SIMPLY SAFE LATTICE CRYPTOGRAPHY

A Dissertation
Presented to
The Academic Faculty

By

Eric Crockett

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

August 2017

SIMPLY SAFE LATTICE CRYPTOGRAPHY

Approved by:

Dr. Chris Peikert, Advisor
Electrical Engineering and Computer Science
University of Michigan

Dr. Alexandra Boldyreva
School of Computer Science
Georgia Institute of Technology

Dr. Richard J. Lipton
School of Computer Science
Georgia Institute of Technology

Dr. J. Alex Halderman
Electrical Engineering and Computer Science
University of Michigan

Dr. Craig Costello
Cryptography and Security Group
Microsoft Research

Date Approved: July 26, 2017

Complexity is still the worst enemy of security.

Bruce Schneier

To Mal Hodges, Dean Isaacs, and Gus Kaufman, Jr.,

who taught me more than a Ph.D. ever could.

ACKNOWLEDGEMENTS

I am extremely grateful to my advisor and faithful collaborator, Chris Peikert. Your expertise, and the patience with which you share it, have been invaluable. Thank you for pushing me to be my absolute best.

I also want to thank my fabulous Atlanta family, The Gaggle. Your friendship allowed me to escape Georgia Tech with both a Ph.D. and my sanity, but you also helped me on a much more important journey to find myself.

Kevin, I can't tell you how much I appreciate having someone to call for advice, or just to listen to me complain about the tribulations of Ph.D. life. I know I can always count on you (unless you're planning to visit me). Mom and Dad, you've sacrificed more than I will ever know to get me where I am. None of this would have been possible without your love and support. Thank you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Lattice cryptography has many compelling features, like security under worst-case hardness assumptions, apparent security against quantum attacks, efficiency and parallelism, and powerful constructions like fully homomorphic encryption. While standard constructions such as lattice-based key exchange are starting to be deployed in real-world scenarios, the most powerful lattice cryptosystems are still limited to research prototypes. This is due in part to the difficulty of *implementing*, *instantiating*, and *using* these schemes.

In this work we present a collection of tools to facilitate broader use of lattice cryptography by improving accessibility and usability. The foundation of this work is $\Lambda\circ\lambda$, a general-purpose software framework for lattice cryptography. The $\Lambda\circ\lambda$ library has several features which distinguish it from prior implementations, including *high-level abstractions* for lattice operations, *advanced functionality* needed for applications like homomorphic encryption, and *safe interfaces*.

Many efficient lattice cryptosystems are based on the relatively new Learning With Errors over Rings (Ring-LWE) problem. In order to attract cryptanalytic effort and improve concrete security estimates for this widely used problem , we publish *challenges* for Ring-LWE and the related Learning With Rounding over Rings problem. Unlike challenges for other cryptographic problems like integer factorization, a dishonest challenger can make Ring-LWE challenges which are much harder to solve than properly generated ones. Thus we propose and implement a non-interactive, publicly verifiable cut-and-choose protocol which provides reasonably convincing evidence that the challenges are properly generated.

Finally, we introduce ALCHEMY, a *domain-specific language* and *compiler* for homomorphic computations. In existing implementations of homomorphic encryption, users must manually represent a desired plaintext computation as a much more complex sequence of operations on ciphertexts. ALCHEMY automates most of the steps in this process, which dramatically reduces the expertise needed to use homomorphic encryption.

CHAPTER 1
INTRODUCTION

The field of cryptography is concerned with all aspects of information security in the presence of an untrusted or malicious party. There are a host of *cryptographic primitives* such as hash functions, pseudo-random functions, public- and private-key encryption, signature schemes, which can be used to solve particular problems in cryptography. At their core, all cryptographic primitives rely on a computationally intractable or "hard" problem. Typically these problems are well-studied and believed to be computationally intractible, e.g., mathematical problems like factoring [RSA78; Rab79], quadratic residuocity [GM84], decoding error correcting codes [McE78], and computing discrete logarithms [DH76]. Since Ajtai's seminal work in 1996 [Ajt04], cryptographers have additionally created primitives which derive their security from hard problems on *lattices*.

1.1 Advantages of Lattice Cryptography

Lattice cryptography refers to a diverse set of cryptographic constructions that derive their security from hard problems on point lattices in $\mathbb{R}^n$, i.e., a discrete additive subgroup of $\mathbb{R}^n$. These objects have been studied since 1842 by the likes of Dirichlet and Minkowski [Ajt04]. Lattice cryptography has many features which make it a compelling alternative to number-theoretic cryptography. Among these are its apparent quantum security, its ability to have security from worst-case hardness assumptions, and powerful constructions like fully homomorphic encryption. We explore the many advantages in more detail below.

Performance. Early lattice cryptosystems [AD97; GGH97] were impractical due to large keys and ciphertexts. In particular, the [AD97] public-key encryption scheme had public keys of size $\tilde{\mathcal{O}}(n^4)$ and ciphertexts of size $\tilde{\mathcal{O}}(n^2)$, with similar runtimes for encryption and

decryption, respectively. However, the NTRU public-key encryption scheme introduced by [HPS98] demonstrated how the use of algebraically *structured* lattices (corresponding to polynomial *rings*) can lead to very efficient cryptography using lattices. Efficiency was further improved with the introduction of the Learning with Errors (LWE) problem [Reg09]. These two improvements were eventually combined into the flexible and efficient Ring-LWE problem [LPR13b], which has been widely used in lattice cryptosystems. These efficient schemes are broadly known as *ring-based* cryptography.

Parallelism. Most modern hardware supports some form of parallelism, e.g., via vector instruction sets, multiple cores, or graphics processing units (GPUs). Lattice cryptosystems are well-poised to take advantage of this hardware parallelism because lattice operations in $\mathbb{R}^n$ can be performed in $\mathcal{O}(\log n)$ or even $\mathcal{O}(1)$ parallel operations on n processors. This has the potential to make expensive applications, like fully homomorphic encryption, usable in practice. Parallelism in lattice cryptography has only recently been explored using hardware vector instructions [Alk+16; Bou+17] and GPUs [Wan+12].

Quantum Security. In some cryptographic applications (like message authentication), we only need to consider the *current* computational abilities of an adversary. With applications like encryption though, we might require that an adversary who collects encrypted data today should not be able to read it for (say) the next 100 years. This means we must account for computational and algorithmic advances which may take place over that period, including the possibility that future attackers may have access to more powerful computational models that do not exist today.

One such model that has been widely studied is the *quantum computer*. Considerable work been done towards actually constructing a large-scale quantum computer. Furthermore, it appears that quantum computers offer additional computation power compared to classical devices. In particular, Peter Shor [Sho97] showed that cryptography relying on the intractibility of factoring large numbers or computing discrete logarithms would be

insecure with mature quantum computing (though these problems are apparently secure against a classical adversary). Researchers have also tried to attack lattice problems with quantum algorithms, but have so far come up empty handed. This gives lattice cryptography the distinguished property of having (apparent) *quantum security*, which has led to interest outside academia [Age15; Bra16a].

Worst-case vs. Average-case hardness. Traditionally, the standard for a "hard problem" was *worst-case hardness*, which says that *some* instances of the problem are hard to solve. There might not be very many of these instances, or they might be difficult to find.

Cryptographic primitives choose a *random* instance of a hard problem from some distribution, so we require that all but a negligible fraction of instances from this distribution are hard to solve. This is known as *average-case hardness*. It can be difficult to choose a distribution for which most instances of the problem are hard, though.

As an example, we consider integer factorization. Although most integers of a fixed size are easy to factor (because they likely have small prime factors), cryptographers believe that when integers have exactly two equal-size prime factors, their product is hard to factor. Thus this is the distribution used for factoring-based cryptography, despite the lack any theoretical evidence suggesting that this integers from this distribution are indeed hard to factor.

One way to avoid the problem of crafting a "hard" distribution is with a *worst-case to average-case reduction*, which says that an algorithm which solves some noticeable fraction of *random* instances of some problem can be used to solve *every* instance of a (possibly different) problem. In 1996, Ajtai showed that lattices admit this strong property [Ajt04]. Specifically, he showed that finding the shortest vector in a lattice chosen randomly from a certain class is as hard as solving three problems on *any* lattice.

Applications. Almost all cryptographic applications that can be constructed from number-theoretic assumptions can also be constructed with lattices. However, some advanced constructions like attribute-based encryption (e.g., [GPV08; GVW13]), which reveals data

only to parties satisfying some arbitrary predicate, and fully homomorphic encryption (e.g., [Gen09b; BGV14; GSW13]), which allows arbitrary computation on encrypted data, have *only* been constructed from lattices. It is not known how to construct these applications from any other cryptographic assumptions, making lattice cryptography the only choice for this advanced functionality.

1.2 Lattice Cryptography Today

Lattice cryptography has seen enormous growth over the past decade. A broad movement toward the *practical implementation* of lattice/ring-based schemes in the past few years has led to an impressive array of results (e.g., [HPS98; Ber+16; Lyu+08; GLP12; Duc+13; Bos+15; Alk+16; Bos+16b; HS; May16; LCP17]). While these have all been research prototypes, there has been very recent progress in experimenting with lattice-based key exchange [Duc+13] on the internet, e.g., in Google's Chrome web browser [Bra16b], the strongSwan IPSec implementation [Ste14], and the Tor protocol [LS16].

The most powerful lattice-based constructions, however, have not yet seen this level of deployment. There are many possible explanations for this state of affairs, but we contend that the challenges facing advanced lattice cryptosystems are primarily *practical* rather than *theoretical* in nature. Specifically, advanced cryptosystems require functionality that is not included in implementations of simpler schemes, hence it is difficult to build and test them. Next, despite its great promise, homomorphic encryption remains difficult to use: only experts can write satisfactory homomorphic computations and select parameters for HE schemes. Another problem facing *all* lattice cryptosystems is that it is difficult to estimate their security in practice. We explore these problems in more detail below.

1.2.1 Lattice Operations

All efficient lattice cryptosystems rely on a handful of shared techniques such as integer modular arithmetic and rounding, error sampling, "gadget" operations including discrete Gaus-

sian sampling, ring switching, ring arithmetic, and inter-ring operations [Mic07; LPR13b; MP12; Gen+13]. Each of these components is much more complex than the tools used in more traditional number-theoretic cryptography. Nevertheless, these primitive lattice operations have been implemented many times in various one-off implementations which, to date, have been specialized to a particular cryptographic primitive, like collision-resistant hashing [Lyu+08], digital signatures [GLP12; Duc+13], key-establishment protocols [Bos+15; Alk+16; Bos+16b], and homomorphic encryption [NLV11; HS].

These tailored implementations typically use fixed parameter sets and have few reusable interfaces, making them hard to implement other primitives upon. Those interfaces that do exist are quite *low-level*; e.g., they require the programmer to explicitly convert between various representations of ring elements, which calls for specialized expertise and can be error prone. Finally, prior implementations either do not support, or use suboptimal algorithms for, the important class of *arbitrary cyclotomic* rings, and thereby lack related classes of homomorphic encryption functionality.

Thus with the current collection of implementations, it is difficult to rapidly prototype lattice cryptosystems (especially those requiring advanced functionality) and to experiment with parameters, parallelism, and more. Lattice cryptography is also in need of well-designed *abstractions* which make it easier and safer to implement lattice cryptosystems.

1.2.2 Complexity of Homomorphic Encryption

Homomorphic Encryption (HE) is a powerful cryptographic concept that allows a worker to perform computations on client-encrypted data, without learning anything about the data itself. There are two types of homomorphic encryption schemes: *somewhat-homomorphic encryption* (SHE) schemes restrict the set of computations that can be performed (e.g., to a certain multiplicative depth), while *fully homomorphic encryption* (FHE) schemes allow *arbitrary* computations.[1] Although first envisioned almost 40 years ago [RAD78]

[1] In much of this work, the distinction between these two concepts is not needed, and we use the generic term "homomorphic encryption" (HE) for statements that apply to both.

as a cryptographic "holy grail," no plausible candidate FHE scheme was known until Gentry's seminal work in 2009 [Gen09b; Gen09a], which showed how to turn somewhat-homomorphic schemes into *fully* homomorphic schemes. Prompted by HE's potential to enable new privacy-aware applications or enhance existing ones, a flurry of research activity has led to schemes with better efficiency, stronger security assurances, and specialized features. (See [Dij+10; SV14; BV11b; Cor+11; CNT12; BV14a; BGV14; Bra12; GHS12b; GHS12a; Che+13; AP13; Gen+13; BV14b; AP14] for a sampling.)

The power of HE translates to a heavy burden on *users* of HE, because there are a large number of tunable parameters and different routes to the user's end goal. In current implementations, merely expressing a homomorphic computation requires expertise in the intricacies of the homomorphic encryption scheme and its particular implementation. Some recent implementations like [LCP17] attempt to partially resolve this complexity by automatically choosing (some) parameters, but many details are still left for the user to manage. This usability challenge limits the impact and usefulness of an otherwise powerful application.

1.2.3 Security Estimates

The security of factoring-based cryptography like RSA is reasonably well-understood: there is a single parameter n (size of the modulus), and increasing n makes the problem harder. Futhermore, the runtime of the general number field sieve, the most efficient known algorithm for factoring large numbers, is easily expressed as a function of n. By contrast, lattice cryptography uses a large number of parameters, all of which interact in complex ways to affect security. Moreover, the best algorithms for attacking lattice problems (like the Block Korkin-Zolotarev (BKZ) basis-reduction algorithm [SE94; CN11]) are poorly understood, and it is notoriously difficult to estimate their runtime. As a result, it is very difficult to accurately estimate the concrete hardness of lattice schemes, for any combination of parameters.

Lacking concrete security estimates, instantiations *could* rely on strong worst-case hardness guarantees. In practice, the parameters needed to obtain this security guarantee are so large as to be impractical when compared to alternative types of cryptography. There remains, however, a large gap between parameters *required* for worst-case guarantees and parameters that are known to be insecure against concrete attacks. Thus many proposed instantiations live somewhere in this gap, with parameters that apparently thwart practical attacks, but that do not support worst-case hardness guarantees [Lyu+08; Duc+13; Alk+16; Bos+16b].

As lattice cryptography becomes more widely used in practice, especially with parameters that lack much (if any) theoretical support, there is an increasing need for further cryptanalytic effort and higher-confidence security estimates for its underlying computational problems.

1.3 Our Contributions

In this work we present a collection of tools which address the practical needs of lattice cryptography. The goal of these tools is to facilitate broader use of lattice cryptography by improving accessibility for researchers, implementors, and end-users. Specifically, we aim to make lattice cryptography easier to get right, simpler to use, and help set the stage for widespread adoption of this leading post-quantum candidate. Our software frameworks emphasize *safety* through programming language features like strong, static typing and domain-specific languages.

The foundation of this thesis is a software framework for lattice cryptography that provides modular and reusable interfaces for operations which appear in a variety of cryptosystems. We also introduce ALCHEMY, a *domain-specific language* and *compiler* for simplifying the process of writing homomorphic computations. Finally, we propose cryptanalytic challenges for a wide range of parameters for two related problems which are broadly used in efficient lattice cryptosystems. We explain these tools in more detail below.

1.3.1 A Functional Library for Lattice Cryptography

At the core of this work is $\Lambda\circ\lambda$, a general-purpose software framework for lattice-based cryptography. The $\Lambda\circ\lambda$ framework has several novel properties that address the limitations of prior implementations of lattice cryptosystems, including the following:

Generality, modularity, concision: $\Lambda\circ\lambda$ defines a collection of general, highly composable interfaces for mathematical operations used across lattice cryptography, allowing for a wide variety of schemes to be expressed very naturally and at a high level of abstraction. For example, we implement an advanced somewhat-homomorphic encryption scheme in as few as 2–5 lines of code per feature, via code that very closely matches the scheme's mathematical definition.

Theory affinity: $\Lambda\circ\lambda$ is designed from the ground-up around the specialized ring representations, fast algorithms, and worst-case hardness proofs that have been developed for the Ring-LWE problem and its cryptographic applications. In particular, it implements fast algorithms for sampling from *theory-recommended* error distributions over *arbitrary* cyclotomic rings, and provides tools for maintaining tight control of error growth in cryptographic schemes.

Safety: $\Lambda\circ\lambda$ has several facilities for reducing code complexity and programming errors, thereby aiding the *correct* implementation of lattice cryptosystems. In particular, it uses strong typing to *statically enforce*—i.e., at compile time—a wide variety of constraints among the various parameters.

Advanced features: $\Lambda\circ\lambda$ exposes the rich *hierarchy* of cyclotomic rings to cryptographic applications. We use this to give the first-ever implementation of an important HE operation known as "ring switching," and also define and analyze a more efficient variant that we call "ring tunneling."

Lastly, this work defines and analyzes a variety of mathematical objects and algorithms for the recommended usage of Ring-LWE in cyclotomic rings, which we believe will serve as a useful knowledge base for future implementations.

1.3.2 A Language and Compiler for Homomorphic Encryption

Homomorphic encryption (HE) allows a worker to perform computations on client-encrypted data, without learning anything about the data itself. Since the first plausible construction in 2009, a variety of real-world HE implementations have been given and used for particular applications of interest. Unfortunately, using HE is currently very complicated, and a great deal of expertise is required to satisfactorily implement a desired homomorphic computation.

This work introduces ALCHEMY, a modular and extensible system that greatly accelerates and simplifies the implementation of homomorphic computations. With ALCHEMY, one expresses a desired "in the clear" computation on plaintexts in a simple *domain-specific language*, and then uses a *compiler* to automatically transform it into a corresponding homomorphic program on ciphertexts. The compiler deals with the cumbersome but rote tasks of tracking the ciphertext "noise" and scheduling appropriate "maintenance" operations to control it, choosing (most of) the parameters, generating keys and hints, etc. In addition, ALCHEMY compilers can be composed together to provide other useful functionality, such as pretty-printing a representation of the programs, logging the empirical noise rates of ciphertexts throughout a computation, etc. In short, ALCHEMY lets programmers write clear and concise code describing what they really care about—the plaintext computation—and easily get a corresponding homomorphic computation without needing any particular expertise in HE.

To demonstrate the simplicity of creating homomorphic computations with ALCHEMY, we propose a design and implementation of *ring-rounding* on encrypted values. This operation is the main component of "bootstrapping" (which makes any somewhat-homomorphic encryption scheme *fully* homomorphic), the Ring-LWR problem, and symmetric encryption

using lattice-based pseudorandom functions like [BPR12; BP14]. The key idea behind our design is to exploit the close "algebraic fit" between ring-rounding and known lattice-based homomorphic encryption constructions.

1.3.3 Cryptanalytic Challenges for Ring Cryptography

Recent lattice cryptography implementations use constructions based on the Learning With Errors (LWE) problem, its more efficient ring-based variant Ring-LWE, and their "deterministic error" counterparts Learning With Rounding (LWR) and Ring-LWR. As these problems are the most widely used in practice (especially the efficient ring variants), it is important to have a better understanding of their concrete security.

The standard approach for attracting cryptanalytic effort and obtaining better security estimates for problems in cryptography is by issuing *challenges* (see, e.g., [91; 97; PS13a; 15; Yas+15]). Following these works, we give a broad collection of challenges for Ring-LWE and Ring-LWR instantiations over cyclotomics rings. The challenges cover a wide variety of instantiations, involving two-power and non-two-power cyclotomics; moduli of various sizes and arithmetic forms; small and large numbers of samples; and error distributions satisfying the bounds from worst-case hardness theorems related to ideal lattices, along with narrower errors that still appear to yield hard instantiations. We estimate the hardness of each challenge by giving the approximate Hermite factor and BKZ block size needed to solve it via lattice-reduction attacks.

A central issue in the creation of challenges for LWE-like problems is that dishonestly generated instances can be much harder to solve than properly generated ones, or even impossible. To address this, we devise and implement a simple, non-interactive, publicly verifiable protocol which gives reasonably convincing evidence that the challenges are properly distributed, or at least not much harder than claimed.

1.3.4 Organization

The rest of this thesis is organized as follows:

Chapter 2 contains the technical background needed to understand this thesis, including mathematical background and a primer for the functional programming language Haskell, which was used to implement $\Lambda\circ\lambda$ and ALCHEMY.

Chapters 3 and 4 are devoted to $\Lambda\circ\lambda$. Chapter 3 introduces the main library component of $\Lambda\circ\lambda$ which contains the primary interfaces. In chapter 4, we implement advanced SHE with $\Lambda\circ\lambda$, and give a detailed evaluation of the overall framework.

Chapters 5 and 6 introduce ALCHEMY. We discuss the design and interfaces in chapter 5, and use it to implement homomorphic evaluation of symmetric-key primitives in chapter 6. This implementation serves as our primary method of evaluating the ALCHEMY system.

Chapter 7 introduces our cryptanalytic challenges for ring-based cryptography, including the cut-and-choose protocol for providing convincing evidence that the challenges are honestly generated.

CHAPTER 2

BACKGROUND

This chapter provides the necessary background to understand the technical content of this thesis. Section 2.1 includes basic concepts related to algebra, the ring of integers, an introduction to lattices, the theory of Gaussian distributions, and other miscellaneous topics. In section 2.2 we introduce the concept and structure of *cyclotomic rings*, and also formally define two important computational problems used in lattice cryptography. Finally, section 2.3 gives a brief introduction to Haskell.

2.1 Mathematical Background and Notation

Notation. We write $\tilde{O}(n)$ for $\mathcal{O}(n \log(n))$ and similarly for $\tilde{\Theta}(n)$ and $\tilde{\Omega}(n)$. For a vector $x \in \mathbb{R}^n$, we write $\|x\|$ to denote the standard Euclidean (or ℓ_2) norm, i.e., $\|x\| = \|x\|_2 = \sqrt{(\langle x, x \rangle)}$.

Rings and Ideals. For an *arbitrary* ring R, an *ideal* $\mathcal{I} \subseteq R$ is a nontrivial additive subgroup that is also closed under multiplication by R, i.e., $x \cdot r \in \mathcal{I}$ for any $x \in \mathcal{I}, r \in R$. When an ideal $I = aR$ for some $a \in I$, we say I is *generated* by a and write $I = (a)$.

Kronecker Product The Kronecker product of two matrices gives their corresponding tensor product. For example $M = A \otimes B$ (where A is $r_A \times c_A$ and B is $r_B \times c_B$) is the $r_A \cdot r_B \times c_A \cdot c_B$ matrix corresponding to replacing each a_{ij} with the *matrix* $a_{ij}B$. We show

two special cases of interest below, namely $A \otimes I_k$ and $I_k \otimes A$:

$$\begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} \otimes I_3 = \begin{bmatrix} a_1 & & & a_2 & & \\ & a_1 & & & a_2 & \\ & & a_1 & & & a_2 \\ a_3 & & & a_4 & & \\ & a_3 & & & a_4 & \\ & & a_3 & & & a_4 \end{bmatrix}, \quad I_3 \otimes \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & & & & \\ a_3 & a_4 & & & & \\ & & a_1 & a_2 & & \\ & & a_3 & a_4 & & \\ & & & & a_1 & a_2 \\ & & & & a_3 & a_4 \end{bmatrix}$$

Tensoring with identity (on either side) corresponds to applying A to certain "slices" of the input vector, giving a simple and efficient *parallel* algorithm for multiplying by a tensor with the identity matrix.

A useful fact about the Kronecker product which we use frequently is called the *mixed-product property*: $(A \otimes B) \cdot (C \otimes D) = AC \otimes BD$ (when we can form the matrix products AC and BD).

2.1.1 Integers

Euler's Totient Function We frequently need Euler's totient function $\varphi(n)$, which counts the number of integers that are both less than n and coprime with n. $\varphi(1) = 1$, and for a prime power p^e with $e \geq 1$, $\varphi(p^e) = (p-1)p^{e-1}$. For an arbitrary positive integer m with prime-power factorization $m = p_1^{e_1} \dots p_k^{e_k}$, $\varphi(m) = \prod_{i=1}^{k} \varphi(p_i^{e_i})$, i.e., the totient function is *multiplicative*.

