scheme $\mathcal{E}$ is semantically secure if (*$\mathsf{KeyGen}_{\mathcal{E}}$*,* $\mathsf{Encrypt}_{\mathcal{E}}$*,* $\mathsf{Decrypt}_{\mathcal{E}}$*) is a semantically secure encryption scheme.*

This is clear because $\mathsf{Evaluate}_{\mathcal{E}}$ only uses public inputs.

1.5.3 Hardness Problems

Concrete instantiations of FHE derive their security from certain computational problems that are conjectured to be hard for polynomial time adversaries. The learning with errors (LWE) problem is one of these widely used problems. We will use problems closer to the ring learning with errors problem (RLWE), a variant of LWE in a ring setting, for better performance, but include LWE as RLWE is directly derived from it.

Definition 1.5.3 (LWE (Decision) [Reg09])**.** *For integers n, $p(n)$, error distribution χ, any $\mathfrak{s} \in \mathbb{Z}_p^n$, the* LWE *problem is to distinguish samples of the form $(a, a \cdot \mathfrak{s} + e)$, and (a, r) in time poly(n), where $a \overset{U}{\leftarrow} \mathbb{Z}_p^n$, $e \overset{\chi}{\leftarrow} \mathbb{Z}_p$, $r \overset{U}{\leftarrow} \mathbb{Z}_p$.*

The ring learning with errors problem looks very similar but uses the same algebraic structures that our FHE constructions will use.

Definition 1.5.4 (RLWE (Decision) [LPR13, BGV12], modified Definition 2.21, 7)**.** *For security parameter λ, let integers $q = q(\lambda)$, $m = m(\lambda)$, distribution $\chi = \chi(\lambda)$ over $\mathcal{R}_{m,q}$. The* RLWE *problem is to distinguish $(a, a\mathfrak{s} + e)$ from (a, b), where $a \overset{U}{\leftarrow} \mathcal{R}_{m,q}$, $\mathfrak{s} \overset{U}{\leftarrow} \mathcal{R}_{m,q}$, $e \overset{\chi}{\leftarrow} \mathcal{R}_{m,q}$, $b \overset{U}{\leftarrow} \mathcal{R}_{m,q}$.*

Notice that this definition of RLWE samples $\mathfrak{s}$ uniformly over $\mathcal{R}_{m,q}$. The FHE constructions we use sample $\mathfrak{s}$ from lower entropy distributions. A low-entropy reduction is only known for LWE (*not* RLWE), thus we present the low-entropy LWE definition.

Definition 1.5.5 (Entropic-LWE [GKPV10], simplified Theorem 4)**.** *Let χ_k be some distribution over $\{0,1\}^n$ with min-entropy k. Then there is a* PPT *reduction from* LWE *using χ_k to uniform* LWE *with dimension $\leq \frac{k - \omega(\log n)}{\log q}$.*

Drawing secrets from lower-entropy (and smaller-magnitude) distributions reduces the noise growth when multiplying FHE ciphertexts, which we will see first in the HElib construction.

1.5.4 Constructions

The HElib library uses a variant of BGV'12 as its FHE scheme. We start with defining the BGV'12 FHE scheme, and then note HElib's variations.

Simplified BGV'12

We first define a simplified BGV'12 from [BGV12] to illustrate the basics of the construction. Next, we will define key switching (which allows decryption of multiplications) and briefly define modulus switching (a noise reduction technique). Note that BGV'12 uses $R_{m,q}$ with $m = 2^{d+1}$ such that $\Phi_m = x^d + 1$ and $n = \phi(2^{d+1}) = 2^d$. Let χ be a noise distribution over $[-B, \dots, B] \subset \mathbb{Z}$. Let the plaintext message space be $\mathcal{R}_{m,2}$.

$\mathsf{KeyGen}_{\mathsf{BGV}}$ Samples secret key $\mathfrak{s} = (1, s \overset{\chi}{\leftarrow} \mathcal{R}_m)$, $a \overset{U}{\leftarrow} \mathcal{R}_{m,q}$, $e \overset{\chi}{\leftarrow} \mathcal{R}_m$ and set public key $pk = (a, b = a\mathfrak{s} + 2e)$. Outputs $\mathfrak{s}, pk$.

$\mathsf{Encrypt}_{\mathsf{BGV}}$ Takes input message $x \in \mathcal{R}_{m,2}$ and $pk = (b, a)$. Samples $r, e_1, e_2 \overset{\chi}{\leftarrow} \mathcal{R}_m$. Outputs $c = (br + 2e_2 + m, -(ar + 2e_1))$.
Note: c can be seen as a new RLWE sample with $a' = ar$ and noise $e' = 2r(e+e_2)+2se_1$.

$\mathsf{Decrypt}_{\mathsf{BGV}}$ Takes input ciphertext c and secret key $\mathfrak{s}$. Outputs $[[c \cdot \mathfrak{s}]_q]_2$.

$\mathsf{EvalAdd}_{\mathsf{BGV}}$ Takes input ciphertexts $a = (a_0, a_1), b = (b_0, b_1)$. Outputs $c = (a_0 + b_0, a_1 + b_1)$

$\mathsf{EvalMult}_{\mathsf{BGV}}$ Takes input ciphertexts a, b. Computes $c = a \otimes b$, an encryption of $x_a x_b$ under key $\mathfrak{s} \otimes \mathfrak{s}$. Performs key switching from [BGV12] on c to produce a ciphertext c' encrypting $x_a x_b$ under key $\mathfrak{s}$. Outputs c'.

We informally show the correctness of BGV'12 given a ciphertext encrypting a bit x under secret key $\mathfrak{s}$ of the form $c = (a\mathfrak{s} + 2e + x, -a)$.

$$\mathsf{Decrypt}_{\mathsf{BGV}}(c, \mathfrak{s}) = [[c \cdot \mathfrak{s}]_q]_2 = [[a\mathfrak{s} + 2e + x - a\mathfrak{s}]_q]_2 = [[2e + x]_q]_2 = x$$

This equality holds as long as the noise (in this case $2e$) is of low magnitude.

Key Switching

While ciphertext addition simply preserves the key the output ciphertext is encrypted under, the tensor product used in multiplication changes the underlying key of the output ciphertext from $\mathfrak{s}$ to $\mathfrak{s} \otimes \mathfrak{s}$. BGV'12 returns the ciphertext back to an encryption under $\mathfrak{s}$ with the key switching technique from [BGV12].

The high level idea of key switching is to encrypt individual bits of $\mathfrak{s} \otimes \mathfrak{s}$ under $\mathfrak{s}$ in a matrix, and then multiply a tensored ciphertext with this matrix to convert it from an encryption under $\mathfrak{s} \otimes \mathfrak{s}$ to an encryption under $\mathfrak{s}$. Since we are encrypting functions of $\mathfrak{s}$ under itself, we require Circular Security of the scheme, which we will not discuss in this work.

The key-switching routine uses two subroutines to work with bit representations of ring element vectors:

- $\mathsf{BitDecomp}(x \in \mathcal{R}_{m,q}^n)$ decomposes the coefficients of x into bits of increasing significance. Let $x = \sum_{j=0}^{\lfloor \lg q \rfloor} 2^j \cdot u_j \in \mathcal{R}_{m,2}$. Outputs $(u_0, \ldots, u_{\lfloor \lg q \rfloor}) \in \mathcal{R}_{m,2}^{n \cdot \lceil \lg q \rceil}$.
- $\mathsf{PowersOf2}(x \in \mathcal{R}_{m,q}^n)$ outputs $(x, 2x, \ldots, 2^{\lfloor \lg q \rfloor} x) \in \mathcal{R}_{m,q}^{n \cdot \lceil \lg q \rceil}$.

The key-switching matrix generation and usage (for $\mathfrak{s} \otimes \mathfrak{s}$ to $\mathfrak{s}$) are as follows:

- $\mathsf{SwitchKeyGen}(\mathfrak{s} = (1, s))$ Generate $4\lceil \lg q \rceil$ encryptions of 0 under $\mathfrak{s}$ arranged as a $4\lceil \lg q \rceil$-by-2 matrix called A. Add $\mathsf{PowersOf2}(\mathfrak{s} \otimes \mathfrak{s})$ to the first column of A to get a matrix called B. Output B.
- $\mathsf{SwitchKey}(c, B)$ Output $\mathsf{BitDecomp}(c)^T \cdot B \in \mathcal{R}_{m,q}^2$.

$\mathsf{SwitchKey}$ essentially performs a bit-wise decryption of c under $\mathfrak{s} \otimes \mathfrak{s}$ within each 0 encryption under $\mathfrak{s}$. These bit-wise decryptions are reconstructed as a result of the dot-product into a ciphertext encrypting the same message under $\mathfrak{s}$. We defer to [BGV12] for a generalized definition of key-switching and a full analysis of its correctness and security.

The drawbacks of key switching come in the form of public key size, computational time, and additional noise. The key switching matrix for a single modulus stores one encryption per bit of $\mathfrak{s}$, requiring the storage of $\approx \lg q$ ring elements (or $\approx \lg B$ if the coefficients of $\mathfrak{s}$ are in $[-B, \ldots, B]$). Any party multiplying ciphertexts must also store this matrix in order to complete the multiplication. The tensor product of ciphertexts requires 4 ring multiplications while the matrix multiplication requires another $\approx \lg q$ ring multiplications. Since the resulting ciphertext is a linear combination of multiple ciphertexts, the noise grows with this sum. This noise increase happens to be small compared to the overall multiplicative increase in noise (which is present in our variants as well).

HElib BGV'12 Variant

The HElib FHE library uses a variant of BGV'12, mostly motivated by better performance in practice. There are two notable differences from [HS13] pertinent to our work.

First, the HElib variant works with settings of m that are not powers of two, as long as there is an m'th root of unity $\xi_m \in \mathbb{Z}_q$. Using m not a power of 2, we find convenient parameter settings for integer addition.

