Practical Homomorphic Encryption Implementations & Applications

by

Leo de Castro

S.B., C.S M.I.T. 2018

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment for the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

February 2020

© 2020 Leo de Castro. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

Author: _____

Department of Electrical Engineering and Computer Science
January 28, 2020

Certified by: _____

Vinod Vaikuntanathan
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor
January 28, 2020

Accepted by: _____

Katrina LaCurts, Chair, Master of Engineering Thesis Committee

Practical Homomorphic Encryption Implementations & Applications

by

Leo de Castro

Submitted to the
Department of Electrical Engineering and Computer Science
on January 28, 2020
in partial fulfillment for the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Homomorphic encryption is an exciting technology that enables computations to be performed over encrypted data. While initial constructions were impractical, recent works have enabled efficiency necessary for many practical application. In this thesis, we present a new library for homomorphic encryption and two of applications built on this library. The first application is a fast oblivious linear evaluation protocol, a fundamental building block for secure computation. The second is a secure data aggregation platform used to study cyber risk.

Thesis Supervisor: Vinod Vaikuntanathan

Title: Associate Professor of Electrical Engineering and Computer Science

To my family.

Acknowledgments

This thesis would not be possible without the help of many people. I would first like to thank my advisor, Professor Vinod Vaikuntanathan, for introducing me to the field of cryptography and agreeing to work with me even when I knew next to nothing. His passion for the field, creativity in research, and openness to teaching others is a continuous inspiration. Next, I would like to thank Dr. Chiraag Juvekar for introducing me to applied homomorphic encryption and, in particular, the Gazelle library. We spent many late nights working on earlier versions of what is now in this thesis, and I look forward to working more with him in the future. I would also like to thank everyone in IPRI who helped with the CyberRisk@CSAIL consortium, especially Taylor Reynolds, Matthew Briggs, Nicolas Xuan-Yi Zhang, and Fransisca Susan. I had great fun working with you guys. Finally, and most importantly, I would like to thank my parents for their unwavering love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A fully homomorphic encryption (FHE) scheme, first proposed by Rivest et al. [RAD78], is an encryption scheme that allows computations to be performed on data while the data is encrypted. Applications for such a scheme are as fantastic as they are plentiful, from securely outsourcing computation to dramatically efficient secure multi-party computation (MPC). The first theoretical construction of a fully homomorphic encryption scheme was given in 2009 in the groundbreaking work of Gentry [Gen09a], although it lacked any practical efficiency and required the security of non-standard assumptions. Nevertheless, thanks to furious efforts over the past decade, constructions of homomorphic encryption schemes have steadily become more capable [CKKS17], secure [GSW13], and practical [BGV12].

In this thesis, we present a homomorphic encryption library that draws from this rich body of work to become truly practical. We demonstrate this practicality through two applications. The first application, and the main result of this thesis, is an oblivious linear evaluation protocol that is both highly efficient as well as secure based on standard assumptions. The second application is a platform to securely and efficiently aggregate and perform computations over private data.

1.1 Road Map

In chapter 2, we give general background useful throughout the rest of the thesis. In 3, we present the Gazelle 2.0 homomorphic encryption library. In chapter 4, we present our fast oblivious linear evaluation protocol. In chapter 5, we present a platform for securely aggregating data built on top of Gazelle.

Chapter 2

General Background

In this chapter, we introduce the several concepts relevant throughout the remainder of this thesis.

2.1 Notation

We will make frequent use of the following ring.

$$\mathcal{R} = \mathbb{Z}[x]/(x^n + 1) \tag{2.1}$$

For a modulus q, let $\mathcal{R}_q$ be $\mathcal{R}$ with all coefficients mod q.

$$\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$$

For a ring $\mathcal{R}_q$, we denote the action of uniformly sampling an element from the ring as $a \leftarrow \mathcal{R}_q$.

For an integer n, we will denote the set $\{0, 1, 2, \ldots, n-1\}$ as $[n]$. For integers $i \leq j$, we denote the range $\{i, i+1, \ldots, j-1\}$ as $[i : j]$.

We denote the rounding function $\lceil \cdot \rfloor : \mathbb{R} \to \mathbb{Z}$ that maps $x_r \in \mathbb{R}$ to the closest integer $x \in \mathbb{Z}$. We denote the flooring function $\lfloor \cdot \rfloor : \mathbb{R} \to \mathbb{Z}$ that maps $x_r \in \mathbb{R}$ to the closest integer $x \in \mathbb{Z}$ such that $x \leq x_r$.

For a positive integer b, we write the modular reduction operation $c \equiv a \mod b$ as $c = [a]_b$. .

The norm notation $||\cdot||$ refers to the ℓ_∞ norm, unless otherwise specified. For a polynomial a with n coefficients each mod q, we have the bound $||a|| \leq q$.

We specify the base-2 logarithm by log.

We say that a function $negl$ is negligible if for every constant $c > 1$ we have $negl(n) < 1/n^c$ for all sufficiently large n.

2.2 Ring Learning With Errors

The homomorphic encryption scheme used in our work is based off of the Ring Learning with Errors (RLWE) problem [LPR10], which is defined over polynomial rings. In our instantiation, we use the ring $\mathcal{R}_q = \mathbb{Z}_q/(x^n + 1)$, where n is a power of 2. We note that the polynomial $f(x) = x^n + 1$ when n is a power of 2 is the $(2n)^{\text{th}}$ cyclotomic polynomial.

We now give an informal definition of the RLWE problem.

Definition 2.2.1 (Decisional Ring Learning with Errors (informal) [LPR10])**.** *For a polynomial ring $\mathcal{R}_q$, let $a, u \leftarrow \mathcal{R}_q$. Let χ be an error distribution with support over $\mathcal{R}_q$, and let $s, e \leftarrow \chi$. The decisional RLWE problem states that the following two tuples are computationally indistinguishable:*

$$(a, as + e) \approx_c (a, u)$$

where all operations are performed over $\mathcal{R}_q$.

Throughout this thesis, we will only consider the RLWE problem over the ring $\mathcal{R}_q =$

$\mathbb{Z}_q/(x^n+1)$ for n a power of 2 and the error distribution χ as the discrete, zero-centered Gaussian distribution over $\mathcal{R}_q$.

2.3 Homomorphic Encryption

In this section, we give high level definitions for the algorithms comprising a homomorphic encryption scheme as well as necessary security definitions.

Definition 2.3.1 (Homomorphic Encryption)**.** *A homomorphic encryption scheme* $\mathcal{E} = (\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Eval}, \mathsf{Decrypt})$ *is a set of PPT algorithms defined as follows:*

- $\mathsf{KeyGen}(1^\lambda) \to (\mathsf{sk}, \mathsf{pk}, \mathsf{evk})$

  Given the security parameter λ*, outputs a key pair consisting of a public encryption key* pk*, a secret decryption key* sk*, and an evaluation key* evk*.*

- $\mathsf{Encrypt}(\mathsf{pk}, m) \to \mathsf{ct}$

  Given a message $m \in \mathcal{M}$ *and an encryption key* pk*, outputs a ciphertext* ct*.*

- $\mathsf{Eval}(\mathsf{evk}, f, \mathsf{ct}_1, \mathsf{ct}_2, \ldots, \mathsf{ct}_n) \to \mathsf{ct}'$

  Given the evaluation key, a description of a function $f\colon \mathcal{M}^n \to \mathcal{M}$*, and* n *ciphertexts encrypting messages* $m_1, \ldots, m_n$*, outputs the result ciphertext* ct' *encrypting* $m' = f(m_1, \ldots, m_n)$*.*

- $\mathsf{Decrypt}(\mathsf{sk}, \mathsf{ct}) = m$ *Given the secret decryption key and a ciphertext* ct *encrypting* m*, outputs* m*.*

When returning a ciphertext output by the Eval function, it is often desirable for this ciphertext to hide the function f that was used to produce it. This property of the scheme is called circuit privacy, which we formally define below.

Definition 2.3.2. *Circuit Privacy ([IP07] definition 7, [BDPMW16] definition 5.1) A homomorphic encryption scheme $\mathcal{E}$ is* circuit private *for functions f of depth $\ell \leq L = poly(\lambda)$ if there exists a PPT simulator algorithm* Sim *such that for all PPT distinguishing algorithms* D *the following holds:*

$$\Bigg| Pr\Big[\mathsf{D}\Big(\mathcal{E}.\mathsf{Eval}\big(\mathsf{evk}, f, \langle \mathsf{ct}_i \rangle\big), \langle \mathsf{ct}_i \rangle, \mathsf{sk}\Big) = 1\Big] - Pr\Big[\mathsf{D}\Big(\mathsf{Sim}\big(1^\ell, \mathsf{pk}, \mathsf{sk}, \mathsf{evk}, f(\langle m_i \rangle), \langle \mathsf{ct}_i \rangle\big), \langle \mathsf{ct}_i \rangle, \mathsf{sk}\Big) = 1\Big]\Bigg| \leq negl(\lambda)$$

where $\langle \mathsf{ct}_i \rangle$ denotes the input ciphertexts to the Eval *function $\mathsf{ct}_1, \ldots, \mathsf{ct}_n$ and $\langle m_i \rangle$ denotes the corresponding input messages to the function f (i.e.* $\mathsf{Decrypt}(\mathsf{sk}, \mathsf{ct}_i) = m_i$ *for all $1 \leq i \leq n$). Note that we assume each input ciphertext ct_i is accompanied by the bounds on the magnitude of its noise term*[1].

2.3.1 Homomorphic Encryption Instruction Set

We now describe the basic operations supported by the homomorphic encryption schemes we consider for the remainder of this thesis. The homomorphic encryption schemes that we consider are all "packed," which means that ciphertexts are able to encrypt vectors of values. The operations listed in this section are used to build a circuit to execute the Eval operation.

- $\mathsf{EvalAdd}(\mathsf{ct}_1, \mathsf{ct}_2) = \mathsf{ct}_+$

  Takes in two ciphertexts encrypting vectors m_1 and m_2 and produces a ciphertext encrypting $m_1 + m_2$, where the addition is component-wise. There is also $\mathsf{EvalAddPlain}$ for when one of the operands is unencrypted.

- $\mathsf{EvalMult}(\mathsf{ct}_1, \mathsf{ct}_2, \mathsf{rlk}) = \mathsf{ct}_\times$

  Takes in two ciphertexts encrypting vectors m_1 and m_2 and produces a ciphertext

[1] This is achieved in [BDPMW16] by simply setting all ct_i to be fresh encryptions of m_i.

encrypting $m_1 \otimes m_2$, where the multiplications is component-wise. This operation requires an evaluation key called a relinearization key. There is also EvalMultPlain for when one of the operands is unencrypted which does not require a relinearization key.