Modular Arithmetic. As usual, $\mathbb{Z}$ denotes the ring of integers, and the quotient ring $\mathbb{Z}_q \cong \mathbb{Z}/(q\mathbb{Z})$ is the ring of integers modulo p, i.e., the cosets $x + q\mathbb{Z}$ with the usual addition and multiplication operations.

Rounding. For integers $q \geq p \geq 2$, we define the rounding function $\lfloor \cdot \rceil : \mathbb{Z}_q \to \mathbb{Z}_p$ by $\lfloor x \rceil_p = \lfloor (p/q) \cdot \bar{x} \rfloor$, where $\bar{x} \in \mathbb{Z} \equiv x \bmod q$. We extend this function to vectors component-wise.

Chinese Remainder Theorem. The Chinese remainder theorem (CRT) gives an isomorphism between $\mathbb{Z}_{q_1 \cdot q_2}$ and the ring product $\mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2}$ when q_1 and q_2 are coprime. In fact, the CRT holds in a more general setting, which we will also need (see below).

2.1.2 Lattices

In cyclotomic ring-based lattice cryptography, we use the space $H \subseteq \mathbb{C}^n$ for some even integer n, defined as

$$H := \{\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{C}^n : x_i = \overline{x_{i+n/2}},\ i \in \{1, \ldots, n/2\}\}.$$

It is easy to check that H, with the inner product $\langle \mathbf{x}, \mathbf{y} \rangle = \sum_i x_i \overline{y_i}$ of the ambient space $\mathbb{C}^n$, is an n-dimensional real inner product space, i.e., it is isomorphic to $\mathbb{R}^n$ via an appropriate rotation. Therefore, the reader may mentally replace H with $\mathbb{R}^n$ in all that follows. We let $\mathcal{B} = \{\mathbf{x} \in H : \|\mathbf{x}\| \leq 1\}$ denote the closed unit ball in H (in the Euclidean norm).

For the purposes of this work, a *lattice* $\mathcal{L}$ is discrete additive subgroup of H that is full rank, i.e., $\mathrm{span}_{\mathbb{R}}(\mathcal{L}) = H$. A lattice is generated as the set of integer linear combinations of some linearly independent basis vectors $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$:

$$\mathcal{L} = \mathcal{L}(\mathbf{B}) := \Big\{ \sum_i z_i \mathbf{b}_i : z_i \in \mathbb{Z} \Big\}.$$

The *volume* (or determinant) of a lattice $\mathcal{L}$ is $\mathrm{vol}(\mathcal{L}) := \mathrm{vol}(H/\mathcal{L}) = |\det(\mathbf{B})|$, where $\mathbf{B}$ denotes any basis of $\mathcal{L}$. The *minimum distance* of $\mathcal{L}$ is $\lambda_1(\mathcal{L}) := \min_{\mathbf{0} \neq \mathbf{v} \in \mathcal{L}} \|\mathbf{v}\|$, the length of a shortest nonzero lattice vector. The *dual lattice* $\mathcal{L}^\vee$ of a lattice $\mathcal{L}$ is the set of all points in H having integer inner products with every vector of the lattice: $\mathcal{L}^\vee := \{\mathbf{w} \in H : \langle \mathbf{w}, \mathcal{L} \rangle \subseteq \mathbb{Z}\}$.

2.1.3 Gaussians.

The Gaussian function $\rho\colon H \to \mathbb{R}^+$ is defined as $\rho(\mathbf{x}) := \exp(-\pi\|\mathbf{x}\|^2)$, and is scaled to have *parameter* (or *width*) $r > 0$ by defining $\rho_r(\mathbf{x}) := \rho(\mathbf{x}/r)$. The (spherical) Gaussian probability distribution D_r over H is defined to have probability density function $r^{-n} \cdot \rho_r$. (We usually omit the subscript when $r = 1$.)

The following bounds use the function

$$f(x) = \sqrt{2\pi e} \cdot x \cdot \exp(-\pi x^2), \tag{2.1.1}$$

which is strictly decreasing and at most 1 for $x \geq 1/\sqrt{2\pi}$.

Lemma 2.1.1 ([Ban93, Lemma 1.5]). *For any $c > 1/\sqrt{2\pi}$ defining $C = f(c) < 1$, and any lattice $\mathcal{L} \subset H$,*

$$\rho(\mathcal{L} \setminus c\sqrt{n}\mathcal{B}) < C^n \cdot \rho(\mathcal{L}).$$

The analogous continuous bound $D(H \setminus c\sqrt{n}\mathcal{B}) < C^n$ follows by taking an arbitrarily dense lattice $\mathcal{L}$ and using a limiting argument. The following is a result of rearranging terms.

Corollary 2.1.2. *If $\pi c^2 - \ln c \geq \frac{1}{n}\ln(\frac{1}{\varepsilon}) + \frac{1}{2}\ln(2\pi e)$ for some $c > 1/\sqrt{2\pi}$ and $\varepsilon > 0$, then $D(H \setminus c\sqrt{n}\mathcal{B}) < \varepsilon$.*

The following is an immediate corollary of Lemma 2.1.1 and [MR07, Lemma 4.1].

Lemma 2.1.3. *For any lattice $\mathcal{L} \subset H$ and $r > \sqrt{n/2\pi}/\lambda_1(\mathcal{L}^\vee)$ defining $C = f(r\lambda_1(\mathcal{L}^\vee)/\sqrt{n}) < 1$, the statistical distance between $D_r \bmod \mathcal{L}$ and the uniform distribution over $H/\mathcal{L}$ is less than $\frac{1}{2}C^n/(1 - C^n)$.*

2.2 Cyclotomic Rings

2.2.1 Introduction

Polynomial Rings. For a ring R and indeterminate X, a polynomial ring $R[X]$ is the set of all finite-degree formal sums $\sum_{i=0}^{n} a_i \cdot X^i$, where each $a_i \in R$ and X is an indeterminate. For example, $3X^2 - 5X + 1 \in \mathbb{Z}[X]$. As with the integers, we can take quotients of the form $R[X]/(f(X))$ where $f(X) \in R[X]$ and $(f(X)) = f(X) \cdot R[X]$ is the ideal generated by $f(X)$. Continuing our example, we set $f(X) = X^2 + 2$ so that $3X^2 - 5X + 1 = 3(X^2 + 2) - 5X - 5 \cong -5x - 5 \in \mathbb{Z}[X]/(X^2 + 2)$. In particular, if $f(X)$ has degree n, the quotient is a polynomial of degree $< n$.

Basic Cyclotomic Rings. For a positive integer m, let $R = \mathbb{Z}[\zeta_m]$ denote the mth *cyclotomic ring*, where ζ_m is an abstract element of multiplicative order m, i.e., $\zeta_m^m = 1$ and $\zeta_m^j \neq 1$ for all positive $j < m$. For example, the first cyclotomic ring is $\mathcal{O}_1 = \mathbb{Z}$. The parameter m is known as the *index* or *conductor* of the cyclotomic ring. For a positive integer q, we frequently use the quotient ring $R_q = R/qR = \mathbb{Z}_q[\zeta_m]$, i.e., the mth cyclotomic over base ring $\mathbb{Z}_q$. As with the integers, we can represent $R_{q_1 \cdot q_2}$ as a ring product $R_{q_1} \times R_{q_2}$, with component-wise operations, via the Chinese Remainder Theorem. (Note that we *also* use the Chinese Remainder Theorem on the factorization of pR into prime ideals below. These two uses are independent, but we refer to their combined use as "double CRT" representation.)

The mth cyclotomic ring is the ring of algebraic integers of (and therefore contained in) the mth *cyclotomic number field* $K = \mathbb{Q}(\zeta_m)$, the ring extension of the rationals $\mathbb{Q}$ obtained by adjoining an element ζ_m. The minimal polynomial $\Phi_m(X)$ (over the rationals) of ζ_m is called the mth *cyclotomic polynomial*. This polynomial has degree $n = \varphi(m)$, so $\deg(K/\mathbb{Q}) = \deg(R/\mathbb{Z}) = n$.

We may also view K (respectively, R, R_q) as a polynomial ring via the isomorphism $\mathbb{Q}(\zeta_m) \cong \mathbb{Q}[X]/(\Phi_m(X))$ (resp. $\mathbb{Z}[\zeta_m] \cong \mathbb{Z}[X]/(\Phi_m(X))$, $\mathbb{Z}_q[\zeta_m] \cong \mathbb{Z}_q[X]/(\Phi_m(X))$, by

identifying ζ_p with X. In particular, this means we write may cyclotomic ring elements as a vector of coefficients with respect to some fixed basis, e.g., the standard polynomial basis $\{1, X, X^2, \ldots\}$. That is, an element of K (respectively, R, R_q) can be uniquely represented as a rational (resp., integral, Z_q) polynomial in X of degree less than n.

Cyclotomic Heierarchy. The mth cyclotomic ring $R = \mathbb{Z}[\zeta_m]$ can be seen as a *subring* of the m'th cyclotomic ring $R' = \mathbb{Z}[\zeta_{m'}]$ if and only if $m|m'$, and in such a case we can *embed* R into R' by identifying ζ_m with $\zeta_{m'}^{m'/m}$. The dimension of the ring extension R/R' is $\dim(R/R') = \varphi(m)/\varphi(m')$.

The *trace* function $\mathrm{Tr}_{R'/R} \colon R' \to R$ is the R-linear function defined as follows: fixing any R-basis of R', multiplication by an $x \in R'$ can be represented as a matrix M_x over R with respect to the basis, which acts on the multiplicand's vector of R-coefficients. Then $\mathrm{Tr}_{R'/R}(x)$ is simply the trace of M_x, i.e., the sum of its diagonal entries. (This is invariant under the choice of basis.) Because R'/R is Galois, the trace can also be defined as the sum of the automorphisms of R' that fix R pointwise. All of this extends to the field of fractions of R' (i.e., its ambient number field) in the same way.

Notice that the trace does *not* fix R (except when $R' = R$), but rather $\mathrm{Tr}_{R'/R}(x) = \deg(R'/R) \cdot x$ for all $x \in R$. For a tower $R''/R'/R$ of ring extensions, the trace satisfies the composition property

$$\mathrm{Tr}_{R''/R} = \mathrm{Tr}_{R'/R} \circ \mathrm{Tr}_{R''/R'} .$$

2.2.2 Tensor Product of Rings

Let R, S be arbitrary rings with common subring $E \subseteq R, S$. The *ring tensor product* of R and S over E, denoted $R \otimes_E S$, is the set of E-linear combinations of *pure tensors* $r \otimes s$

for $r \in R$, $s \in S$, with ring operations defined by E-bilinearity, i.e.,

$$(r_1 \otimes s) + (r_2 \otimes s) = (r_1 + r_2) \otimes s$$
$$(r \otimes s_1) + (r \otimes s_2) = r \otimes (s_1 + s_2)$$
$$e(r \otimes s) = (er) \otimes s = r \otimes (es)$$

for any $e \in E$, and the mixed-product property

$$(r_1 \otimes s_1) \cdot (r_2 \otimes s_2) = (r_1 r_2) \otimes (s_1 s_2).$$

We need the following facts about tensor products of cyclotomic rings. Let $R = \mathcal{O}_{m_1}$ and $S = \mathcal{O}_{m_2}$. Their largest common subring and smallest common extension ring (called the *compositum*) are, respectively,

$$E = \mathcal{O}_{m_1} \cap \mathcal{O}_{m_2} = \mathcal{O}_{\gcd(m_1, m_2)}$$
$$T = \mathcal{O}_{m_1} + \mathcal{O}_{m_2} = \mathcal{O}_{\mathrm{lcm}(m_1, m_2)}.$$

Moreover, the ring tensor product $R \otimes_E S$ is isomorphic to T, via the E-linear map defined by sending $r \otimes s$ to $r \cdot s \in T$. In particular, for coprime m_1, m_2, we have $\mathcal{O}_{m_1} \otimes_{\mathbb{Z}} \mathcal{O}_{m_2} \cong \mathcal{O}_{m_1 m_2}$.

2.2.3 Powerful Basis

Prime cyclotomics. For a prime p, the pth cyclotomic ring is $\mathcal{O}_p = \mathbb{Z}[\zeta_p]$, where ζ_p denotes a primitive pth root of unity, i.e., ζ_p has multiplicative order p. The minimal polynomial over $\mathbb{Z}$ of ζ_p is $\Phi_p(X) = 1 + X + X^2 + \cdots + X^{p-1}$, so $\mathcal{O}_p$ has degree $\varphi(p) = p - 1$ over $\mathbb{Z}$, and we have the ring isomorphism $\mathcal{O}_p \cong \mathbb{Z}[X]/(\Phi_p(X))$ by identifying ζ_p with X. The

power basis $\vec{p}_p$ of $\mathcal{O}_p$ is the $\mathbb{Z}$-basis consisting of the first $p-1$ powers of ζ_p, i.e.,

$$\vec{p}_p := (1, \zeta_p, \zeta_p^2, \ldots, \zeta_p^{p-2}).$$

For example, the 5th cyclotomic polynomial is $1+X+X^2+X^3$, and the 5th cyclotomic ring is isomorphic to $\mathbb{Z}[X]/(1+X+X^2+X^3)$. The power basis for $\mathcal{O}_5$ is $(1, \zeta_5, \zeta_5^2, \zeta_5^3)$.

Prime-power cyclotomics. Now let $m = p^e$ for $e \geq 2$ be a power of a prime p. Then we can inductively define $\mathcal{O}_m = \mathcal{O}_{m/p}[\zeta_m]$, where ζ_m denotes a primitive pth root of $\zeta_{m/p}$. Its minimal polynomial over $\mathcal{O}_{m/p}$ is $X^p - \zeta_{m/p}$, so $\mathcal{O}_m$ has degree p over $\mathcal{O}_{m/p}$, and hence has degree $\varphi(m) = (p-1)p^{e-1}$ over $\mathbb{Z}$.

The above naturally yields the *relative* power basis of the extension $\mathcal{O}_m/\mathcal{O}_{m/p}$, which is the $\mathcal{O}_{m/p}$-basis

$$\vec{p}_{m,m/p} := (1, \zeta_m, \ldots, \zeta_m^{p-1}).$$

More generally, for any powers m, m' of p where $m|m'$, we define the relative power basis $\vec{p}_{m',m}$ of $\mathcal{O}_{m'}/\mathcal{O}_m$ to be the $\mathcal{O}_m$-basis obtained as the Kronecker product of the relative power bases for each level of the tower:

$$\vec{p}_{m',m} := \vec{p}_{m',m'/p} \otimes \vec{p}_{m'/p,m'/p^2} \otimes \cdots \otimes \vec{p}_{mp,m}. \tag{2.2.1}$$

Notice that because $\zeta_{p^i} = \zeta_{m'}^{m'/p^i}$ for $p^i \leq m'$, the relative power basis $\vec{p}_{m',m}$ consists of all the powers $0, \ldots, \varphi(m')/\varphi(m) - 1$ of $\zeta_{m'}$, but in "base-p digit-reversed" order (which turns out to be more convenient for implementation). Finally, we also define $\vec{p}_m := \vec{p}_{m,1}$ and simply call it the powerful basis of $\mathcal{O}_m$.

Of special interest are the *two-power* cyclotomic rings, which have especially simple representations and are widely used in practical instantiations of lattice cryptography. When $m = 2^k \geq 2$ is a power of two, the mth cyclotomic polynomial is $\Phi_m(X) = X^n + 1$, where $n = \varphi(m) = 2^{k-1}$. Thus the 8th cyclotomic field is $K = \mathbb{Q}[X]/(X^4+1)$ and the

corresponding ring is $R = \mathbb{Z}[X]/(X^4+1)$. For this special case, the power basis is identical to the *powerful* basis $\vec{p}$ and the *"tweaked" decoding* basis $t \cdot \vec{d}$ of R as defined below.

Arbitrary cyclotomics. Now let m be any positive integer, and let $m = \prod_{\ell=1}^{t} m_\ell$ be its factorization into maximal prime-power divisors m_ℓ (in some canonical order). Then we can define

$$\mathcal{O}_m := \mathbb{Z}[\zeta_{m_1}, \zeta_{m_2}, \ldots, \zeta_{m_t}].^{1}$$

It is known that the rings $\mathbb{Z}[\zeta_\ell]$ are linearly disjoint over $\mathbb{Z}$, i.e., for any $\mathbb{Z}$-bases of the individual rings, their Kronecker product is a $\mathbb{Z}$-basis of $\mathcal{O}_m$. In particular, the *powerful basis* of $\mathcal{O}_m$ is defined as the Kronecker product of the component powerful bases:

$$\vec{p}_m := \bigotimes_\ell \vec{p}_{m_\ell}. \tag{2.2.2}$$

Similarly, for $m|m'$ having factorizations $m = \prod_\ell m_\ell$, $m' = \prod_\ell m'_\ell$, where each m_ℓ, m'_ℓ is a power of a distinct prime p_ℓ (so some m_ℓ may be 1), the *relative* powerful basis of $\mathcal{O}_{m'}/\mathcal{O}_m$ is

$$\vec{p}_{m',m} := \bigotimes_\ell \vec{p}_{m'_\ell, m_\ell}. \tag{2.2.3}$$

Notice that for $m|m'|m''$, we have that $\vec{p}_{m'',m}$ and $\vec{p}_{m'',m'} \otimes \vec{p}_{m',m}$ are equivalent *up to order*, because they are tensor products of the same components, but possibly in different orders.

2.2.4 Canonical Embedding

There are n distinct ring embeddings (i.e., injective ring homomorphisms) $\sigma_i \colon K \to \mathbb{C}$, indexed by $i \in \mathbb{Z}_m^*$, which are defined by $\sigma_i(\zeta_m) = \omega_m^i$ where $\omega_m = \exp(2\pi\sqrt{-1}/m) \in \mathbb{C}$ is the principal mth complex root of unity. These embeddings come in conjugate pairs (σ_i, σ_{m-i}), because ω_m^i is the complex conjugate of $\omega_m^{m-i} = \omega_m^{-i}$. The *canonical embedding* is the concatenation of all the embeddings (under a suitable reindexing of $\mathbb{Z}_m^*$ as $\{1, \ldots, n\}$),

[1] Equivalently, $\mathcal{O}_m = \bigotimes_\ell \mathcal{O}_{m_\ell}$ is the ring tensor product over $\mathbb{Z}$ of all the m_ℓth cyclotomic rings.

i.e., the injective function

$$\sigma \colon K \to H$$
$$\sigma(a) = (\sigma_i(a))_{i \in \mathbb{Z}_m^*}$$

where $H \subset \mathbb{C}^n$ is the subspace defined above in subsection 2.1.2.

We endow K and R with a geometry using the canonical embedding σ, i.e., all geometric quantities on K and R are defined in terms of the canonical embedding. For example, we define the ℓ_2 norm on K as $\|x\|_2 = \|\sigma(x)\|_2 = \sqrt{\langle \sigma(x), \sigma(x) \rangle}$, and use this to define the continuous Gaussian distribution D_r over K.[2] A key property is that both addition and multiplication in the ring are coordinate-wise in the canonical embedding:

$$\sigma(a + b) = \sigma(a) + \sigma(b)$$
$$\sigma(a \cdot b) = \sigma(a) \odot \sigma(b).$$

This property aids analysis and allows for sharp bounds on the growth of errors in cryptographic applications.

For two-power cyclotomics, this geometry is particulary simple: σ is just a scaling by a $\sqrt{n}$ factor, followed by a rigid rotation (an isometry). Therefore, a sample from the Gaussian distribution D_r over H (and over K, via σ^{-1}) has independent power-basis coefficients, drawn from $D_{r/\sqrt{n}}$.

2.2.5 (Tweaked) Decoding Basis

Ideal lattices. Recall that an ideal $\mathcal{I} \subseteq R$ is a nontrivial additive subgroup that is also closed under multiplication by R. The *norm* is defined as $\mathrm{N}(\mathcal{I}) := |R/\mathcal{I}|$, the index

[2] To be formal, the continuous Gaussian is defined over $K_{\mathbb{R}} := K \otimes_{\mathbb{Q}} \mathbb{R}$, which is analogous to K as the reals $\mathbb{R}$ are to the rationals $\mathbb{Q}$, and which is in bijective correspondence with H via the natural extension of σ. Because precision is always finite in any computational context, in this work we ignore the formal distinction between K and $K_{\mathbb{R}}$.

of $\mathcal{I}$ in R. A *fractional ideal* $\mathcal{J} \subset K$ is a set that can be expressed as $\mathcal{J} = d^{-1} \cdot \mathcal{I}$ for some ideal $\mathcal{I} \subseteq R$ and $d \in R$. (We sometimes omit the word "fractional" when it is clear from context.) Its norm is defined as $\mathrm{N}(\mathcal{J}) := \mathrm{N}(\mathcal{I})/\mathrm{N}(d)$. The fractional ideals form a group under multiplication (with R as the identity), where ideal multiplication is defined by $\mathcal{I}\mathcal{J} = \{\sum_i x_i y_i : x_i \in \mathcal{I}, y_i \in \mathcal{J}\}$. The norm map is then multiplicative: $\mathrm{N}(\mathcal{I}\mathcal{J}) = \mathrm{N}(\mathcal{I})\,\mathrm{N}(\mathcal{J})$.

Any (fractional) ideal $\mathcal{I}$ yields a lattice $\sigma(\mathcal{I}) \subset H$ under the canonical embedding. As usual, we often leave σ implicit and refer to $\mathcal{I}$ itself as a lattice. The following lower bound on the minimum distance of an ideal lattice is an immediate consequence of the arithmetic-mean/geometric-mean inequality.

Lemma 2.2.1. *For any fractional ideal $\mathcal{I} \subset K$, we have $\lambda_1(\mathcal{I}) \geq \sqrt{n} \cdot \mathrm{N}(\mathcal{I})^{1/n}$.*

The dual ideal, and a "tweak." Any fractional ideal $\mathcal{I} \subset K$ has a *dual* (fractional) ideal $\mathcal{I}^\vee$, which under the canonical embedding corresponds to (the complex conjugate of) the dual lattice of $\mathcal{I}$, i.e., $\sigma(\mathcal{I})$ and $\overline{\sigma(\mathcal{I}^\vee)}$ are duals. In particular, the dual ideal $R^\vee$ of R, also called the *codifferent* ideal, is defined as the dual of R under the trace, i.e.,

$$R^\vee := \{\text{fractional } a : \mathrm{Tr}_{R/\mathbb{Z}}(a \cdot R) \subseteq \mathbb{Z}\}.$$

The dual ideal $I^\vee$ is related to the inverse ideal via the codifferent: $\mathcal{I}^\vee = \mathcal{I}^{-1} R^\vee$. (See, e.g., [Con09] for further details and proofs.) By the composition property of the trace, $(R')^\vee$ is the set of all fractional a such that $\mathrm{Tr}_{R'/R}(a \cdot R') \subseteq R^\vee$. In particular, we have $\mathrm{Tr}_{R'/R}((R')^\vee) = R^\vee$.

Concretely, the dual ideal is the principal fractional ideal $R^\vee = (g_m/\hat{m})R$, where $\hat{m} = m/2$ if m is even and $\hat{m} = m$ otherwise, and the special element $g_m \in R$ is as follows:

- for $m = p^e$ for prime p and $e \geq 1$, we have $g_m = g_p := 1 - \zeta_p$ if p is odd, and $g_m = g_p := 1$ if $p = 2$;

- for $m = \prod_\ell m_\ell$ where the m_ℓ are powers of distinct primes, we have $g_m = \prod_\ell g_{m_\ell}$.