Second, HElib generates secret keys using a modified distribution. Instead of generating each coefficient of $\mathfrak{s}$ from an error distribution, HElib generates secret keys with a specified Hamming weight (number of non-zero coefficients). By default, the library generates keys with Hamming weight 64, meaning that exactly 64 randomly-chosen coefficients of $\mathfrak{s}$ are sampled uniformly at random from $\{-1, 1\}$, and the rest are set to 0. Generating $\mathfrak{s}$ with Hamming weight 64 means $||\mathfrak{s}|| = \sqrt{64}$, and $\mathfrak{s}$ will have very low noise contribution. The security of generating $\mathfrak{s}$ this way has not been proven.

Coefficient vs. Evaluation Representation

Polynomials, including elements of $\mathcal{R}_{m,q}$, may be represented both by their polynomial coefficients, or by their evaluations at n different points. The coefficient representation of an element $r \in \mathcal{R}_{m,q}$ is a vector $(a_0, \ldots, a_{n-1})$ where $a_i \in \mathbb{Z}_q$ such that $r = \sum_i a_i x^i$. While addition occurs component-wise in the coefficient representation, polynomial multiplication requires computation for every pair of terms, making it computationally cumbersome.

The evaluation representation of an element $r \in \mathcal{R}_{m,q}$ is a vector $(y_0, \ldots, y_{n-1})$ where y_i is the evaluation of r at some point. Generally, these points are chosen as powers of an m-th root of unity, ξ_m^i. In this way, a coefficient representation may be converted to an evaluation representation using the Discrete Fast Fourier Transform, and back with the inverse transform. In the evaluation representation, both addition and multiplication are computed component-wise, making it the desired representation when computing on ciphertexts.

Modulus Switching

The authors of [BGV12] developed a noise reduction technique called modulus switching wherein the ring used has a composite modulus with a “chain” of primes. An encryption begins modulo all of the primes. Once the noise of a ciphertext grows to a certain threshold, the modulus is decreased by removing a prime. Changing to a smaller modulus actually *reduces* the overall noise ratio of the ciphertext. When a ciphertext is modulo a single prime, it must be bootstrapped to continue computation on it.

The mechanics behind the modulus switching technique from [BGV12] revolve around scaling a ciphertext to the new modulus and rounding to preserve the correctness of the ciphertext.

This rounding operation incurs some noise but still results in a net decrease in the relative noise magnitude of the ciphertext. In order to modulus switch a ciphertext c modulo q with plaintext modulus 2 to a ciphertext c' modulo q', we compute c' as the closest ring vector to the scaled ciphertext $\frac{p}{q}c$ such that $c \equiv c' \mod 2$. In this way, c' will decrypt to the same message as c using the same secret key. The resulting noise of c' is dependent on the magnitude of the coefficient representation of $\mathfrak{s}$, which must be much smaller than q (preventing modulus switching in the SS scheme).

Bootstrapping

Every homomorphic addition and multiplication increases the underlying noise of the ciphertext. Once this noise exceeds $\frac{q}{p}$, where p is the plaintext space, the plaintext will no longer be recoverable upon decryption. If an upper-bound on the depth of a circuit is known, then parameters for an FHE scheme can be chosen such that the noise in any of the ciphertexts does not exceed this threshold with high probability. On the other hand, circuits with larger depth will require re-encrypted ciphertexts using larger parameters, which may be inefficient or inconvenient for the owner of the plaintext data.

Craig Gentry proposed the idea of *bootstrapping* a ciphertext in [Gen09b], in which the decryption circuit ($\mathsf{Decrypt}_{\mathcal{E}}$) is evaluated homomorphically on a ciphertext, resulting in a ciphertext encrypting the same plaintext under the same secret key. Since part of the decryption circuit removes the noise from the ciphertext, the bootstrapped ciphertext's noise comes only from the evaluation of the decryption circuit. Reducing any ciphertext's noise to a constant allows evaluation of an arbitrary-depth circuit by bootstrapping whenever a ciphertext's noise nears the decryption threshold minus the noise increase during bootstrapping.

Ciphertext Packing

Just as vector registers in CPUs support running a single instruction on multiple data (SIMD), Smart and Vercauteren developed methods in [SV14] for encrypting multiple plaintexts in a single ciphertext and performing homomorphic operations on two ciphertexts component-wise. Ciphertext packing presents a tradeoff between larger ciphertexts and parallelized computation. Even bootstrapping may be performed using packed ciphertexts, which we utilize in our implementation of addition circuits.

1.5.5 HElib Open Source FHE Library

Considerable work has gone into implementing FHE schemes. The HElib library [HS13, GHS12], by Shai Halevi and Victor Shoup, is an open source implementation of the aforementioned BGV'12 variant. The implementation is primitive-complete in the sense that it can perform key generation, encryption, decryption, ciphertext addition, ciphertext multiplication, and bootstrapping. All of the implementations in this work build off of HElib because of its accessibility and completeness.

In evaluating performance of our scheme variants, we compare their performance with that of unmodified HElib so that both implementations utilize the same optimizations (such as $\mathfrak{s}$ having very low entropy).

The PALISADE lattice cryptography library, by the authors of [GPRR16], is another implementation of FHE written in C++ that we did not have the time to explore.

1.5.6 Addition Circuits

Since FHE evaluation occurs over circuits, developments and optimizations in addition circuits apply also to evaluations of integer addition in FHE schemes. We note that if the final sum of ciphertexts does not exceed the plaintext space size, then encrypted integers or vectors of integers may be summed simply using $\mathsf{EvalAdd}_{\mathcal{E}}$. Since the plaintext space is usually quite small, and in this work always set $p = 2$, we explore addition methods that support sums larger than the plaintext space.

Here we define the types and properties of addition circuits used in our implementations of 32-bit integer addition. First, we define two important signals commonly referred to in adder circuits.

Definition 1.5.6. *A* **propagate signal** *for a block of one or more contiguous adder input bits is* 1 *if the block's carry output is equal to its carry input and* 0 *otherwise. In other words, the propagate signal indicates whether the block would propagate a carry input to a carry output.*

Definition 1.5.7. *A* **generate signal** *for a block of one or more contiguous adder input bits is* 1 *if it generates a carry output with value* 1 *and* 0 *otherwise.*

Full Adder

A full adder captures the entire logic for computing a single bit of the sum of two integers. A full adder for bit i, $F(a_i, b_i, c_i)$, takes the i-th bits of two integers a and b, as well as the i-th carry input bit c_i, and outputs the i-th bit of $a + b$, s_i, and the $i + 1$-th carry input bit c_{i+1} such that

$$s_i = a_i \otimes b_i \otimes c_i$$

$$c_{i+1} = (s_i \wedge c_i) \vee (a_i \wedge b_i)$$

$$c_0 = 0$$

Notice that the input to a full adder is dependent on the outputs of every full adder before it (particularly the carry bit). Each of the following addition circuit types mainly vary in how they compute these carry bits.

Ripple Carry

The simplest addition circuit, a ripple carry, functions by chaining together n full adders, where n is the bit-length of the longest summand. In this way, while inputs a_i and b_i are already known, the output for full adder i can only be computed after waiting for the output of full adder $i - 1$. Thus a ripple adder has a circuit depth of n which we will see to be computationally taxing when evaluating it using an FHE scheme.

Carry-Lookahead

A carry-lookahead adder splits the summands into blocks and computes the propagate signal for these blocks. The concept dates back to the 1950s, forms of which were patented by IBM [B60]. The propagate signal for a given block determines how to compute its carry out. If the propagate signal for an entire block is 1, then the carry out is determined by the carry in. Otherwise, the carry out of the block is determined by the generate signal of the block independent of the carry input. Intuitively, if the propagate signal is 1, then every $a_i \otimes b_i$ in the block is 1, which will propagate a carry bit all the way through the block exactly when the carry input is 1. When the propagate signal is 0, then for some i in the block, $a_i \otimes b_i = 0$, and any carry input will simply propagate until it reaches bit i and turns this bit into a 1. A carry output will be generated in this case only if it comes from within the block itself.

By the above logic, the propagate and generate signals for each block may be computed in parallel. Next, the carry output for each block may be determined serially, using the propagate signal of the block to select between the carry output of the previous block or the generate signal of the current block. Finally, with correct carry inputs for each block, the final sum bits may be computed for each block in parallel.

The circuit depth of a carry-lookahead adder is $\Theta(B + \frac{n}{B})$ where B is the size of an input block. Thus we minimize the circuit depth with a block size of $B = \Theta(\sqrt{n})$.

Kogge-Stone

A Kogge-Stone adder utilizes an algorithm called **recursive doubling** to compute the propagate and generate signals for every prefix of the input [KS73]. First we define the type of problem **recursive doubling** applies to.

Definition 1.5.8 (Adapted [KS73] (2))**.** *An* **m-th order recurrence problem** *is the computation of values $x_1, \ldots, x_n$ where $x_i = f_i(x_{i-1}, \ldots, x_{i-m})$ for some function f_i. Problems of this form may be solved with recursive doubling with computation depth* $\lg n$.

In the case of integer addition, every carry output depends on every previous input. Thus integer addition of two n-bit integers is an n-th order recurrence problem.

In the Kogge-Stone adder, there are $\lceil \lg n \rceil$ levels, starting with level 0. In level k, the current propagate and generate signals for slot i are combined at slot $i + 2^k$. After $\lceil \lg n \rceil$ levels, the propagate and generate signals for every slot are correct and the final sum output is computed from them.

1.6 Previous Work

The authors of HElib have incorporated countless optimizations into their FHE implementation. Our scheme variants and integer addition implementations all build on top of these optimizations detailed in [HS13, GHS12, HS15]. The authors of HElib have also incorporated a timing and testing environment useful for measuring the performance impact of changes on the underlying subroutines such as `reLinearize` and `reCrypt`.

In [GHS12], the authors reason about parameter security. Most notably, they argue that n should be approximately $1000 * L$, where L is the number of levels in the ciphertext modulus.

Using these parameter guidelines, the authors implemented and benchmarked homomorphic evaluation of AES encryption and decryption. They present benchmarks both with bootstrapping and without bootstrapping (resulting in a ciphertext that cannot be computed on anymore). With a secure parameter setting, they measured AES decryption to take 394 seconds without bootstrapping and 1630 seconds with bootstrapping.