- $\mathsf{EvalAutomorphism}(\mathsf{ct}, i, \mathsf{rotket}) = \mathsf{ct}_{rot}$

  Since the available arithmetic operations are component-wise, this operation gives a method to permute the encrypted vector. This allows for operations to be performed on elements that are initially in different slots of a ciphertext. This operation requires an evaluation key called a rotation key.

With these operations, we are able to implement any arithmetic circuit, which is sufficient to implement the Eval function.

2.4 Security of an MPC Scheme

In this section, we give some definitions of the security of MPC schemes. These definitions will be mostly informal and they are presented with the intention of improving understanding.

2.4.1 Real-Ideal Model

We begin by describing a useful model for proving security of an MPC scheme. For a more formal definition, we refer the reader to [Can00]. Consider a protocol Π between parties $P_1, \ldots, P_k$ that that evaluates a function $f \colon \mathcal{X}_1 \times \ldots \times \mathcal{X}_k \to \mathcal{Y}_1 \times \ldots \times \mathcal{Y}_k$, where each party P_i inputs $x_i \in \mathcal{X}_i$ and receives the result $y_i \in \mathcal{Y}_i$. We can prove that Π is secure if it is indistinguishable from an "ideal" protocol that employs a trusted third party $\mathcal{T}$ to which all parties send their inputs over perfectly private channels. The party $\mathcal{T}$ then computes f and returns y_i to each party P_i over a perfectly private channel. If we can prove that Π is indistinguishable from this ideal protocol, then we can conclude that Π is secure in the

real-ideal model. Note that this model captures the freedom of parties to pick any input that they choose as well as removes from the security definition any leakage the function output may reveal about the parties' private inputs.

2.4.2 Honest-but-Curious Model

In the MPC protocols described in this thesis, we will only consider protocols that are secure in the "honest-but-curious" or "passive" security setting. This setting captures an adversary with limited capability. Namely, this adversary must follow the prescribed protocol exactly, but is able to use the information learned from the protocol to try and extract additional information beyond what could be learned in the ideal setting. For one of the many uses of protocols secure in this setting, see [IPS08]. This model is in contrast to the malicious security, in which the adversary is permitted to deviate from the protocol.

Chapter 3

GAZELLE 2.0: A Faster Homomorphic Encryption Library

In this chapter, we present the Gazelle homomorphic encryption library, focusing primarily on its optimizations and performance features. The first version of this library was written by Dr. Chiraag Juvekar [Juv18] with the goal of securely evaluating neural networks on encrypted data [JVC18]. In this next version, we maintain this goal while expanding the capability of the library to support a variety of advanced secure computation features.

As a general overview, Gazelle is a lattice cryptography library that implements the homomorphic encryption scheme of Brakerski, Fan, and Vercuteran [Bra12, FV12] (from hereon refered to as the BFV scheme) and the homomorphic encryption scheme of Cheon et al. [CKKS17] (from hereon refered to as the CKKS scheme). More specifically, the library implements the variants of these schemes that are optimized for the residue number system (RNS) representation [HPS19, CKK$^+$19] to allow all operations to use only standard machine words.

Using these schemes, the library includes implementations of several useful secure computation algorithms, including protocols for both diagonalized and hybrid secure matrix-vector

product [HS14, JVC18], oblivious linear evaluation (presented in chapter 4), and secure neural network layers for the evaluation of deep convolution neural networks on encryption data. In the remainder of this section, we will present the goals and justification for this library as well as the major design decisions made during its writing. We will also discuss some of the optimizations the library includes, focusing on the more rare and novel optimizations that most other HE libraries do not include.

3.1 Library Goals & Justification

Motivated by the potential applications of FHE, a rich area of applied cryptography research has developed around the implementation of various homomorphic encryption schemes and algorithms that use these schemes. This work primarily revolves around the development of a few main C++ libraries for homomorphic encryption; namely, HElib [HS] developed by IBM, SEAL [SEA19] developed by Microsoft Research, PALISADE [PAL] developed by Duality Technologies, and most recently the HEAAN [HEA] group of libraries developed primarily by researchers at Seoul National University. While this list is not exhaustive, it represents the major libraries that are under active development as new optimizations and techniques are researched. With this seemingly extensive selection of libraries that implement homomorphic encryption, it may not seem necessary to introduce yet another HE library.

However, when considering using homomorphic encryption for a performance intensive application, each of these libraries falls short in a significant way. They either sacrifice performance for the sake of usability, are designed in ways that make including new optimizations incredibly challenging, or simply lack the resource management infrastructure to scale to memory intensive applications. In contrast the Gazelle library was built with performance as the primary goal from the very beginning, allowing it to be uncompromising in this effort. In addition, the library is incredibly modular, allowing different optimizations and tech-

niques to mix-and-match together to easily create tailored solutions for various applications. The code uses the latest C++17 features to ensure resource management is automatically integrated into all applications using the library. The result is a feature-rich library that outperforms all of the aforementioned libraries in a variety of applications, even outperforming far more specialized libraries [ADI+17] for certain custom tasks.

3.2 NTT Engine

In essentially all homomorphic encryption schemes based on RingLWE (definition 2.2.1), the bottleneck operation is the number-theoretic transform for polynomial multiplication. This is essentially an analog of the fast Fourier transform over a field $\mathbb{Z}_q^*$. For a ring $\mathcal{R} = \mathbb{Z}_q[x]/(x^n + 1)$, nearly all operations that we perform over elements in the ring $\mathcal{R}$ take linear time $\Theta(n)$, but the runtime of the NTT is $\Theta(n \log(n))$. As such, the speed of the NTT operation limits the speed of any homomorphic encryption library.

With the goal of writing the fastest possible HE library, we began with the fastest NTT implementation. Our implementation is based on the NFLlib library [AMBG+16], which is a C library designed for an optimized NTT implementation. While the HE library built atop NFLlib is limited in scope, several implementation techniques from this library have been transfered over to Gazelle. A major source of speed for the library is compile-time optimizations, achievable through pre-selection of the prime modulus q and dimension n over the ring $\mathbb{Z}_q[x]/(x^n + 1)$ over which the NTT is performed. By dictating that these parameters be known at compile time, further optimizations are available when compared to all other known HE libraries that select their parameters at runtime. Gazelle contains a variety of parameter sets available to easily indicate which set an application should be used at compile-time as well as code-generation scripts to generate new parameter sets.

3.2.1 Newton Modulus Reduction

Since we are able to dictate the exact primes that are used, we are able to ensure that they satisfy additional constraints as well as precompute parameters for fast operations. One crucial operation that is accelerated by careful selection of primes is the modular reduction operation. In Gazelle, we make use of the Newton reduction algorithm ([AMBG+16], Algorithm 2), which requires an additional precomputed parameter that is included in the parameter sets in the library and automatically computed by the parameter generation scripts.

This reduction technique is independent of the input; it does not require an input-dependent pre-computed parameter, as in Barrett reduction [Bar87]. However, it also restricts the primes that we may use. For a machine word size s, let $\beta = 2^s$ (in most cases, $s = 64$). For a prime p, let $1 \leq s_0 \leq s - 1$ be an integer denoting the gap between the bitwidth of our prime p and the machine word size, i.e. $p < \beta/2^{s_0}$. The full Newton reduction constraint for the prime p is as follows ([AMBG+16] equation 1):

$$(1 + 1/2^{3s_0}) \cdot \beta/(2^{s_0} + 1) < p < \beta/2^{s_0} \tag{3.1}$$

When compared to the standard modulus reduction operation supported by the `clang++` compiler, Newton reduction provides a speedup of over 20%. As the bottle-neck operation of most arithmetic tasks in homomorphic encryption, this is a significant improvement.

3.3 RNS Representation

The next major design choice for this library was to use no multi-precision arithmetic outside of parameter computation. More specifically, we enforce that all moduli q that are larger than the machine word are represented as a product of moduli $\prod q_i = q$, where each q_i fit in a standard machine word.

3.3.1 Background

In leveled homomorphic encryption, one fixes an arithmetic circuit depth ℓ and then chooses parameters of the homomorphic scheme to support noise growth up to ℓ levels. This leads to a ciphertext modulus q that is often much larger than a standard machine word (typically 64 bits). In order to avoid expensive extended-precision arithmetic over these large integers, the ciphertext modulus q is represented as a product of primes each smaller than the machine word size. Let $q = \prod_{i=0}^{k-1} q_i$, where the q_i are all pair-wise coprime. By the Chinese Remainder Theorem, we can use the isomorphism between the ring $\mathbb{Z}_q$ and the tensor of rings mod each of the factors of q.

$$\mathbb{Z}_q \simeq \mathbb{Z}_{q_0} \otimes \mathbb{Z}_{q_1} \otimes \ldots \otimes \mathbb{Z}_{q_{k-1}}$$

This allows us to represent a number mod q as a vector of length k of integers that each fit nicely into a standard machine word. We refer to one of these elements of the vector as a *limb* of the integer. Since this map is an isomorphism, we can perform arithmetic in this representation as we would over the original ring $\mathbb{Z}_q$.

The structure of this isomorphism and it's effective use when implementing RLWE schemes is described in section 2.1 of [HPS19]. For an integer $x \in \mathbb{Z}_q$, let $x_i = [x]_{q_i}$ for all $i \in [k]$. Let $q_i^* = q/q_i$ and $\tilde{q}_i \equiv (q_i^*)^{-1} \mod q_i$. We make use of the following equation:

$$x = \Big(\sum_{i=0}^{k-1} [x_i \cdot \tilde{q}_i]_{q_i} \cdot q_i^* \Big) - v \cdot q = \Big[\sum_{i=0}^{k-1} [x_i \cdot \tilde{q}_i]_{q_i} \cdot q_i^* \Big]_q \tag{3.2}$$

for some $v \in \mathbb{Z}$.

3.3.2 BFV RNS Ciphertext Compression

In this section, we present an RNS operation in the BFV homomorphic encryption scheme for compressing a ciphertext with integers comprising of many limbs into a ciphertext that consists only of single-limb integers. This operation is useful when homomorphic operations on a ciphertext have concluded and the encrypted result is ready to be sent over a network, as the compressed form of the ciphertext saves network bandwidth. In addition, we show later in section 4.3 that careful selection of parameters results in the output of the operation having desirable *circuit privacy* properties.

Let $Q = \prod_{i=0}^{k-1} q_i$ be the ciphertext modulus, and let $q_0^* = Q/q_0$. At its core, this operation involves taking each ciphertext coefficient and dividing it by q_0^*. The naïve approach to this operation is to take an integer x in DCRT representation $\{x_0, \ldots, x_{k-1}\}$, recombine according to equation 3.2 above, then perform a multi-precision divide and floor by q_0^*, then take the result mod q_0. In total, this requires $2k$ integer multiplications, k integer additions, 1 multi-precision divide-and-floor, and $k+1$ modular reductions.