The dual ideal $R^\vee$ plays a very important role in the definition, hardness proofs, and cryptographic applications of Ring-LWE (see [LPR13b; LPR13a] for details). However, for implementations it seems preferable to work entirely in R, so that we do not to have to contend with fractional values or the dual ideal explicitly. Following [AP13], we achieve this by multiplying all values related to $R^\vee$ by the "tweak" factor $t_m = \hat{m}/g_m \in R$; recall that $t_m R^\vee = R$. To compensate for this implicit tweak factor, we replace the trace by what we call the *twace* (for "tweaked trace") function $\mathrm{Tw}_{m',m} = \mathrm{Tw}_{R'/R} \colon R' \to R$, defined as

$$\mathrm{Tw}_{R'/R}(x) := t_m \cdot \mathrm{Tr}_{R'/R}(x/t_{m'}) = (\hat{m}/\hat{m}') \cdot \mathrm{Tr}_{R'/R}(x \cdot g_{m'}/g_m). \tag{2.2.4}$$

A nice feature of the twace is that it fixes the base ring pointwise, i.e., $\mathrm{Tw}_{R'/R}(x) = x$ for every $x \in R$. It is also easy to verify that it satisfies the same composition property that the trace does.

We stress that this "tweaked" perspective is mathematically and computationally equivalent to using $R^\vee$, and all the results from [LPR13b; LPR13a] can translate to this setting without any loss.

Decoding Basis. The work of [LPR13a] defines a certain $\mathbb{Z}$-basis $\vec{b}_m = (b_j)$ of $R^\vee$, called the *decoding* basis. It is defined as the dual of the conjugated powerful basis $\vec{p}_m = (p_j)$ under the trace:

$$\mathrm{Tr}_{R/\mathbb{Z}}(b_j \cdot p_{j'}^{-1}) = \delta_{j,j'}$$

for all j, j'. The key geometric property of the decoding basis is, informally, that the $\mathbb{Z}$-coefficients of any $e \in R^\vee$ with respect to $\vec{b}_m$ are optimally small in relation to $\sigma(x)$, the canonical embedding of e. In other words, short elements like Gaussian errors have small decoding-basis coefficients.

With the above-described "tweak" that replaces $R^\vee$ by R, we get the $\mathbb{Z}$-basis

$$\vec{d}_m = (d_j) := t_m \cdot \vec{b}_m \ ,$$

which we call the (tweaked) decoding basis of R. By definition, this basis is dual to the conjugated powerful basis $\vec{p}_m$ under the *twace*:

$$\mathrm{Tw}_{R/\mathbb{Z}}(d_j \cdot p_{j'}^{-1}) = \delta_{j,j'}.$$

Because $g_m \cdot t_m = \hat{m}$, it follows that the coefficients of any $e \in R$ with respect to $\vec{d}_m$ are identical to those of $g_m \cdot e \in g_m R = \hat{m} R^\vee$ with respect to the $\mathbb{Z}$-basis $g_m \cdot \vec{d}_m = \hat{m} \cdot \vec{b}_m$ of $g_m R$. Hence, they are optimally small in relation to $\sigma(g_m \cdot e)$.[3]

Relative decoding basis. Generalizing the above, the *relative* decoding basis $\vec{d}_{m',m}$ of R'/R is dual to the (conjugated) relative powerful basis $\vec{p}_{m',m}$ under $\mathrm{Tw}_{R'/R}$. As such, $\vec{d}_{m',m}$ (and in particular, $\vec{d}_{m'}$ itself) has a Kronecker-product structure mirroring that of $\vec{p}_{m',m}$ from Equations (2.2.1) and (2.2.3). Furthermore, by the results of [LPR13a, Section 6], for a positive power m of a prime p we have

$$\vec{d}^t_{m,m/p} = \begin{cases} \vec{p}^t_{m,m/p} \cdot L_p & \text{if } m = p \\ \vec{p}^t_{m,m/p} & \text{otherwise,} \end{cases} \tag{2.2.5}$$

where L_p is the lower-triangular matrix with 1s throughout its lower triangle.

[3] This is why Invariant 4.2.2 of our somewhat-homomorphic encryption scheme (section 4.3) requires $\sigma(e \cdot g_m)$ to be short, where e is the error in the ciphertext.

2.2.6 Chinese Remainder Bases

This section contains a relatively brief summary of the Chinese Remainder sets and bases used throughout this thesis; see [LPR13b; LPR13a] for many more mathematical and computational details.

Prime splitting. As usual, let R denote the mth cyclotomic ring and $n = \varphi(m)$. Let $p \in \mathbb{Z}$ be a prime integer, which for simplicity we assume does not divide m. The factorization of the ideal pR into prime ideals is as follows. Let d be the order of p modulo m, i.e., the smallest positive integer such that $p^d = 1 \pmod m$, and note that $d \mid n$. Let $\langle p \rangle = \{1, p, p^2, \ldots, p^{d-1}\} \subseteq \mathbb{Z}_m^*$ denote the multiplicative subgroup generated by p. Then pR factors as

$$pR = \prod_i \mathfrak{p}_i \ ,$$

where the $\mathfrak{p}_i$ are indexed by the quotient group $G = \mathbb{Z}_m^*/\langle p \rangle$, i.e., the multiplicative group of cosets $i\langle p \rangle$ of the subgroup $\langle p \rangle$ of $\mathbb{Z}_m^*$. These are called the prime ideals *lying over* p in R, and their number n/d is called the *splitting number* of p in R.

Concretely, the ideals lying over p are as follows: let ω_m be some arbitrary element of order m in the finite field $\mathbb{F}_{p^d}$. (Such an element exists because the multiplicative group $\mathbb{F}_{p^d}^*$ is cyclic and has order $p^d - 1 = 0 \pmod m$.) For each $i \in \mathbb{Z}_m^*$, define a ring homomorphism $h_i \colon R \to \mathbb{F}_{p^d}$ by $h_i(\zeta_m) = \omega_m^i$. Then the prime ideal $\mathfrak{p}_I$ corresponding to the coset I of $\langle p \rangle$ is the kernel of the homomorphism h_i, where $i \in I$ denotes some arbitrary element of the coset. It is easy to verify that this is an ideal, and that it is invariant under the choice of representative, because $h_{i \cdot p}(r) = h_i(r)^p$ for any $r \in R$ since $\mathbb{F}_{p^d}$ has characteristic p and therefore $(a + b)^p = a^p + b^p$ for any $a, b \in \mathbb{F}_{p^d}$. Because $\mathfrak{p}_I$ is the kernel of h_i, the induced ring homomorphisms $h_i \colon R/\mathfrak{p}_I \to \mathbb{F}_{p^d}$ for all $i \in I$ are in fact isomorphisms.

The Chinese Remainder Theorem states (in particular) that the natural ring homomorphism from $R_p := R/pR$ to the product ring $\prod_I (R/\mathfrak{p}_I) \cong (\mathbb{F}_{p^d})^{n/d}$, where I ranges

over all cosets of $\langle p \rangle \subseteq \mathbb{Z}_m^*$, is a ring isomorphism. In particular, the concatenation $h = (h_i)_{i \in \mathbb{Z}_m^*} : R_p \to \mathbb{F}_{2^d}^n$ is a ring embedding (an injective ring homomorphism). We refer to the set $\vec{c} = \{c_I\} \subset R_p$, where $c_I = 1 \pmod{\mathfrak{p}_I}$ and $c_I = 0 \pmod{\mathfrak{p}_J}$ for all cosets $I \neq J$ of $\langle p \rangle$, as the *mod-p CRT set* of R. In particular, for the CRT set $\vec{c}$ of R_p, for any $z \in R_p$ we have

$$\mathrm{Tr}_{R_p/\mathbb{Z}_p}(z \cdot \vec{c}) = \mathrm{Tr}_{\mathbb{F}_{p^d}/\mathbb{F}_p}(h(z)). \tag{2.2.6}$$

Similarly, for a prime power p^ℓ the natural ring homomorphism from R_{p^ℓ} to $\prod_I (R/\mathfrak{p}_I^\ell)$ is a ring isomorphism, and the mod-p^ℓ CRT set is defined analogously.

Finally, consider the general case where p may divide m. It turns out that this case easily reduces to the one where p does not divide m, as follows. Let $m = p^k \cdot \bar{m}$ for $p \nmid \bar{m}$, and let $\bar{R} = \mathcal{O}_{\bar{m}}$ and $p\bar{R} = \prod_i \bar{\mathfrak{p}}_i$ be the prime-ideal factorization of $p\bar{R}$ as described above. Then the ideals $\bar{\mathfrak{p}}_i \subset \bar{R}$ are *totally ramified* in R, i.e., we have $\bar{\mathfrak{p}}_i R = \mathfrak{p}_i^{\varphi(m)/\varphi(\bar{m})}$ for some distinct prime ideals $\mathfrak{p}_i \subset R$. This implies that the CRT set for R_p is *exactly* the CRT set for $\bar{R}_p$, embedded into R_p. Therefore, in what follows we restrict our attention to the case where p does not divide m.

CRT Set Extensions. As above, let p be a prime integer not dividing m, let p have order $d' | n'$ in $\mathbb{Z}_{m'}^*$, and let $\mathfrak{p}_{I'} \subset R'$ be the prime ideals lying over p in R', where I' ranges over the cosets of $\langle p \rangle \subseteq \mathbb{Z}_{m'}^*$. Then each $\mathfrak{p}_I$ lies over exactly one $\mathfrak{p}_{I'}$, i.e., it is a divisor of exactly one ideal $\mathfrak{p}_{I'} R$, namely, the one for which $I' = I \bmod m'$. Therefore, there are exactly $(n/d)/(n'/d')$ prime ideals lying over each $\mathfrak{p}_{I'}$; this number is called the *relative splitting number* of p in the extension R/R'.

CRT Basis When the order of p modulo m is 1, pR factors into n distinct prime ideals, and the mod-p CRT set becomes a *Chinese remainder* (or *CRT*) $\mathbb{Z}_p$-basis $\vec{c} = \vec{c}_m \in R_p^{\varphi(m)}$, whose entries are indexed by $\mathbb{Z}_m^*$. This happens precisely when p is a prime congruent to

$1 \pmod m$. The key property satisfied by this basis is

$$c_i \cdot c_{i'} = \delta_{i,i'} \cdot c_i \tag{2.2.7}$$

for all $i, i' \in \mathbb{Z}_m^*$. Therefore, multiplication of ring elements represented in the CRT basis is coefficient-wise (and hence linear time): for any coefficient vectors $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_p^{\varphi(m)}$, we have

$$(\vec{c}^t \cdot \mathbf{a}) \cdot (\vec{c}^t \cdot \mathbf{b}) = \vec{c}^t \cdot (\mathbf{a} \odot \mathbf{b}).$$

Also by Equation (2.2.7), the matrix corresponding to multiplication by c_i (with respect to the CRT basis) has one in the ith diagonal entry and zeros everywhere else, so the trace of every CRT basis element is unity: $\mathrm{Tr}_{R/\mathbb{Z}}(\vec{c}) = \mathbf{1} \pmod p$. For completeness, in what follows we describe the explicit construction of the CRT basis.

Arbitrary cyclotomics. For an arbitrary index m, the CRT basis is defined in terms of the prime-power factorization $m = \prod_{\ell=1}^{t} m_\ell$. Recall that $R_p = \mathbb{Z}_p[\zeta_{m_1}, \ldots, \zeta_{m_t}]$, and that the natural homomorphism $\phi\colon \mathbb{Z}_m^* \to \prod_\ell \mathbb{Z}_{m_\ell}^*$ is a group isomorphism. Using this, we can equivalently index the CRT basis by $\prod_\ell \mathbb{Z}_{m_\ell}^*$. With this indexing, the CRT basis $\vec{c}_m$ of R_p is the Kronecker product of the CRT bases $\vec{c}_{m_\ell}$ of $\mathbb{Z}_p[\zeta_{m_\ell}]$:

$$\vec{c}_m = \bigotimes_\ell \vec{c}_{m_\ell},$$

i.e., the $\phi(i)$th entry of $\vec{c}_m$ is the product of the $\phi(i)_\ell$th entry of $\vec{c}_{m_\ell}$, taken over all ℓ. It is easy to verify that Equation (2.2.7) holds for $\vec{c}_m$, because it does for all the $\vec{c}_{m_\ell}$.

Prime-power cyclotomics. Now let m be a positive power of a prime p, and let $\omega_m \in \mathbb{Z}_p^*$ be an element of order m (i.e., a primitive mth root of unity), which exists because $\mathbb{Z}_p^*$ is a cyclic group of order $p-1$, which is divisible by m. We rely on two standard facts:

1. the Kummer-Dedekind Theorem, which implies that the ideal $pR = \prod_{i \in \mathbb{Z}_m^*} \mathfrak{p}_i$ factors into the product of $\varphi(m)$ distinct prime ideals $\mathfrak{p}_i = (\zeta_m - \omega_m^i)R + pR \subset R$; and

2. the Chinese Remainder Theorem (CRT), which implies that the natural homomorphism from R_p to the product ring $\prod_{i \in \mathbb{Z}_m^*} R/\mathfrak{p}_i$ is a ring isomorphism.

Using this isomorphism, the basis $\vec{c}_m$ is defined so that its ith entry $c_i \in R_p$ satisfies $c_i = \delta_{i,i'} \pmod{\mathfrak{p}_{i'}}$ for all $i, i' \in \mathbb{Z}_m^*$. Observe that this definition clearly satisfies Equation (2.2.7).

Like the powerful and decoding bases, for any extension R'_p/R_p where $R' = \mathcal{O}_{m'}$, $R = \mathcal{O}_m$ for powers $m|m'$ of p, there is a *relative* CRT R_p-basis $\vec{c}_{m',m}$ of R'_p, which has a Kronecker-product factorization mirroring the one in Equation (2.2.1). The elements of this R_p-basis satisfy Equation (2.2.7), and hence their traces into R_p are all unity.

2.2.7 Computational Problems for Cyclotomic Rings

Learning with Errors Ring-Learning-With-Errors (Ring-LWE) is a family of computational problems that was defined and analyzed in [LPR13b; LPR13a]. Those works use a form of Ring-LWE involving the dual ideal $R^\vee$. Formally, for an integer $q > 1$ defining $R_q := R/qR$, a secret $s \in R^\vee$, and an error distribution ψ over $K_\mathbb{R}$, R-LWE is a *distribution* $A_{q,s,\psi}$ where samples are generated by choosing a uniformly random $a \in R_q$, $e \leftarrow \psi$, and outputting

$$(a, b = a \cdot s + e \bmod qR^\vee) \in R_q \times (K_\mathbb{R}/(qR^\vee)).$$

Typically, ψ is either a continuous spherical Gaussian or its discretization to $R^\vee$; these respectively give us *continuous* (where $b_i \in K/qR^\vee$) and *discrete* (where $b_i \in R_q^\vee$) forms of the following problems:

Search-Ring-LWE The *search* problem is to recover the secret s given polynomially many samples from $A_{q,s,\psi}$.

Decision-Ring-LWE The decision problem is to distinguish (with non-negligible advantage in n) between independent samples from $A_{q,s,\psi}$ and uniformly random samples.

It is often more convenient for implementations to use an equivalent form of Ring-LWE that does not involve $R^\vee$. As first suggested in [AP13], this can be done with no loss in security or efficiency by working with an equivalent "tweaked" form of the problem, which is obtained by multiplying the noisy products b_i by the tweak factor $t = t_m = \hat{m}/g_m \in R$. Doing so yields new noisy products

$$b'_i := t \cdot b_i = a_i \cdot (t \cdot s) + (t \cdot e_i) = a_i \cdot s' + e'_i \bmod qR,$$

where both a_i and $s' = t \cdot s$ reside in R/qR, and the error terms $e'_i = t \cdot e_i$ come from the "tweaked" distribution $t \cdot \psi$. Note that when ψ corresponds to a spherical Gaussian (in the canonical embedding), its tweaked form $t \cdot \psi$ may be *highly non-spherical*, but this is not a problem: the tweaked form of Ring-LWE is entirely equivalent to the above one involving $R^\vee$, because the tweak is reversible.

In this paper, our exposition primarily uses the original form of Ring-LWE involving $R^\vee$, so that we can use sharp concentration bounds on spherical Gaussians. Our implementations, however, uses the tweaked form, where equivalent bounds follow by $\|g \cdot e'\| = \|g \cdot t \cdot e\| = \hat{m} \cdot \|e\|$, where e is the original error term and $e' = t \cdot e$ is its tweaked counterpart.

Learning with Rounding Ring-Learning-With-Rounding (Ring-LWR) is closely related to Ring-LWE. It replaces the error term from a distribution with deterministic roundoff error. For two integers $q \geq p \geq 2$ and a secret $s \in R_q$, R-LWR is a distribution $L_{s,q,p}$ over $R_q \times R_p$. Samples are obtained by sampling a uniformly random $a \in R_q$ and outputting

$$(a, b = \lfloor a \cdot s \rceil_p).$$

Like Ring-LWE, there is a search and decision variant of Ring-LWR , defined analogously.

2.3 Haskell Background

In this section we give a brief primer on the basic syntax, concepts, and features of Haskell needed to understand the material in the rest of the paper. For further details, see the excellent tutorial [Lip11].

2.3.1 Types

Every well-formed Haskell expression has a particular *type*, which is known statically (i.e., at compile time). An expression's type can be explicitly specified by a *type signature* using the `::` symbol, e.g., `3 :: Integer` or `True :: Bool`. However, such low-level type annotations are usually not necessary, because Haskell has very powerful *type inference*, which can automatically determine the types of arbitrarily complex expressions (or declare that they are ill-typed).

Every *function*, being a legal expression, has a type, which is written by separating the types of the input(s) and the output with the arrow `->` symbol, e.g., `xor :: Bool -> Bool -> Bool`. Functions can be either fully or only partially applied to arguments having the appropriate types, e.g., we have the expressions `xor False False :: Bool` and `xor True :: Bool -> Bool`, but not the ill-typed `xor 3`. Partial application works because `->` is right-associative, so the "true" type of `xor` is `Bool -> (Bool -> Bool)`, i.e., it takes a boolean as input and outputs a *function* that itself maps a boolean to a boolean. Functions can also take functions as *inputs*, e.g.,

```
selfCompose :: (Integer -> Integer) -> (Integer -> Integer)
```

takes any `f :: Integer -> Integer` as input and outputs another function (presumably representing `f` ∘ `f`).

The names of *concrete* types, such as `Integer` or `Bool`, are always capitalized. This is in contrast with lower-case *type variables*, which can stand for any type (possibly subject to some constraints; see the next subsection). For example, the function `alwaysTrue :: a`

`-> Bool` takes a value of any type, and outputs a boolean value (presumably `True`). More interestingly, `cons :: a -> [a] -> [a]` takes a value of any type, and a *list* of values all having that *same* type, and outputs a list of values of that type.

Types can be parameterized by other types. For example:

- The type `[]` seen just above is the generic "(ordered) list" type, whose single argument is the type of the listed values, e.g., `[Bool]` is the "list of booleans" type. (Note that `[a]` is just syntactic sugar for `[] a`.)
- The type `Maybe` represents "either a value (of a particular type), or nothing at all;" the latter is typically used to signify an exception. Its single argument is the underlying type, e.g., `Maybe Integer`.
- The generic "pair" type `(,)` takes two arguments that specify the types being paired together, e.g., `(Integer,Bool)`.

Only fully applied types can admit values, e.g., there are no values of type `[]`, `Maybe`, or `(Integer,)`.

2.3.2 Type Classes

Type classes, or just *classes*, define abstract interfaces that types can implement, and are therefore a primary mechanism for obtaining polymorphism. For example, the `Additive` class (from the numeric prelude [TTJ15]) represents types that form abelian additive groups. As such, it introduces the terms[4]

```
zero     :: Additive a => a
negate   :: Additive a => a -> a
(+), (-) :: Additive a => a -> a -> a
```

[4]Operators like `+`, `-`, `*`, `/`, and `==` are merely functions introduced by various type classes. Function names consisting solely of special characters can be used in infix form in the expected way, but in all other contexts they must be surrounded by parentheses.

In type signatures like the ones above, the text preceding the `=>` symbol specifies the *class constraint(s)* on the type variable(s). The constraints `Additive a` seen above simply mean that the type represented by `a` must be an *instance* of the `Additive` class. A type is made an instance of a class via an *instance declaration*, which simply defines the actual behavior of the class's terms for that particular type. For example, `Integer` and `Double` are instances of `Additive`. While `Bool` is not, it could be made one via the instance declaration

```
instance Additive Bool where
  zero   = False
  negate = id
  (+)    = xor    -- same for (-)
```

Using class constraints, one can write polymorphic expressions using the terms associated with the corresponding classes. For example, we can define `double :: Additive a => a -> a` as `double x = x + x`. The use of `(+)` here is legal because the input `x` has type `a`, which is constrained to be an instance of `Additive` by the type of `double`. As a slightly richer example, we can define

```
isZero :: (Eq a, Additive a) => a -> Bool
isZero x = x == zero
```

where the class `Eq` introduces the function `(==) :: Eq a => a -> a -> Bool` to represent types whose values can be tested for equality.[5]

The definition of a class `C` can declare other classes as *superclasses*, which means that any type that is an instance of `C` must also be an instance of each superclass. For example, the class `Ring` from numeric prelude, which represents types that form rings with identity, has `Additive` as a superclass; this is done by writing `class Additive r => Ring r` in the class definition.[6] One advantage of superclasses is that they help reduce the complexity

[5]Notice the type inference here: the use of `(==)` means that `x` and `zero` must have the *same* type `a` (which must be an instance of `Additive`), so there is no ambiguity about which implementation of `zero` to use.

[6]It is generally agreed that the arrow points in the wrong direction, but for historical reasons we are stuck with this syntax.

of class constraints. For example, we can define `f :: Ring r => r -> r` as `f x = one + double x`, where the term `one :: Ring r => r` is introduced by **Ring**, and `double` is as defined above. The use of `(+)` and `double` is legal here, because `f`'s input `x` has type `r`, which (by the class constraint on `f`) is an instance of **Ring** and hence also of **Additive**.

So far, the discussion has been limited to *single-parameter* classes: a type either is, or is not, an instance of the class. In other words, such a class can be seen as merely the *set* of its instance types. More generally, *multi-parameter* classes express *relations* among types. For example, the two-argument class definition `class (Ring r, Additive a) => Module r a` represents that the additive group `a` is a module over the ring `r`, via the scalar multiplication function `(*>) :: Module r a => r -> a -> a`.

CHAPTER 3

$\Lambda \circ \lambda$: FUNCTIONAL LATTICE CRYPTOGRAPHY

Recent theoretical improvements in lattice cryptography have paved the way for the *practical implementation* of lattice/ring-based schemes, with many impressive results. To date, each such implementation has been specialized to a particular cryptographic primitive (and sometimes even to a specific computational platform), e.g., collision-resistant hashing (using SIMD instruction sets) [Lyu+08], digital signatures [GLP12; Duc+13], key-establishment protocols [Bos+15; Alk+16; Bos+16b], and homomorphic encryption (HE) [NLV11; HS] (using GPUs and FPGAs [Wan+12; Cou+14]), to name a few.

However, the state of lattice cryptography implementations is also highly *fragmented*: they are usually focused on a single cryptosystem for fixed parameter sets, and have few reusable interfaces, making them hard to implement other primitives upon. Those interfaces that do exist are quite *low-level*; e.g., they require the programmer to explicitly convert between various representations of ring elements, which calls for specialized expertise and can be error prone. Finally, prior implementations either do not support, or use suboptimal algorithms for, the important class of *arbitrary cyclotomic* rings, and thereby lack related classes of HE functionality. (See subsection 3.1.4 for a more detailed review of related work.)

With all this in mind, we contend that there is a need for a *general-purpose*, *high-level*, and *feature-rich framework* that will allow researchers to more easily implement and experiment with the wide variety of lattice-based cryptographic schemes, particularly more complex ones like HE.