With bootstrapping frequently consuming the majority of computation time when evaluating a complex circuit, optimizing which intermediate ciphertexts are bootstrapped can have a large impact on the overall runtime. This optimization problem is formalized in [BLMZ17] as the *bootstrap problem*. The authors assume a fixed modulus level L for the scheme and present an L-approximation algorithm for the *bootstrap problem* given a circuit to evaluate. We first take the approach of investigating a desirable parameter selection, including whether and how to pack ciphertexts. These choices alter the circuit structure and thus make it difficult to apply their exact model. We also found that our circuits generally had notable points of convergence where it made the most sense to bootstrap, and where all ciphertexts needed bootstrapping could be packed into one.

Chapter 2

New BGV Variants

We present two new BGV variants developed in the course of [BV16]. Both scheme variants remove the need to relinearize ciphertexts after a multiplication. The first variant (SS) generates secret keys from a special distribution. While SS removes the need for relinearization after a multiplication, its modifications cause an incompatibility with modulus switching. The second variant (QE) publishes an equation quadratic in the secret key publicly. In addition to removing the need for relinearization after multiplication, QE also slightly reduces the noise growth after multiplication compared to HElib's BGV variant. For both variants, we will assume a plaintext space of single bits for simplicity.

2.1 Secret Squared (SS) Variant

The SS BGV variant draws secret keys from a special distribution such that $\mathfrak{s} = \mathfrak{s}^2$. A simple distribution satisfying this property comes about when considering ring elements in the evaluation domain. Since multiplication occurs coordinate-wise in the evaluation domain, we sample secret keys with evaluations sampled uniformly from $\{0, 1\}$. Squaring any elements in this distribution will clearly preserve the value of each evaluation, resulting in the same ring element. We then modify the multiplication of ciphertexts to use this equality.

$\mathsf{KeyGen}_{\mathsf{SS}}$ Samples secret key $\mathfrak{s}_e \leftarrow U\{0, 1\}^n$ in evaluation representation. Converts $\mathfrak{s}_e$ to $\mathfrak{s}$ in coefficient representation. Outputs $\mathfrak{s}$.

$\mathsf{EvalMult}_{\mathsf{SS}}$ Takes inputs $c_0 = (a_0, b_0)$ and $c_1 = (a_1, b_1)$. Computes $c_2 = (a_2, b_2)$ such that

$$\begin{aligned} a_2 &= a_0 b_1 + a_1 b_0 - a_0 a_1 \\ b_2 &= b_0 b_1 \end{aligned}$$

Outputs c_2.

2.1.1 Derivation of $\mathsf{EvalMult}_{\mathsf{SS}}$

First we show the derivation of the computation of c_2. We represent the input ciphertexts as encryptions

$$\begin{aligned} c_0 &= (a_0, b_0 = a_0 \mathfrak{s} + 2e_0 + m_0) \\ c_1 &= (a_1, b_1 = a_1 \mathfrak{s} + 2e_1 + m_1) \end{aligned}$$

Next, we rewrite these ciphertexts as their correctness equations.

$$\begin{aligned} b_0 - a_0 \mathfrak{s} &= 2e_0 + m_0 \\ b_1 - a_1 \mathfrak{s} &= 2e_1 + m_1 \end{aligned}$$

We then consider the product of these equations and group common terms.

$$\begin{aligned} (b_0 - a_0 \mathfrak{s})(b_1 - a_1 \mathfrak{s}) &= (2e_0 + m_0)(2e_1 + m_1) \\ b_0 b_1 - a_0 b_1 \mathfrak{s} - a_1 b_0 \mathfrak{s} + a_0 a_1 \mathfrak{s}^2 &= 4e_0 e_1 + 2e_0 m_1 + 2e_1 m_0 + m_0 m_1 \\ b_0 b_1 - a_0 b_1 \mathfrak{s} - a_1 b_0 \mathfrak{s} + a_0 a_1 \mathfrak{s} &= 4e_0 e_1 + 2e_0 m_1 + 2e_1 m_0 + m_0 m_1 \qquad (\mathfrak{s} = \mathfrak{s}^2) \\ b_0 b_1 - (a_0 b_1 + a_1 b_0 - a_0 a_1) \mathfrak{s} &= 2(2e_0 e_1 + e_0 m_1 + e_1 m_0) + m_0 m_1 \\ b_2 - a_2 \mathfrak{s} &= 2(2e_0 e_1 + e_0 m_1 + e_1 m_0) + m_0 m_1 \end{aligned}$$

Finally, we note that the resulting form is a correctness equation for a ciphertext corresponding to the product of c_0 and c_1, as long as the error term remains small.

2.1.2 Multiplication Correctness

We show the correctness of $\mathsf{EvalMult}_{\mathsf{SS}}$ after a single multiplication $c_2 = \mathsf{EvalMult}_{\mathsf{SS}}(c_0, c_1)$.

$$\begin{aligned}
&\mathsf{Decrypt}_{\mathsf{SS}}(c_2, \mathfrak{s})\\
&\quad= [[b_2 - a_2\mathfrak{s}]_q]_2\\
&\quad= [[b_0b_1 - (a_0b_1 + a_1b_0 - a_0a_1)\mathfrak{s}]_q]_2\\
&\quad= [[a_0a_1\mathfrak{s}^2 + 2a_0e_1\mathfrak{s} + a_0m_1\mathfrak{s} + 2a_1e_0\mathfrak{s}\\
&\qquad + 4e_0e_1 + 2e_0m_1 + a_1m_0\mathfrak{s} + 2e_1m_0 + m_0m_1\\
&\qquad - a_0a_1\mathfrak{s}^2 - 2a_0e_1\mathfrak{s} - a_0m_1\mathfrak{s} - a_0a_1\mathfrak{s}^2 - 2a_1e_0\mathfrak{s} - a_1m_0\mathfrak{s} + a_0a_1\mathfrak{s}]_q]_2\\
&\quad= [[a_0a_1\mathfrak{s} + 2a_0e_1\mathfrak{s} + a_0m_1\mathfrak{s} + 2a_1e_0\mathfrak{s} && (\mathfrak{s} = \mathfrak{s}^2)\\
&\qquad + 4e_0e_1 + 2e_0m_1 + a_1m_0\mathfrak{s} + 2e_1m_0 + m_0m_1\\
&\qquad - a_0a_1\mathfrak{s} - 2a_0e_1\mathfrak{s} - a_0m_1\mathfrak{s} - a_0a_1\mathfrak{s} - 2a_1e_0\mathfrak{s} - a_1m_0\mathfrak{s} + a_0a_1\mathfrak{s}]_q]_2\\
&\quad= [[4e_0e_1 + 2e_0m_1 + 2e_1m_0 + m_0m_1]_q]_2 && (\mathfrak{s}\text{ terms eliminated})\\
&\quad= [4e_0e_1 + 2e_0m_1 + 2e_1m_0 + m_0m_1]_2 && (\text{error terms} << q)\\
&\quad= [m_0m_1]_2\\
&\quad= m_0m_1
\end{aligned}$$

Thus we get back the multiplication of the two messages.

2.1.3 Multiplication Noise Analysis

The noise terms in a SS ciphertext after a single multiplication are $4e_0e_1 + 2e_0m_1 + 2e_1m_0$. Since the plaintext space is $\mathcal{R}_{1,2}$, the guaranteed factor of 2 in each of these terms causes each to be congruent to 0 mod 2, as long as their total magnitude has not exceeded q. Since messages and error distribution samples are very small compared to q, the terms $2e_0m_1$ and $2e_1m_0$ are negligibly larger than e_0 and e_1, the error terms of the input ciphertexts. The $4e_0e_1$ term causes a multiplicative error growth of ciphertexts which is present in both BGV'12 and HElib's variant, but cannot be dealt with in the normal way using modulus switching.

2.1.4 Incompatibility with Modulus Switching

The distribution of secret keys in SS has very low magnitude in the evaluation representation on average, since the evaluations are in $\{0, 1\}$ by definition. Unfortunately, a small magnitude in the evaluation representation does not correlate with a small magnitude in the coefficient representation. We have not found a way to upper-bound the coefficient representation magnitude of SS secret keys to a reasonable level while maintaining the $s = s^2$ property. We have observed in practice that modulus switching SS ciphertexts introduces too much noise to maintain correctness of the ciphertext.

2.1.5 Security

Reasoning about the security of SS requires the definition of a new assumption based on low-entropy distributions in the evaluation representation.

Assumption 2.1.1 (Entropic-Evaluation RLWE (EE-RLWE))**.** *For security parameter λ, integers $m = m(\lambda)$ and $q = q(\lambda)$, and $\mathfrak{s} \leftarrow \mathsf{KeyGen}_{\mathsf{SS}}$, a probabilistic polynomial-time adversary cannot distinguish between a polynomial number of RLWE samples using secret $\mathfrak{s}$ and samples of uniformly random elements in $\mathcal{R}_{m,q}$ with non-negligible advantage.*

While this assumption is similar to the Entropic-RLWE assumption, we note that both are still open questions. The authors of [GKPV10] prove that there is a probabilistic polynomial-time reduction from Decision-LWE to Decision-LWE over a binary distribution with lower entropy. The reduction comes with a loss of dimension from n to $l \leq \frac{k-\omega(\lg n)}{\lg q}$ where k is the entropy of the binary distribution.

We loosely compare EE-RLWE to Entropic-LWE by first assuming that EE-RLWE holds for the same parameters as Entropic-LWE. Since there are n evaluations, each chosen over the uniform binary distribution, the distribution of keys from $\mathsf{KeyGen}_{\mathsf{SS}}$ has n bits of entropy. In this work our larger parameter settings use $n \approx 23000$ and $\lg q \approx 510$, yielding an upper bound of $l \leq 45$. Thus even under this assumption, Entropic-LWE does not imply security of EE-RLWE.

While the SS scheme mainly served as an avenue for exploring the sampling of secret keys from a special distribution to facilitate ciphertext multiplication, the next variant explores publishing of additional public information. Unlike SS, we end up with a fully-functional scheme compatible with modulus switching and bootstrapping.