To avoid multi-precision arithmetic required when operating over elements modulo the full ciphertext modulus, we made use of the linearity of the DCRT recombination described in equation 3.2. From this equation, we have the following expression for division by q_0^*:

$$\left[\frac{x}{q_0^*}\right]_{q_0} = \left[\frac{1}{q_0^*}\Big(\Big(\sum_{i=0}^{k-1}[x_i \cdot \tilde{q}_i]_{q_i} \cdot q_i^*\Big) - v \cdot q\Big)\right]_{q_0} \tag{3.3}$$

$$= \left[\frac{1}{q_0^*}\Big(\sum_{i=0}^{k-1}[x_i \cdot \tilde{q}_i]_{q_i} \cdot q_i^*\Big)\right]_{q_0} \tag{3.4}$$

$$= \left[\sum_{i=0}^{k-1}[x_i \cdot \tilde{q}_i]_{q_i} \cdot \frac{q_0}{q_i}\right]_{q_0} \tag{3.5}$$

From equation 3.5 above, we have that only the terms $\tilde{q}_i$ and q_0/q_i for all $i \in [k]$ are needed to compute the desired term $\left[x/q_0^*\right]_{q_0}$. Since these values only depend on the choice of factors

of the ciphertext modulus, we can easily precompute them, which allows us to perform ciphertext compression with k integer multiplications, k floating point multiplications, k floating point additions, and $k+1$ modular reductions. Comparing with the naïve approach, this technique replaces the entire multi-precision divide-and-floor operation with only the marginal cost increase of k floating point addition and multiplication operations versus k integer addition and multiplication operations. We refer the reader to [HPS19] for further analysis of the benefits of the DCRT representations.

3.3.3 Avoiding Floating Point Operations

In section 3.3.2, we gave a generic algorithm for the DCRT rounding operation. However, in the case where the number of limbs is small, we can further constrain our primes to remove even the floating-point operations in this compression procedure. The following two lemmas are from [Juv18], which we state here.

Lemma 3.3.1 (Fast Two-Limb Compression ([Juv18] lemma 6.2.1))**.** *Given primes* $(q_0, q_1) = (2q_1 - 1, q_1)$ *and limbs* (x_0, x_1) *of a positive integer* $x < q_0 q_1$ *such that* $x_i = [x]_{q_i}$ *for* $i \in \{0, 1\}$*, the equation holds for* $-1 \leq B \leq 1$*:*

$$\frac{x}{q_1} \equiv 2(x_1 - x_0) + B \mod q_0$$

Lemma 3.3.2 (Fast Three-Limb Compression ([Juv18] lemma 6.2.2))**.** *Given primes* $(q_0, q_1, q_2) = (4q_2 - 3, 2q_2 - 1, q_2)$ *and limbs* (x_0, x_1, x_2) *of a positive integer* $x < q_0 q_1 q_2$ *such that* $x_i = [x]_{q_i}$ *for* $i \in \{0, 1, 2\}$*, the equation holds for* $-2 \leq B \leq 2$*:*

$$\frac{x}{q_1 q_2} \equiv (4 \cdot 3_{q_2}^{-1} - 2) \cdot x_2 - 4x_1 + (8 \cdot 3_{q_0}^{-1}) \cdot x_0 + B \mod q_0$$

Table 3.1 gives the relative improvements of this techniques over the ones used when the

primes are not picked according to the constraints in lemmas 3.3.1 and 3.3.1.

	Two Limbs	**Three Limbs**
Regular Reduction	458 μs	1592 μs
Fast Reduction	242 μs	752 μs
Percent Speedup	47.16%	52.76%

Table 3.1: Fast DCRT Compression vs. Regular DCRT Compression

The modularity of the Gazelle library allows us to switch to this fast DCRT reduction method with a simple, one-line change at the application level. This is a prime example of how the library's modularity allows for the incorporation of new optimizations with virtually no change to the application code itself.

Chapter 4

Fast Oblivious Linear Evaluation

4.1 Introduction

Oblivious linear evaluation (OLE) is a fundamental building block in many secure computation protocols. In an OLE protocol over the ring $\mathbb{Z}_p$, there are two parties, a sender S with values $\alpha, \beta \in \mathbb{Z}_p$ and a receiver R with a value $x \in \mathbb{Z}_p$. At the end of the protocol, R will learn the value $\gamma \in \mathbb{Z}_p$ where $\gamma = \alpha \cdot x + \beta$ while S will learn nothing. Vector OLE (VOLE) can be viewed as many OLE protocols running in parallel where the receiver has the same x in all the protocols. Batch OLE (BOLE) can be viewed as many OLE protocols running in parallel where the receiver has a vector $\mathbf{x}$ of values over $\mathbb{Z}_p^m$. In this work, we will primarily focus on VOLE with a simple extension to BOLE.

Our approach to implementing a VOLE protocol is to use the packed additively homomorphic encryption (PAHE) of Brakerski [Bra12] and Fan and Vercuteran [FV12], from hereon referred to as the BFV scheme. PAHE is a natural choice of primitive to implement such a protocol, especially in the honest-but-curious model. At a high level, our protocol has the receiver R encrypt the value x and send the encryption $[\![x]\!]$ to the sender S. Using the PAHE operations, the sender can then compute the ciphertext $[\![\boldsymbol{\gamma}]\!] = [\![\boldsymbol{\alpha} \cdot x + \boldsymbol{\beta}]\!]$ and return it

to the receiver, who can decrypt and learn the VOLE output $\boldsymbol{\gamma}$. As long as the PAHE scheme achieves *circuit privacy*, which, informally, says that a ciphertext does not leak information about the circuit used to compute it, then security is achieved against honest-but-curious adversaries.

4.1.1 Related Work

Recent related work on vector OLE has focused primarily on achieving impressive efficiency through nonstandard assumptions. Our main point of comparison is the work of Applebaum et al. [ADI+17], which implements a fast vector OLE protocol using linear codes. However, they require a nonstandard coding assumption for their security proof, and our main result is to achieve slightly better computational efficiency than this work using only standard assumptions.

It is also worth mentioning the recent beautiful work of Boyle et al. [BCGI18] that uses techniques from function secret sharing to generate random vector OLE correlations by communicating only function shares. Using standard reductions from random OLE to arbitrary OLE, this work achieves incredible efficiency; however, they require further coding assumptions not only for their security proof but also to avoid an exponential runtime. Constructing such a scheme from standard assumptions is a fascinating open question.

4.2 Background & Preliminaries

4.2.1 OLE Definition and Security Requirements

In this section, we give formal definitions of OLE, vector OLE, and batch OLE, along with security definitions.

Definition 4.2.1 (Oblivious Linear Evaluation)**.** *Given a ring $\mathcal{R}$, OLE is a protocol between*

two parties, a sender S and a receiver R. At the start of the protocol, S has two values $\alpha, \beta \in \mathbb{Z}_p$ and R has a value $x \in \mathbb{Z}_p$. At the end of the protocol, R receives the value $\gamma \in \mathbb{Z}_p$ where $\gamma = \alpha \cdot x + \beta$, and S receives no additional information.

Definition 4.2.2 (Vector Oblivious Linear Evaluation)**.** *Given a ring $\mathbb{Z}_p$ and a positive integer m, vector OLE is a protocol between two parties, a sender S and a receiver R. At the start of the protocol, S has two vectors $\boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathbb{Z}_p^m$ and R has a scalar $x \in \mathbb{Z}_p$. At the end of the protocol, R receives the vector $\boldsymbol{\gamma} \in \mathbb{Z}_p^m$ where $\boldsymbol{\gamma} = \boldsymbol{\alpha} \cdot x + \boldsymbol{\beta}$, where the arithmetic operations are performed component-wise over $\mathbb{Z}_p$, and S receives no additional information.*

Definition 4.2.3 (Batch Oblivious Linear Evaluation)**.** *Given a finite field $\mathbb{Z}_p$ and a positive integer m, batch OLE is a protocol between two parties, a sender S and a receiver R. At the start of the protocol, S has two vectors $\boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathbb{Z}_p^m$ and R has a a vector $\mathbf{x} \in \mathbb{Z}_p^m$. At the end of the protocol, R receives the vector $\boldsymbol{\gamma} = \boldsymbol{\alpha} \cdot \mathbf{x} + \boldsymbol{\beta}$, where the arithmetic operations are performed component-wise over $\mathbb{Z}_p$, and S receives no additional information.*

In both VOLE and BOLE, we will refer to the dimension m as the "length" or "batch size" of the protocol.

We now give security definitions for OLE. These definitions extend naturally to VOLE and BOLE.

Definition 4.2.4 (Sender View)**.** *For an OLE protocol Π, public parameters pp, and values $\alpha, \beta \in \mathbb{Z}_p$, let $\mathsf{View}_\Pi(S(pp, \alpha, \beta))$ be the view of S during Π, which contains all messages generated and received by S as well as all random bits sampled by S.*

Definition 4.2.5 (Receiver View)**.** *For an OLE protocol Π, public parameters pp, and vector $x \in \mathbb{Z}_p$, let $\mathsf{View}_\Pi(R(pp, x))$ be the view of R during Π, which contains all messages generated and received by R (including the output of the protocol) as well as all random bits sampled by R.*

We define passive security with respect to each party with simulation-based definitions.

Definition 4.2.6 (Security against Sender)**.** *We say that an OLE protocol* Π *is computationally secure against semi-honest senders* S *if there exists a PPT algorithm* Sim *such that for all PPT distinguishing algorithms* D *we have that*

$$\left|Pr[\mathsf{D}(\mathsf{View}_{\Pi}(S(pp,\alpha,\beta))) = 1] - Pr[\mathsf{D}(\mathsf{Sim}(pp,\alpha,\beta) = 1]\right| \leq negl(\lambda)$$

Definition 4.2.7 (Security against Receiver)**.** *We say that an OLE protocol* Π *is computationally secure against semi-honest receivers* R *if there exists a PPT algorithm* Sim *such that for all PPT distinguishing algorithms* D *we have that*

$$\left|Pr[\mathsf{D}(\mathsf{View}_{\Pi}(R(pp,x))) = 1] - Pr[\mathsf{D}(\mathsf{Sim}(pp,x,\gamma) = 1]\right| \leq negl(\lambda)$$

where γ *is the output that is received by* R *at the end of* Π*.*

Note in definitions 4.2.6 and 4.2.7 the View variables we consider are only those of a valid OLE protocol. We do not consider View variables resulting from either party deviating from the protocol. In this way, we only consider passively-secure OLE protocols.