3.1 Contributions

This work describes the design, implementation, and evaluation of $\Lambda\circ\lambda$, a general-purpose framework for lattice-based cryptography in the compiled, functional, strongly typed programming language Haskell.[1,2] Our primary goals for $\Lambda\circ\lambda$ include: (1) the ability to implement both basic and advanced lattice cryptosystems correctly, concisely, and at a high level of abstraction; (2) alignment with the current best theory concerning security and algorithmic efficiency; and (3) acceptable performance on commodity CPUs, along with the capacity to integrate specialized backends (e.g., GPUs) without affecting application code.

3.1.1 Novel Attributes of $\Lambda\circ\lambda$

The $\Lambda\circ\lambda$ framework has several novel properties that distinguish it from prior lattice-crypto implementations.

Generality, modularity, and concision: $\Lambda\circ\lambda$ defines a collection of simple, modular interfaces and implementations for the lattice cryptography "toolbox," i.e., the collection of operations that are used across a wide variety of modern cryptographic constructions. This generality allows cryptographic schemes to be expressed very naturally and concisely, via code that closely mirrors their mathematical definitions. For example in chapter 4, we implement a full-featured SHE scheme (which includes never-before-implemented functionality) in as few as 2–5 lines of code per feature.

While $\Lambda\circ\lambda$'s interfaces are general enough to support most modern lattice-based cryptosystems, our main focus (as with most prior implementations) is on systems defined over *cyclotomic rings*, because they lie at the heart of practically efficient lattice-based cryptography (see, e.g., [HPS98; Mic07; LPR13b; LPR13a]). However, while almost all prior

[1] The name $\Lambda\circ\lambda$ refers to the combination of lattices and functional programming, which are often signified by Λ and λ, respectively. The recommended pronunciation is "L O L."

[2] $\Lambda\circ\lambda$ is available under the free and open-source GNU GPL2 license. It can be installed from Hackage, the Haskell community's central repository, via `stack install lol`. The source repository is also available at `https://github.com/cpeikert/Lol`.

implementations are limited to the narrow subclass of *power-of-two* cyclotomics (which are the algorithmically simplest case), $\Lambda \circ \lambda$ supports *arbitrary* cyclotomic rings. Such support is essential in a general framework, because many advanced techniques in ring-based cryptography, such as "plaintext packing" and homomorphic SIMD operations [SV10; SV14], inherently require non-power-of-two cyclotomics when using characteristic-two plaintext spaces (e.g., $\mathbb{F}_{2^k}$).

Theory affinity: $\Lambda \circ \lambda$ is designed from the ground-up around the specialized ring representations, fast algorithms, and worst-case hardness proofs developed in [LPR13b; LPR13a] for the design and analysis of ring-based cryptosystems (over arbitrary cyclotomic rings), particularly those relying on Ring-LWE. To our knowledge, $\Lambda \circ \lambda$ is the first-ever implementation of these techniques, which include:

- fast and modular algorithms for converting among the three most useful representations of ring elements, corresponding to the *powerful*, *decoding*, and *Chinese Remainder Theorem (CRT)* bases;
- fast algorithms for sampling from "theory-recommended" error distributions—i.e., those for which the Ring-LWE problem has provable worst-case hardness—for use in encryption and related operations;
- proper use of the powerful- and decoding-basis representations to maintain tight control of error growth under cryptographic operations, and for the best error tolerance in decryption.

We especially emphasize the importance of using appropriate error distributions for Ring-LWE, because ad-hoc instantiations with narrow error can be completely broken by certain attacks [Eli+15; CLS15; CIV16], whereas theory-recommended distributions are provably immune to the same class of attacks [Pei16].

In addition, $\Lambda \circ \lambda$ is the first lattice cryptography implementation to expose the rich *hierarchy* of cyclotomic rings, making subring and extension-ring relationships accessible

to applications. In particular, $\Lambda\circ\lambda$ support the homomorphic operations known as *ring-switching* [BGV14; Gen+13; AP13], which enables efficient homomorphic evaluation of certain structured linear transforms. Ring-switching has multiple applications, such as ciphertext compression [BGV14; Gen+13] and asymptotically efficient "bootstrapping" algorithms for FHE [AP13].

Safety: Building on its host language Haskell, $\Lambda\circ\lambda$ has several facilities for reducing programming errors and code complexity, thereby aiding the *correct* implementation of lattice cryptosystems. This is particularly important for advanced constructions like HE, which involve a host of parameters, mathematical objects, and algebraic operations that must satisfy a variety of constraints for the scheme to work as intended.

More specifically, $\Lambda\circ\lambda$ uses advanced features of Haskell's type system to *statically enforce* (i.e., at compile time) a variety of mathematical constraints. This catches many common programming errors early on, and guarantees that any execution will perform only legal operations.[3] For example, $\Lambda\circ\lambda$ represents integer moduli and cyclotomic indices as specialized *types*, which allows it to statically enforce that all inputs to modular arithmetic operations have the same modulus, and that to embed from one cyclotomic ring to another, the former must be a subring of the latter. We emphasize that representing moduli and indices as types does not require fixing their values at compile time; instead, one can (and we often do) *reify* runtime values into types, checking any necessary constraints just once at reification.

Additionally, $\Lambda\circ\lambda$ aids safety by defining *high-level abstractions* and *narrow interfaces* for algebraic objects and cryptographic operations. For example, it provides an abstract data type for cyclotomic rings, which hides its choice of internal representation (powerful or CRT basis, subring element, etc.), and automatically performs any necessary conversions. Moreover, it exposes only high-level operations like ring addition and multiplication, bit decomposition, sampling uniform or Gaussian ring elements, etc.

[3] A popular joke about Haskell code is "if you can get it to compile, it must be correct."

Finally, Haskell itself also greatly aids safety because computations are by default *pure*: they cannot mutate state or otherwise modify their environment. This makes code easier to reason about, test, or even formally verify, and is a natural fit for algebra-intensive applications like lattice cryptography. We stress that "effectful" computations like input/output or random number generation are still possible, but must be embedded in a structure that precisely delineates what effects are allowed.

Multiple backends: $\Lambda\circ\lambda$'s architecture sharply separates its interface of cyclotomic ring operations from the implementations of their corresponding linear transforms. This allows for multiple "backends,", e.g., based on specialized hardware like GPUs or FPGAs via tools like [Cha+11], without requiring any changes to cryptographic application code. (By contrast, prior implementations exhibit rather tight coupling between their application and backend code.) We have implemented two interchangeable backends, one in the pure-Haskell Repa array library [Kel+10; Lip+12], and one in C++.

3.1.2 Other Technical Contributions

Our work on $\Lambda\circ\lambda$ has also led to several technical novelties of broader interest and applicability.

Abstractions for lattice cryptography. As already mentioned, $\Lambda\circ\lambda$ defines *composable* abstractions and algorithms for widely used lattice operations, such as *rounding* (or rescaling) $\mathbb{Z}_q$ to another modulus, *(bit) decomposition*, and other operations associated with "gadgets" (including in "Chinese remainder" representations). Prior works have documented and/or implemented subsets of these operations, but at lower levels of generality and composability. For example, we derive generic algorithms for all the above operations on *product rings*, using any corresponding algorithms for the component rings. And we show how to generically "promote" these operations on $\mathbb{Z}$ or $\mathbb{Z}_q$ to arbitrary cyclotomic rings. Such

modularity makes our code easier to understand and verify, and is also pedagogically helpful to newcomers to the area.

DSL for sparse decompositions. As shown in [LPR13a] and further in this work, most cryptographically relevant operations on cyclotomic rings correspond to linear transforms having *sparse decompositions*, i.e., factorizations into relatively sparse matrices, or tensor products thereof. Such factorizations directly yield fast and highly parallel algorithms; e.g., the Cooley-Tukey FFT algorithm arises from a sparse decomposition of the Discrete Fourier Transform.

To concisely and systematically implement the wide variety of linear transforms associated with general cyclotomics, $\Lambda \circ \lambda$ includes an embedded *domain-specific language* (DSL) for expressing sparse decompositions using natural matrix notation, and a "compiler" that produces corresponding fast and parallel implementations. This compiler includes generic combinators that "lift" any class of transform from the primitive case of prime cyclotomics, to the prime-power case, and then to arbitrary cyclotomics. (See section 3.4 for details.)

Algorithms for the cyclotomic hierarchy. Recall that $\Lambda \circ \lambda$ is the first lattice cryptography implementation to expose the rich hierarchy of cyclotomic rings, i.e., their subring and extension-ring relationships. As the foundation for this functionality, in section 3.3 we derive sparse decompositions for a variety of objects and linear transforms related to the cyclotomic hierarchy. In particular, we obtain simple linear-time algorithms for the *embed* and *"tweaked" trace* operations in the three main bases of interest (powerful, decoding, and CRT), and for computing the *relative* analogues of these bases for cyclotomic extension rings. To our knowledge, almost all of this material is new. (For comparison, the Ring-LWE "toolkit" [LPR13a] deals almost entirely with transforms and algorithms for a *single* cyclotomic ring, not inter-ring operations.)

3.1.3 Limitations and Future Work

Security. While $\Lambda\circ\lambda$ has many attractive functionality and safety features, we stress that it is still an early-stage research prototype, and is not yet recommended for production purposes—especially in scenarios requiring high security assurances. Potential issues include, but may not be limited to:

- Most functions in $\Lambda\circ\lambda$ are not constant time, and may therefore leak secret information via timing or other side channels. (Systematically protecting lattice cryptography from side-channel attacks is an important area of research.)

- While $\Lambda\circ\lambda$ implements a fast algorithm for sampling from theory-recommended error distributions, the current implementation is somewhat naïve in terms of precision. By default, some $\Lambda\circ\lambda$ functions use double-precision floating-point arithmetic to approximate a sample from a continuous Gaussian, before rounding. (But one can specify an alternative data type having more precision.) We have not yet analyzed the associated security implications, if any. We do note, however, that Ring-LWE is robust to small variations in the error distribution (see, e.g., [LPR13b, Section 5]).

Discrete Gaussian sampling. Many lattice-based cryptosytems, such as digital signatures and identity-based or attribute-based encryption schemes following [GPV08], require sampling from a *discrete Gaussian* probability distribution over a given lattice coset, using an appropriate kind of "trapdoor." Supporting this operation in $\Lambda\circ\lambda$ is left to future work, for the following reasons. While it is straightforward to give a clean *interface* for discrete Gaussian sampling (similar to the `Decompose` class described in subsection 3.2.4), providing a secure and practical *implementation* is very subtle, especially for arbitrary cyclotomic rings: one needs to account for the non-orthogonality of the standard bases, use practically efficient algorithms, and ensure high statistical fidelity to the desired distribution using finite precision. Although there has been good progress in addressing these issues at the

theoretical level (see, e.g., [DN12; LPR13a; DP15a; DP15b]), a complete practical solution still requires further research.

Applications. Our focus in this chapter is mainly on the $\Lambda \circ \lambda$ framework itself. We provide two reference implementations in other chapters: chapter 4 has an implementation of somewhat-homomorphic encryption [BGV14], and section 6.6 includes the weak pseudorandom function from [BPR12]. We leave further implementations of lattice-based cryptosystems with $\Lambda \circ \lambda$ for future work. While digital signatures and identity/attribute-based encryption use discrete Gaussian sampling, many other primitives should be straightforward to implement using $\Lambda \circ \lambda$'s existing functionality. These include standard Ring-LWE-based [LPR13b; LPR13a] and NTRU-style encryption [HPS98; SS11], public-key encryption with security under chosen-ciphertext attacks [MP12], and strong pseudorandom functions (PRFs) [BPR12; Bon+13; BP14].

3.1.4 Comparison to Related Work

As mentioned above, there are many implementations of various lattice- and ring-based cryptographic schemes, such as NTRU (Prime) encryption [HPS98; Ber+16], the SWIFFT hash function [Lyu+08], digital signature schemes like [GLP12] and BLISS [Duc+13], key-exchange protocols [Bos+15; Alk+16; Bos+16b], and HE libraries like HElib [HS]. In addition, there are some high-performance backends for power-of-two cyclotomics, like NFLlib [Mel+16] and [Wan+12], which can potentially be plugged into these other systems. Also, in a Masters thesis developed concurrently with this work, Mayer [May16] implemented the "toolkit" algorithms from [LPR13a] for arbitrary cyclotomic rings (though not the inter-ring operations that $\Lambda \circ \lambda$ supports).

On the whole, the prior works each implement just one cryptographic primitive (sometimes even on a specific computational platform), and typically opt for performance over generality and modularity. In particular, none of them provide any abstract data types

for cyclotomic rings, but instead require the programmer to explicitly manage the representations of ring elements (e.g., as polynomials) and ensure that operations on them are mathematically meaningful. Moreover, with the exception of [May16], they do not support general cyclotomic rings using the current best theory for cryptographic purposes.

HElib. Our work compares most closely to HElib [HS], which is an "assembly language" for BGV-style HE over cyclotomic rings [BGV14]. It holds speed records for a variety of HE benchmarks (e.g., homomorphic AES computation [GHS12c]), and appears to be the sole public implementation of many advanced HE features, like bootstrapping for "packed" ciphertexts [HS15].

On the downside, HElib does not use the best known algorithms for cryptographic operations in general (non-power-of-two) cyclotomics. Most significantly, it uses the *univariate* representation modulo cyclotomic polynomials, rather than the multivariate/tensored representations from [LPR13a], which results in more complex and less efficient algorithms, and suboptimal noise growth in cryptographic schemes. The practical effects of this can be seen in our performance evaluation (subsection 4.4.2), which shows that $\Lambda \circ \lambda$'s C++ backend is about nine times slower than HElib for power-of-two cyclotomics, but is significantly *faster* (by factors of two or more) for indices involving two or more small primes. Finally, HELib is targeted toward just one class of cryptographic construction (HE), so it lacks functionality necessary to implement a broader selection of lattice schemes (e.g., CCA-secure encryption).

Computational algebra systems. Algebra packages like Sage and Magma provide very general-purpose support for computational number theory. While these systems do offer higher-level abstractions and operations for cyclotomic rings, they are not a suitable platform for attaining our goals. First, their existing implementations of cyclotomic rings do not use the "tensored" representations (i.e., powerful and decoding bases, and CRT bases over $\mathbb{Z}_q$) and associated fast algorithms that are preferred for cryptographic purposes. Nor do they include support for special lattice operations like bit decomposition and other

"gadget" operations, so to use such systems we would have to reimplement essentially all the mathematical algorithms from scratch. Perhaps more significantly, the programming languages of these systems are relatively weakly and *dynamically* (not statically) typed, so all type-checking is deferred to runtime, where errors can be much harder to debug.

3.1.5 Architecture and Chapter Organization

The components of $\Lambda \circ \lambda$ are arranged in a few main layers, and the remainder of the chapter is organized correspondingly. From the bottom up, the layers are:

Integer layer (section 3.2): This layer contains abstract interfaces and implementations for domains like the integers $\mathbb{Z}$ and its quotient rings $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$, including specialized operations like rescaling and "(bit) decomposition." It also contains tools for working with moduli and cyclotomic indices at the *type level*, which enables static enforcement of mathematical constraints.

Tensor layer (section 3.3 and 3.4): This layer's main abstract interface, called `Tensor`, defines all the linear transformations and special values needed for working efficiently in cyclotomic rings (building on the framework developed in [LPR13a]), and permits multiple implementations. This low-level interface is completely hidden from typical cryptographic applications by the cyclotomic layer (below). These sections describe the interface and include the definitions and analysis of several linear transforms and algorithms that, to our knowledge, have not previously appeared in the literature. Additionally, section 3.4 describes the "sparse decomposition" DSL and compiler that underlie our pure-Haskell `Tensor` implementation.

Cyclotomic layer (section 3.5): This layer defines data types and high-level interfaces for cyclotomic rings and their cryptographically relevant operations. Our implementations are relatively thin wrappers which modularly combine the integer and tensor layers,

and automatically manage the internal representations of ring elements for more efficient operations.

Cryptography layer: This layer consists of implementations of cryptographic schemes. We defer our main application to chapter 4, which uses $\Lambda \circ \lambda$ to implement a full-featured somewhat-homomorphic encryption scheme. We expand this layer with a second application in chapter 6.

Acknowledgments. We thank the anonymous CCS'16 reviewers for many useful comments.

3.2 Integer and Modular Arithmetic

At its core, lattice-based cryptography is built around arithmetic in the ring of integers $\mathbb{Z}$ and quotient rings $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ of integers modulo q. In addition, a variety of specialized operations are also widely used, e.g., *lifting* a coset in $\mathbb{Z}_q$ to its smallest representative in $\mathbb{Z}$, *rescaling* (or *rounding*) one quotient ring $\mathbb{Z}_q$ to another, and *decomposing* a $\mathbb{Z}_q$-element as a vector of small $\mathbb{Z}$-elements with respect to a "gadget" vector.

Here we recall the relevant mathematical background for all these domains and operations, and describe how they are represented and implemented in $\Lambda \circ \lambda$. This will provide a foundation for the next section, where we show how all these operations are very easily "promoted" from base rings like $\mathbb{Z}$ and $\mathbb{Z}_q$ to cyclotomic rings, to support ring-based cryptosystems. (Similar promotions can also easily be done to support cryptosystems based on *plain*-LWE, but we elect not to do so in $\Lambda \circ \lambda$, mainly because those systems are not as practically efficient.)

3.2.1 Representing $\mathbb{Z}$ and $\mathbb{Z}_q$

We exclusively use fixed-precision primitive Haskell types like `Int` and `Int64` to represent the integers $\mathbb{Z}$, and define our own specialized types like `ZqBasic q z` to represent $\mathbb{Z}_q$. Here

the q parameter is a "phantom" type that represents the value of the modulus q, while z is an integer type (like `Int64`) specifying the underlying representation of the integer residues modulo q.

This approach has many advantages: by defining `ZqBasic q z` as an instance of `Ring`, we can use the `(+)` and `(*)` operators without any explicit modular reductions. More importantly, at compile time the type system disallows operations on incompatible types—e.g., attempting to add a `ZqBasic q1 z` to a `ZqBasic q2 z` for distinct `q1`, `q2`—with no runtime overhead. Finally, we implement `ZqBasic q z` as a `newtype` for z, which means that they have identical runtime representations, with no additional overhead.

CRT/RNS representation. Some applications, like homomorphic encryption, can require moduli q that are too large for standard fixed-precision integer types. Many languages have support for unbounded integers (e.g., Haskell's `Integer` type), but the operations are relatively slow. Moreover, the values have varying sizes, which means they cannot be stored efficiently in "unboxed" form in arrays. A standard solution is to use the Chinese Remainder Theorem (CRT), also known as Residue Number System (RNS), representation: choose q to be the product of several pairwise coprime and sufficiently small $q_1, \ldots, q_t$, and use the natural ring isomorphism from $\mathbb{Z}_q$ to the product ring $\mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_t}$, where addition and multiplication are both component-wise.

In Haskell, using the CRT representation—and more generally, working in product rings—is very natural using the generic pair type `(,)`: whenever types a and b respectively represent rings A and B, the pair type `(a,b)` represents the product ring $A \times B$. This just requires defining the obvious instances of `Additive` and `Ring` for `(a,b)`—which in fact has already been done for us by the numeric prelude. Products of more than two rings are immediately supported by nesting pairs, e.g., `((a,b),c)`, or by using higher-arity tuples like `(a,b,c)`. A final nice feature is that a pair (or tuple) has fixed representation size if

all its components do, so arrays of pairs can be stored directly in "unboxed" form, without requiring any layer of indirection.

3.2.2 `Reduce` and `Lift`

Two basic, widely used operations are *reducing* a $\mathbb{Z}$-element to its residue class in $\mathbb{Z}_q$, and *lifting* a $\mathbb{Z}_q$-element to its smallest integer representative, i.e., in $\mathbb{Z} \cap [-\frac{q}{2}, \frac{q}{2})$. These operations are examples of the natural homomorphism, and canonical representative map, for arbitrary quotient groups. Therefore, we define `class (Additive a, Additive b) => Reduce a b` to represent that `b` is a quotient group of `a`, and `class Reduce a b => Lift b a` for computing canonical representatives.[4] These classes respectively introduce the functions

```
reduce :: Reduce a b => a -> b
lift   :: Lift   b a => b -> a
```

where `reduce` $\circ$ `lift` should be the identity function.

Instances of these classes are straightforward. We define an instance `Reduce z (ZqBasic q z)` for any suitable integer type `z` and `q` representing a modulus that fits within the precision of `z`, and a corresponding instance for `Lift`. For product groups (pairs) used for CRT representation, we define the natural instance `Reduce a (b1,b2)` whenever we have instances `Reduce a b1` and `Reduce a b2`. However, we do not have (nor do we need) a corresponding `Lift` instance, because there is no sufficiently generic algorithm to combine canonical representatives from two quotient groups.

3.2.3 `Rescale`

Another operation commonly used in lattice cryptography is *rescaling* (sometimes also called *rounding*) $\mathbb{Z}_q$ to a different modulus. Mathematically, the rescaling operation $\lfloor \cdot \rceil_{q'} : \mathbb{Z}_q \to$

[4] Precision issues prevent us from merging `Lift` and `Reduce` into one class. For example, we can reduce an `Int` into $\mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2}$ if both components can be represented by `Int`, but lifting may cause overflow.

$\mathbb{Z}_{q'}$ is defined as

$$\lfloor x + q\mathbb{Z} \rceil_{q'} := \left\lfloor \tfrac{q'}{q} \cdot (x + q\mathbb{Z}) \right\rceil = \left\lfloor \tfrac{q'}{q} \cdot x \right\rceil + q'\mathbb{Z} \in \mathbb{Z}_{q'}, \tag{3.2.1}$$

where $\lfloor \cdot \rceil$ denotes rounding to the nearest integer. (Notice that the choice of representative $x \in \mathbb{Z}$ has no effect on the result.) In terms of the additive groups, this operation is at least an "approximate" homomorphism: $\lfloor x + y \rceil_{q'} \approx \lfloor x \rceil_{q'} + \lfloor y \rceil_{q'}$, with equality when $q|q'$. We represent the rescaling operation via `class (Additive a, Additive b) => Rescale a b`, which introduces the function

```
rescale :: Rescale a b => a -> b
```

Instances. A straightforward instance, whose implementation just follows the mathematical definition, is `Rescale (ZqBasic q1 z) (ZqBasic q2 z)` for any integer type `z` and types `q1`, `q2` representing moduli that fit within the precision of `z`.

More interesting are the instances involving product groups (pairs) used for CRT representation. A naïve implementation would apply Equation (3.2.1) to the canonical representative of $x + q\mathbb{Z}$, but for large q this would require unbounded-integer arithmetic. Instead, following ideas from [GHS12c], here we describe algorithms that avoid this drawback.

To "scale up" $x \in \mathbb{Z}_{q_1}$ to $\mathbb{Z}_{q_1 q_2} \cong \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2}$ where q_1 and q_2 are coprime, i.e., to multiply by q_2, simply output $(x \cdot q_2 \bmod q_1, 0)$. This translates easily into code that implements the instance `Rescale a (a,b)`. Notice, though, that the algorithm uses the value of the modulus q_2 associated with `b`. We therefore require `b` to be an instance of `class Mod`, which exposes the modulus value associated with the instance type. The instance `Rescale b (a,b)` works symmetrically.

To "scale down" $x = (x_1, x_2) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \cong \mathbb{Z}_{q_1 q_2}$ to $\mathbb{Z}_{q_1}$, we essentially need to divide by q_2, discarding the (signed) remainder. To do this,

1. Compute the canonical representative $\bar{x}_2 \in \mathbb{Z}$ of x_2.

(Observe that $(x_1' = x_1 - (\bar{x}_2 \bmod q_1), 0) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2}$ is the multiple of q_2 closest to $x = (x_1, x_2)$.)

2. Divide by q_2, outputting $q_2^{-1} \cdot x_1' \in \mathbb{Z}_{q_1}$.

The above easily translates into code that implements the instance `Rescale (a,b) a`, using the `Lift` and `Reduce` classes described above. The instance `Rescale (a,b) b` works symmetrically.