2.2 Quadratic Equation (QE) Variant

The QE variant publishes a random quadratic equation over $\mathfrak{s}$ to allow multiplication of ciphertexts without key-switching. The scheme has a modified ciphertext multiplication to take advantage of the quadratic equation. It requires a relaxed security assumption that we reason about later. We first describe the modified public key generation and multiplication algorithms.

$\mathsf{KeyGen}_{\mathsf{QE}}$ Generates $\mathfrak{s}, pk_0 \leftarrow \mathsf{KeyGen}_{\mathsf{BGV}}$. Samples $a \overset{U}{\leftarrow} \mathcal{R}_{m,q}$. Computes $b = a\mathfrak{s} + \mathfrak{s}^2$. Outputs $sk = \mathfrak{s}$ and $pk = (pk_0, a, b)$.

$\mathsf{EvalMult}_{\mathsf{QE}}$ Takes two ciphertexts $c_0 = (a_0, b_0)$ and $c_1 = (a_1, b_1)$ and public key $pk = (pk_0, a, b)$. Computes $c_2 = (a_2, b_2)$ such that

$$\begin{aligned} a_2 &= a_0 b_1 + a_1 b_0 + a_0 a_1 a \\ b_2 &= b_0 b_1 + a_0 a_1 b \end{aligned}$$

Outputs c_2.

Note that the published (a, b) have no noise associated with them, requiring a new security assumption that we will explore after showing correctness of the scheme. Also note that (a, b) are required to carry out multiplication, but are not required for any other operations.

2.2.1 Correctness of $\mathsf{EvalMult}_{\mathsf{QE}}$

Given ciphertexts $c_0 = (a_0, b_0)$ and $c_1 = (a_1, b_1)$, we show that $c_2 = (a_2, b_2) = \mathsf{EvalMult}_{\mathsf{QE}}(c_0, c_1)$ decrypts to the product of messages of c_0 and c_1. The published quadratic equation will provide a way of eliminating the s^2 terms by substituting $b = a\mathfrak{s} + \mathfrak{s}^2$.

$$\begin{aligned}
\mathsf{Decrypt}_{\mathsf{QE}}(c_2) &= [[b_2 - a_2\mathfrak{s}]_q]_2 \\
&= [[b_0b_1 + a_0a_1b - (a_0b_1 + a_1b_0 + a_0a_1a)\mathfrak{s}]_q]_2 \\
&= [[(a_0\mathfrak{s} + 2e_0 + m_0)(a_1\mathfrak{s} + 2e_1 + m_1) + a_0a_1b \\
&\qquad - (a_0(a_1\mathfrak{s} + 2e_1 + m_1) + a_1(a_0\mathfrak{s} + 2e_0 + m_0) + a_0a_1a)\mathfrak{s}]_q]_2 \\
&= [[a_0a_1\mathfrak{s}^2 + 2a_0e_1\mathfrak{s} + a_0m_1\mathfrak{s} + 2a_1e_0\mathfrak{s} + 4e_0e_1 + 2e_0m_1 \\
&\qquad + a_1m_0\mathfrak{s} + 2e_1m_0 + m_0m_1 + a_0a_1b \\
&\qquad - (a_0a_1\mathfrak{s} + 2a_0e_1 + a_0m_1 + a_1a_0\mathfrak{s} + 2a_1e_0 + a_1m_0 + a_0a_1a)\mathfrak{s}]_q]_2 \\
&= [[a_0a_1\mathfrak{s}^2 + 4e_0e_1 + 2e_0m_1 + 2e_1m_0 + m_0m_1 + a_0a_1b \\
&\qquad - (2a_0a_1\mathfrak{s} + a_0a_1a)\mathfrak{s}]_q]_2 \\
&= [[4e_0e_1 + 2e_0m_1 + 2e_1m_0 + m_0m_1 + a_0a_1(a\mathfrak{s} + s^2) \\
&\qquad - a_0a_1\mathfrak{s}^2 - a_0a_1a\mathfrak{s}]_q]_2 \\
&= [[4e_0e_1 + 2e_0m_1 + 2e_1m_0 + m_0m_1]_q]_2 \\
&= \; m_0m_1
\end{aligned}$$

Thus we successfully recover the product of messages after a ciphertext multiplication. Next we state the assumption underlying the security of QE.

2.2.2 Security

Publishing additional information about $\mathfrak{s}$ in the public key is not a new idea, as the key-switching operation in [BGV12] uses published key-switching matrices that contain encryptions of the bits of $\mathfrak{s}$. However, these encryptions are created as RLWE samples using an error distribution, where as the quadratic equation in QE contains no additive noise, necessitating a new assumption. We begin by defining the RLWE with Quadratic Equation problem.

Informally, the RLWEQE problem is to distinguish RLWE samples from random samples with the help of a single random equation quadratic in $\mathfrak{s}$.

Definition 2.2.1 (RLWE with Quadratic Equation Problem (RLWEQE)). *For security parameter λ, let integers $q = q(\lambda)$, $m = m(\lambda)$, distribution $\chi = chi(\lambda)$ over $\mathcal{R}_{m,q}$. Let*

$b = a\mathfrak{s} + \mathfrak{s}^2$. The RLWEQE *problem is to distinguish $\{(a, b, a_i, a_i\mathfrak{s} + e_i)\}_i$ from $\{(a, b, a_i, b_i)\}_i$, where $a, a_i, b_i, \mathfrak{s} \overset{U}{\leftarrow} \mathcal{R}_{m,q}$, $e_i \overset{\chi}{\leftarrow} \mathcal{R}_{m,q}$.*

Note that only a single sample of the form $b = a\mathfrak{s} + \mathfrak{s}^2$ is provided for the adversary. If an additional quadratic sample was provided, the adversary could easily solve for $\mathfrak{s}$.

Assumption 2.2.2 (RLWEQE Assumption)**.** *The* RLWEQE *assumption states that a* PPT *adversary cannot solve the* RLWEQE *problem with non-negligible advantage over the randomness of the experiment.*

If the RLWEQE assumption holds, then RLWEQE reduces directly to RLWE without loss of parameters.

Theorem 2.2.3 (RLWE $\implies$ RLWEQE)**.** *Given an oracle $\mathcal{O}$ for* RLWE*, there exists a* PPT *adversary $\mathcal{A}$ that solves* RLWEQE*.*

Proof.
We define $\mathcal{A}$ such that on RLWEQE challenge (a, b, c, d) $\mathcal{A}$ outputs $\mathcal{O}(c, d)$. By the definition of RLWEQE, (c, d) is either the uniform distribution over $\mathcal{R}_{m,q}$ or the distribution of RLWE samples. Since $\mathcal{O}$ has non-negligible advantage in distinguishing RLWE samples from random, $\mathcal{A}$ distinguishes RLWEQE challenges with the same non-negligible advantage. □

Assuming RLWEQE does not afford us anything without first cryptanalyzing the assumption. We begin with a problem definition, RQE, that examines the pseudorandomness of the published quadratic equation in QE. Unfortunately, we later find that solving RQE is feasible.

Definition 2.2.4 (Ring Quadratic Equation Problem (RQE))**.** *For security parameter λ, let integers $q = q(\lambda)$, $m = m(\lambda)$. Let $b = a\mathfrak{s} + \mathfrak{s}^2$. The* RQE *problem is to distinguish (a, b) from (c, d), where $a, c, d, \mathfrak{s} \overset{U}{\leftarrow} \mathcal{R}_{m,q}$.*

Here we show that RQE is indeed solvable in polynomial time.

Theorem 2.2.5 (RQE Feasible)**.** *There exists a* PPT *adversary that can solve* RQE *with non-negligible advantage.*

Let q be an odd prime. We create a distinguisher $\mathcal{D}$ that distinguishes RQE samples from uniform random samples with non-negligible advantage as follows:

Procedure 1 $\mathcal{D}$: Distinguish $(a, b = a\mathfrak{s} + \mathfrak{s}^2)$ from random (r_0, r_1)

Input: (a, b) an RQE or uniform random sample from $\mathcal{R}_{m,q}$

Output: 0 or 1

```
  $c \leftarrow b + \frac{a^2}{4}$                                  // Potentially "complete the square"
  for all $i \in \mathbb{Z}_m^*$ do
    $l \leftarrow c \bmod (x - \xi_m^i)$
    if $l^{\frac{q-1}{2}} \not\equiv 1 \bmod q \wedge l \not\equiv 0 \bmod q$ then      // Check if $l$ is not a quadratic residue
      return 0
    end if
  end for
  return 1
```

Informally, $\mathcal{D}$ assumes that the input is an RQE sample. It "completes the square" such that if (a, b) is an RQE sample, then c is a quadratic residue. Finally, $\mathcal{D}$ checks whether c is a quadratic residue by checking for quadratic residuosity with respect to each ξ_m^i. We now prove some lemmata to later show the correctness of $\mathcal{D}$.

Lemma 2.2.6. $c \in \mathrm{QR}(\mathcal{R}_{m,q}) \Longleftrightarrow \forall i \in \mathbb{Z}_m^* c \bmod (x - \xi_m^i) \in \mathrm{QR}(\mathbb{Z}_q)$

Proof. Follows from the Chinese Remainder Theorem. □

Lemma 2.2.7. $l^{\frac{q-1}{2}} \equiv 1 \bmod q \vee l \equiv 0 \bmod q \Longleftrightarrow l \in \mathrm{QR}(\mathbb{Z}_q)$

Proof. Follows from Euler's Criterion. □

Lemma 2.2.8. $(a, b) \leftarrow \mathsf{RQE} \Longrightarrow b + \frac{a^2}{4} \in \mathrm{QR}(\mathcal{R}_{m,q})$

Proof. $b + \frac{a^2}{4} = \mathfrak{s}^2 + a\mathfrak{s} + \frac{a^2}{4} = (\mathfrak{s} + \frac{a}{2})^2$ □

Lemma 2.2.9. $(a, b) \overset{U}{\leftarrow} \mathcal{R}_{m,q} \Longrightarrow \Pr[b + \frac{a^2}{4} \in \mathrm{QR}(\mathcal{R}_{m,q})] = \left(\frac{1}{2} + \frac{1}{q}\right)^n$

Proof.