Satisfying definition 2.3.2 above will be necessary to satisfy defintion 4.2.7.

4.2.2 Flooring Error

It will be useful in our analysis below to define the error introduced by the divide-and-floor operation as an exact function of the operands. Given two positive integers a and b, the following identity holds:

$$\left\lfloor \frac{a}{b} \right\rfloor = \frac{a - [a]_b}{b} = \frac{a}{b} - \frac{[a]_b}{b}$$

It will also be useful to upper bound the probability that an error term e added to a has any effect on the term $\left\lfloor \frac{a+e}{b} \right\rfloor$. This can be represented as the probability that $\left\lfloor \frac{a+e}{b} \right\rfloor \neq \left\lfloor \frac{a}{b} \right\rfloor$.

In particular, we are interested in the case where a is a uniformly random element of some range $[k \cdot b]$ for an integer $k > 1$, meaning that $[a]_b$ is uniformly random over $[b]$.

The following lemma will be very useful in our results below.

Lemma 4.2.1 (Flooring Error)**.** *Let $\alpha \in [k \cdot b]$ be a uniformly sampled element from the range $[k \cdot b]$, and let e be a small error term such that $|e| < \lfloor b/2 \rfloor$ with overwhelming probability. We have the following upper bound:*

$$Pr\Big[\Big\lfloor \frac{\alpha + e}{b} \Big\rfloor \neq \Big\lfloor \frac{\alpha}{b} \Big\rfloor\Big] \leq \frac{|e|}{b}$$

where the probability is taken over the choice of α given e.

Proof. We can rewrite α as $\alpha = d \cdot b + r$ for non-negative integers d and r, where $r < b$. We have that $\lfloor \frac{\alpha}{b} \rfloor = d$. In order for $\lfloor \frac{\alpha+e}{b} \rfloor \neq d$, we must have that either $r + e \geq b$ or $r + e < 0$. Since $0 \leq r < b$, exactly one of these cases is possible, with the first case corresponding to $e \geq 0$ and the second corresponding to $e < 0$. In both cases, there is a range of size $|e|$ into which r must fall in order for $\lfloor \frac{\alpha+e}{b} \rfloor \neq d$. This range is $[b - e, b - 1]$ in the first case and $[0, -e - 1]$ in the second case. From the discussion above, we have that $r = [a]_b$ is uniformly random over the range $[b]$, so the probability that r falls into the problematic range in either case is $\frac{|e|}{b}$. □

4.2.3 Ring Expansion Factor

In order to effectively choose parameters for our leveled homomorphic encryption scheme, we must accurately upper bound the noise growth due to homomorphic operations. When multiplying two elements of $\mathcal{R}_q$, we need an upper bound on the norm of the product by a function of the norms of the operands as well as properties of $\mathcal{R}_q$ itself. We use the definition

of the ring expansion factor from [LM06, Gen09b] as follows:

$$\gamma_{\mathcal{R}} = \max_{a,b \in \mathcal{R}} \frac{||a \cdot b||}{||a|| \cdot ||b||} \tag{4.1}$$

We can upper bound $\gamma_{\mathcal{R}}$ by n, for $\mathcal{R} = \mathbb{Z}[x]/(x^n + 1)$. Therefore, for any elements $a, b \in \mathcal{R}$, we can upper bound the norm of the product $c = a \cdot b$ over $\mathcal{R}$ by $||c|| \leq n \cdot ||a|| \cdot ||b||$.

4.2.4 BFV Homomorphic Encryption Scheme

In this section , we describe the algorithms that define the Brakerski-Fan-Vercuteran ([Bra12], [FV12]) homomorphic encryption scheme based on the Ring Learning with Errors (RLWE) problem [LPR10]. While this scheme is fully homomorphic, we will only be using encryption and decryption, the plaintext addition, and plaintext multiplication functions for our VOLE protocol.

For an integer q and n a power of two, we define the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n+1)$. Let χ be the error distribution of the RLWE problem (typically a discrete, zero-centered Gaussian mod q), and let p be an integer much smaller than q. Let $\Delta = \lfloor q/p \rfloor$.

First, let's define the algorithms for public key and secret key encryption of the BFV scheme.

- $\mathsf{KeyGen}(1^\lambda) \to (\mathsf{sk}, \mathsf{pk})$.

  Outputs the secret key sk and public key pk. The secret key $\mathsf{sk} = s$ is generated by sampling from the error distribution $s \leftarrow \chi$. The public key is generated by first sampling a uniformly random element $a \leftarrow \mathcal{R}_q$ and an error term $e \leftarrow \chi$. We then set $\mathsf{pk} = (a, a \cdot s + e)$.

- $\mathsf{Encrypt}_{q,\chi}(\mathsf{sk}, m) \to \mathsf{ct}$.

  For an error distribution χ, outputs a ciphertext encrypting the message $m \in \mathcal{R}_p$.

Samples a uniformly random element $a \leftarrow \mathcal{R}_q$ and an error term $e \leftarrow \chi$ and outputs the tuple $\mathsf{ct} = (a, a \cdot s + \Delta m + e)$.

- $\mathsf{Encrypt}_{q,\chi}(\mathsf{pk}, m) \rightarrow \mathsf{ct}$.

  For an error distribution χ, outputs a ciphertext encrypting the message $m \in \mathcal{R}_p$. For a public key $\mathsf{pk} = (\mathsf{pk}[0], \mathsf{pk}[1])$, this algorithm samples three error terms $u, e_1, e_2 \leftarrow \chi$. It then outputs the ciphertext

$$\mathsf{ct} = (\mathsf{pk}[0] \cdot u + e_1, \mathsf{pk}[1] \cdot u + \Delta \cdot m + e_2)$$

- $\mathsf{Decrypt}(\mathsf{sk}, \mathsf{ct}) = m$. Outputs the message m that the ciphertext $\mathsf{ct} = (\mathsf{ct}[0], \mathsf{ct}[1])$ encrypts. Computes and outputs the following:

$$m = \left\lceil \frac{\mathsf{ct}[1] - s \cdot \mathsf{ct}[0]}{\Delta} \right\rfloor$$

From the structure of the ciphertext, the algorithms for addition and plaintext multiplication follow naturally. Note that these are specific instantiations of the algorithms listed in 2.3.1.

- $\mathsf{EvalAdd}(\mathsf{ct}_1, \mathsf{ct}_2) = \mathsf{ct}_3$.

  For $\mathsf{ct}_1 = (\mathsf{ct}_1[0], \mathsf{ct}_1[1])$ and $\mathsf{ct}_2 = (\mathsf{ct}_2[0], \mathsf{ct}_2[1])$ that encrypt m_1 and m_2, the ciphertext ct_3 encrypts $m_1 + m_2$, where addition is over $\mathcal{R}_p$. The ciphertext

  $\mathsf{ct}_3 = (\mathsf{ct}_1[0] + \mathsf{ct}_2[0], \mathsf{ct}_1[1] + \mathsf{ct}_2[1])$, where all operations are over $\mathcal{R}_q$.

- $\mathsf{EvalAddPlain}(\mathsf{ct}_1, m_2) = \mathsf{ct}_3$.

  For $\mathsf{ct}_1 = (\mathsf{ct}_1[0], \mathsf{ct}_1[1])$ encrypting the message $m_1 \in \mathcal{R}_p$ and $m_2 \in \mathcal{R}_p$, the ciphertext ct_3 encrypts the message $m_1 + m_2$, where addition is over $\mathcal{R}_p$. The ciphertext

  $\mathsf{ct}_3 = (\mathsf{ct}_1[0], \mathsf{ct}_1[1] + \Delta m_2)$, where all operations are over $\mathcal{R}_q$.

- $\mathsf{EvalMultPlain}(\mathsf{ct}_1, m_2) = \mathsf{ct}_3$.

  For $\mathsf{ct}_1 = (\mathsf{ct}_1[0], \mathsf{ct}_1[1])$ encrypting the message $m_1 \in \mathcal{R}_p$ and $m_2 \in \mathcal{R}_p$, the ciphertext ct_3 encrypts the message $m_1 \cdot m_2$, where multiplication is over $\mathcal{R}_p$. The ciphertext $\mathsf{ct}_3 = (\mathsf{ct}_1[0] \cdot m_2, \mathsf{ct}_1[1] \cdot m_2)$, where all operations are over $\mathcal{R}_q$.

Note that the $\mathsf{EvalMultPlain}$ function described above does not, on its own, achieve circuit privacy as defined in definition 2.3.2.

Encoding Inputs as Polynomials

The homomorphic encryption scheme described above encrypts messages over the polynomial ring $\mathcal{R}_p$. In order to operate on encrypted vectors and perform component-wise operations, we encode the vectors as polynomials. A vector x of length n is encoded as a polynomial in $\mathcal{R}_p$ by finding a polynomial m such that evaluation of m at n roots of unity is x. This results in polynomial multiplication on m_1 and m_2 mapping to component-wise multiplication over the evaluations x_1 and x_2.

To encode scalar values, we note that it suffices to treat a scalar value $x \in \mathbb{Z}_p$ as an element of $\mathcal{R}_p$, since the evaluation of this element in $\mathcal{R}_p$ at any input will be x. This naturally distributes the scalar to all elements of the other encoded operand in the homomorphic operations defined above. Because of this, our VOLE and BOLE protocols differ only in these encoding and decoding steps.

4.2.5 OLE Leakage

Let's consider a naïve application of the homomorphic operations described in section 4.2.4 to implement an VOLE protocol. We will show in this section that this results in leakage of the sender's private values.

In the ideal world [Can00], all that is revealed to the receiver is $\boldsymbol{\gamma} = \boldsymbol{\alpha} \cdot x + \boldsymbol{\beta}$, which

effectively hides the sender's inputs $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$. However, we see that if the sender receives a ciphertext of the form $(a, as + \Delta x + e)$ and then performs a single EvalMultPlain and EvalAddPlain as defined in section 4.2.4, the receiver will receive a ciphertext that has the following form:

$$\big(a \cdot \boldsymbol{\alpha}, \quad a \cdot s \cdot \boldsymbol{\alpha} + \Delta \cdot (x \cdot \boldsymbol{\alpha} + \boldsymbol{\beta}) + e \cdot \boldsymbol{\alpha}\big) \tag{4.2}$$

It is clear from equation 4.2 above that the vector $\boldsymbol{\alpha}$ is easily recoverable from either the first term by factoring out the a polynomial or the error term by factoring out the original error term e.

If the receiver knows $\boldsymbol{\alpha}$, the OLE output also leaks $\boldsymbol{\beta}$, so the sender's privacy is completely lost. In sections 4.3 and 4.4.1 below, we discuss our approach to achieving leakage resilience by removing the dependence of $\boldsymbol{\alpha}$ from both terms of the ciphertext, achieving the condition in definition 2.3.2.