3.2.4 `Gadget`, `Decompose`, and `Correct`

Many advanced lattice cryptosystems use special objects called *gadgets* [MP12], which support certain operations as described below. For the purposes of this work, a gadget is a tuple over a quotient ring $R_q = R/qR$, where R is a ring that admits a meaningful "geometry." For concreteness, one can think of R as merely being the integers $\mathbb{Z}$, but later on we generalize to cyclotomic rings.

Perhaps the simplest gadget is the powers-of-two vector $\mathbf{g} = (1, 2, 4, 8, \ldots, 2^{\ell-1})$ over $\mathbb{Z}_q$, where $\ell = \lceil \lg q \rceil$. There are many other ways of constructing gadgets, either "from scratch" or by combining gadgets. For example, one may use powers of integers other than two, mixed products, the Chinese Remainder Theorem, etc. The salient property of a gadget $\mathbf{g}$ is that it admits efficient algorithms for the following tasks:

1. *Decomposition*: given $u \in R_q$, output a *short* vector $\mathbf{x}$ over R such that $\langle \mathbf{g}, \mathbf{x} \rangle = \mathbf{g}^t \cdot \mathbf{x} = u \pmod q$.

2. *Error correction*: given a "noisy encoding" of the gadget $\mathbf{b}^t = s \cdot \mathbf{g}^t + \mathbf{e}^t \bmod q$, where $s \in R_q$ and $\mathbf{e}$ is a sufficiently short error vector over R, output s and $\mathbf{e}$.

A key property is that decomposition and error-tolerant encoding relate in the following way (where the notation is as above, and $\approx$ hides a short error vector over R):

$$s \cdot u = (s \cdot \mathbf{g}^t) \cdot \mathbf{x} \approx \mathbf{b}^t \cdot \mathbf{x} \pmod{q}.$$

We represent gadget vectors and their associated operations via the following classes:

```
class Ring u => Gadget gad u where
  gadget    ::      Tagged gad [u]
  encode    :: u -> Tagged gad [u]

class (Gadget gad u, Reduce r u) => Decompose gad u r where
  decompose :: u -> Tagged gad [r]

class Gadget gad u => Correct gad u where
  correct   :: Tagged gad [u] -> (u, [LiftOf u])
```

The class `Gadget gad u` says that the ring `u` supports a gadget vector indexed by the type `gad`; the gadget vector itself is given by the term `gadget`. Note that its type is actually `Tagged gad [u]`: this is a `newtype` for `[u]`, with the additional type-level context `Tagged gad` indicating which gadget the vector represents (recall that there are many possible gadgets over a given ring). This tagging aids safety, by preventing the nonsensical mixing of values associated with different kinds of gadgets. In addition, Haskell provides generic ways of "promoting" ordinary operations to work within this extra context. (Formally, this is because `Tagged gad` is an instance of the `Functor` class.)

The class `Decompose gad u r` says that a `u`-element can be decomposed into a vector of `r`-elements (with respect to the gadget index by `gad`), via the `decompose` method.[5] The

[5]For simplicity, here we have depicted `r` as an additional parameter of the `Decompose` class. Our actual code adopts the more idiomatic practice of using a *type family* `DecompOf u`, which is defined by each instance of `Decompose`.

class `Correct gad u` says that a noisy encoding of a `u`-element (with respect to the gadget) can be error-corrected, via the `correct` method.

Note that we split the above functionality into three separate classes, both because their arguments are slightly different (e.g., `Correct` has no need for the `r` type), and because in some cases we have meaningful instances for some classes but not others.

Instances. For our type `ZqBasic q z` representing $\mathbb{Z}_q$, we give a straightforward instantiation of the "base-b" gadget $\mathbf{g} = (1, b, b^2, \ldots)$ and error correction and decomposition algorithms, for any positive integer b (which is represented as a parameter to the gadget type). In addition, we implement the trivial gadget $\mathbf{g} = (1) \in \mathbb{Z}_q^1$, where the decomposition algorithm merely outputs the canonical $\mathbb{Z}$-representative of its $\mathbb{Z}_q$-input. This gadget turns out to be useful for building nontrivial gadgets and algorithms for product rings, as described next.

For the pair type (which, to recall, we use to represent product rings in CRT representation), we give instances of `Gadget` and `Decompose` that work as follows. Suppose we have gadget vectors $\mathbf{g}_1, \mathbf{g}_2$ over R_{q_1}, R_{q_2}, respectively. Then the gadget for the product ring $R_{q_1} \times R_{q_2}$ is essentially the concatenation of $\mathbf{g}_1$ and $\mathbf{g}_2$, where we first attach $0 \in R_{q_2}$ components to the entries of $\mathbf{g}_1$, and similarly for $\mathbf{g}_2$. The decomposition of $(u_1, u_2) \in R_{q_1} \times R_{q_2}$ with respect to this gadget is the concatenation of the decompositions of u_1, u_2. All this translates easily to the implementations

```
gadget = (++) <$> (map (,zero) <$> gadget) <*> (map (zero,) <$> gadget)
decompose (a,b) = (++) <$> decompose a <*> decompose b
```

In the definition of `gadget`, the two calls to `map` attach zero components to the entries of $\mathbf{g}_1, \mathbf{g}_2$, and `(++)` appends the two lists. (The syntax `<$>`, `<*>` is standard applicative notation, which promotes normal functions into the `Tagged gad` context.)

3.2.5 `CRTrans`

Fast multiplication in cyclotomic rings is made possible by converting ring elements to the *Chinese remainder* representation, using the Chinese Remainder Transform (CRT) over the base ring. This is an invertible linear transform akin to the Discrete Fourier Transform (over $\mathbb{C}$) or the Number Theoretic Transform (over appropriate $\mathbb{Z}_q$), which has a fast algorithm corresponding to its "sparse decomposition" (see Equation 3.3.1 and [LPR13a, Section 3] for further details).

Applying the CRT and its inverse requires knowledge of certain roots of unity, and the inverse of a certain integer, in the base ring. So we define the synonym `type CRTInfo r = (Int -> r, r)`, where the two components are (1) a function that takes an integer i to the ith power of a certain principal[6] mth root of unity ω_m in `r`, and (2) the multiplicative inverse of $\hat{m}$ in `r`, where $\hat{m} = m/2$ if m is even, else $\hat{m} = m$. We also define the class `CRTrans`, which exposes the CRT information:

```
class (Monad mon, Ring r) => CRTrans mon r where
  crtInfo :: Int -> mon (CRTInfo r)
```

Note that the output of `crtInfo` is embedded in a `Monad mon`, the choice of which can reflect the fact that the CRT might not exist for certain m. For example, the `CRTrans` instance for the complex numbers $\mathbb{C}$ uses the trivial `Identity` monad, because the complex CRT exists for every m, whereas the instance for `ZqBasic q z` uses the `Maybe` monad to reflect the fact that the CRT may not exist for certain combinations of m and moduli q.

We give nontrivial instances of `CRTrans` for `ZqBasic q z` (representing $\mathbb{Z}_q$) for prime q, and for `Complex Double` (representing $\mathbb{C}$). In addition, because we use tensors and cyclotomic rings over base rings like $\mathbb{Z}$ and $\mathbb{Q}$, we must also define trivial instances of `CRTrans` for `Int`, `Int64`, `Double`, etc., for which `crtInfo` always returns `Nothing`.

[6] A principal mth root of unity in `r` is an element ω_m such that $\omega_m^m = 1$, and $\omega_m^{m/t} - 1$ is not a zero divisor for every prime t dividing m. Along with the invertibility of $\hat{m}$ in `r`, these are sufficient conditions for the index-m CRT over `r` to be invertible.

3.2.6 Type-Level Cyclotomic Indices

Recall that there is one cyclotomic ring for every positive integer m. The index m of a cyclotomic ring, and in particular its factorization, plays a major role in the definitions of the ring operations. For example, the index-m "Chinese remainder transform" is similar to a mixed-radix FFT, where the radices are the prime divisors of m. In addition, cyclotomic rings can sometimes be related to each other based on their indices. For example, the mth cyclotomic can be seen as a subring of the m'th cyclotomic if and only if $m|m'$; the largest common subring of the m_1th and m_2th cyclotomics is the $\gcd(m_1, m_2)$th cyclotomic, etc.

In $\Lambda\circ\lambda$, a cyclotomic index m is specified by an appropriate type `m`, and the data types representing cyclotomic rings (and their underlying coefficient tensors) are parameterized by such an `m`. Based on this parameter, $\Lambda\circ\lambda$ *generically derives* algorithms for all the relevant operations in the corresponding cyclotomic. In addition, for operations that involve more than one cyclotomic, $\Lambda\circ\lambda$ expresses and *statically enforces* (at compile time) the laws governing when these operations are well defined.

We achieve the above properties using Haskell's type system, with the help of the powerful *data kinds* extension [Yor+12] and the *singletons* library [EW12; ES14]. Essentially, these tools enable the "promotion" of ordinary values and functions from the data level to the type level. More specifically, they promote every value to a corresponding type, and promote every function to a corresponding *type family*, i.e., a function on the promoted types. We stress that all type-level computations are performed at compile time, yielding the dual benefits of static safety guarantees and no runtime overhead.

We provide a brief overview of the interface for type-level factored numbers below. In subsection 3.2.7 below we give more details on how cyclotomic indices are represented and operated upon at the type level. Then in subsection 3.2.8 we describe how all this is used to generically derive algorithms for arbitrary cyclotomics.

Interface. Concretely, $\Lambda \circ \lambda$ defines a special data type `Factored` that represents positive integers by their factorizations, along with several functions on such values. Singletons then promotes all of this to the type level. This yields concrete "factored types" `Fm` for various useful values of `m`, e.g., `F1`, ..., `F100`, `F128`, `F256`, `F512`, etc. In addition, it yields the following type families, where `m1`, `m2` are variables representing any factored types:

- `FMul m1 m2` (synonym: `m1 * m2`) and `FDiv m1 m2` (synonym: `m1 / m2`) respectively yield the factored types representing $m_1 \cdot m_2$ and m_1/m_2 (if it is an integer; else it yields a compile-time error);
- `FGCD m1 m2` and `FLCM m1 m2` respectively yield the factored types representing $\gcd(m_1, m_2)$ and $\mathrm{lcm}(m_1, m_2)$;
- `FDivides m1 m2` yields the (promoted) boolean type `True` or `False`, depending on whether $m_1 | m_2$. In addition, `m1 `Divides` m2` is a convenient synonym for the constraint `True ~ Divides m1 m2`. (This constraint is used section 3.5 below.)

Finally, $\Lambda \circ \lambda$ also provides several *entailments* representing number-theoretic laws that the compiler itself cannot derive from our data-level code. For example, transitivity of the "divides" relation is represented by the entailment

```
(k `Divides` l, l `Divides` m) :- (k `Divides` m)
```

which allows the programmer to satisfy the constraint $k|m$ in any context where the constraints $k|\ell$ and $\ell|m$ are satisfied.

3.2.7 Promoting Factored Naturals

Operations in a cyclotomic ring are governed by the prime-power factorization of its index. Therefore, we define the data types `PrimeBin`, `PrimePower`, and `Factored` to represent factored positive integers (here the types `Pos` and `Bin` are standard Peano and binary encodings, respectively, of the natural numbers):

```
-- Invariant: argument is prime
newtype PrimeBin   = P  Bin
-- (prime, exponent) pair
newtype PrimePower = PP (PrimeBin, Pos)
-- List invariant: primes appear in strictly increasing order
-- (no duplicates).
newtype Factored   = F  [PrimePower]
```

To enforce the invariants, we hide the `P`, `PP`, and `F` constructors from clients, and instead only export operations that verify and maintain the invariants. In particular, we provide functions that construct valid `PrimeBin`, `PrimePower`, and `Factored` values for any appropriate positive integer, and we define the following arithmetic operations, whose implementations are straightforward:

```
fDivides         :: Factored -> Factored -> Bool
fMul, fGCD, fLCM :: Factored -> Factored -> Factored
```

We use data kinds and singletons to mechanically promote the above data-level definitions to the type level. Specifically, data kinds defines an (uninhabited) `Factored` *type* corresponding to each `Factored` *value*, while singletons produces *type families* `FDivides`, `FMul`, etc. that operate on these promoted types. We also provide compile-time "macros" that define `F`m as a synonym for the `Factored` type corresponding to positive integer m, and similarly for `PrimeBin` and `PrimePower` types. Combining all this, e.g., `FMul F2 F2` yields the type `F4`, as does `FGCD F12 F8`. Similarly, `FDivides F5 F30` yields the promoted type `True`.

In addition, for each `Factored` type `m`, singletons defines a type `Sing m` that is inhabited by a single value, which can be obtained as `sing :: Sing m`. This value has an internal structure mirroring that of the corresponding `Factored` value, i.e., it is essentially a list of singleton values corresponding to the appropriate `PrimePower` types. (The same goes for

the singletons for `PrimePower` and `PrimeBin` types.) Lastly, the `withSingI` function lets us go in the reverse direction, i.e., it lets us "elevate" a particular singleton *value* to instantiate a corresponding *type variable* in a polymorphic expression.

3.2.8 Applying the Promotions

Here we summarize how we use the promoted types and singletons to generically derive algorithms for operations in arbitrary cyclotomics. We rely on the "sparse decomposition" framework described in section 3.4 below; for our purposes here, we only need that a value of type `Trans r` represents a linear transform over a base ring `r` via some sparse decomposition.

A detailed example will illustrate our approach. Consider the polymorphic function

```
crt :: (Fact m, CRTrans r, ...) => Tagged m (Trans r)
```

which represents the index-`m` Chinese Remainder Transform (CRT) over a base ring `r` (e.g., $\mathbb{Z}_q$ or $\mathbb{C}$). Equation (3.3.1) gives a sparse decomposition of CRT in terms of prime-power indices, and Equations (3.3.2) and (3.3.3) give sparse decompositions for the prime-power case in terms of the CRT and DFT for prime indices, and the "twiddle" transforms for prime-power indices.

Following these decompositions, our implementation of `crt` works as follows:

1. It first obtains the singleton corresponding to the `Factored` type `m`, using `sing :: Sing m`, and extracts the list of singletons for its `PrimePower` factors. It then takes the Kronecker product of the corresponding specializations of the *prime power*-index CRT function

   ```
   crtPP :: (PPow pp, CRTrans r, ...) => Tagged pp (Trans r)
   ```

   The specializations are obtained by "elevating" the `PrimePower` singletons to instantiate the `pp` type variable using `withSingI`, as described above.

(The above-described transformation from `Factored` to `PrimePower` types applies equally well to *all* our transforms of interest. Therefore, we implement a generic combinator that builds a transform indexed by `Factored` types from any given one indexed by `PrimePower` types.)

2. Similarly, `crtPP` obtains the singleton corresponding to the `PrimePower` type `pp`, extracts the singletons for its `PrimeBin` (base) and `Pos` (exponent) types, and composes the appropriate specializations of the *prime-index* CRT and DFT functions

   ```
   crtP, dftP :: (Prim p, CRTrans r, ...) => Tagged p (Trans r)
   ```

   along with prime power-indexed transforms that apply the appropriate "twiddle" factors.

3. Finally, `crtP` and `dftP` obtain the singleton corresponding to the `PrimeBin` type `p`, and apply the CRT/DFT transformations indexed by this value, using naïve matrix-vector multiplication. This requires the pth roots of unity in `r`, which are obtained via the `CRTrans` interface.

3.3 `Tensor` Interface and Sparse Decompositions

In this section we detail the "backend" representations and algorithms for computing in cyclotomic rings. We implement these algorithms using the sparse decomposition framework outlined in section 3.4. This section relies heavily on the background and notation given in section 2.2.

An element of the mth cyclotomic ring over a base ring `r` (e.g., $\mathbb{Q}$, $\mathbb{Z}$, or $\mathbb{Z}_q$) can be represented as a vector of $n = \varphi(m)$ coefficients from `r`, with respect to a particular `r`-basis of the cyclotomic ring. We call such a vector a *(coefficient) tensor* to emphasize its implicit multidimensional nature, which arises from the tensor-product structure of the bases we use.

The class `Tensor` (see Figure 3.1) represents the cryptographically relevant operations on coefficient tensors with respect to the powerful, decoding, and CRT bases. An instance

of `Tensor` is a data type `t` that itself takes two type parameters: an `m` representing the cyclotomic index, and an `r` representing the base ring. So the fully applied type `t m r` represents an index-`m` cyclotomic tensor over `r`.

The `Tensor` class introduces a variety of methods representing linear transformations that either convert between two particular bases (e.g., `lInv`, `crt`), or perform operations with respect to certain bases (e.g., `mulGPow`, `embedDec`). It also exposes some important fixed values related to cyclotomic ring extensions (e.g., `powBasisPow`, `crtSetDec`). An instance `t` of `Tensor` must implement all these methods and values for arbitrary (legal) cyclotomic indices.

3.3.1 Single-Index Transforms

In this and the next subsection we describe sparse decompositions for all the `Tensor` operations. We start here with the dimension-preserving transforms involving a single index m, i.e., they take an index-m tensor as input and produce one as output.

Prime-Power Factorization

For an arbitrary index m, every transform of interest factors into the tensor product of the corresponding transforms for prime-power indices. More specifically, let T_m denote the matrix for any of the linear transforms on index-m tensors that we consider below. Then letting $m = \prod_\ell m_\ell$ be the factorization of m into its maximal prime-power divisors m_ℓ (in some canonical order), we have the factorization

$$T_m = \bigotimes_\ell T_{m_\ell} \quad . \tag{3.3.1}$$

This follows directly from the Kronecker-product factorizations of the powerful, decoding, and CRT bases (e.g., Equation (2.2.2)), and the mixed-product property. Therefore, for the

```
class Tensor t where
  -- single-index transforms

  scalarPow :: (Ring        r, Fact m) =>      r -> t m r
  scalarCRT :: (CRTrans mon r, Fact m) => mon (r -> t m r)

  l, lInv   :: (Ring r, Fact m) => t m r -> t m r

  mulGPow, mulGDec :: (Ring           r, Fact m)
    => t m r ->        t m r
  divGPow, divGDec :: (IntegralDomain r, Fact m)
    => t m r -> Maybe (t m r)

  crt, crtInv, mulGCRT, divGCRT :: (CRTrans mon r, Fact m)
    => mon (t m r -> t m r)

  tGaussianDec :: (OrdFloat q, Fact m, MonadRandom rnd, ...)
    => v -> rnd (t m q)

  gSqNormDec   :: (Ring r, Fact m) => t m r -> r

  -- two-index transforms and values

  embedPow, embedDec :: (Ring r, m `Divides` m') => t m  r -> t m' r
  twacePowDec        :: (Ring r, m `Divides` m') => t m' r -> t m  r

  embedCRT :: (CRTrans mon r, m `Divides` m') => mon (t m  r -> t m' r)
  twaceCRT :: (CRTrans mon r, m `Divides` m') => mon (t m' r -> t m  r)

  coeffs :: (Ring r, m `Divides` m') => t m' r -> [t m r]

  powBasisPow :: (Ring r, m `Divides` m') => Tagged m [t m' r]

  crtSetDec :: (PrimeField fp, m `Divides` m', ...)
    => Tagged m [t m' fp]
```

Figure 3.1: Representative methods from the `Tensor` class. For the sake of concision, the constraint `TElt t r` is omitted from every method.

remainder of this subsection we only deal with prime-power indices $m = p^e$ for a prime p and positive integer e.

Embedding Scalars

Consider a scalar element a from the base ring, represented relative to the powerful basis $\vec{p}_m$. Because the first element of $\vec{p}_m$ is unity, we have

$$a = \vec{p}_m^t \cdot (a \cdot \mathbf{e}_1),$$

where $\mathbf{e}_1 = (1, 0, \ldots, 0)$. Similarly, in the CRT basis $\vec{c}_m$ (when it exists), unity has the all-ones coefficient vector $\mathbf{1}$. Therefore,

$$a = \vec{c}_m^t \cdot (a \cdot \mathbf{1}).$$

The `Tensor` methods `scalarPow` and `scalarCRT` use the above equations to represent a scalar from the base ring as a coefficient vector relative to the powerful and CRT bases, respectively. Note that `scalarCRT` itself is wrapped by `Maybe`, so that it can be defined as `Nothing` if there is no CRT basis over the base ring.

Converting Between Powerful and Decoding Bases

Let L_m denote the matrix of the linear transform that converts from the decoding basis to the powerful basis:

$$\vec{d}_m^t = \vec{p}_m^t \cdot L_m \ ,$$

i.e., a ring element with coefficient vector $\mathbf{v}$ in the decoding basis has coefficient vector $L_m \cdot \mathbf{v}$ in the powerful basis. Because $\vec{d}_m = \vec{p}_{m,p} \otimes \vec{d}_{p,1}$ and $\vec{d}_{p,1}^t = \vec{p}_{p,1}^t \cdot L_p$ (both by

Equation (2.2.5)), we have

$$\begin{aligned}\vec{d}_m^{\,t} &= (\vec{p}_{m,p}^{\,t} \cdot I_{m/p}) \otimes (\vec{p}_p^{\,t} \cdot L_p) \\ &= \vec{p}_m^{\,t} \cdot \underbrace{(I_{m/p} \otimes L_p)}_{L_m} \quad .\end{aligned}$$

Recall that L_p is the square $\varphi(p)$-dimensional lower-triangular matrix with 1s throughout its lower-left triangle, and L_p^{-1} is the lower-triangular matrix with 1s on the diagonal, -1s on the subdiagonal, and 0s elsewhere. We can apply both L_p and L_p^{-1} using just $p-1$ additions, by taking partial sums and successive differences, respectively.

The `Tensor` methods `l` and `lInv` represent multiplication by L_m and L_m^{-1}, respectively.

Multiplication by g_m

Let G_m^{pow} denote the matrix of the linear transform representing multiplication by g_m in the powerful basis, i.e.,

$$g_m \cdot \vec{p}_m^{\,t} = \vec{p}_m^{\,t} \cdot G_m^{\text{pow}} \quad .$$

Because $g_m = g_p \in \mathcal{O}_p$ and $\vec{p}_m = \vec{p}_{m,p} \otimes \vec{p}_p$, we have

$$\begin{aligned}g_m \cdot \vec{p}_m &= \vec{p}_{m,p} \otimes (g_p \cdot \vec{p}_p) \\ &= (\vec{p}_{m,p} \cdot I_{m/p}) \otimes (\vec{p}_p \cdot G_p^{\text{pow}}) \\ &= \vec{p}_m \cdot \underbrace{(I_{m/p} \otimes G_p^{\text{pow}})}_{G_m^{\text{pow}}} \ ,\end{aligned}$$

where G_p^{pow} and its inverse (which represents division by g_p in the powerful basis) are the square $(p-1)$-dimensional matrices

$$G_p^{\text{pow}} = \begin{pmatrix} 1 & & & & 1 \\ -1 & \ddots & & & 1 \\ & \ddots & 1 & & \vdots \\ & & -1 & 1 & 1 \\ & & & -1 & 2 \end{pmatrix}, \quad (G_p^{\text{pow}})^{-1} = p^{-1} \cdot \begin{pmatrix} p-1 & \cdots & -1 & -1 & -1 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ 3 & \cdots & 3 & 3-p & 3-p \\ 2 & \cdots & 2 & 2 & 2-p \\ 1 & \cdots & 1 & 1 & 1 \end{pmatrix}.$$

Identical decompositions hold for G_m^{dec} and G_m^{crt} (which represent multiplication by g_m in the decoding and CRT bases, respectively), where

$$G_p^{\text{dec}} = \begin{pmatrix} 2 & 1 & \cdots & & 1 \\ -1 & 1 & & & \\ & \ddots & \ddots & & \\ & & -1 & 1 & \\ & & & -1 & 1 \end{pmatrix}, \quad (G_p^{\text{dec}})^{-1} = p^{-1} \cdot \begin{pmatrix} 1 & 2-p & 3-p & \cdots & -1 \\ 1 & 2 & 3-p & \cdots & -1 \\ 1 & 2 & 3 & \cdots & -1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 2 & 3 & \cdots & p-1 \end{pmatrix},$$

and G_p^{crt} is the diagonal matrix with $1 - \omega_p^i$ in the ith diagonal entry (indexed from 1 to $p-1$), where ω_p is the same primitive pth root of unity in the base ring used to define the CRT basis.