Note $|QR(\mathbb{Z}_q)| = \frac{q-1}{2} + 1 = \frac{q+1}{2}$ since $f(x) = x^2$ is 2-to-1 for all x (besides $f(0) = 0$), for odd q. $c = b + \frac{a^2}{4}$ is uniformly random over $\mathcal{R}_{m,q}$ by the independent uniform random sampling of b. Thus $\forall i \in \mathbb{Z}_m^*, c \bmod (x - \xi_m^i)$ is uniformly random over $\mathbb{Z}_q$ and $\Pr[c \bmod (x - \xi_m^i) \in \mathrm{QR}(\mathbb{Z}_q)] = \frac{\frac{q+1}{2}}{q} = \frac{1}{2} + \frac{1}{q}$. Finally,

$$\Pr[c \in \mathrm{QR}(\mathcal{R}_{m,q})] = \prod_{i \in \mathbb{Z}_m^*} \Pr[c \bmod (x - \xi_m^i) \in \mathrm{QR}(\mathbb{Z}_q)] = \prod_{i \in \mathbb{Z}_m^*} \frac{1}{2} + \frac{1}{q} = \left(\frac{1}{2} + \frac{1}{q}\right)^n$$

where the first equality follows from Lemma 2.2.6. □

We next observe that $\mathcal{D}$ solves RQE.

Theorem 2.2.10. *$\mathcal{D}$ is a* PPT *algorithm that solves* RQE *with non-negligible advantage.*

Proof.

By Lemma 2.2.6 and Lemma 2.2.7, $\mathcal{D}$ outputs 1 if $b + \frac{a^2}{4}$ is a quadratic residue and 0 otherwise. By Lemma 2.2.8, $\Pr[(a,b) \xleftarrow{U} \mathcal{R}_{m,q} : \mathcal{D}(a,b) = 1] = 1$. By Lemma 2.2.9, $\Pr[(a,b) \leftarrow \mathsf{RQE} : \mathcal{D}(a,b) = 1] = \left(\frac{1}{2} + \frac{1}{q}\right)^n$. Thus $\mathcal{D}$ has advantage

$$\left|\Pr\left[(a,b) \xleftarrow{U} \mathcal{R}_{m,q} : \mathcal{D}(a,b) = 1\right] - \Pr\left[(a,b) \leftarrow \mathsf{RQE} : \mathcal{D}(a,b) = 1\right]\right| = 1 - \left(\frac{1}{2} + \frac{1}{q}\right)^n$$

It is clear that $\mathcal{D}$ is PPT as modular exponentiation may be computed with repeated squaring.

□

Given that RQE is feasible, we explore an approach to break RLWEQE directly by attempting to recover $\mathfrak{s}$ from the quadratic equation. Clearly if $\mathfrak{s}$ can be recovered, RLWEQE samples are trivially distinguishable and the QE scheme is broken.

We show an upper-bound on the time to break QE by defining an adversary $\mathcal{A}_{\mathsf{QE}}$ in Procedure 2 that searches all quadratic residues of $b + \frac{a^2}{4}$ to find $\mathfrak{s} + \frac{a}{2}$ and thus find $\mathfrak{s}$.

Procedure 2 $\mathcal{A}_{\mathsf{QE}}$: Recover $\mathfrak{s}$ from $(a, b = a\mathfrak{s} + \mathfrak{s}^2)$

```
Input: $(a, b = a\mathfrak{s} + \mathfrak{s}^2)$, $a, s \xleftarrow{U} \mathcal{R}_{m,q}$
Output: $\mathfrak{s}$
  $c \leftarrow b + \frac{a^2}{4}$                                   // "Complete the square"
  $Q \leftarrow []$
  for all $i \in \mathbb{Z}_m^*$ do
    $Q \leftarrow Q + \mathsf{TonelliShanks}(c \bmod (x - \xi_m^i))$      // Find CRT squareroots
  end for
  for all reconstructed squareroots $r$ do                 // Using $\pm Q_i$
    $\mathfrak{s}' \leftarrow r - \frac{a}{2}$
    if $b = a\mathfrak{s}' + \mathfrak{s}'^2$ then
      return $\mathfrak{s}'$
    end if
  end for
```

Theorem 2.2.11. $\mathcal{A}_{\mathsf{QE}}$ *correctly outputs* $\mathfrak{s}$.

Proof.

We first note that $c = b + \frac{a^2}{4} \in \text{QR}(\mathcal{R}_{m,q})$ since $c = (\mathfrak{s} + \frac{a}{2})^2$. By the correctness of TonelliShanks [Lin99], Q contains squareroots of $c \bmod (x - \xi_m^i)$. Assuming odd prime q, for each squareroot $r \in Q$, we compute the only other squareroot as $-r$. By Chinese Remainder Theorem, every squareroot of c uniquely corresponds to n squareroots of $c \bmod (x - \xi_m^i)$ for $i \in \mathbb{Z}_m^*$. By enumerating over all squareroots r corresponding to positive and negative elements of Q, we enumerate over all possible squareroots of c. Thus exactly one of these squareroots must be $\mathfrak{s} + \frac{a}{2}$, in which case the if condition holds and $\mathcal{A}_{\mathsf{QE}}$ outputs $\mathfrak{s}$. $\square$

Runtime.
The runtime of $\mathcal{A}_{\mathsf{QE}}$ depends on the runtime of TonelliShanks as well as the number of enumerated squareroots. The runtime of TonelliShanks is trivially lower-bounded by the bit-length of q, $\lg q$, modular multiplications. If $c \equiv 0 \bmod (x - \xi_m^i)$, 0 is its only squareroot. Otherwise, c has two squareroots $\text{mod}(x - \xi_m^i)$. Thus we must count the expected number of zero squareroots. Note that

$$c = \mathfrak{s}^2 + a\mathfrak{s} + \frac{a^2}{4} = \left(\mathfrak{s} + \frac{a}{2}\right)^2 \equiv 0 \bmod (x - \xi_m^i) \Longleftrightarrow \mathfrak{s} + \frac{a}{2} \equiv 0 \bmod (x - \xi_m^i)$$

because the only squareroot of 0 is 0. Since $\mathfrak{s}$ and a are independent and uniform samples, $\Pr[\mathfrak{s} + \frac{a}{2} \equiv 0 \bmod (x - \xi_m^i)] = \frac{1}{q}$. Thus the expected number of 0 squareroots is $\frac{n}{q}$ and the expected number of pairs of squareroots is $n\left(1 - \frac{1}{q}\right)$. Therefore the expected number of enumerated squareroots is $2^{n\left(1-\frac{1}{q}\right)}$, yielding an expected runtime of

$$\Omega\left(\lg q + 2^{n\left(1-\frac{1}{q}\right)}\right)$$

Note that the expected runtime is exponential in the ring dimension, which is already sufficiently large for normal parameter settings.

2.2.3 Theoretical Performance

Like SS, QE avoids the need for key-switching after every multiplication. In addition, only ring multiplications and additions are performed, as opposed to the tensor product of coefficients required in BGV and HElib. Thus we predict a large speedup in multiplication-heavy circuits.

2.3 HElib Implementation

We present a report on implementations of both the SS and QE FHE variants, including performance benchmarks where feasible. The implementations are built on top of HElib git commit `a5921a08e8b418f154be54f4e39a849e74489319` hosted publicly at https://github.com/shaih/helib/tree/a5921a08e8b418f154be54f4e39a849e74489319. When we refer to vanilla HElib, we are referring to HElib at this commit hash.

2.3.1 SS Implementation

The SS implementation required changes to both key generation and ciphertext multiplication.

HElib Changes

Below we list the notable changes to the HElib FHE library to implement the SS scheme.

`DoubleCRT::randomizeZeroOne()` samples a ring element in evaluation form such that each evaluation is 0 or 1 with $\frac{1}{2}$ probability.

`FHESecKey::GenSecKeyZeroOne()` generates a secret key ring element using `DoubleCRT::randomizeZeroOne`. The key is imported using `FHESecKey::ImportSecKey`. Vanilla HElib stores a secret key's Hamming weight in coefficient form along with its public key in order to facilitate computation of the noise of a ciphertext encrypted using the public key. We note that secret keys in SS still have an obvious Hamming weight, but in evaluation form instead of coefficient form. From [GHS12] appendix A.5, we observe that the resulting noise variance from a Hamming weight h key in coefficient form is the same as Hamming weight h in evaluation form. A major benefit of SS key generation is that the key is generated in evaluation form and does not have to be converted from coefficient form to perform efficient computations. Vanilla HElib samples keys of a specified Hamming weight in coefficient form first, then coverts them into `DoubleCRT` representation.

`Ctxt::multiplyByWithSquareKey(Ctxt c1, Ctxt c2)` evaluates the product of `c1` and `c2` under the assumption that the underlying secret key is sampled as in $\mathsf{KeyGen}_{\mathsf{SS}}$. The result is stored in `this`. Depending on a compiler flag, the vanilla HElib multiplication procedure is overridden with this method so that all uses of multiplication are automatically converted to $\mathsf{EvalMult}_{\mathsf{SS}}$.

Single Multiplication Performance

Given the incompatibility of SS with modulus switching, we restricted benchmarks to the performance of generating a secret key, encrypting a ciphertext, and squaring that ciphertext. We compare the SS implementation to vanilla HElib. The time spent in seconds for each implementation in important subroutines is as follows:

	Vanilla		SS	
	Time (s)	# Occurrences	Time (s)	# Occurrences
`GenSecKey`	0.0557	1	0.0129	1
`GenKeySWmatrix`	0.0418	2	-	-
`operator*=`	0.0011	1	0.0013	1
`reLinearize`	0.0207	1	-	-

Comparison of runtimes between vanilla HElib and SS in the course of a secret key generation, encryption, and squaring using default HElib parameters `m = 7781` with three ciphertext primes.

The generation of the secret key is about four times faster in SS than in vanilla HElib. We attribute this to SS not having to generate any key switching matrices for multiplication. Note that `GenKeySWmatrix` is a subroutine of `GenSecKey` that is no longer called in the SS implementation. As for multiplication, the time spent in `operator*=` represents the tensor product for vanilla HElib, while it represents the entire multiplication evaluation for SS. The tensor product and $\mathsf{EvalMult}_{\mathsf{SS}}$ take about the same amount of time, but vanilla HElib must perform an expensive relinearization after the tensor product. Thus the entire multiplication operation is about sixteen times faster for SS.