4.3 Circuit Privacy from Ciphertext Compression

Let Q be the original ciphertext modulus of our scheme, and let $\{q_i\}_{i=0}^{\ell-1}$ be the primes such that $\prod_{i=0}^{\ell-1} q_i = Q$. Given a ciphertext ct_Q, where the elements are in $\mathcal{R}_Q$ and the decryption operations are performed over $\mathcal{R}_Q$, the goal of this operation is to obtain a ciphertext ct_{q_0} that encrypts the same message. This new ciphertext ct_{q_0} will have elements in $\mathcal{R}_{q_0}$ and the decryption of this ciphertext will be performed over $\mathcal{R}_{q_0}$ as well. Recall that we choose this primes q_i to fit in a standard machine word, so decryption of ct_{q_0} is far more efficient than the decryption of ct_Q. In addition, the communication cost of sending ct_{q_0} over a network is significantly less than sending ct_Q. Note that this operation can only be correct if the plaintext modulus p is sufficiently less than q_0.

Let $q_0^* = Q/q_0$. Below, we write the ciphertext ct_Q and expand out the terms to anticipate

the division by q_0^*.

$$\begin{aligned}
\mathsf{ct}_Q &= \Big(a,\ a\cdot s + \Big\lfloor \frac{Q}{p} \Big\rfloor m + e\Big) \\
&= \Big(a' \cdot q_0^* + [a]_{q_0^*},\ (a' \cdot q_0^* + [a]_{q_0^*}) \cdot s + \Big\lfloor \frac{Q}{p} \Big\rfloor m + e\Big) \\
&= \left(a' \cdot q_0^* + [a]_{q_0^*},\ (a' \cdot q_0^* + [a]_{q_0^*}) \cdot s + \Big(\frac{Q}{p} - \frac{[Q]_p}{p}\Big) m + e\right)
\end{aligned}$$

The ciphertext ct_{q_0} is obtained by dividing the two components of ct_Q by q_0^*.

$$\mathsf{ct}_{q_0} = \left\lfloor \frac{\mathsf{ct}_Q}{q_0^*} \right\rceil = \Big(\left\lfloor \frac{a}{q_0^*} \right\rceil,\ \left\lfloor \frac{a\cdot s + \left\lfloor \frac{Q}{p} \right\rfloor m + e}{q_0^*} \right\rceil \Big) \tag{4.3}$$

$$= \Big(\left\lfloor \frac{a' \cdot q_0^* + [a]_{q_0^*}}{q_0^*} \right\rceil, \left\lfloor \frac{(a' \cdot q_0^* + [a]_{q_0^*}) \cdot s + \left(\frac{Q}{p} - \frac{[Q]_p}{p}\right) m + e}{q_0^*} \right\rceil \Big) \tag{4.4}$$

$$= \Big(a',\ a' \cdot s + \left\lfloor \frac{[a]_{q_0^*} \cdot s + \left(\frac{Q}{p} - \frac{[Q]_p}{p}\right) m + e}{q_0^*} \right\rceil \Big) \tag{4.5}$$

$$= \Big(a',\ a' \cdot s + \frac{q_0}{p} m + \left\lfloor \frac{[a]_{q_0^*} \cdot s - \frac{[Q]_p}{p} m + e}{q_0^*} \right\rceil \Big) \tag{4.6}$$

$$= \Big(a',\ a' \cdot s + \Big\lfloor \frac{q_0}{p} \Big\rfloor m + \frac{[q_0]_p}{p} m + \left\lfloor \frac{[a]_{q_0^*} \cdot s - \frac{[Q]_p}{p} m + e}{q_0^*} \right\rceil \Big) \tag{4.7}$$

Note that we made the substitution $\frac{q_0}{p} = \left\lfloor \frac{q_0}{p} \right\rfloor + \frac{[q_0]_p}{p}$ in the last line because we divide by $\left\lfloor \frac{q_0}{p} \right\rfloor$ during decryption.

The new error term in ct_{q_0} is

$$e' = \frac{[q_0]_p}{p} m + \left\lfloor \frac{[a]_{q_0^*} \cdot s - \frac{[Q]_p}{p} m + e}{q_0^*} \right\rceil = \frac{[q_0]_p}{p} m + v \tag{4.8}$$

Lemma 4.3.1 (Compressed Error Independence)**.** *The compressed error e' is independent of the original error term e with probability $1 - \frac{|e|}{q_0^*}$.*

Proof. This follows directly from an application of lemma 4.2.1, which is valid since the term v from equation 4.8 can be viewed as a uniformly random element plus a small error term. As long as $|e| < q_0^*/2$, probability that e has any effect on the resulting term is at most $\frac{|e|}{q_0^*}$. □

We will now argue that this compression technique can be used to achieve circuit privacy from definition 2.3.2. At a high level, the homomorphic Eval function will first homomorphically evaluate f on the encrypted inputs, then randomize the a component of the ciphertext using the public key. The resulting ciphertext can then be compressed, and this compressed result will be output by the Eval function. As long as the error magnitude of the pre-compressed ciphertext is not too great, which is a function of the error levels of the input ciphertexts and the depth of f, then by lemma 4.3.1 this final ciphertext is simulatable.

Lemma 4.3.2 (Re-randomization using Encryption of Zero)**.** *Given a ciphertext* $\mathsf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1)$ *and public key* $\mathsf{pk} = (a, a \cdot s + e)$*, let* $\mathsf{ct}^{(0)} \leftarrow \mathsf{Encrypt}(\mathsf{pk}, 0)$ *the ciphertext* $\mathsf{ct}' = \mathsf{ct} + \mathsf{ct}^{(0)} = (\mathsf{ct}'_0, \mathsf{ct}'_1)$ *has a term* ct'_0 *that is indistinguishable from a uniformly random element over* $\mathcal{R}_Q$*, even given* $\mathsf{sk} = s$*.*

Proof. This follows from the RLWE assumption, since the structure of $\mathsf{ct}_0^{(0)} = a \cdot u + e'$, where u and e' are sampled from the RLWE noise distribution. Even knowing a, this term is indistinguishable from uniform over $\mathcal{R}_Q$, so $\mathsf{ct}'_0 = \mathsf{ct}_0 + \mathsf{ct}_0^{(0)}$ is also indistinguishable from a random element of $\mathcal{R}_Q$. □

Lemma 4.3.3 (Circuit Privacy from Compression)**.** *For the BFV homomorphic* Eval *algorithm over a ring* $\mathcal{R}_Q$ *where* $Q = \prod_i q_i$*, consider the following modified homomorphic evaluation algorithm* Eval_{cp} *that first runs* Eval*, then adds a fresh encryption of zero, then outputs the compressed result. If the error term of the ciphertext output by* Eval *has magnitude bound* B*, then* Eval_{cp} *is circuit private (w.r.t. defintion 2.3.2) with probability* $1 - \frac{B}{q_0^*}$*, where* $q_0^* = Q/q_0$*.*

Proof. In order to show that Eval_{cp} is circuit private, we will construct a simulator that produces a ciphertext indistinguishable from the output of Eval_{cp} with probability $1 - \frac{B}{q_0^*}$. Recall that the simulator in definition 2.3.2 has the form

$$\mathsf{Sim}\big(1^\ell, \mathsf{pk}, \mathsf{sk}, \mathsf{evk}, \gamma = f(\langle m_i \rangle), \langle \mathsf{ct}_i \rangle\big)$$

The ciphertext output by Eval_{cp} will have the following form, where $\gamma = f(\langle m_i \rangle)$ is the result of the corresponding plaintext computation:

$$\mathsf{ct}_E = \left(a',\ a' \cdot s + \left\lfloor \frac{q_0}{p} \right\rceil \gamma + \frac{[q_0]_p}{p}\gamma + \left\lfloor \frac{[a]_{q_0^*} \cdot s - \frac{[Q]_p}{p}\gamma + e}{q_0^*} \right\rceil \right) \in \mathcal{R}_{q_0}^2$$

which is from equation 4.7 above. We first note that even when given the secret key the term a is computationally indistinguishable from a random element of $\mathcal{R}_Q$, since this term was randomized with a public key encryption of zero prior to compression. This means that it suffices for our simulator to output the compression of a ciphertext encrypting γ where the a term is freshly sampled from $\mathcal{R}_Q$. Our simulator outputs a ciphertext of the following form:

$$\mathsf{ct}_S = \left(a',\ a' \cdot s + \left\lfloor \frac{q_0}{p} \right\rceil \gamma + \frac{[q_0]_p}{p}\gamma + \left\lfloor \frac{[a]_{q_0^*} \cdot s - \frac{[Q]_p}{p}\gamma}{q_0^*} \right\rceil \right) \in \mathcal{R}_{q_0}^2$$

where $a' = \lfloor a/q_0^* \rceil$ and a is uniformly sampled over $\mathcal{R}_Q$. The a used in the compression of ct_S and the a used in the compression of ct_E are computationally indistinguishable by the security of RLWE. Therefore, the only remaining difference between ct_S and ct_E are the error terms. Since $[a]_{q_0^*} \cdot s$ is uniformly random (recall that this a term has been randomized by adding an encryption of zero), then we can invoke lemma 4.2.1 to get the following:

$$Pr\left[\left\lfloor \frac{[a]_{q_0^*} \cdot s - \frac{[Q]_p}{p}\gamma + e}{q_0^*} \right\rceil = \left\lfloor \frac{[a]_{q_0^*} \cdot s - \frac{[Q]_p}{p}\gamma}{q_0^*} \right\rceil\right] \geq 1 - \frac{|e|}{q_0^*}$$

Bounding the magnitude of e with B, we have the statement in the lemma. □

In section 3.3.2, we give several methods for efficiently performing this operation.

4.4 A Fast VOLE Protocol

In this section, we present our VOLE protocol. This VOLE protocol executes a vector OLE protocol with length n, where n is the parameter in the ring $\mathcal{R}$. Below, the steps of our VOLE protocol are given.

Protocol 1 Vector Oblivious Linear Evaluation

Roles: Sender S and Receiver R.

Inputs: S inputs $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, R inputs x.

Outputs: S gets nothing. R gets $\boldsymbol{\alpha} \cdot x + \boldsymbol{\beta}$.

Protocol:

1. **One-time Setup**
   (a) R generates a BFV public key pk.
   (b) R sends pk to S.
2. **Per VOLE Setup**
   (a) S uses pk to encrypt zero to produce ct_0.
3. **Online.**
   (a) R receives x, S receives $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$.
   (b) R encrypts x to obtain ct_x, S encodes $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$.
   (c) R sends ct_x to S.
   (d) S multiplies ct_x by $\boldsymbol{\alpha}$ and adds $\boldsymbol{\beta}$ and ct_0
   (e) S compresses the result ciphertext.
   (f) S returns the final ciphertext to R.
   (g) R decrypts the final ciphertext to obtain the plaintext result.