The linear transforms represented by the above matrices can be applied in time linear in the dimension. For G_p^{pow}, G_p^{dec}, and G_p^{crt} and its inverse this is obvious, due to their sparsity. For $(G_p^{\text{dec}})^{-1}$, this follows from the fact that every row (apart from the top one) differs from the preceding one by a single entry. For $(G_p^{\text{pow}})^{-1}$, we can compute the entries of the output vector from the bottom up, by computing the sum of all the input entries and their partial sums from the bottom up.

The **Tensor** methods `mulGPow` and `mulGDec` represent multiplication by G_m^{pow} and G_m^{dec}, respectively. Similarly, the methods `divGPow` and `divGDec` represent division by these matrices; note that their outputs are wrapped by **Maybe**, so that the output can be **Nothing** when division fails. Finally, `mulGCRT` and `divGCRT` represent multiplication and division by G_m^{crt}; note that these methods *themselves* are wrapped by **Maybe**, because G_m^{crt} and its inverse are well-defined over the base ring exactly when a CRT basis exists. (In this case, division always succeeds, hence no **Maybe** is needed for the *output* of `divGCRT`.)

Chinese Remainder and Discrete Fourier Transforms

Consider a base ring, like $\mathbb{Z}_q$ or $\mathbb{C}$, that admits an invertible index-m Chinese Remainder Transform CRT_m, defined by a principal mth root of unity ω_m. Then as shown in [LPR13a, Section 3], this transform converts from the powerful basis to the CRT basis (defined by the same ω_m), i.e.,

$$\vec{p}_m^{\,t} = \vec{c}_m^{\,t} \cdot \text{CRT}_m \quad .$$

Also as shown in [LPR13a, Section 3], CRT_m admits the following sparse decompositions for $m > p$:[7]

$$\mathrm{CRT}_m = (\mathrm{DFT}_{m/p} \otimes I_{p-1}) \cdot \hat{T}_m \cdot (I_{m/p} \otimes \mathrm{CRT}_p) \tag{3.3.2}$$

$$\mathrm{DFT}_m = (\mathrm{DFT}_{m/p} \otimes I_p) \cdot T_m \cdot (I_{m/p} \otimes \mathrm{DFT}_p) \ . \tag{3.3.3}$$

(These decompositions can be applied recursively until all the CRT and DFT terms have subscript p.) Here DFT_p is a square p-dimensional matrix with rows and columns indexed from zero, and CRT_p is its lower-left $(p-1)$-dimensional square submatrix, with rows indexed from one and columns indexed from zero. The (i,j)th entry of each matrix is ω_p^{ij}, where $\omega_p = \omega_m^{m/p}$. Finally, $\hat{T}_m, T_m$ are diagonal "twiddle" matrices whose diagonal entries are certain powers of ω_m.

For the inverses CRT_m^{-1} and DFT_m^{-1}, by standard properties of matrix and Kronecker products, we have sparse decompositions mirroring those in Equations (3.3.2) and (3.3.3). Note that DFT_p is invertible if and only if p is invertible in the base ring, and the same goes for CRT_p, except that CRT_2 (which is just unity) is always invertible. More specifically, $\mathrm{DFT}_p^{-1} = p^{-1} \cdot \mathrm{DFT}_p^*$, the (scaled) conjugate transpose of DFT_p, whose (i,j)th entry is ω_p^{-ij}. For CRT_p^{-1}, it can be verified that for $p > 2$,

$$\mathrm{CRT}_p^{-1} = p^{-1} \cdot \left(X - \mathbf{1} \cdot (\omega_p^1, \omega_p^2, \ldots, \omega_p^{p-1})^t\right),$$

where X is the upper-right $(p-1)$-dimensional square submatrix of DFT_p^*. Finally, note that in the sparse decomposition for CRT_m^{-1} (for aribtrary m), we can collect all the individual p^{-1} factors from the CRT_p^{-1} and DFT_p^{-1} terms into a single $\hat{m}^{-1}$ factor. (This factor is exposed by the **CRTrans** interface; see subsection 3.2.5.)

[7] In these decompositions, the order of arguments to the Kronecker products is swapped as compared with those appearing in [LPR13a]. This is due to our corresponding reversal of the factors in the Kronecker-product decompositions of the powerful and CRT bases. The ordering here is more convenient for implementation, but note that it yields bases and twiddle factors in "digit-reversed" order. In particular, the twiddle matrices $\hat{T}_m, T_m$ here are permuted versions of the ones defined in [LPR13a].

The **Tensor** methods `crt` and `crtInv` respectively represent multiplication by CRT_m and its inverse. These methods themselves are wrapped by **Maybe**, so that they can be **Nothing** when there is no CRT basis over the base ring.

Generating (Tweaked) Gaussians in the Decoding Basis

Cryptographic applications often need to sample secret error terms from a prescribed distribution. For the original definition of Ring-LWE involving the dual ideal $R^\vee$ (see section 2.2), it is particularly useful to use distributions D_r that correspond to (continuous) spherical Gaussians in the canonical embedding. For sufficiently large r, these distributions are supported by worst-case hardness proofs [LPR13b]. Note that the error can be discretized in a variety of ways, with no loss in hardness.

With the "tweaked" perspective that replaces $R^\vee$ by R via the tweak factor $t_m \in R$, we are interested in sampling from tweaked distributions $t_m \cdot D_r$. More precisely, we want a randomized algorithm that samples a coefficient vector over $\mathbb{R}$, with respect to one of the standard bases of R, of a random element that is distributed as $t_m \cdot D_r$. This is not entirely trivial because (except in the power-of-two case) R does not have an orthogonal basis, so the output coefficients will not be independent.

The material in [LPR13a, Section 6.3] yields a specialized, fast algorithm for sampling from D_r with output represented in the decoding basis $\vec{b}_m$ of $R^\vee$. Equivalently, the very same algorithm samples from the tweaked Gaussian $t_m \cdot D_r$ relative to the decoding basis $\vec{d}_m = t_m \cdot \vec{b}_m$ of R. The algorithm is faster (often much moreso) than the naïve one that applies a full CRT^*_m (over $\mathbb{C}$) to a Gaussian in the canonical embedding. The efficiency comes from skipping several layers of orthogonal transforms (namely, scaled DFTs and twiddle matrices), which is possible due to the rotation-invariance of spherical Gaussians. The algorithm also avoids complex numbers entirely, instead using only reals.

The algorithm. The sampling algorithm simply applies a certain linear transform over $\mathbb{R}$, whose matrix E_m has a sparse decomposition as described below, to a vector of i.i.d. real Gaussian samples with parameter r, and outputs the resulting vector. The `Tensor` method `tGaussianDec` implements the algorithm, given $v = r^2$. (Note that its output type `rnd (t m q)` for `MonadRandom rnd` is necessarily monadic, because the algorithm is randomized.)

As with all the transforms considered above, we describe the sparse decomposition of E_m where m is a power of a prime p, which then generalizes to arbitrary m as described in subsection 3.3.1. For $m > p$, we have

$$E_m = \sqrt{m/p} \cdot (I_{m/p} \otimes E_p),$$

where E_2 is unity and E_p for $p > 2$ is

$$E_p = \tfrac{1}{\sqrt{2}} \cdot \mathrm{CRT}_p^* \cdot \begin{pmatrix} I & -\sqrt{-1}J \\ J & \sqrt{-1}I \end{pmatrix} \in \mathbb{R}^{(p-1)\times(p-1)},$$

where CRT_p is over $\mathbb{C}$, and J is the "reversal" matrix obtained by reversing the columns of the identity matrix.[8] Expanding the above product, E_p has rows indexed from zero and columns indexed from one, and its (i, j)th entry is

$$\sqrt{2} \cdot \begin{cases} \cos\theta_{i\cdot j} & \text{for } 1 \le j < p/2 \\ \sin\theta_{i\cdot j} & \text{for } p/2 < j \le p-1 \end{cases}, \qquad \theta_k = 2\pi k/p.$$

Finally, note that in the sampling algorithm, when applying E_m for arbitrary m with prime-power factorization $m_\ell = \prod_\ell m_\ell$, we can apply all the $\sqrt{m_\ell/p_\ell}$ scaling factors

[8] We remark that the signs of the rightmost block of the above matrix (containing $-\sqrt{-1}J$ and $\sqrt{-1}I$) is swapped as compared with what appears in [LPR13a, Section 6.3]. The choice of sign is arbitrary, because any orthonormal basis of the subspace spanned by the columns works equally well.

(from the E_{m_ℓ} terms) to the parameter r of the Gaussian input vector, i.e., use parameter $r\sqrt{m/\operatorname{rad}(m)}$ instead.

Gram Matrix of Decoding Basis

Certain cryptographic applications need to obtain the Euclidean norm, under the canonical embedding σ, of cyclotomic ring elements (usually, error terms). Let $\vec{b}$ denote any $\mathbb{Q}$-basis of the ambient number field and let τ denote conjugation, which maps any root of unity to its inverse. Then the squared norm of $\sigma(e)$, where $e = \vec{b}^t \cdot \mathbf{e}$ for some rational coefficient vector $\mathbf{e}$, is

$$\|\sigma(e)\|^2 = \langle \sigma(e), \sigma(e) \rangle = \operatorname{Tr}_{R/\mathbb{Z}}(e \cdot \tau(e)) = \mathbf{e}^t \cdot \operatorname{Tr}_{R/\mathbb{Z}}(\vec{b} \cdot \tau(\vec{b}^t)) \cdot \mathbf{e} = \langle \mathbf{e}, G\mathbf{e} \rangle \ ,$$

where $G = \operatorname{Tr}_{R/\mathbb{Z}}(\vec{b} \cdot \tau(\vec{b}^t))$ denotes the Gram matrix of the basis $\vec{b}$. So computing the squared norm mainly involves multiplication by the Gram matrix.

As shown below, the Gram matrix of the decoding basis $\vec{b}_m$ of $R^\vee$ has a particularly simple sparse decomposition. Now, because the *tweaked* decoding basis $\vec{d}_m = t_m \cdot \vec{b}_m$ of R satisfies $g_m \cdot \vec{d}_m = \hat{m} \cdot \vec{b}_m$, the same Gram matrix also yields $\|\sigma(g_m \cdot e)\|^2$ (up to a $\hat{m}^2$ scaling factor) from the coefficient tensor of e with respect to $\vec{d}_m$. This is exactly what is needed when using tweaked Gaussian errors $e \in R$, because the "untweaked" error $g_m \cdot e$ is short and (near-)spherical in the canonical embedding (see, e.g., Invariant 4.2.2). The `Tensor` method `gSqNormDec` maps the coefficient tensor of e (with respect to $\vec{d}_m$) to $\hat{m}^{-1} \cdot \|\sigma(g_m \cdot e)\|^2$.[9]

[9] The $\hat{m}^{-1}$ factor compensates for the implicit scaling between $\vec{b}_m$ and $g_m \cdot \vec{d}_m$, and is the smallest such factor that guarantees an integer output when the input coefficients are integral.

Recall that $\vec{b}_m$ is defined as the dual, under $\mathrm{Tr}_{R/\mathbb{Z}}$, of the conjugate powerful basis $\tau(\vec{p}_m)$. From this it can be verified that

$$\begin{aligned} \vec{b}_p &= p^{-1} \cdot \left(\zeta_p^j - \zeta_p^{-1}\right)_{j=0,\ldots,p-2} \\ \vec{b}_{m,p} &= (m/p)^{-1} \cdot \vec{p}_{m,p} \ . \end{aligned}$$

Using the above, an elementary calculation shows that

$$\begin{aligned} p \cdot \mathrm{Tr}_{p,1}(\vec{b}_p \cdot \tau(\vec{b}_p)) &= I_{p-1} + \mathbf{1} \\ (m/p) \cdot \mathrm{Tr}_{m,p}(\vec{b}_{m,p} \cdot \tau(\vec{b}_{m,p})) &= I_{m/p} \ , \end{aligned}$$

where $\mathbf{1}$ denotes the all-1s matrix. (Note that for $p = 2$, the Gram matrix of $\vec{b}_p$ is just unity.) Combining these, we have

$$\begin{aligned} m \cdot \mathrm{Tr}_{R/\mathbb{Z}}(\vec{b}_m \cdot \tau(\vec{b}_m)^t) &= p \cdot \mathrm{Tr}_{p,1}((m/p) \cdot \mathrm{Tr}_{m,p}(\vec{b}_{m,p} \cdot \tau(\vec{b}_{m,p}^t)) \otimes (\vec{b}_p \cdot \tau(\vec{b}_p^t))) \\ &= I_{m/p} \otimes p \cdot \mathrm{Tr}_{p,1}(\vec{b}_p \cdot \vec{b}_p^t) \\ &= I_{m/p} \otimes (I_{p-1} + \mathbf{1}) \ . \end{aligned}$$

3.3.2 Two-Index Transforms and Values

We now consider transforms and special values relating the mth and m'th cyclotomic rings, for $m|m'$. These are used for computing the embed and twace functions, the relative powerful basis, and the relative CRT set.

Prime-Power Factorization

As in the subsection 3.3.1, every transform of interest for arbitrary $m|m'$ factors into the tensor product of the corresponding transforms for prime-power indices having the same prime base. More specifically, let $T_{m,m'}$ denote the matrix of any of the linear transforms

we consider below. Suppose we have factorization $m = \prod_\ell m_\ell$, $m' = \prod_\ell m'_\ell$ where each m_ℓ, m'_ℓ is a power of a distinct prime p_ℓ (so some m_ℓ may be 1). Then we have the factorization

$$T_{m,m'} = \bigotimes_\ell T_{m_\ell,m'_\ell} \ ,$$

which follows directly from the Kronecker-product factorizations of the powerful and decoding bases, and the mixed-product property. Therefore, from this point onward we deal only with prime-power indices $m = p^e$, $m' = p^{e'}$ for a prime p and integers $e' > e \geq 0$.

We mention that for the transforms we consider below, the fully expanded matrices $T_{m,m'}$ have very compact representations and can be applied directly to the input vector, without computing a sequence of intermediate vectors via the sparse decomposition. For efficiency, our implementation does exactly this.

Coefficients in Relative Bases

We start with transforms that let us represent elements with respect to *relative* bases, i.e., to represent an element of the m'th cyclotomic as a vector of elements in the mth cyclotomic, with respect to a relative basis. Due to the Kronecker-product structure of the powerful, decoding, and CRT bases, it turns out that the same transformation works for all of them. The `coeffs` method of **`Tensor`** implements this transformation.

One can verify the identity $(\vec{x} \otimes \vec{y})^t \cdot \mathbf{a} = \vec{x}^t \cdot A \cdot \vec{y}$, where A is the "matricization" of the vector $\mathbf{a}$, whose rows are (the transposes of) the consecutive $\dim(\vec{y})$-dimensional blocks of $\mathbf{a}$. Letting $\vec{b}_\ell$ denote either the powerful, decoding, or CRT basis in the ℓth cyclotomic, which has factorization $\vec{b}_{m'} = \vec{b}_{m',m} \otimes \vec{b}_m$, we have

$$\vec{b}_{m'}^t \cdot \mathbf{a} = \vec{b}_{m',m}^t \cdot (A \cdot \vec{b}_m).$$

Therefore, $A \cdot \vec{b}_m$ is the desired vector of R-coefficients of $a = \vec{b}^t_{m'} \cdot \mathbf{a} \in R'$. In other words, the $\varphi(m)$-dimensional blocks of $\mathbf{a}$ are the coefficient vectors (with respect to basis $\vec{b}_m$) of the R-coefficients of a with respect to the relative basis $\vec{b}_{m',m}$.

Embed Transforms

We now consider transforms that convert from a basis in the mth cyclotomic to the same type of basis in the m'th cyclotomic. That is, for particular bases $\vec{b}_{m'}, \vec{b}_m$ of the m'th and mth cyclotomics (respectively), we write

$$\vec{b}^t_m = \vec{b}^t_{m'} \cdot T$$

for some integer matrix T. So embedding a ring element from the mth to the m'th cyclotomic (with respect to these bases) corresponds to left-multiplication by T. The `embedB` methods of **`Tensor`**, for $\mathsf{B} \in \{\mathsf{Pow}, \mathsf{Dec}, \mathsf{CRT}\}$, implement these transforms.

We start with the powerful basis. Because $\vec{p}_{m'} = \vec{p}_{m',m} \otimes \vec{p}_m$ and the first entry of $\vec{p}_{m',m}$ is unity,

$$\begin{aligned} \vec{p}^t_m &= (\vec{p}^t_{m',m} \cdot \mathbf{e}_1) \otimes (\vec{p}^t_m \cdot I_{\varphi(m)}) \\ &= \vec{p}^t_{m'} \cdot (\mathbf{e}_1 \otimes I_{\varphi(m)}) \ , \end{aligned}$$

where $\mathbf{e}_1 = (1, 0, \ldots, 0) \in \mathbb{Z}^{\varphi(m')/\varphi(m)}$. Note that $(\mathbf{e}_1 \otimes I_{\varphi(m)})$ is the identity matrix stacked on top of an all-zeros matrix, so left-multiplication by it simply pads the input vector by zeros.

For the decoding bases $\vec{d}_{m'}$, $\vec{d}_m$, an identical derivation holds when $m > 1$, because $\vec{d}_{m'} = \vec{p}_{m',m} \otimes \vec{d}_m$. Otherwise, we have $\vec{d}_{m'} = \vec{p}_{m',p} \otimes \vec{d}_p$ and $\vec{d}^t_m = (1) = \vec{d}^t_p \cdot \mathbf{v}$, where

$\mathbf{v} = (1, -1, 0, \ldots, 0) \in \mathbb{Z}^{\varphi(p)}$. Combining these cases, we have

$$\vec{d}_m^t = \vec{d}_{m'}^t \cdot \begin{cases} \mathbf{e}_1 \otimes I_{\varphi(m)} & \text{if } m > 1 \\ \mathbf{e}_1 \otimes \mathbf{v} & \text{if } m = 1. \end{cases}$$

For the CRT bases $\vec{c}_{m'}$, $\vec{c}_m$, because $\vec{c}_m = \vec{c}_{m',m} \otimes \vec{c}_m$ and the sum of the elements of any (relative) CRT basis is unity, we have

$$\begin{aligned} \vec{c}_m^t &= (\vec{c}_{m',m}^t \cdot \mathbf{1}) \otimes (\vec{c}_m^t \cdot I_{\varphi(m)}) \\ &= \vec{c}_{m'}^t \cdot (\mathbf{1} \otimes I_{\varphi(m)}) \ . \end{aligned}$$

Notice that $(\mathbf{1} \otimes I_{\varphi(m)})$ is just a stack of identity matrices, so left-multiplication by it just stacks up several copies of the input vector.

Finally, we express the *relative* powerful basis $\vec{p}_{m',m}$ with respect to the powerful basis $\vec{p}_{m'}$; this is used in the `powBasisPow` method of **`Tensor`**. We simply have

$$\begin{aligned} \vec{p}_{m',m}^t &= (\vec{p}_{m',m}^t \cdot I_{\varphi(m')/\varphi(m)}) \otimes (\vec{p}_m \cdot \mathbf{e}_1) \\ &= \vec{p}_{m'}^t \cdot (I_{\varphi(m')/\varphi(m)} \otimes \mathbf{e}_1) \ . \end{aligned}$$

Twace Transforms

We now consider transforms that represent the twace function from the m'th to the mth cyclotomic for the three basis types of interest. That is, for particular bases $\vec{b}_{m'}, \vec{b}_m$ of the m'th and mth cyclotomics (respectively), we write

$$\mathrm{Tw}_{m',m}(\vec{b}_{m'}^t) = \vec{b}_m^t \cdot T$$

for some integer matrix T, which by linearity of twace implies

$$\mathrm{Tw}_{m',m}(\vec{b}^t_{m'} \cdot \mathbf{v}) = \vec{b}^t_m \cdot (T \cdot \mathbf{v}).$$

In other words, the twace function (relative to the these bases) corresponds to left-multiplication by T. The `twacePowDec` and `twaceCRT` methods of **`Tensor`** implement these transforms.

To start, we claim that

$$\mathrm{Tw}_{m',m}(\vec{p}_{m',m}) = \mathrm{Tw}_{m',m}(\vec{d}_{m',m}) = \mathbf{e}_1 \in \mathbb{Z}^{\varphi(m')/\varphi(m)}. \tag{3.3.4}$$

This holds for $\vec{d}_{m',m}$ because it is dual to (conjugated) $\vec{p}_{m',m}$ under $\mathrm{Tw}_{m',m}$, and the first entry of $\vec{p}_{m',m}$ is unity. It holds for $\vec{p}_{m',m}$ because $\vec{p}_{m',m} = \vec{d}_{m',m}$ for $m > 1$, and for $m = 1$ one can verify that

$$\mathrm{Tw}_{m',1}(\vec{p}_{m',1}) = \mathrm{Tw}_{p,1}(\mathrm{Tw}_{m',p}(\vec{p}_{m',p}) \otimes \vec{p}_{p,1}) = (1, 0, \ldots, 0) \otimes \mathrm{Tw}_{p,1}(\vec{p}_{p,1}) = \mathbf{e}_1.$$

Now for the powerful basis, by linearity of twace and Equation (3.3.4) we have

$$\begin{aligned}
\mathrm{Tw}_{m',m}(\vec{p}^t_{m'}) &= \mathrm{Tw}_{m',m}(\vec{p}^t_{m',m}) \otimes \vec{p}^t_m \\
&= (1 \cdot \mathbf{e}_1^t) \otimes (\vec{p}^t_m \cdot I_{\varphi(m)}) \\
&= \vec{p}^t_m \cdot (\mathbf{e}_1^t \otimes I_{\varphi(m)}) \ .
\end{aligned}$$

An identical derivation holds for the decoding basis as well. Notice that left-multiplication by the matrix $(\mathbf{e}_1^t \otimes I_{\varphi(m)})$ just returns the first $\varphi(m')/\varphi(m)$ entries of the input vector.