2.3.2 QE Implementation

The implementation of QE in HElib uses the same fixed Hamming weight key generation as vanilla HElib. Upon secret key import, along with storing the key's Hamming weight, the QE implementation generates a random quadratic equation over $\mathfrak{s}$ and stores its coefficients in the public key. HElib has the ability to store multiple keys in a single `FHESecKey` class. Thus we store a list of pairs of quadratic equation coefficients in `FHEPubKey::s2Relations` where the coefficient pair at index i corresponds to the secret key at the same index.

HElib Changes

`vector<pair<DoubleCRT>> FHEPubKey::s2Relations` stores a list of pairs of ring elements in evaluation form. It is part of the public key as it is needed for evaluating multiplications of ciphertexts.

`FHESecKey::ImportSecKey(DoubleCRT sk)` is an existing helper method to store a newly generated secret key ring element `sk` and create the public encryption key (basically an encryption of 0). We modified this routine to additionally generate the random equation quadratic in `sk`. We first generate a uniformly random ring element `a` in coefficient form using `DoubleCRT::randomize()`. We then compute `b` as in $\mathsf{KeyGen}_{\mathsf{QE}}$ and store `pair(a, b)` in `s2Relations`. It is important to note that when working with a modulus chain, `a` and `b` should be generated over all of the ciphertext primes, just as the secret key is.

`Ctxt::multiplyByWithS2Relation(Ctxt c1, Ctxt c2, pair<DoubleCRT> ab)` computes the product of `c1` and `c2` using the quadratic equation coefficients `ab` and stores it in `this`. It is assumed that `c1` and `c2` are on the same level of the modulus chain, i.e. they are stored modulo the same primes. However, their level may not match that of `ab` which was generated at the top level. For this reason, ring multiplications with coefficients `ab` are done with `matchPrimeSets` set to `false`, indicating to HElib that primes in the modulus of `ab` that are not present in `c1` or `c2` should be ignored. After the product is computed, the noise variance of the product is updated to roughly `c1.noiseVar * c2.noiseVar + c1.noiseVar + c2.noiseVar`. As in the SS implementation, the compiler flag `PUBLIC_RELATION` replaces the vanilla HElib multiplication routine with this method.

We note that the implementations of addition, automorphism, and multiplication by constant required no changes from vanilla HElib. Since QE multiplication preserves correctness for circuit depth > 1, the higher-level functionalities in HElib that are built on top of multiplication still function using QE. These higher-level functionalities include bootstrapping and homomorphic evaluation of the Advanced Encryption Standard (AES) circuit. Next, we present benchmarks of various FHE operations comparing the performance of QE to vanilla HElib.

Single Multiplication Performance

We first present a benchmark of a single secret key generation, followed by a multiplication of two ciphertexts.

	Vanilla		QE	
	Time (s)	# Occurrences	Time (s)	# Occurrences
`GenSecKey`	0.0557	1	0.0169	1
`GenKeySWmatrix`	0.0418	2	-	-
`operator*=`	0.0011	1	0.0013	1
`reLinearize`	0.0207	1	-	-

Comparison of runtimes between vanilla HElib and QE in the course of a secret key generation, encryption, and multiplication using default HElib parameters `m = 7781` with three ciphertext primes.

As with SS, we observe a significant speedup of secret key generation with QE. Again, we attribute this to QE not needing any key switching matrices for multiplication. We note that $\texttt{GenSecKey}_{\mathsf{QE}}$ takes 0.004 seconds longer than $\texttt{GenSecKey}_{\mathsf{SS}}$, accounting for the time to generate and store the quadratic equation in `s2Relations`.

Exponentiation Performance

While the performance of a single ciphertext multiplication provides a simple validation of the performance of QE, FHE really only becomes useful with larger circuits. Next, we observe the performance in a multiplication tree of depth 45. In this benchmark, we generate a secret key, encrypt a ciphertext, and then square the ciphertext 45 times, effectively computing $c^{2^{45}}$. Since this circuit is much deeper, we choose a modulus chain length of 23 to handle the noise growth.

	Vanilla		QE	
	Time (s)	# Occurrences	Time (s)	# Occurrences
`GenSecKey`	2.364	1	0.869	1
`GenKeySWmatrix`	1.767	2	-	-
`operator*=`	32.150	45	12.138	45
⌙ `modDownToSet`	30.549	44	10.369	44
`reLinearize`	28.274	45	-	-

Comparison of runtimes between vanilla HElib and QE during secret key generation, encryption, and ciphertext exponentiation ($c^{2^{45}}$) using HElib parameters $\mathtt{m} = \mathtt{43691}$ with 23 ciphertext primes.

We first note that secret key generation for this larger parameter set is still about four times faster for QE than vanilla HElib. `modDownToSet` is the most expensive part of modulus switching. `modDownToSet` now appears in the runtime report because HElib must modulus switch a ciphertext that has experienced noise growth from multiplication before multiplying that ciphertext again. The check for whether to modulus switch, and the modulus switching itself, are performed in `operator*=` before the actual multiplication happens. For this reason, we denote `modDownToSet` with a " ⌙ " symbol to indicate it is a subroutine of `operator*=`. The remainder of the multiplication time is spent in the tensor product for Vanilla HElib and in the $c_2 = (a_2, b_2)$ computation from the definition of $\mathsf{EvalMult}_{\mathsf{QE}}$ for QE. Thus the tensor product and (a_2, b_2) computations take approximately the same amount of time. One might expect that the `modDownToSet` operation consumes the same amount of time between vanilla HElib and QE, since we did not change its implementation at all. However, in order to key-switch a ciphertext, vanilla HElib scales that ciphertext up to a modulus with many additional primes called "special" primes. In the next call to `modDownToSet`, a key-switched ciphertext must have all of the special primes removed in addition to the ciphertext prime that is being removed. Thus, even though a ciphertext in either implementation ends up with the same composite modulus after each multiplication, the runtime of modulus switching for vanilla HElib is about three times longer because of the added special primes from key switching. As with the single multiplication, QE does not use `reLinearize` as the linearity in $\mathfrak{s}$ is preserved by the definition of $\mathsf{EvalMult}_{\mathsf{QE}}$. Taking into account each of the multiplication subroutines, we observe approximately a four times speed increase of multiplication with QE compared to vanilla HElib.

In terms of noise growth caused by multiplication, we observe that the multiplicative factor in both schemes dominates the growth compared to any additional noise brought on by key-switching. Thus QE experiences effectively the same noise growth per multiplication as vanilla HElib and cannot evaluate deeper circuits using the same parameters. For example, both schemes result in incorrect decryptions if an additional squaring of the ciphertext in the previous benchmark is performed.

AES Performance

Moving on from exclusively multiplicative circuits, we next present a benchmark of an evaluation of AES without bootstrapping. We use the HElib provided parameters of `m = 28679` and 21 ciphertext primes.

	Vanilla		QE	
	Time (s)	# Occurrences	Time (s)	# Occurrences
`AESEncrypt`	130.836	1	109.749	1
`AESDecrypt`	198.836	1	160.727	1
`GenSecKey`	1.327	1	0.364	1
`GenKeySWmatrix`	22.404	40	22.170	40
`operator*=`	68.038	40	58.698	40
`modDownToSet`	123.246	335	101.683	296
`reLinearize`	221.115	394	175.937	314
`smartAutomorph`	194.404	314	177.509	314

Comparison of runtimes between vanilla HElib and QE during secret key generation, encryption, and evaluation of AES encryption and decryption using HElib parameters `m` = `28679` with 21 ciphertext primes.

From a high level, we observe that QE performs about 1.19 times better than vanilla HElib for AES encryption and 1.24 times better for AES decryption. We note that this is a significantly smaller, but non-negligible, performance increase compared to multiplication. The main difference between the AES benchmark and the multiplication benchmarks is that AES evaluation uses automorphisms over the ciphertexts in order to shift packed plaintexts and compute using ciphertext SIMD operations. HElib performs automorphisms in the `smartAutomorph` routine, which in turn makes calls to `reLinearize` to perform key-switching from a key $\mathfrak{s}(x^i)$ to key $\mathfrak{s}(x)$. We observe that `reLinearize` is called exactly once per `smartAutomorph` operation in `QE`, consistent with `QE` not using `reLinearize` for multiplication. Note that `modDownToSet` is no longer strictly a subroutine of `operator*=` since special primes must be removed using this method after an automorphism key-switching is performed as well. Even though both schemes must remove special primes after an automorphism, the difference in time spent in `modDownToSet` comes from `QE` not having to remove special primes in the case of multiplication. Both executions consumed approximately 1.11GB of memory and spent about the same amount of time generating key-switching matrices, which is consistent with the fact that `QE` now needs key-switching matrices in order to perform the automorphisms necessary for AES evaluation.

Bootstrapping Performance

Another fundamental higher-level FHE procedure is bootstrapping. We present a benchmark comparing a ciphertext bootstrapping in vanilla HElib to the same in `QE` with parameter $m = 28679$ and 18 primes in the modulus chain.

	Vanilla		QE	
	Time (s)	# Occurrences	Time (s)	# Occurrences
`reCrypt`	340.225	1	168.856	1
`GenKeySWmatrix`	24.696	64	23.784	64
`operator*=`	119.686	504	69.819	504
`modDownToSet`	128.429	570	77.133	570
`reLinearize`	182.749	650	62.717	146
`smartAutomorph`	59.564	148	63.322	148

Comparison of runtimes between vanilla HElib and `QE` during secret key generation, encryption, and bootstrapping using HElib parameters `m = 28679` with 18 ciphertext primes.

Overall, we observe a significant bootstrapping speedup of about two times in the `QE` implementation. We can see that this speedup results from `modDownToSet` not having to process special primes after a multiplication in `QE`, as well as 504 fewer calls to `reLinearize` (as it is not needed after a multiplication in `QE`). From the occurrence counts, we note that bootstrapping is a much more multiplication-intensive operation compared to evaluation of AES. Specifically, bootstrapping performs about thirty times more calls to `operator*=` per call to `smartAutomorph` compared to AES evaluation.