4.4.1 Security

In this section, we argue that protocol 1 satisfies the security definitions given in section 4.2.1.

Lemma 4.4.1 (Secure Against Sender)**.** *Protocol 1 satisfies definition 4.2.6.*

Proof. This proof follows directly from the semantic security of the encryption scheme. Since the view of the sender consists entirely of messages that are computationally indistinguishable from uniformly random elements of $\mathcal{R}_Q$, the simulator Sim can easily generate a transcript that is computationally indistinguishable from the real transcript by simply sampling uniform elements of $\mathcal{R}_Q$ for each element of $\mathcal{R}_Q$ the sender receives. The compression operation can be performed entirely with public parameters, which completes the transcript. □

Lemma 4.4.2 (Secure Against Receiver)**.** *Protocol 1 satisfies definition 4.2.7.*

Proof. This proof follows from lemma 4.3.3. The only information about the sender's $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ values that the receiver can learn must come from the result ciphertext. By lemma 4.3.3, this result ciphertext is indistinguishable from a ciphertext generated without access to $\boldsymbol{\alpha}$ or $\boldsymbol{\beta}$, so no computationally bonded receiver can extract information about $\boldsymbol{\alpha}$ or $\boldsymbol{\beta}$ from the resulting ciphertext. Note that this necessarily requires that $\frac{B'}{q_0^*} \leq 2^{-\lambda}$, for security parameter λ, where B' is the upper bound on the error term prior to compression. Using equation 4.1, we can bound the magnitude of this error term by npB, where n is the ring dimension, p is the plaintext modulus, and B is the magnitude bound on the original error term. □

4.4.2 Simple Extension to Batched OLE

The extension of protocol 1 from a VOLE protocol to a batch-OLE protocol is straightforward and preserves the security proofs of the previous section. Only the receiver's role is

modified in this extension. The receiver begins with a vector $\mathbf{x}$ of values, which she encodes as a polynomial as described in section 4.2.4. The operations performed by the sender are identical. To obtain the BOLE output, the receiver must decode the decryption result by evaluating the resulting polynomial at the roots of unity determined by the encoding NTT parameters.

4.5 Performance

4.5.1 Parameter Selection

In both of our OLE protocols, there are three parameters that we must select: the plaintext modulus p, the ciphertext modulus q, and the ring dimension n. These parameter sets must satisfy both the computational security requirements of the RLWE problem as well as the statistical security requirements of the circuit-privacy compression operation. In the VOLE protocol, there are no restrictions on the plaintext modulus p other than its size, so we opt for powers of two for ease of comparison. In the BOLE protocol, the plaintext modulus must support the NTT operation to encode the vectors as polynomials for component-wise operations. This requires that $\mathbb{Z}_p$ contain a $2n^{\text{th}}$ root of unity.

In table 4.1 below, we give the parameter settings for the benchmarks below. Each parameter setting is given an ID, and the benchmarks associated with a given ID were generated using the parameter setting of the same ID. These settings are focused on the sizes of the parameters and the security parameters that result from these sizes. All of the limbs in the ciphertext moduli are 55 bits.

The parameters were selected to all have at least 128 bits of computation security based on the homomorphic encryption security standard [ACC$^+$18]. The statistical privacy parameter refers to the sender's privacy due to the compression operation. This is computed

by taking - $\log\left(\frac{B}{q_0^*}\right)$, where B is the upper bound on the ℓ_∞ norm of the noise just before the compression operation is performed. For the discrete Gaussian sampler for our noise terms, we used a standard deviation of $\sigma = 3.2$. We employ the discrete Gaussian sampling method of [DG14], which means we can upper bound the magnitude of the output of this distribution by $20\sigma < 64$, or 6 bits. We can then get an upper bound on $\log(B)$ by adding $\log(n) + \log(p)$, as per the expansion factor described in section 4.2.3. This gives us the following formula for the statistical privacy parameter.

$$\log(q_0^*) - \log(B) = \log(q_0) - \log(n) - \log(p) - 6$$

ID	$\log(p)$	$\log(q)$	$\log(n)$	Statistical Privacy Parameter
1	8	165	13	83
2	16	220	13	130
3	20	220	13	126
4	24	220	13	122
5	28	220	13	118
6	32	220	13	114
7	40	220	13	106

Table 4.1: Parameter set sizes for OLE benchmarks.

In the tables below listing the results, we group parameters together with essentially identical performance. This occurs for a variety of reasons, but it is often mainly due to all data-types being identical in the benchmarks of the parameter sets. One set of times is reported for these joined rows.

4.5.2 Experimental Setup

We benchmark our protocols using AWS machines with 16 virtual CPUs running at 3.1GHz and 32GB of RAM. In order to give the most versatile benchmarks, we do not give bench-

marks that ran over the Internet. Instead, we give benchmarks of protocols that ran locally between two threads that are connected via `localhost`. These benchmarks along with the communication complexity of the protocol modes given in table 4.6 should allow for the calculation of the performance of the OLE protocols in a variety of network settings. Our experiments show that the round-trip time of a `localhost` connection is less than 60 μs, which we consider as negligible in the the computation time benchmarks given below.

4.5.3 Micro Benchmarks

In this subsection, we give micro-benchmarks for the VOLE protocol as well as the BOLE variant. We note that only the receiver's actions change in the BOLE variants, so only the benchmarks for the receiver's role is given for BOLE.

Furthermore, both protocols have two modes of operation: a compute-optimized mode and a communication optimized mode. These two modes affect both the sender and the receiver, and we give benchmarks for both modes. In the communication-optimized mode, the random a elements in the ciphertexts are not sent over the network. Instead, we make use of a standard technique, which is to generate the a values by sampling a small, random PRG seed and then send the seed used to generate the a term rather than the a term itself. This reduces our communication cost by nearly a factor of two, but running the PRG to generate the a term is more computationally expensive than sampling the a term directly, and the PRG generation must also be done by the sender.

All of the runtimes reported in tables 4.2, 4.3, 4.4, and 4.5 are for a single thread.

ID	Sender Per-OLE Preprocess Time (ms)	Receiver Encrypt Time (ms)	Sender Online Time (ms)	Receiver Postprocess Time (ms)
1	3.722	2.91066	1.95537	1.5216
2-7	4.0139	3.6036	2.81912	1.51536

Table 4.2: Vector OLE microbenchmarks for compute-optimized mode

ID	Receiver Encrypt Time (ms)	Sender Online Time (ms)
1	3.0569	2.9040
2-7	3.782	4.0876

Table 4.3: Vector OLE microbenchmarks for communication-optimized mode

Note that in table 4.3 the times for the sender preprocessing and the receiver postprocessing are not reported because they are unchanged from table 4.2.

ID	Sender Per-OLE Preprocess Time (ms)	Receiver Encrypt Time (ms)	Sender Online Time (ms)	Receiver Postprocess Time (ms)
2-5	4.41155	3.80401	2.82767	1.72952
6	5.58623	4.3955	2.82219	2.33956
7	7.32377	5.26913	2.82383	3.20131

Table 4.4: Batch OLE microbenchmarks for compute-optimized mode

ID	Receiver Encrypt Time (ms)	Sender Online Time (ms)
2-5	4.00123	4.10862
6	4.3955	4.11112
7	5.45984	4.08604

Table 4.5: Batch OLE microbenchmarks for communication-optimized mode

We now report the communication complexity for the two modes of the OLE protocols. The communication complexity for both the VOLE and BOLE variants are the same, so we only report the VOLE variant. All numbers below are relative to the receiver (i.e. how much data the receiver sends and receives). These numbers do not include the initial public key sent before the OLE protocol is run; however, this message is identical in size to the receiver's first message in the OLE protocol. In both the computation and communication optimized variants, the amount of data received by the receiver is the same, so only one number is given for these cases.

ID	Receiver Data Sent Computation Optimized (KB)	Receiver Data Sent Communication Optimized (KB)	Receiver Data Received (KB)
1	393.216	196.624	131.072
2-7	524.288	262.160	131.072

Table 4.6: OLE Communication Complexity

4.5.4 Pipelined Benchmarks

In this section, we present the main performance result of this chapter. In the tables below, we give a series of benchmarks for high-throughput VOLE and BOLE protocols. The implementations to obtain these benchmarks are the same as in section 4.5.3. However, the times below include *both* the per-ole preprocessing and the online phase, as described in protocol 1 above. In other words, these times indicate the efficiency of executing VOLE and BOLE protocols with no setup. If setup is allowed, then simply multiplying the online time from section 4.5.3 will give the performance of this setting.

Computation-Optimized VOLE Benchmarks

ID	1 threads		2 threads	
	Wall-clock Time (ms)	Per OLE Time (us)	Wall-clock Time	Per OLE Time
1	6.0312	0.7362	4.469	0.5456
2 - 4	7.2743	0.8880	5.1131	0.6242
5 - 7	7.2929	0.8902	5.1097	0.6237

Table 4.7: Vector OLE Times for 1 & 2 threads

The parameter groupings in table 4.7 are largely due to the fact that the data-types representing the plaintext data are identical.

ID	4 threads		8 threads	
	Wall-clock Time (ms)	Per OLE Time (us)	Wall-clock Time	Per OLE Time
1	3.2665	0.3987	3.4604	0.4224
2 - 7	3.7873	**0.4623**	4.0839	0.4985

Table 4.8: Vector OLE Times for 4 & 8 threads

The groupings in table 4.8 is largely due to the fact that span of the parallelized computation[1] does not involve any plaintext processing. Thus, variability in the plaintext processing times has no effect on the overall runtime. Note that the overhead of thread creation in the eight thread case results in slightly slower runtimes than in the four thread case. The eight thread case is mainly to compare against [ADI+17], whose fastest times use eight threads.

The runtimes given in [ADI+17] include communication time over a 1Gib network with $0.15ms$ latency and were run on a slightly faster machine (3.5 GHz). Their communication did not consume the full bandwidth of the network, so subtracting the network latency and accounting for the difference in clock speed gives approximately $0.6153\mu s$ of computation per OLE for a VOLE protocol of width 10,000 over a 32-bit field when using eight threads. Our protocol gives nearly 25% reduction in computation time over this protocol (where we use the time in bold in table 4.8).

[1]The longest running sequence of threads.