Finally, we consider the CRT basis. Because $g_{m'} = g_p$ (recall that $m' \geq p$), by definition of twace in terms of trace we have

$$\mathrm{Tw}_{m',m}(x) = (\hat{m}/\hat{m}') \cdot g_m^{-1} \cdot \mathrm{Tr}_{m',m}(g_p \cdot x). \tag{3.3.5}$$

Also recall that the traces of all relative CRT set elements are unity: $\mathrm{Tr}_{m',\ell}(\vec{c}_{m',\ell}) = \mathbf{1}_{\varphi(m')/\varphi(\ell)}$ for any $\ell | m'$. We now need to consider two cases. For $m > 1$, we have $g_m = g_p$, so by Equation (3.3.5) and linearity of trace,

$$\mathrm{Tw}_{m',m}(\vec{c}_{m',m}) = (\hat{m}/\hat{m}') \cdot \mathbf{1}_{\varphi(m')/\varphi(m)} \ .$$

For $m = 1$, we have $g_m = 1$, so by $\vec{c}_{m',1} = \vec{c}_{m',p} \otimes \vec{c}_{p,1}$ and linearity of trace we have

$$\begin{aligned}\mathrm{Tw}_{m',1}(\vec{c}_{m',1}) &= (\hat{m}/\hat{m}') \cdot \mathrm{Tr}_{p,1}(\mathrm{Tr}_{m',p}(\vec{c}_{m',p}) \otimes (g_p \cdot \vec{c}_{p,1})) \\ &= (\hat{m}/\hat{m}') \cdot \mathbf{1}_{\varphi(m')/\varphi(p)} \otimes \mathrm{Tr}_{p,1}(g_p \cdot \vec{c}_{p,1}) \ .\end{aligned}$$

Applying the two cases, we finally have

$$\begin{aligned}\mathrm{Tw}_{m',m}(\vec{c}^{\,t}_{m'}) &= (1 \cdot \mathrm{Tw}_{m',m}(\vec{c}^{\,t}_{m',m})) \otimes (\vec{c}^{\,t}_{m} \cdot I_{\varphi(m)}) \\ &= \vec{c}^{\,t}_{m} \cdot (\hat{m}/\hat{m}') \cdot \begin{cases} \mathbf{1}^t_{\varphi(m')/\varphi(m)} \otimes I_{\varphi(m)} & \text{if } m > 1 \\ \mathbf{1}^t_{\varphi(m')/\varphi(p)} \otimes \mathrm{Tr}_{p,1}(g_p \cdot \vec{c}^{\,t}_{p,1}) & \text{if } m = 1. \end{cases}\end{aligned}$$

Again because $\mathrm{Tr}_{p,1}(\vec{c}_{p,1}) = \mathbf{1}_{\varphi(p)}$, the entries of $\mathrm{Tr}_{p,1}(g_p \cdot \vec{c}_{p,1})$ are merely the CRT coefficients of g_p. That is, the ith entry (indexed from one) is $1 - \omega_p^i$, where $\omega_p = \omega_{m'}^{m'/p}$ for the value of $\omega_{m'}$ used to define the CRT set of the m'th cyclotomic.

3.3.3 CRT Sets

In this final subsection we describe an algorithm for computing a representation of the *relative CRT set* $\vec{c}_{m',m}$ modulo a prime-power integer. CRT *sets* are a generalization of CRT *bases* to the case where the prime modulus may not be 1 modulo the cyclotomic index (i.e., it does not split completely), and therefore the cardinality of the set may be less than the dimension of the ring. CRT sets are used for homomorphic SIMD operations [SV14] and in

the bootstrapping algorithm of [AP13]. See subsection 2.2.6 for the necessary background information.

Computing CRT Sets

We start with an easy calculation that, for a prime integer p, "lifts" the mod-p CRT set to the mod-p^e CRT set.

Lemma 3.3.1. *For $R = \mathcal{O}_m$, a prime integer p where $p \nmid m$, and a positive integer e, let $(c_i)_i$ be the CRT set of R_{p^e}, and let $\bar{c}_i \in R$ be any representative of c_i. Then $(\bar{c}_i^p \bmod p^{e+1}R)_i$ is the CRT set of $R_{p^{e+1}}$.*

Corollary 3.3.2. *If $\bar{c}_i \in R$ are representatives for the mod-p CRT set $(c_i)_i$ of R_p, then $(\bar{c}_i^{p^{e-1}} \bmod p^e R)_i$ is the CRT set of R_{p^e}.*

Proof of Lemma 3.3.1. Let $pR = \prod_i \mathfrak{p}_i$ be the factorization of pR into distinct prime ideals $\mathfrak{p}_i \subset R$. By hypothesis, we have $\bar{c}_i \in \delta_{i,i'} + \mathfrak{p}_{i'}^e$ for all i, i'. Then

$$\bar{c}_i^p \in \delta_{i,i'} + p \cdot \mathfrak{p}_{i'}^e + \mathfrak{p}_{i'}^{ep} \subseteq \delta_{i,i'} + \mathfrak{p}_{i'}^{e+1},$$

because p divides the binomial coefficient $\binom{p}{k}$ for $0 < k < p$, because $pR \subseteq \mathfrak{p}_{i'}$, and because $\mathfrak{p}_{i'}^{ep} \subseteq \mathfrak{p}_{i'}^{e+1}$. □

CRT sets modulo a prime. We now describe the mod-p CRT set for a prime integer p, and an efficient algorithm for computing representations of its elements. To motivate the approach, notice that the coefficient vector of $x \in R_p$ with respect to some arbitrary $\mathbb{Z}_p$-basis $\vec{b}$ of R_p can be obtained via the twace and the dual $\mathbb{Z}_p$-basis $\vec{b}^\vee$ (under the twace):

$$x = \vec{b}^t \cdot \mathrm{Tw}_{R_p/\mathbb{Z}_p}(x \cdot \vec{b}^\vee).$$

In what follows we let $\vec{b}$ be the *decoding* basis, because its dual basis is the conjugated powerful basis, which has a particularly simple form. The following lemma is a direct consequence of Equation (2.2.6) and the definition of twace (Equation (2.2.4)).

Lemma 3.3.3. *For $R = \mathcal{O}_m$ and a prime integer $p \nmid m$, let $\vec{c} = (c_i)$ be the CRT set of R_p, let $\vec{d} = \vec{d}_m$ denote the decoding $\mathbb{Z}_p$-basis of R_p, and let $\tau(\vec{p}) = (p_j^{-1})$ denote its dual, the conjugate powerful basis. Then*

$$\vec{c}^t = \vec{d}^t \cdot \mathrm{Tw}_{R_p/\mathbb{Z}_p}(\tau(\vec{p}) \cdot \vec{c}^t) = \vec{d}^t \cdot \hat{m}^{-1} \cdot \mathrm{Tr}_{\mathbb{F}_{p^d}/\mathbb{F}_p}(C),$$

where C is the matrix over $\mathbb{F}_{q^d}$ whose $(j, \bar{\imath})$th element is $\rho_{\bar{\imath}}(g_m) \cdot \rho_{\bar{\imath}}(p_j^{-1})$.

Notice that $\rho_{\bar{\imath}}(p_j^{-1})$ is merely the inverse of the $(\bar{\imath}, j)$th entry of the matrix CRT_m over $\mathbb{F}_{p^d}$, which is the Kronecker product of CRT_{m_ℓ} over all maximal prime-power divisors of m. In turn, the entries of CRT_{m_ℓ} are all just appropriate powers of $\omega_{m_\ell} \in \mathbb{F}_{p^d}$. Similarly, $\rho_{\bar{\imath}}(g_m)$ is the product of all $\rho_{\bar{\imath} \bmod m_\ell}(g_{m_\ell}) = 1 - \omega_{m_\ell}^{\bar{\imath}}$. So we can straightforwardly compute the entries of the matrix C and takes their traces into $\mathbb{F}_p$, yielding the decoding-basis coefficient vectors for the CRT set elements.

Relative CRT sets. We conclude by describing the *relative* CRT set $\vec{c}_{m',m}$ modulo a prime p, where $R = \mathcal{O}_m, R' = \mathcal{O}_{m'}$ for $m | m'$ and $p \nmid m'$. The key property of $\vec{c}_{m',m}$ is that the CRT sets $\vec{c}_{m'}, \vec{c}_m$ for R_p, R'_p (respectively) satisfy the Kronecker-product factorization

$$\vec{c}_{m'} = \vec{c}_{m',m} \otimes \vec{c}_m \ . \tag{3.3.6}$$

The definition of $\vec{c}_{m',m}$ arises from the splitting of the prime ideal divisors $\mathfrak{p}_i$ (of pR) in R', as described next.

Recall from above that the prime ideal divisors $\mathfrak{p}'_{i'} \subset R'$ of pR' and the CRT set $\vec{c}_{m'} = (c'_{i'})$ are indexed by $i' \in G' = \mathbb{Z}^*_{m'}/\langle p \rangle$, and similarly for $\mathfrak{p}_i \subset R$ and $\vec{c}_m = (c_i)$. For

each $i \in G = \mathbb{Z}_m^*/\langle p \rangle$, the ideal $\mathfrak{p}_i R'$ factors as the product of those $\mathfrak{p}'_{i'}$ such that $i' = i \pmod m$, i.e., those $i' \in \phi^{-1}(i)$ where $\phi\colon G' \to G$ is the natural mod-m homomorphism. Therefore,

$$c_i = \sum_{i' \in \phi^{-1}(i)} c'_{i'} \ . \tag{3.3.7}$$

To define $\vec{c}_{m',m}$, we partition G' into a collection $\mathcal{I}'$ of $|G'|/|G|$ equal-sized subsets I', such that $\phi(I') = G$ for every $I' \in \mathcal{I}'$. In other words, ϕ is a bijection between each I' and G. This induces a bijection $\psi\colon G' \to \mathcal{I}' \times G$, where the projection of ψ onto its second component is ϕ. We index the relative CRT set $\vec{c}_{m',m} = (c_{I'})$ by $I' \in \mathcal{I}'$, defining

$$c_{I'} := \sum_{i' \in I'} c'_{i'} \ .$$

By Equation (3.3.7) and the fact that $(c'_{i'})$ is the CRT set of R'_p, it can be verified that $c_{i'} = c_{I'} \cdot c_i$ for $\psi(i') = (I', i)$, thus confirming Equation (3.3.6).

3.4 Sparse Decompositions and Haskell Framework

As shown in section 3.3, the structure of the powerful, decoding, and CRT bases yield *sparse decompositions*, and thereby efficient algorithms, for cryptographically important linear transforms relating to these bases. Here we explain the principles of sparse decompositions, and summarize our Haskell framework for expressing and evaluating them.

3.4.1 Sparse Decompositions

A sparse decomposition of a matrix (or the linear transform it represents) is a factorization into sparser or more "structured" matrices, such as diagonal matrices or Kronecker products. Recall that the Kronecker (or tensor) product $A \otimes B$ of two matrices or vectors $A \in \mathcal{R}^{m_1 \times n_1}, B \in \mathcal{R}^{m_2 \times n_2}$ over a ring $\mathcal{R}$ is a matrix in $\mathcal{R}^{m_1 m_2 \times n_1 n_2}$. Specifically, it is the m_1-by-n_1 block matrix (or vector) made up of m_2-by-n_2 blocks, whose (i,j)th block is

$a_{i,j} \cdot B \in \mathcal{R}^{m_2 \times n_2}$, where $A = (a_{i,j})$. The Kronecker product satisfies the properties

$$(A \otimes B)^t = (A^t \otimes B^t)$$
$$(A \otimes B)^{-1} = (A^{-1} \otimes B^{-1})$$

and the *mixed-product* property

$$(A \otimes B) \cdot (C \otimes D) = (AC) \otimes (BD),$$

which we use extensively in what follows.

A sparse decomposition of a matrix A naturally yields an algorithm for multiplication by A, which can be much more efficient and parallel than the naïve algorithm. For example, multiplication by $I_n \otimes A$ can be done using n parallel multiplications by A on appropriate chunks of the input, and similarly for $A \otimes I_n$ and $I_l \otimes A \otimes I_r$. More generally, the Kronecker product of any two matrices can be expressed in terms of the previous cases, as follows:

$$A \otimes B = (A \otimes I_{\text{height}(B)}) \cdot (I_{\text{width}(A)} \otimes B) = (I_{\text{height}(A)} \otimes B) \cdot (A \otimes I_{\text{width}(B)}).$$

If the matrices A, B themselves have sparse decompositions, then these rules can be applied further to yield a "fully expanded" decomposition. All the decompositions we consider in this work can be fully expanded as products of terms of the form $I_l \otimes A \otimes I_r$, where multiplication by A is relatively fast, e.g., because A is diagonal or has small dimensions.

3.4.2 Haskell Framework

We now describe a simple, deeply embedded domain-specific language for expressing and evaluating sparse decompositions in Haskell. It allows the programmer to write such factorizations recursively in natural mathematical notation, and it automatically yields fast evaluation algorithms corresponding to fully expanded decompositions. For simplicity, our

implementation is restricted to square matrices (which suffices for our purposes), but it could easily be generalized to rectangular ones.

As a usage example, to express the decompositions

$$A = B \otimes C$$

$$B = (I_n \otimes D) \cdot E$$

where C, D, and E are "atomic," one simply writes

```
transA =  transB @* transC                -- B ⊗ C
transB = ( Id n  @* transD) .* transE     -- (I_n ⊗ D) · E
transC = trans functionC                  -- similarly for transD, transE
```

where `functionC` is (essentially) an ordinary Haskell function that left-multiplies its input vector by C. The above code causes `transA` to be internally represented as the fully expanded decomposition

$$A = (I_n \otimes D \otimes I_{\dim(C)}) \cdot (E \otimes I_{\dim(C)}) \cdot (I_{\dim(E)} \otimes C).$$

Finally, one simply writes `eval transA` to get an ordinary Haskell function that left-multiplies by A according to the above decomposition.

Data types. We first define the data types that represent transforms and their decompositions (here `Array r` stands for some arbitrary array type that holds elements of type `r`)

```
-- (dim(f), f) such that (f l r) applies I_l ⊗ f ⊗ I_r
type Tensorable r = (Int, Int -> Int -> Array r -> Array r)


-- transform component: a Tensorable with particular I_l, I_r
type TransC r = (Tensorable r, Int, Int)
```

```
-- full transform: a sequence of zero or more components
data Trans r = Id Int           -- identity sentinel
             | TSnoc (Trans r) (TransC r)
```

- The client-visible type alias `Tensorable r` represents an "atomic" transform (over the base type `r`) that can be augmented (tensored) on the left and right by identity transforms of any dimension. It has two components: the dimension d of the atomic transform f itself, and a function that, given any dimensions l, r, applies the ldr-dimensional transform $I_l \otimes f \otimes I_r$ to an array of `r`-elements. (Such a function could use parallelism internally, as already described.)
- The type alias `TransC r` represents a *transform component*, namely, a `Tensorable r` with particular values for l, r. `TransC` is only used internally; it is not visible to external clients.
- The client-visible type `Trans r` represents a full transform, as a sequence of zero or more components terminated by a sentinel representing the identity transform. For such a sequence to be well-formed, all the components (including the sentinel) must have the same dimension. Therefore, we export the `Id` constructor, but not `TSnoc`, so the only way for a client to construct a nontrivial `Trans r` is to use the functions described below (which maintain the appropriate invariant).

Evaluation. Evaluating a transform is straightforward. Simply evaluate each component in sequence:

```
evalC :: TransC r -> Array r -> Array r
evalC ((_,f), l, r) = f l r

eval :: Trans r -> Array r -> Array r
```

```
eval (Id _)         = id  -- identity function
eval (TSnoc rest f) = eval rest . evalC f
```

Constructing transforms. We now explain how transforms of type `Trans r` are constructed. The function `trans` wraps a `Tensorable` as a full-fledged transform:

```
trans :: Tensorable r -> Trans r
trans f@(d,_) = TSnoc (Id d) (f, 1, 1)  -- I_d · f
```

More interesting are the functions for composing and tensoring transforms, respectively denoted by the operators `(.*), (@*) :: Trans r -> Trans r -> Trans r`. Composition just appends the two sequences of components, after checking that their dimensions match; we omit its straightforward implementation. The Kronecker-product operator `(@*)` simply applies the appropriate rules to get a fully expanded decomposition:

```
-- I_m ⊗ I_n = I_mn
(Id m) @* (Id n) = Id (m*n)

-- I_n ⊗ (A · B) = (I_n ⊗ A) · (I_n ⊗ B), and similarly
i@(Id n) @* (TSnoc a (b, l, r)) = TSnoc (i @* a) (b, (n*l), r)
(TSnoc a (b, l, r)) @* i@(Id n) = TSnoc (a @* i) (b, l, (r*n))

-- (A ⊗ B) = (A ⊗ I) · (I ⊗ B)
a @* b = (a @* Id (dim b)) .* (Id (dim a) @* b)
```

(The `dim` function simply returns the dimension of a transform, via the expected implementation.)

3.5 Cyclotomic Rings

In this section we summarize $\Lambda\circ\lambda$'s interfaces and implementations for cyclotomic rings. In subsection 3.5.1 we describe the interfaces of the two data types, `Cyc` and `UCyc`, that represent cyclotomic rings: `Cyc` completely hides and transparently manages the internal representation of ring elements (i.e., the choice of basis in which they are represented), whereas `UCyc` is a lower-level type that safely exposes and allows explicit control over the choice of representation. Lastly, in subsection 3.5.2 we describe key aspects of the implementations, such as `Cyc`'s subring optimizations, and how we generically "promote" base-ring operations to cyclotomic rings.

3.5.1 Cyclotomic Types: `Cyc` and `UCyc`

In this subsection we describe the interfaces of the two data types, `Cyc` and `UCyc`, that represent cyclotomic rings.

- `Cyc t m r` represents the `m`th cyclotomic ring over a base ring `r`—typically, one of $\mathbb{Q}$, $\mathbb{Z}$, or $\mathbb{Z}_q$—backed by an underlying `Tensor` type `t` (see section 3.3 for details on `Tensor`). The interface for `Cyc` completely hides the internal representations of ring elements (e.g., the choice of basis) from the client, and automatically manages the choice of representation so that the various ring operations are usually as efficient as possible. Therefore, most cryptographic applications can and should use `Cyc`.

- `UCyc t m rep r` represents the same cyclotomic ring as `Cyc t m r`, but as a coefficient vector relative to the basis indicated by `rep`. This argument is one of the four valueless types `P`, `D`, `C`, `E`, which respectively denote the powerful basis, decoding basis, CRT `r`-basis (if it exists), and CRT basis over an appropriate extension ring of `r`. Exposing the representation at the type level in this way allows—indeed, requires—the client to manage the choice of representation. (`Cyc` is one such client.) This can lead to more efficient computations in certain cases where `Cyc`'s management may

be suboptimal. More importantly, it safely enables a wide class of operations on the underlying coefficient vector, via category-theoretic classes like `Functor`; see sections 3.5.1 and 3.5.2 for further details.

Clients can easily switch between `Cyc` and `UCyc` as needed. Indeed, `Cyc` is just a relatively thin wrapper around `UCyc`, which mainly just manages the choice of representation, and provides some other optimizations related to subrings (see subsection 3.5.2 for details).

Instances

The `Cyc` and `UCyc` types are instances of many classes, which comprise a large portion of their interfaces.

Algebraic classes. As one might expect, `Cyc t m r` and `UCyc t m rep r` are instances of `Eq`, `Additive`, `Ring`, and various other algebraic classes for any appropriate choices of `t`, `m`, `rep`, and `r`. Therefore, the standard operators `(==)`, `(+)`, `(*)`, etc. are well-defined for `Cyc` and `UCyc` values, with semantics matching the mathematical definitions.

We remark that `UCyc t m rep r` is an instance of `Ring` only for the CRT representations `rep` = `C`, `E`, where multiplication is coefficient-wise. In the other representations, multiplication is algorithmically more complicated and less efficient, so we simply do not implement it. This means that clients of `UCyc` must explicitly convert values to a CRT representation before multiplying them, whereas `Cyc` performs such conversions automatically.

Category-theoretic classes. Because `UCyc t m rep r` for `rep` = `P`, `D`, `C` (but not `rep` =`E`) is represented as a vector of `r`-coefficients with respect to the basis indicated by `rep`, we define the *partially applied* types `UCyc t m rep` (note the missing base type `r`) to be instances of the classes `Functor`, `Applicative`, `Foldable`, and `Traversable`. For example, our instantiation of `Functor` for `f` = `UCyc t m rep` defines `fmap :: (r -> r') -> f r -> f r'` to apply the given `r -> r'` function independently on each of the `r`-coefficients.

By contrast, `Cyc t m` is *not* an instance of any category-theoretic classes. This is because by design, `Cyc` hides the choice of representation from the client, so it is unclear how (say) `fmap` should be defined: using the current internal representation (whatever it happens to be) would lead to unpredictable and often unintended behavior, whereas always using a particular representation (e.g., the powerful basis) would not be flexible enough to support operations that ought to be performed in a different representation.

Lattice cryptography classes. Lastly, we "promote" instances of our specialized lattice cryptography classes like `Reduce`, `Lift`, `Rescale`, `Gadget`, etc. from base types to `UCyc` and/or `Cyc`, as appropriate. For example, the instance `Reduce z zq`, which represents modular reduction from $\mathbb{Z}$ to $\mathbb{Z}_q$, induces the instance `Reduce (Cyc t m z) (Cyc t m zq)`, which represents reduction from R to R_q. All these instances have very concise and generic implementations using the just-described category-theoretic instances for `UCyc`; see subsection 3.5.2 for further details.

Functions

We now describe the remaining functions that define the interface for `Cyc`; see Figure 3.2 for their type signatures. (`UCyc` admits a very similar collection of functions, which we omit from the discussion.) We start with functions that involve a single cyclotomic index `m`.

`scalarCyc` embeds a scalar element from the base ring `r` into the `m`th cyclotomic ring over `r`.

`mulG`, `divG` respectively multiply and divide by the special element g_m in the mth cyclotomic ring. These operations are commonly used in applications, and have efficient algorithms in all our representations, which is why we define them as special functions (rather than, say, just exposing a value representing g_m). Note that because the input may not always be divisible by g_m, the output type of `divG` is a `Maybe`.

```
scalarCyc :: (Fact m, CElt t r) =>         r ->        Cyc t m r
mulG      :: (Fact m, CElt t r) => Cyc t m r ->        Cyc t m r
divG      :: (Fact m, CElt t r) => Cyc t m r -> Maybe (Cyc t m r)
liftPow, liftDec
          :: (Fact m, Lift b a, ...) => Cyc t m b ->   Cyc t m a
advisePow, adviseDec, adviseCRT
          :: (Fact m, CElt t r)      => Cyc t m r ->   Cyc t m r

-- error sampling
tGaussian    ::
  (OrdFloat  q, ToRational v, MonadRandom rnd, CElt t q, ...)
                                  => v              -> rnd (Cyc t m q)
errorRounded :: (ToInteger z, ...) => v             -> rnd (Cyc t m z)
errorCoset   :: (ToInteger z, ...) =>
  v -> Cyc t m zp -> rnd (Cyc t m z)
gSqNorm      :: (Fact m, CElt t r) => Cyc t m r       -> r

-- inter-ring operations
embed     :: (m `Divides` m', CElt t r) => Cyc t m  r ->  Cyc t m' r
twace     :: (m `Divides` m', CElt t r) => Cyc t m' r ->  Cyc t m  r
coeffsPow, coeffsDec
          :: (m `Divides` m', CElt t r) => Cyc t m' r -> [Cyc t m  r]
powBasis  :: (m `Divides` m', CElt t r)      => Tagged m [Cyc t m' r]
crtSet    :: (m `Divides` m', CElt t r, ...) => Tagged m [Cyc t m' r]
```

Figure 3.2: Representative functions for the `Cyc` data type. (The `CElt t r` constraint is a synonym for a collection of constraints that include `Tensor t`, along with various constraints on the base type `r`.)

`liftB` for `B` = `Pow`, `Dec` lifts a cyclotomic ring element coordinate-wise with respect to the specified basis (powerful or decoding).

`adviseB` for `B` = `Pow`, `Dec`, `CRT` returns an equivalent ring element whose internal representation *might* be with respect to (respectively) the powerful, decoding, or a Chinese Remainder Theorem basis. These functions have no externally visible effect on the results of any computations, but they can serve as useful optimization hints. E.g., if one needs to compute `v * w1`, `v * w2`, etc., then advising that `v` be in CRT

representation can speed up these operations by avoiding duplicate CRT conversions across the operations.

The following functions relate to sampling error terms from cryptographically relevant distributions:

`tGaussian` samples an element of the number field K from the "tweaked" continuous Gaussian distribution $t \cdot D_r$, given $v = r^2$. (See section 2.2 above for background on, and the relevance of, tweaked Gaussians. The input is $v = r^2$ because that is more convenient for implementation.) Because the output is random, its type must be monadic: `rnd (Cyc t m r)` for `MonadRandom rnd`.

`errorRounded` is a discretized version of `tGaussian`, which samples from the tweaked Gaussian and rounds each decoding-basis coefficient to the nearest integer, thereby producing an output in R.