2.3.3 Summary

Overall, while SS cannot perform modulus switching, its multiplication performance shares the same benefits as QE. We note that the distribution of secret keys underlying SS is very similar to HElib's Hamming-weight scheme, but sampled in the evaluation representation instead of the coefficient representation. Thus we think the security of SS may be more closely related to that of vanilla HElib than QE, though we have no formal proof of this. Regarding QE, our results indicate that the performance benefits of QE are best harnessed with multiplication-intensive circuits, yielding a performance increase of two to four times between multiplication and bootstrapping.

We note that both SS and QE are particularly well-suited to circuits with a small amount of multiplications. In previous schemes, performing multiplications required a tradeoff between ciphertext size and performance (by either allowing the underlying secret key to grow, or performing key-switching). With both SS and QE, we avoid expensive key-switching while also maintaining the size of ciphertexts.

Chapter 3

Homomorphic Integer Addition

3.1 Approach

While FHE trivially supports addition of integers modulo the plaintext space (by simply using $\mathsf{EvalAdd}_{\mathcal{E}}$), many applications require addition of larger numbers. We see integer addition as one of the simple but universally-necessary building blocks to any FHE application, leading us to explore how fast we can add integers with HElib. The main opportunities for optimization of FHE evaluation of an integer addition circuit are the type of addition circuit used, the amount of ciphertext packing, and when and how frequently bootstrapping is performed. We break the analysis in this chapter up by the type of addition circuit being explored, starting with the simplest of the three, ripple carry, followed by carry-lookahead, and finally Kogge-Stone.

It is important to first note the correlation between bit-wise logical operations and ciphertext operations. When in plaintext space $p = 2$, we use a ciphertext multiplication as a binary-AND gate and ciphertext addition as a binary-XOR gate (think of multiplication and addition $\mod 2$). To perform a binary-inverse, we add an encryption of 1. To zero a ciphertext, we add the ciphertext to itself. Some circuits will require the use of an OR gate. Unfortunately, there is no native ciphertext operation with plaintext space $p = 2$ that corresponds to binary OR. Instead, we use the universality of NAND gates to convert any OR gate between two signals in the following way:

$$A \vee B \Longleftrightarrow \neg(\neg A \wedge \neg B)$$

Thus we can satisfy OR logic using ciphertext multiplication for AND and adding an encryption of 1 for inversion.

With HElib's recommendation for secure parameters, one ciphertext has more than enough plaintext slots to fit every encrypted bit of a 32-bit integer. The main difference between ripple carry computation using individually-encrypted bits and a single ciphertext per 32-bit integer is that two bits at different indices in a packed ciphertext can only interact with the use of a ciphertext shift, as opposed to simply indexing into a vector in the individually-encrypted case. We note that automorphism is not a cheap operation as it requires a key-switching operation afterwards to return the ciphertexts to encryptions under $\mathfrak{s}(x)$. A benefit of ciphertext packing is that up to 32 operations may be carried out at once. Whether ciphertext packing is beneficial becomes a tradeoff between shifting and SIMD versus cheap indexing and one bit operation at a time.

3.2 Ripple Carry Circuit

With the ripple carry circuit being the simplest of the three main types of adder circuits, we present its implementation and benchmarks first.

3.2.1 Bit-Wise

In the first implementation of a ripple carry circuit, we use plaintext space $p = 2$ and encode one plaintext bit per ciphertext. Thus to encrypt two 32-bit integers, we create two vectors, each of 32 ciphertexts. With these two vectors of ciphertexts, we perform the ripple carry addition roughly as below:

Procedure 3 Add two vectors of ciphertexts, `a` and `b`, with ripple carry

Input: `a`, `b`, two vectors of ciphertexts encoding the bits of integers increasing in significance

Output: `s`, a vector of ciphertexts encoding `a + b`

```
c ← Encrypt_$\mathcal{E}$(0)
for i = 0...31 do
  s[i] ← a[i] + b[i] + c
  c ← (a[i] * b[i]) || ((a[i] + b[i]) * c)          // (a ∧ b) ∨ ((a ⊕ b) ∧ c)
  if c.noisy then
    bootstrap(c)
  end if
end for
return s
```

In the above procedure, `||` corresponds to binary OR. As previously mentioned, there is no native ciphertext operation for binary OR. We use `||` here for brevity but expand the logic to the NAND form in the actual implementation. We also abbreviate the logic of whether to bootstrap the carry ciphertext as `c.noisy`. In the actual implementation, this logic is dependent on the modulus level of the ciphertext and the parameters of the scheme. Finally, the choice of ciphertext to bootstrap in this circuit is straightforward, as `c` is the only ciphertext whose depth increases with each loop iteration.

3.2.2 Packed Ciphertexts

We next present a rough sketch of our ciphertext-packed ripple carry adder. While the ripple carry circuit does not lend itself well to SIMD operations, we present the algorithm as an instructive example.

Procedure 4 Add two packed ciphertexts, `a` and `b`, with ripple carry

Input: `a`, `b`, two ciphertexts with the bits of integers packed increasing in significance

Output: `s`, a ciphertext encoding `a + b`

```
s ← a + b
c ← a * b
for i = 1...31 do
  shift(c, 1)
  c' ← c
  c ← c * s
  s ← s + c'
  bootstrapIfNoisy(c)
  bootstrapIfNoisy(s)
end for
return s
```

On each iteration, the algorithm shifts the ciphertext holding the carry bits, effectively performing the same rippling as incrementing the index into a vector in the bit-wise algorithm. We note that the depth of both `c` and `s` grow with each iteration, requiring the bootstrapping of both ciphertexts once the noise threshold is reached.

One notable difference in the packed algorithm is that it contains no explicit OR logic. This SIMD adaptation of ripple carry computes the carry at every position on each iteration and moves those carry bits to the next position, adding them to the result `s`. The nature of packed ciphertexts almost necessitates this type of adaptation, as it is only possible to isolate a single bit using masking ciphertexts. We demonstrate this technique in the following adder circuits.

3.2.3 Performance

We present a benchmark comparing bit-wise and packed ripple carry algorithms using parameters recommended by the HElib authors. We settle on the parameter choice of `m = 31775` based on how many levels are required to allow bootstrapping as well as a few circuit operations. This parameter setting is expected to provide security while using between 1 and 2 gigabytes of RAM for each benchmark.

	Bit-Wise		Packed	
	Time (s)	# Occurrences	Time (s)	# Occurrences
`rippleAdd`	1667.74	1	2815.57	1
`reCrypt`	1646.63	6	2775.85	10
`operator*=`	334.508	2613	537.549	4233
`modDownToSet`	400.180	3122	623.622	5135
`reLinearize`	420.967	1164	728.296	2004
`smartAutomorph`	425.474	1176	715.892	2024

Comparison of runtimes between bit-wise and packed ciphertext encodings during a single ripple carry addition evaluation using HElib parameters $\mathtt{m} = \mathtt{31775}$ with 12 ciphertext primes and the QE scheme.

We observe that the performance of the packed algorithm is considerably slower than the bit-wise version. Approximately 98% of the total computation time is spent in bootstrapping (the `reCrypt` routine) for both algorithms. While this indicates that larger parameters would likely allow more circuit operations per bootstrap and reduce the proportion of time spent bootstrapping, we elected to keep the memory usage safely below 2 gigabytes to prevent any swapping or memory compression. Since the call to `shift` in the packed version consumes a modulus level, the noise grows twice as fast as in the bit-wise version. In addition, the packed algorithm bootstraps two ciphertexts instead of one. These two aspects account for the performance difference mostly caused by additional bootstrapping. On a side note, the bit-wise algorithm consumes 1.9 gigabytes of memory, 0.5 gigabytes more than the packed version, since it stores more ciphertexts to accomplish the computation.

3.3 Carry-Lookahead Circuit

A carry-lookahead circuit computes a propagate and generate signal for contiguous blocks, computes the carry input for each block from these, and finally computes the real sum for each block. The first and last steps may be computed in parallel over all of the blocks. As opposed to the ripple adder implementation, we only present a packed ciphertext algorithm for carry-lookahead addition. Note that some details such as bootstrapping logic, masking and shifts between phases are omitted for brevity.

Procedure 5 Add two packed ciphertexts, `a` and `b`, with carry-lookahead

Input: `a`, `b`, two ciphertexts with the bits of integers packed increasing in significance

Output: `s`, a ciphertext encoding `a + b`

```
B ← 4, p ← a + b                                   // block size B
for lgShift = 0..lg(B) do                          // Phase 1: block-level propagate
  p' ← p,
  shift(p', pow(2, lgShift))
  p ← p * p'
end for
s ← a + b,  cB ← a * b
for i = 1...B do                                   // Phase 2: block-level generate
  cB' ← cB
  shift(cB', 1)
  cB ← cB + (cB' * s)
  s ← s + cB'
end for
cIn ← p * cB
for i = 1...32/B do                                // Phase 3: carry-lookahead
  cIn' ← cIn
  shift(cIn', B)
  cIn ← (p * cIn') || cB
end for
s ← a + b,  c ← (cIn * s) || (a * b)
for i = 1..B do                                    // Phase 4: final sum
  shift(c', 1)
  c' ← c
  c ← c * s
  s ← s + c'
end for
return s
```

The first phase of carry-lookahead computes the propagate signal of the block, which is precisely a binary-AND over all $a_i \oplus b_i$ in the block. We compute the signal using parallel-prefix so that only $\lg(B)$ shifts are required. The second phase computes the carry output of each block in parallel. The computation is basically the same as the packed ciphertext ripple carry algorithm, but only ripples through `B = 4` bits. The third phase performs a carry look-ahead once for each block, recording either a propagated carry bit (`p * cIn'`) or a generated carry bit (`cB`). The final phase is almost identical to the second phase in the packed ripple carry adder, but begins with the correct carry inputs stored in `cIn`.

3.3.1 Performance

We present a performance benchmark of carry-lookahead addition of two 32-bit integers using packed ciphertexts.