Communication-Optimized VOLE Benchmarks

ID	1 threads		2 threads	
	Wall-clock Time (ms)	Per OLE Time (us)	Wall-clock Time	Per OLE Time
1	7.00786	0.85545	4.87763	0.59541
2 - 7	8.55401	1.0442	6.14207	0.74976

Table 4.9: Communication-Optimized VOLE Times for 1 & 2 threads

ID	4 threads		8 threads	
	Wall-clock Time (ms)	Per OLE Time (us)	Wall-clock Time	Per OLE Time
1	4.05134	0.49455	4.05766	0.49532
2 - 7	4.9333	0.60221	4.85493	0.592643

Table 4.10: Communication-Optimized VOLE Times for 4 & 8 threads

Computation-Optimized BOLE Benchmarks

ID	1 threads		2 threads	
	Per Batch Time (ms)	Per OLE Time (us)	Per Batch Time (ms)	Per OLE Time (us)
2	7.6914	0.9389	5.5072	0.6723
3	7.6641	0.9356	5.4717	0.6679
4	7.7443	0.9453	5.4631	0.6669
5	7.7522	0.9463	5.5366	0.6759
6	8.8391	1.0790	6.5233	0.7963
7	10.667	1.3021	8.5471	1.0433

Table 4.11: Batch OLE Times for 1 & 2 threads

ID	4 threads		8 threads	
	Per Batch Time (ms)	Per OLE Time (us)	Per Batch Time (ms)	Per OLE Time (us)
2	3.9930	0.4874	4.2072	0.5136
3	4.0157	0.4902	4.2351	0.5170
4	3.9897	0.4870	4.2742	0.5218
5	4.0533	0.4948	4.2988	0.5248
6	4.5526	0.5557	5.0155	0.6122
7	5.7565	0.7027	6.1831	0.7548

Table 4.12: Batch OLE Times for 4 & 8 threads

Communication-Optimized BOLE Benchmarks

ID	1 threads		2 threads	
	Per Batch Time (ms)	Per OLE Time (us)	Per Batch Time (ms)	Per OLE Time (us)
2	8.98039	1.09624	6.34768	0.774863
3	8.95177	1.09275	6.34857	0.774972
4	8.95314	1.0929	6.36998	0.77759
5	9.0164	1.10063	6.38082	0.77891
6	10.219	1.24744	6.98307	0.85243
7	12.086	1.475341797	8.175	0.99792

Table 4.13: Communication-Optimized BOLE Times for 1 & 2 threads

ID	4 threads		8 threads	
	Per Batch Time (ms)	Per OLE Time (us)	Per Batch Time (ms)	Per OLE Time (us)
2	4.89069	0.59701	5.1113	0.623938
3	4.86141	0.59343	5.05564	0.6171
4	5.0829	0.620471	5.00712	0.61122
5	5.0632	0.6181	5.10315	0.6229
6	5.61014	0.6848	5.65156	0.68989
7	6.43982	0.786111	6.75374	0.82443

Table 4.14: Communication-Optimized BOLE Times for 4 & 8 threads

4.6 Future Work

The efficiency on this protocol from standard assumptions presents exciting directions for future works. Many of these directions involve using this protocol as a building block for more sophisticated protocols. In particular, we plan to instantiate protocols in the malicious security model using the IPS compiler [IPS08].

In addition, there are new directions allowing maliciously secure OLE to be instantiated directly using several calls to a passively secure OLE protocol [HIMV19]. In fact, full passive security is not required to run this instantiation; the underlying OLE protocol is allowed to be "leaky." This provides motivation to formally quantify the leakage that occurs in the setting where the parameters are relaxed to have a weaker statistical privacy parameter. This could allow for even more efficient OLE protocols to become crucial building blocks to more sophisticated maliciously secure protocols in the future.

Chapter 5

Computing Statistics on Private Data

5.1 Introduction

In this section, we present a platform to securely aggregate private data and perform computations over the joint dataset. This is a form of multi-party computation (MPC). In general, MPC computations consider the case where a group of mutually distrusting parties P wish to compute a function f on private inputs x_i, for $i \in P$. The MPC computations we consider are optimized for the situations that are specialize in two ways. The first specialization is that each party holds a large amount of data relative to the number of parties participating in the computation. The batched operations of our HE schemes naturally support this case. The second specialization is when the function f being computed has a succinct representation as an arithmetic circuit. Again, this specialization is naturally supported by our HE schemes, but it is worth noting that most MPC techniques are optimized for evaluation of Boolean circuits, which can cause major performance degradation when implementing arithmetic operations.

5.2 MPC from HE

In this section, we describe in detail our instantiation of MPC from homomorphic encryption. This instantiation builds off of the theoretical work of Asharov et al. [AJL+12]. At a high level, the intuition for the approach is as follows. In our schemes, homomorphic operations are only defined for ciphertexts that are decryptable with the same secret key. Our approach is to design a distributed key generation algorithm that is run by each party. The result of this key generation algorithm is a public encryption key that is shared by all participating parties as well as an additive share of the corresponding secret key. Since each party has the same public key, they can all encrypt their data with this public key to create ciphertexts that are all decryptable with the same secret key. However, none of the parties actually have this secret key. Instead, they each have additive shares of this secret key, which they can use as inputs to a distributed decryption algorithm that allows a ciphertext encrypted under the shared public key to be decrypted.

Successfully implementing the distributed key generation and decryption functions as described yields a secure MPC protocol. This protocol begins by having the parties run the distributed key generation to obtain the shared encryption key as well as their share of the secret key. Then, each party encrypts their inputs x_i and distributes the resulting ciphertexts ct_i. Next, the ciphertext $\mathsf{ct}_r = \mathsf{Eval}(f, \mathsf{ct}_1, \ldots, \mathsf{ct}_{|P|})$ is computed either by each party or by a trusted central server. Since Eval is deterministic and we only consider deterministic functions f, each party can check that the computation was done correctly to all obtain the same ciphertext ct_r. It is crucial that ct_r is the *same* for all parties. Once each party has ct_r, they can run the distributed decryption protocol to obtain $f(x_1, \ldots, x_{|P|})$, which is the desired output of the protocol. The security of the homomorphic encryption scheme guarantees that no information about the inputs other than the output of f is learned by the participants.

In order to show that this MPC scheme is secure, it suffices to show that the distributed key generation and distributed decryption algorithms do not compromise the security of the underlying HE scheme. We discuss these protocols in more detail below. Note that we abuse notation slightly by denoting uniformly random elements in $\mathcal{R}_q$ as a and error terms as e. We denote when it is necessary that different a terms are equal. Note that it will never be necessary that error terms are equal.

5.2.1 Distributed Encryption Key Generation

In both of the schemes implemented in the Gazelle library, the public encryption key has the structure $\mathsf{pk} = (a, a \cdot s + e)$, where a is a uniformly random element over the ring and s and e are drawn from the error distribution over the ring. The key generation protocol begins by using a common random seed for a pseudorandom number generator (PRNG). Each party then runs this PRNG to generate the a polynomial for the shared public key. Since each party has the same PRNG seed, they will all sample the same a. They then proceed with the remainder of the HE key generation protocol with this a value to obtain the following:

$$(\mathsf{pk}_i, \mathsf{sk}_i) = \big((a, a \cdot s_i + e_i), s_i\big) \tag{5.1}$$

Each party then distributes pk_i to all other parties. The parties can then all compute the following shared public key.

$$\mathsf{pk} = \Big(a,\ \sum_i \mathsf{pk}_i[1]\Big) = \Big(a,\ a \cdot \sum_i s_i + \sum_i e_i\Big) \tag{5.2}$$

Lemma 5.2.1. *If the number of parties in $O(1)$, then the tuple described in equation 5.2 is a well-formed public key.*

Proof. Consider the polynomial $s = \sum_i s_i$ and $e = \sum_i e_i$. The tuple in equation 5.2 is

a public key with secret key s and error e. This is a well-formed public key for an error distribution $\chi' = \sum_i \chi$, the sum of n samples of the original error distribution. Since we only consider error distributions that are Gaussians, this sum is also a Gaussian. The variance of χ' will be larger than the original χ, but if the number of parties is $O(1)$, then this additional magnitude is negligible. □

5.2.2 Distributed Decryption

Consider a ciphertext in the standard BFV scheme, which has the following form:

$$\mathsf{ct} = \big(a,\ a \cdot s + \Delta m + e\big)$$

where Δ is a public scaling factor to prevent the small error term e from corrupting the message m. Decryption is performed by computing the following function on $\mathsf{ct} = (\mathsf{ct}[0], \mathsf{ct}[1])$.

$$m = \left\lfloor \frac{\mathsf{ct}[1] - s \cdot \mathsf{ct}[0]}{\Delta} \right\rceil = \left\lfloor \frac{\Delta m + e}{\Delta} \right\rceil$$

Our distributed decryption protocol collaboratively computes the numerator from "decryption shares" generated by each party, then the division and flooring is computed in the clear. Recall from the previous section that the secret key has the form $s = \sum_{i=1}^{k} s_i$, and each party i has a share s_i of the secret key. We assume that all parties know the total number of parties k as well as the ciphertext $\mathsf{ct} = (a,\ a \cdot s + \Delta m + e)$ that is to be decrypted. Each client i takes their share of the secret key s_i and generates the following decryption share:

$$d_i = a \cdot s + \Delta m + e - k \cdot (a \cdot s_i + e'_i)$$

where all operations are over $\mathcal{R}_q$ and e'_i is an error term sampled from a discrete Gaussian

with large standard deviation[1]. The additive term $(a \cdot s_i + e')$ is multiplied by k because when these shares are summed there are k terms with $a \cdot s = a \sum_i s_i$, each with a component $a \cdot s_i$ that must be subtracted away.

Each sample d_i is then published. The security of this step follows from the fact that the tuple $(a, a \cdot s_i + e'_i)$ is a well-formed Ring-LWE sample, which is computationally indistinguishable from (a, u) for a uniformly sampled $u \leftarrow \mathcal{R}_q$.

When all of the decryption shares have been published, the numerator in the decryption equation can be computed as follows:

$$\begin{aligned} d = \sum_{i=1}^{k} d_i &= k \cdot a \cdot s + k \cdot \Delta m + k \cdot e - \sum_{i=1}^{k} (k \cdot a \cdot s_i + e'_i) \\ &= k \cdot a \cdot s + k \cdot \Delta m + k \cdot e - k \cdot a \sum_{i=1}^{k} (s_i) - e' \\ &= k \cdot a \cdot s + k \cdot \Delta m + k \cdot e - k \cdot a \cdot s - e' \\ &= k \cdot \Delta m + k \cdot e - e' \end{aligned}$$

Note that the message m is multiplied by k in this numerator.

Once this numerator is computed, the message m can be recovered by simply dividing by $k\Delta$ and flooring.

$$m = \left\lfloor \frac{d}{k\Delta} \right\rfloor = \left\lfloor \frac{k \cdot \Delta m + k \cdot e - e'}{k\Delta} \right\rfloor$$

Correctness holds as long as the magnitude of the error term $ke - e'$ remains less than $k\Delta/2$.

[1]Such noise terms are sometimes referred to as "flooding" terms.

5.2.3 Distributed Automorphism Key Generation

The protocol to generate automorphism keys is essentially the same as the protocol to generate encryption keys. This is due to the structure of the automorphism key, which is the following set of pairs of elements in $\mathcal{R}_q$:

$$\mathsf{rotket} = \{(a,\ a \cdot s + e + 2^w s_{\mathsf{rot}})\}_{w=0}^{\lfloor \log q \rfloor}$$

The term s_{rot} is simply a permutation of the terms of s that correspond to the desired rotation on the message. Letting $s = \sum_{i=1}^{k} s_i$, we can then use the linearity of this permutation operation to get the following equation:

$$\sum_{i=1}^{k} (s_i)_{\mathsf{rot}} = s_{\mathsf{rot}}$$

Using identical techniques as in section 5.2.1, we can generate keys for the automorphism operation.

5.2.4 Distributed Multiplicaiton Key Generation

In order to perform the homomorphic evaluation algorithm, we need to generate evaluation keys that corresponds to the shared secret key of the protocol. The evaluation key has the following form:

$$\mathsf{evk} = \{(a,\ a \cdot s + e + 2^w s^2)\}_{w=0}^{\lfloor \log q \rfloor} = \{(a,\ a \cdot s + e + 2^w (\sum_{i=1}^{k} s_i)^2)\}_{w=0}^{\lfloor \log q \rfloor} \tag{5.3}$$

To compute the quadratic function of this sum of secret keys, we consider the function as the sum of the following terms:

$$\Big(\sum_{i=1}^{k} s_i\Big)^2 = \sum_{i=1}^{k}\sum_{j=1}^{k} s_i s_j = \sum_{i=1}^{k} s_i \cdot s$$

In order to compute each of these terms, we define the following two round protocol.

1. The first round is essentially identical to the single round of communication required to generate the shared public key described in section 5.2.1. Each party generates a set of pair of elements in $\mathcal{R}_q$ that has the form $\{(a,\ a \cdot s_i + e + 2^w s_i)\}_{w=0}^{\lfloor \log q \rfloor}$ and distributes this set of pairs to all other parties. Note that for a given value of w, all the a terms across the sets distributed by different parties must be the same. However, the a terms for different values of w need not be the same.

2. Each party now generates sets of the form $\{(a, a \cdot s + e + 2^w s)\}_{w=0}^{\lfloor \log q \rfloor}$. Now, each party i must multiply through by s_i. To perform this operation in a way that the result is safe to publish, noise must be added to each term in the set. The result is a set of elements of the following form:

$$\begin{aligned}\mathsf{evk}_i &= \{(a s_i + e',\ a \cdot s \cdot s_i + e s_i + 2^w s \cdot s_i + e'')\}_{w=0}^{\lfloor \log q \rfloor} \\ &= \{(a',\ a' \cdot s + e''' + 2^w s \cdot s_i)\}_{w=0}^{\lfloor \log q \rfloor}\end{aligned}$$

   Note that a' is now different for every party and every pair in the evk_i set. Our error term e''' has grown by a multiplicative factor of s_i. However, s_i is sampled from the error distribution, so this term remains small. Each party i broadcasts evk_i to all other parties.

3. As the final step, the parties then all compute $\mathsf{evk} = \sum_{i=1}^{k} \mathsf{evk}_i$. Since all sets evk_i

are already under the shared secret key s, the resulting evk will be well formed as in equation 5.3.

5.3 CyberRisk@CSAIL: Securely Measuring Cyber Risk

In this section, we discuss a major application of the techniques above: the Secure Cyber Risk Aggregation and Measurement (SCRAM) platform. Built atop the Gazelle library, this platform is the primary piece of technical infrastructure for the CyberRisk@CSAIL industry consortium. CyperRisk@CSAIL is an industry consortium formed as a joint project with the MIT Internet Policy Research Initiative (IPRI) to better understand the causes of successful cyber attacks and how best to prevent them.

Consortium members consist of companies with annual revenue higher than USD 1 billion and as well as a high level of security sophistication, including a Chief Information Security Officer (CISO). We recruited seven firms as our initial members, representing the financial sector, health care, communications and retail services. The participating firms had an average annual revenue of USD 24 billion (median of USD 18 billion) and and average of 50,000 employees.

5.3.1 Computing on Real Data

After a few sessions with computations on dummy data, we ran a computation in April 2019 on real data from our participants. We ran two computations in the April 2019 session: the first benchmarking defenses and the second associating monetary losses to control failures. We worked closely with the participating companies over four months developing the two computations and their associated input formats. The inputs and outputs of the computations are explained below.

1. **Benchmarking:** In computation one, we benchmark the adoption of a broad set of

cyber defenses to allow firms to compare their own security posture to adoption levels across the group. We use the Center for Internet Security's (CIS) list of 171 critical security sub-controls that are meant to help organizations better defend against known attacks by distilling security concepts into actionable controls. The benchmarking consisted of a questionnaire where firms indicated if they currently implement each of the CIS 171 sub-controls (Yes = 1, No = 0). The result of the computation was the adoption rate of each subcontrol across all participants.

2. **Linking monetary losses to failed sub-controls:** In the second computation, we gather data on losses and implicated security control failures in order to identify problematic controls across the group. We asked firms to submit individual losses and implicate which sub-control failures were responsible. For the computation, firms submitted a table with individual incidents on each row. Each participating firm assigned a monetary loss (in USD thousands) to each of their security incidents and then indicated up to 5 sub-controls (Yes = 1) that were responsible for each loss (either because they were in place and failed, or because they were not implemented). For this round, each implicated sub-control received an equal proportion of the total loss during the computation. We implemented a minimum loss threshold of USD 5,000 in order to exclude routine costs such as reformatting infected machines and focus specifically on larger incidents. The output of this computation was the total loss attributed to each subcontrol across all submitted incidents.

To avoid digressing too far into security policy, we forgo the presentation of these results of these computations. However, at a high level, it is clear that the insights that are able to be learned from this securely pooled dataset are greater than any individual party could learn with their data alone. Furthermore, the removal of any trusted third party allows the participation of companies that are much more sensitive to data leaks as well as allows for

the use of much more sensitive and accurate datasets for computing our results.

5.4 Future Work

Much of the future work of this direction revolves around the expansion of the SCRAM platform and the CyberRisk@CSAIL industry consortium. We are actively developing an application that can better integrate into the systems of the participating firms. This will allow us to run more computations on larger datasets as well as more easily on-board new firms. The high-level goal of these efforts is to increase the scope and sophistication of the measures we can compute to ultimately provide a rich understanding of cyber risk and effective methods to reduce it.

Bibliography

[ACC+18] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

[ADI+17] Benny Applebaum, Ivan Damgård, Yuval Ishai, Michael Nielsen, and Lior Zichron. Secure arithmetic computation with constant computational overhead. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 223–254, Cham, 2017. Springer International Publishing.

[AJL+12] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, pages 483–501, 2012.

[AMBG+16] Carlos Aguilar-Melchor, Joris Barrier, Serge Guelton, Adrien Guinet, Marc-Olivier Killijian, and Tancrède Lepoint. Nfllib: Ntt-based fast lattice library. pages 341–356, 02 2016.

[Bar87] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 311–323, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[BCGI18] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector ole. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS 18, page 896912, New York, NY, USA, 2018. Association for Computing Machinery.

[BDPMW16] Florian Bourse, Rafaël Del Pino, Michele Minelli, and Hoeteck Wee. Fhe circuit privacy almost for free. In Matthew Robshaw and Jonathan Katz, editors,

Advances in Cryptology – CRYPTO 2016, pages 62–89, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, 2012.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. *CRYPTO*, 2012.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, Jan 2000.

[CKK+19] Jung Cheon, Han Kyoohyung, Andrey Kim, Miran Kim, and Yongsoo Song. *A Full RNS Variant of Approximate Homomorphic Encryption: 25th International Conference, Calgary, AB, Canada, August 1517, 2018, Revised Selected Papers*, pages 347–368. 01 2019.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT*, 2017.

[DG14] Nagarjun Dwarakanath and Steven Galbraith. Sampling from discrete gaussians for lattice-based cryptography on a constrained device. *Applicable Algebra in Engineering, Communication and Computing*, 25, 06 2014.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption, 2012.

[Gen09a] Craig Gentry. PhD thesis, Stanford University, 2009.

[Gen09b] Craig Gentry. Fully homomorphic encryption using ideal lattices. 2009.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology – CRYPTO*, 2013.

[HEA] Library for homomorphic encryption for arithmetic of approximate numbers. `https://github.com/snucrypto/HEAAN`.

[HIMV19] Carmit Hazay, Yuval Ishai, Antonio Marcedone, and Muthuramakrishnan Venkitasubramaniam. Leviosa: Lightweight secure arithmetic computation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS 19, page 327344, New York, NY, USA, 2019. Association for Computing Machinery.

[HPS19] Shai Halevi, Yuriy Polyakov, and Victor Shoup. An improved rns variant of the bfv homomorphic encryption scheme. In Mitsuru Matsui, editor, *Topics in Cryptology – CT-RSA 2019 - The Cryptographers' Track at the RSA Conference 2019, Proceedings*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 83–105. Springer-Verlag, 1 2019.

[HS] Shai Halevi and Victor Shoup. Helib - an implementation of homomorphic encryption. `https://github.com/shaih/HElib/`.

[HS14] Shai Halevi and Victor Shoup. Algorithms in helib. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pages 554–571, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[IP07] Yuval Ishai and Anat Paskin. Evaluating branching programs on encrypted data. In Salil P. Vadhan, editor, *Theory of Cryptography*, pages 575–594, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer efficiently. pages 572–591, 08 2008.

[Juv18] Chiraag Juvekar. PhD thesis, MIT, 2018.

[JVC18] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. Gazelle: A low latency framework for secure neural network inference. *27th USENIX Security Symposium*, 2018.

[LM06] Vadim Lyubashevsky and Daniele Micciancio. Generalized compact knapsacks are collision resistant. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming*, pages 144–155, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *EUROCRYPT*, 2010.

[PAL] Palisade homomorphic encryption software library. `https://palisade-crypto.org/`.

[RAD78] R. Rivest, L. Adleman, and M. Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation*, 1978.

[SEA19] Microsoft SEAL (release 3.4). `https://github.com/Microsoft/SEAL`, October 2019. Microsoft Research, Redmond, WA.