	Packed	
	Time (s)	# Occurrences
`carryLookaheadAdd`	556.249	1
`reCrypt`	534.978	2
`operator*=`	107.192	859
`modDownToSet`	135.391	1069
`reLinearize`	148.652	420
`smartAutomorph`	150.222	424

Runtimes of packed ciphertext carry-lookahead addition evaluation using HElib parameters `m` = `31775` with 12 ciphertext primes and the `QE` scheme. The execution consumed 1.3 gigabytes of memory.

We observe about a three times performance increase in the carry-lookahead implementation compared to the bit-wise ripple carry implementation. These results are consistent with the smaller multiplication depth of the carry-lookahead circuit. In this benchmark, bootstrapping consumes approximately 95% of the total runtime (as a subroutine of `carryLookaheadAdd`).

3.4 Kogge-Stone Circuit

In the carry-lookahead algorithm, we used parallel prefix, or recursive doubling, to compute each block's propagate signal. The Kogge-Stone circuit takes this idea to the extreme by computing every bit's propagate and generate signals using parallel prefix. We present both bit-wise and packed ciphertext algorithms for evaluating the Kogge-Stone addition circuit.

3.4.1 Bit-Wise

The bit-wise implementation of Kogge-Stone maintains three vectors containing the propagate signal, generate signal, and final sum. A simplified version is presented below.

Procedure 6 Add two vectors of ciphertexts, `a` and `b`, with Kogge-Stone

Input: `a`, `b`, two vectors of ciphertexts encrypting bits of integers in increasing significance
Output: `s`, a vector of ciphertexts encoding `a + b`

```
p ← [],  g ← [], s ← []
for i = 0...31 do                          // Initialize propagate, generate and sum vectors
  p ← p + (a[i] + b[i])
  g ← g + (a[i] * b[i])
  s ← s + p[i]
end for
for lgShift = 0...lg(32) do                                               // Parallel prefix
  for i = 31...pow(2, lgShift) do
    p' ← p[i]
    p[i] ← p[i] * p[i - pow(2, lgShift)]
    g[i] ← g[i] || (p' * g[i - pow(2, lgShift)])
  end for
end for
for i = 1...31 do                                                      // Final sum update
  s[i] ← s[i] + g[i-1]
end for
return s
```

The parallel prefix for loop performs one iteration per power of two up to the bit length of the inputs. In each iteration, the propagate and generate signals are updated in decreasing significance, as positions of greater significance depend on the less significant but not vice-versa. We found the Kogge-Stone circuit simpler to implement than carry-lookahead as both propagate and generate can be computed in the same loop. While it is not shown in the algorithm pseudocode, we use the same technique as the authors in [GHS12] to bootstrap multiple bit-wise ciphertexts at the same time by packing all of them into a single ciphertext, bootstrapping that ciphertext, and then unpacking them.

3.4.2 Packed Ciphertext

We present an additional (simplified) Kogge-Stone implementation that uses a single packed ciphertext per integer instead of vectors of encrypted bits.

Procedure 7 Add two ciphertexts, `a` and `b`, with Kogge-Stone

Input: `a`, `b`, two packed ciphertexts encrypting bits of integers in increasing significance

Output: `s`, a ciphertext encoding `a + b`

```
p ← a + b,  g ← a * b
for lgShift = 0...lg(32) do                         // Parallel prefix
  p' ← p,  g' ← g
  shift(p', pow(2, lgShift))
  shift(g', pow(2, lgShift))
  g ← g || (p * g')
  p ← p * p'
end for
shift(g, 1)
s ← s + g
return s
```

This algorithm closely resembles the logic of the bit-wise implementation. By utilizing the SIMD nature of the packed ciphertexts, we have no need for an inner for loop. On the other hand, we now perform `2*lg(32)` shifts, increasing the noise growth of `p` and `g`.

3.4.3 Performance

Next we present the benchmark results for a single Kogge-Stone circuit evaluation in both bit-wise and packed form.

	Bit-Wise		Packed	
	Time (s)	# Occurrences	Time (s)	# Occurrences
`koggeStoneAdd`	376.631	1	294.766	1
`reCrypt`	276.674	1	278.164	1
`operator*=`	136.965	839	58.460	439
`modDownToSet`	153.881	1279	72.463	552
`reLinearize`	114.876	282	79.606	222
`smartAutomorph`	116.082	284	80.454	224

Comparison of runtimes between bit-wise and packed ciphertext encodings during a single Kogge-Stone addition evaluation using HElib parameters $\texttt{m} = \texttt{31775}$ with 12 ciphertext primes and the QE scheme.

We observe that the packed Kogge-Stone implementation yields the best performance of any of the presented implementations. We perform just one bootstrapping in both implementations, accomplishing this in the bit-wise implementation by first packing a single ciphertext with all ciphertexts requiring bootstrapping, bootstrapping that single ciphertext, and then unpacking. The memory footprint of the bit-wise version was 1.9 gigabytes, while the memory footprint of the packed version was 1.3 gigabytes.

3.4.4 QE Versus Vanilla HElib

Since the ciphertext-packed Kogge-Stone adder implementation proved to be fastest overall, we present a benchmark comparing the performance of the adder with QE as the underlying FHE scheme versus vanilla HElib.

	Vanilla HElib		`QE`	
	Time (s)	# Occurrences	Time (s)	# Occurrences
`koggeStoneAdd`	427.760	1	294.766	1
`reCrypt`	403.476	1	278.164	1
`operator*=`	101.418	439	58.460	439
`modDownToSet`	118.251	565	72.463	552
`reLinearize`	173.824	661	79.606	222
`smartAutomorph`	79.733	224	80.454	224

Comparison of runtimes between vanilla HElib and `QE` evaluating packed-ciphertext Kogge-Stone addition using HElib parameters `m` = `31775` with 12 ciphertext primes. Both executions consumed about 1.3GB of memory.

We observe that `QE` performs about 1.45 times faster than vanilla HElib, mostly due to increased performance in bootstrapping. We expected the speedup of `QE` to be closer to 2 times compared to vanilla HElib, but have observed that modulus switching and automorphism-caused key-switching become greater portions of the overall runtime in this parameter setting. Since `QE` does not affect the runtime of either of these, its performance impact is smaller for this Kogge-Stone circuit.

Chapter 4

Conclusion

4.1 Future Work

The presented SS scheme shows promise in terms of performance and a potentially more firmly-grounded basis for security than QE. That being said, we see value in further understanding and proving the security behind both of these schemes in relation to existing assumptions and hardness problems. We also hope that SS can be modified to support modulus switching.

Given that both SS and QE remove expensive key-switching from the multiplication of ciphertexts, we wonder whether new scheme variants can do the same for automorphisms of ciphertexts. Improving the efficiency of automorphisms would have a direct impact on the performance of integer addition, especially during bootstrapping.

We note that requiring evaluation of bootstrapping places a lower-bound on the security parameters that quickly turns memory into the hardware bottleneck. Developing a shallower or less memory-intensive bootstrapping evaluation or reducing the memory footprint of the existing implementation would allow larger parameters and less-frequent bootstrapping.

The choice of parameters in our addition circuits was carried out heuristically while developing the algorithms and was informed by the HElib authors' choices of parameters for AES evaluation. We see the need for the development of a framework that chooses an approximation of optimal parameters and bootstrapping locations given the definition of a circuit to evaluate.

As integer addition is just the beginning, we see the need for additional arithmetic primitives, such as rational numbers. With these primitives in place, higher-level applications such as machine-learning algorithms may be pursued.

4.2 Acknowledgements

I greatly appreciate having the opportunity to collaborate with Prof. Brakerski and Prof. Vaikuntanathan, from whom the ideas behind the SS and QE schemes originated. I would like to thank my parents for the consistent motivation and my colleague in crime Alex Grinman for always being there to bounce ideas or coffee off of. A large part of this work was made possible by the talented authors of HElib, Dr. Halevi and Prof. Shoup.

Bibliography

[B60] R.G. B. Simultaneous carry adder, December 27 1960. US Patent 2,966,305. 16

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 309–325. ACM, 2012. 6, 7, 9, 10, 11, 12, 13, 24

[BLMZ17] Fabrice Benhamouda, Tancrède Lepoint, Claire Mathieu, and Hang Zhou. Optimization of bootstrapping circuits. In *Symposium on Discrete Algorithms*, 2017. 18

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In *Annual Cryptology Conference*, pages 505–524. Springer, 2011. 6

[BV16] Zvika Brakerski and Vinod Vaikuntanathan. personal communication, 2016. 7, 19

[Gen09a] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig. 6

[Gen09b] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM. 14

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the aes circuit. Cryptology ePrint Archive, Report 2012/099, 2012. https://eprint.iacr.org/2012/099. 15, 17, 29, 44

[GKPV10] Shafi Goldwasser, Yael Tauman Kalai, Chris Peikert, and Vinod Vaikuntanathan. Robustness of the learning with errors assumption. 2010. 10, 22

[GPRR16] Arnab Deb Gupta, Yuriy Polyakov, Kurt Rohloff, and Gerard Ryan. Securely sharing encrypted medical information. In *2016 IEEE First International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*, pages 330–331. IEEE, 2016. 15

[HS13] Shai Halevi and Victor Shoup. Design and implementation of a homomorphic-encryption library. *IBM Research (Manuscript)*, 2013. 6, 12, 15, 17

[HS15] Shai Halevi and Victor Shoup. Bootstrapping for helib. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 641–670. Springer, 2015. 17

[IGI+15] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud. Cryptology ePrint Archive, Report 2015/898, 2015. http://eprint.iacr.org/2015/898. 6

[KS73] Peter M Kogge and Harold S Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE transactions on computers*, 100(8):786–793, 1973. 8, 17

[Lin99] Scott Lindhurst. An analysis of shanks's algorithm for computing square roots in finite fields. In *Number theory (Ottawa, 1996), CRM Proc. Lecture Notes*, volume 19, pages 231–242, 1999. 28

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for ring-lwe cryptography. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 35–54. Springer, 2013. 10

[RAD78] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978. 6

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):34, 2009. 10

[SV14] Nigel P Smart and Frederik Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71(1):57–81, 2014. 14