`errorCoset` samples an error term from a (discretized) tweaked Gaussian of parameter $p \cdot r$ over a given coset of $R_p = R/pR$. This operation is often used in encryption schemes when encrypting a desired message from the plaintext space R_p.[10]

`gSqNorm` yields the scaled squared norm of $g_m \cdot e$ (typically for a short error term e) under the canonical embedding, namely, $\hat{m}^{-1} \cdot \|\sigma(g_m \cdot e)\|^2$.

Finally, the following functions involve **`Cyc`** data types for two indices `m|m'`; recall that this means the `m`th cyclotomic ring can be viewed as a subring of the `m'`th one. Notice that in the type signatures, the divisibility constraint is expressed as `m `Divides` m'`, and recall from subsection 3.2.6 that this constraint is statically checked by the compiler and carries no runtime overhead.

[10] The extra factor of p in the Gaussian parameter reflects the connection between coset sampling as used in cryptosystems, and the underlying Ring-LWE error distribution actually used in their security proofs. This scaling gives the input v a consistent meaning across all the error-sampling functions.

embed, twace are respectively the embedding and "tweaked trace" functions between the `m`th and `m'`th cyclotomic rings.

coeffsB for `B` $=$ `Pow`, `Dec` expresses an element of the `m'`th cyclotomic ring with respect to the relative powerful or decoding basis ($\vec{p}_{m',m}$ and $\vec{d}_{m',m}$, respectively), as a list of coefficients from the `m`th cyclotomic.

powBasis is the relative powerful basis $\vec{p}_{m',m}$ of the `m'`th cyclotomic over the `m`th one.[11] Note that the **Tagged** `m` type annotation is needed to specify which subring the basis is relative to.

crtSet is the relative CRT set $\vec{c}_{m',m}$ of the `m'`th cyclotomic ring over the `m`th one, modulo a prime power. (See subsection 3.3.3 for its formal definition and a novel algorithm for computing it.) We have elided some constraints which say that the base type `r` must represent $\mathbb{Z}_{p^e}$ for a prime p.

We emphasize that both `powBasis` and `crtSet` are *values* (of type **Tagged** `m [`**Cyc** `t m' r]`), not functions. Due to Haskell's laziness, only those values that are actually used in a computation are ever computed; moreover, the compiler usually ensures that they are computed only once each and then memoized.

In addition to the above, we also could have included functions that apply *automorphisms* of cyclotomic rings, which would be straightforward to implement in our framework. We leave this for future work, merely because we have not yet needed automorphisms in any of our applications.

3.5.2 Implementation

We now describe some notable aspects of the **Cyc** and **UCyc** implementations. As previously mentioned, **Cyc** is mainly a thin wrapper around **UCyc** that automatically manages the choice

[11] We also could have defined `decBasis`, but it is slightly more complicated to implement, and we have not needed it in any of our applications.

of representation `rep`, and also includes some important optimizations for ring elements that are known to reside in cyclotomic subrings. In turn, **`UCyc`** is a thin wrapper around an instance of the **`Tensor`** class. (Recall that **`Tensor`** encapsulates the cryptographically relevant linear transforms on coefficient vectors for cyclotomic rings; see section 3.3 for details.)

Representations

`Cyc` `t m r` can represent an element of the `m`th cyclotomic ring over base ring `r` in a few possible ways:

- as a **`UCyc`** `t m rep r` for some `rep` = **`P`**, **`D`**, **`C`**, **`E`**;
- when applicable, as a *scalar* from the base ring `r`, or more generally, as an element of the `k`th cyclotomic *subring* for some `k`|`m`, i.e., as a **`Cyc`** `t k r`.

The latter subring representations enable some very useful optimizations in memory and running time: while cryptographic applications often need to treat scalars and subring elements as residing in some larger cyclotomic ring, **`Cyc`** can exploit knowledge of their "true" domains to operate more efficiently, as described in subsection 3.5.2 below.

`UCyc` represents a cyclotomic ring element by its coefficients tensor with respect to the basis indicated by `rep`. That is, for `rep` = **`P`**, **`D`**, **`C`**, a value of type **`UCyc`** `t m rep r` is simply a value of type `(t m r)`. However, a CRT basis over `r` does not always exist, e.g., if `r` represents the integers $\mathbb{Z}$, or $\mathbb{Z}_q$ for a modulus q that does not meet certain criteria. To handle such cases we use `rep` =**`E`**, which indicates that the representation is relative to a CRT basis over a certain *extension* ring **`CRTExt`** `r` that *always* admits such a basis, e.g., the complex numbers $\mathbb{C}$. That is, a **`UCyc`** `t m E r` is a value of type `(t m (`**`CRTExt`**` r))`.

We emphasize that the extension ring **`CRTExt`** `r` is determined by `r` itself, and **`UCyc`** is entirely agnostic to it. For example, **`ZqBasic`** uses the complex numbers, whereas the pair type `(a,b)` (which, to recall, represents a product ring) uses the product ring `(`**`CRTExt`**` a, `**`CRTExt`**` b)`.

Operations

Most of the `Cyc` functions shown in Figure 3.2 (e.g., `mulG`, `divG`, the error-sampling functions, `coeffsB`, `powBasis`, `crtSet`) simply call their `UCyc` counterparts for an appropriate representation `rep` (after converting any subring inputs to the full ring). Similarly, most of the `UCyc` operations for a given representation just call the appropriate `Tensor` method. In what follows we describe some operations that depart from these patterns.

The algebraic instances for `Cyc` implement operations like `(==)`, `(+)`, and `(*)` in the following way: first they convert the inputs to "compatible" representations in the most efficient way possible, then they compute the output in an associated representation. A few representative rules for how this is done are as follows:

- For two scalars from the base ring `r`, the result is just computed and stored as a scalar, thus making the operation very fast.
- Inputs from (possibly different) subrings of indices $\mathtt{k1}, \mathtt{k2} | \mathtt{m}$ are converted to the *compositum* of the two subrings, i.e., the cyclotomic of index $\mathtt{k} = \mathrm{lcm}(\mathtt{k1}, \mathtt{k2})$ (which divides `m`), then the result is computed there and stored as a subring element.
- For `(+)`, the inputs are converted to a common representation and added entry-wise.
- For `(*)`, if one of the inputs is a scalar from the base ring `r`, it is simply multiplied by the coefficients of the other input (this works for any `r`-basis representation). Otherwise, the two inputs are converted to the same CRT representation and multiplied entry-wise.

The implementation of the inter-ring operations `embed` and `twace` for `Cyc` is as follows: `embed` is "lazy," merely storing its input as a subring element and returning instantly. For `twace` from index `m'` to `m`, there are two cases: if the input is represented as a `UCyc` value (i.e., not as a subring element), then we just invoke the appropriate representation-specific `twace` function on that value (which in turn just invokes a method from `Tensor`). Otherwise, the input is represented as an element of the `k'`th cyclotomic for some $\mathtt{k'} | \mathtt{m'}$, in which case

we apply `twace` from index `k'` to index `k` $= \gcd(\texttt{m}, \texttt{k'})$, which is the smallest index where the result is guaranteed to reside, and store the result as a subring element.

Promoting Base-Ring Operations

Many cryptographic operations on cyclotomic rings are defined as working entry-wise on the ring element's coefficient vector with respect to some basis (either a particular or arbitrary one). For example, reducing from R to R_q is equivalent to reducing the coefficients from $\mathbb{Z}$ to $\mathbb{Z}_q$ in *any* basis, while "decoding" R_q to R (as used in decryption) is defined as lifting the $\mathbb{Z}_q$-coefficients, relative to the *decoding* basis, to their smallest representatives in $\mathbb{Z}$. To implement these and many other operations, we generically "promote" operations on the base ring to corresponding operations on cyclotomic rings, using the fact that `UCyc t m rep` is an instance of the category-theoretic classes `Functor`, `Applicative`, `Traversable`, etc.

As a first example, consider the `Functor` class, which introduces the method

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

Our `Functor` instance for `UCyc t m rep` defines `fmap g c` to apply `g` to each of `c`'s coefficients (in the basis indicated by `rep`). This lets us easily promote our specialized lattice operations from section 3.2. For example, an instance `Reduce z zq` can be promoted to an instance `Reduce (UCyc t m P z) (UCyc t m P zq)` simply by defining `reduce = fmap reduce`. We similarly promote other base-ring operations, including lifting from $\mathbb{Z}_q$ to $\mathbb{Z}$, rescaling from $\mathbb{Z}_q$ to $\mathbb{Z}_{q'}$, discretization of $\mathbb{Q}$ to either $\mathbb{Z}$ or to a desired coset of $\mathbb{Z}_p$, and more.

As a richer example, consider gadgets and decomposition (subsection 3.2.4) for a cyclotomic ring R_q over base ring $\mathbb{Z}_q$. For any gadget vector over $\mathbb{Z}_q$, there is a corresponding

gadget vector over R_q, obtained simply by embedding $\mathbb{Z}_q$ into R_q. This lets us promote a `Gadget` instance for `zq` to one for `UCyc t m rep zq`:[12,13]

```
gadget = fmap (fmap scalarCyc) gadget
```

Mathematically, decomposing an R_q-element into a short vector over R is defined coefficient-wise with respect to the powerful basis. That is, we decompose each $\mathbb{Z}_q$-coefficient into a short vector over $\mathbb{Z}$, then collect the corresponding entries of these vectors to yield a vector of short R-elements. To implement this strategy, one might try to promote the function (here with slightly simplified signature)

```
decompose :: Decompose zq z => zq -> [z]
```

to `Cyc t m zq` using `fmap`, as we did with `reduce` and `lift` above. However, a moment's thought reveals that this does not work: it yields output of type `Cyc t m [z]`, whereas we want `[Cyc t m z]`. The solution is to use the `Traversable` class, which introduces the method

```
traverse :: (Traversable v, Applicative f) =>
  (a -> f b) -> v a -> f (v b)
```

In our setting, `v` is `UCyc t m P`, and `f` is the list type `[]`, which is indeed an instance of `Applicative`.[14] We can therefore easily promote an instance of `Decompose` from `zq` to `UCyc t m P zq`, essentially via:

```
decompose v = traverse decompose v
```

We similarly promote the error-correction operation `correct :: Correct zq z => [zq] -> (zq, [z])`.

[12] The double calls to `fmap` are needed because there are two `Functor` layers around the `zq`-entries of `gadget :: Tagged gad [zq]`: the list `[]`, and the `Tagged gad` context.

[13] Technically, we only instantiate the gadget-related classes for `Cyc t m zq`, not `UCyc t m rep zq`. This is because `Gadget` has `Ring` as a superclass, which is instantiated by `UCyc` only for the CRT representations `rep` $=$ `C`, `E`; however, for geometric reasons the gadget operations on cyclotomic rings must be defined in terms of the `P` or `D` representations. This does not affect the essential nature of the present discussion.

[14] Actually, the `Applicative` instance for `[]` models *nondeterminism*, not the entry-wise operations we need. Fortunately, there is a costless `newtype` wrapper around `[]`, called `ZipList`, that instantiates `Applicative` in the desired way.

Rescaling. Mathematically, rescaling R_q to $R_{q'}$ is defined as applying $\lfloor \cdot \rceil_{q'} : \mathbb{Z}_q \to \mathbb{Z}_{q'}$ (represented by the function `rescale :: Rescale a b => a -> b`; see subsection 3.2.3) coefficient-wise in either the powerful or decoding basis (for geometrical reasons). However, there are at least two distinct algorithms that implement this operation, depending on the representation of the ring element and of $\mathbb{Z}_q$ and $\mathbb{Z}_{q'}$. The generic algorithm simply converts the input to the required basis and then rescales coefficient-wise. But there is also a more efficient, specialized algorithm [GHS12c] for rescaling a product ring $R_q = R_{q_1} \times R_{q_2}$ to R_{q_1}. For the typical case of rescaling an input in the CRT representation to an output in the CRT representation, the algorithm requires only one CRT transformation for each of R_{q_1} and R_{q_2}, as opposed to two and one (respectively) for the generic algorithm. In applications like HE where R_{q_1} itself can be a product of multiple component rings, this reduces the work by nearly a factor of two.

In more detail, the specialized algorithm is analogous to the one for product rings $\mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2}$ described at the end of subsection 3.2.3. To rescale $a = (a_1, a_2) \in R_{q_1} \times R_{q_2}$ to R_{q_1}, we lift $a_2 \in R_{q_2}$ to a relatively short representative $\bar{a}_2 \in R$ using the powerful or decoding basis, which involves an inverse-CRT for R_{q_2}. We then compute $\bar{a}'_2 = \bar{a}_2 \bmod q_1 R$ and output $q_2^{-1} \cdot (a_1 - \bar{a}'_2) \in R_{q_1}$, which involves a CRT for R_{q_1} on $\bar{a}'_2$.

To capture the polymorphism represented by the above algorithms, we define a class called `RescaleCyc`, which introduces the method `rescaleCyc`. We give two distinct instances of `RescaleCyc` for the generic and specialized algorithms, and the compiler automatically chooses the appropriate one based on the concrete types representing the base ring.

CHAPTER 4

STATE-OF-THE-ART HOMOMORPHIC ENCRYPTION WITH $\Lambda\circ\lambda$

Homomorphic encryption is a powerful cryptographic construction which allows computation on encrypted data. It has numerous applications, such as securely offloading computation to an untrusted third party, private information retrieval [Yi+13], multi-party computation [MW15], statistical analysis on a large-scale multidimensional corpus [WH12], and advertising [NLV11], to name a few. Following the first plausible construction by Gentry in 2009 [Gen09b; Gen09a], improvement in the theory of homomorphic encryption has led to schemes with better efficiency, stronger security assurances, and specialized features (see, e.g., [Dij+10; SV14; BV11b; Cor+11; CNT12; BV14a; BGV14; Bra12; GHS12b; GHS12a; Che+13; AP13; Gen+13; BV14b; AP14].)

The promise of efficient homomorphic encryption has led to several implementations of somewhat-homomorphic encryption (SHE) schemes, all of which highlight particular aspects of SHE/FHE (e.g., efficient bootstrapping [DM15], good performance [HS], parallelism using GPUs [Wan+12], and partial parameter generation [LCP17]). However, each lacks important theoretical developments in homomorphic encryption which results in suboptimal performance and functionality.

In this chapter, we define an advanced SHE scheme that incorporates and refines a wide collection of features from a long series of works [LPR13b; BV11b; BV14a; BGV14; GHS12c; Gen+13; LPR13a; AP13]. Our scheme has several distinguishing features, including:

- advanced SHE functionality like efficient ring switching;
- support for large plaintext spaces, which is more efficient than encrypting individual bits;

- the ability for plaintext and ciphertext spaces to be defined over different cyclotomic rings, which permits certain optimizations;
- and strict separation of the interface from the computational details, so it is easy to use our SHE scheme on, e.g., multi-core CPUs, GPUs, etc.

4.1 SHE with $\Lambda\circ\lambda$

Our implementation uses the $\Lambda\circ\lambda$ library defined in chapter 3 to achieve its advanced functionality. The high level interfaces exposed in $\Lambda\circ\lambda$ make our implementation particularly simple, closely and concisely matching the SHE scheme's mathematical definition.

Using $\Lambda\circ\lambda$'s support for the cyclotomic hierarchy, we also devise and implement a more efficient variant of ring-switching for HE, which we call *ring tunneling*. A prior technique [AP13] homomorphically evaluates a linear function by "hopping" from one ring to another through a common *extension* ring. The extension ring can be very large (dimension 200,000 or more), leading to a significant performance bottleneck. Our new approach avoids this problem by "tunneling" through a common *subring*, which has a much smaller dimension resulting in improved performance. Moreover, we show that the linear function can be integrated into the accompanying key-switching step, thus unifying two operations into a simpler and even more efficient one. (See section 4.2 for details.) This implementation is the foundation for the homomorphic evaluation of a lattice-based symmetric-key primitives (chapter 6).

4.1.1 Example: SHE in $\Lambda\circ\lambda$

For illustration, here we briefly give a flavor of our SHE implementation in $\Lambda\circ\lambda$; see Figure 4.1 for representative code, and section 4.3 for many more details of the scheme's mathematical definition and implementation. While we do not expect the reader (especially one who is not conversant with Haskell) to understand all the details of the code, it should be clear that even complex operations like modulus-switching and key-switching/relinearization

have very concise and natural implementations in terms of Λ∘λ's interfaces (which include the functions `errorCoset`, `reduce`, `embed`, `twace`, `liftDec`, etc.). Indeed, the implementations of the SHE functions are often shorter than their type declarations! (For the reader who is new to Haskell, section 2.3 gives a brief tutorial that provides sufficient background to understand the code fragments appearing in this paper.)

As a reader's guide to the code from Figure 4.1, by convention the type variables `z`, `zp`, `zq` always represent (respectively) the integer ring $\mathbb{Z}$ and quotient rings $\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z}, \mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$, where $p \ll q$ are respectively the plaintext and ciphertext moduli. The types `m`, `m'` respectively represent the indices m, m' of the cyclotomic rings R, R', where we need $m|m'$ so that R can be seen as a subring of R'. Combining all this, the types `Cyc m' z`, `Cyc m zp`, and `Cyc m' zq` respectively represent R', the plaintext ring $R_p = R/pR$, and the ciphertext ring $R'_q = R'/qR'$.

The declaration `encrypt :: (m `Divides` m', ...) => ...` defines the *type* of the function `encrypt` (and similarly for `decrypt`, `rescaleCT`, etc.). Preceding the arrow `=>`, the text `(m `Divides` m', ...)` lists the *constraints* that the types must satisfy at compile time; here the first constraint enforces that $m|m'$. The text following the arrow `=>` defines the types of the inputs and output. For `encrypt`, the inputs are a secret key in R' and a plaintext in R'_p, and the output is a random ciphertext over R'_q. Notice that the full ciphertext type also includes the types `m` and `zp`, which indicate that the plaintext is from R_p. This aids safety: thanks to the type of `decrypt`, the type system prevents the programmer from incorrectly attempting to decrypt the ciphertext into a ring other than R_p.

Finally, each function declaration is followed by an implementation, which describes how the output is computed from the input(s). Because the implementations rely on the mathematical definition of the scheme, we defer further discussion to section 4.3.

```
encrypt :: (m `Divides` m', MonadRandom rnd, ...)
  => SK (Cyc m' z)                -- secret key ∈ R'
  -> PT (Cyc m  zp)               -- plaintext ∈ R_p
  -> rnd (CT m zp (Cyc m' zq))    -- ciphertext over R'_q

encrypt (SK s) mu = do            -- in randomness monad
  e  <- errorCoset (embed mu)     -- error ← μ + pR'
  c1 <- getRandom                 -- uniform from R'_q
  return $ CT LSD 0 1 [reduce e - c1 * reduce s, c1]

decrypt :: (Lift zq z, Reduce z zp, ...)
  => SK (Cyc m' z)                -- secret key ∈ R'
  -> CT m zp (Cyc m' zq)          -- ciphertext over R'_q
  -> PT (Cyc m zp)                -- plaintext in R_p

decrypt (SK s) (CT LSD k l c) =
  let e = liftDec $ evaluate c (reduce s)
  in l *> twace (iterate divG (reduce e) !! k)

-- homomorphic multiplication
(CT LSD k1 l1 c1) * (CT _ k2 l2 c2) =
  CT d2 (k1+k2+1) (l1*l2) (mulG <$> c1 * c2)

-- ciphertext modulus switching
rescaleCT :: (Rescale zq zq', ...)
  => CT m zp (Cyc m' zq )         -- ciphertext over R'_q
  -> CT m zp (Cyc m' zq')         -- to R'_q'

rescaleCT (CT MSD k l [c0,c1]) =
  CT MSD k l [rescaleDec c0, rescalePow c1]

-- key switching/linearization
keySwitchQuad :: (MonadRandom rnd, ...)
  => SK r' -> SK r'               -- target, source keys
  -> rnd (CT m zp r'q -> CT m zp r'q) -- recrypt function

keySwitchQuad sout sin = do       -- in randomness monad
  hint <- ksHint sout sin
  return $ \(CT MSD k l [c0,c1,c2]) ->
    CT MSD k l $ [c0,c1] + switch hint c2

switch hint c =
  sum $ zipWith (*>) (reduce <$> decompose c) hint
```

Figure 4.1: Representative (and approximate) code from our implementation of an SHE scheme in $\Lambda \circ \lambda$.

4.1.2 Related Work

FHEW. FHEW [DM15] is an implementation of a very fast bootstrapping algorithm for "third-generation" FHE schemes [GSW13; AP14]. However, it is not intended for general-purpose homomorphic computations, since the scheme encrypts only one bit per ciphertext. Our implementation supports large plaintext rings, which allows much higher throughtput.

HElib. HElib [HS] is an "assembly language" for BGV-style HE over cyclotomic rings [BGV14]. It holds speed records for a variety of HE benchmarks (e.g., homomorphic AES computation [GHS12c]), and appears to be the sole public implementation of many advanced HE features, like bootstrapping for "packed" ciphertexts [HS15]. However, it does not use the best known algorithms for cryptographic operations in general (non-power-of-two) cyclotomics, which results in more complex and less efficient algorithms, and suboptimal noise growth in cryptographic schemes.

Our SHE scheme is implemented with $\Lambda\circ\lambda$, which uses a much better representation for arbitrary cyclotomic rings. This results in improved efficiency compared to HELib, despite its emphasis on performance (see subsection 4.4.2 for details.)

Computational Platform. Several SHE implementations target specialized computational platforms like FPGAs [Cou+14] and GPUs [Wan+12]. Since our implementation uses $\Lambda\circ\lambda$, the hardware platform is completely abstracted away from the FHE functionality. This means it is easy to make our SHE scheme run on FPGAs, GPUs, use vector instruction sets, multi-core CPUs, and more. In particular, it is possible to include the highly-optimized code from [HS] for two-power cyclotomic rings into an $\Lambda\circ\lambda$ backend to obtain the efficiency of HELib, while simultaneoulsy enjoying the safety and advanced functionality of our implementation.

4.1.3 Organization

The rest of this chapter is organized as follows:

Section 4.2 describes *ring-tunneling* for HE, a method of ring-switching which improves upon prior work of [AP13].

Section 4.3 gives the design and implementation of our SHE scheme using $\Lambda \circ \lambda$, including the implementation of ring-tunneling.

Section 4.4 uses the SHE implementation to evaluate $\Lambda \circ \lambda$ in terms of code quality and runtime performance, and gives a comparison to HElib [HS].

Acknowledgments. We thank Tancrède Lepoint for providing HElib benchmark code and Victor Shoup for helpful discussions regarding HElib performance.

4.2 Efficient Ring-Switching

The term "ring switching" encompasses a collection of techniques, introduced in [BGV14; Gen+13; AP13], that allow one to change the ciphertext ring for various purposes. These techniques can also induce a corresponding change in the plaintext ring, at the same time applying a desired linear function to the underlying plaintext.

In this section we describe an new, more efficient instantiation of homomorphic ring-switching which we call "ring-tunneling". This operation was first described in [Gen+12], and an improved version called ring-hopping was given in [AP13]. These prior works focus mainly on the mathematical description and analysis and the procedures, and do not give many details regarding efficient algorithms or concrete implementation.

Ring-switching provides the following functionality: given a ciphertext over a certain cyclotomic ring R, it transforms it into a ciphertext over another cyclotomic ring S, with the effect of applying a linear function to the original plaintext coefficients (with respect to a certain basis of R). The transformation is implemented by passing through a sequence

of "hybrid" rings which gradually interpolate between R and S, while also gradually transforming the coefficients via a sequence of linear functions..

We observed that the ring-hopping procedure as described in [AP13] has a significant bottleneck in its use of so-called *compositum* rings, which in practice can be very large (of dimension 200,000 or more) and thus expensive to work in. Here we describe an alternative procedure that avoids compositum rings altogether, working entirely within rings whose dimensions are essentially only as large as they need to be for security (e.g., in the low thousands in our application). This yields a major runtime improvement, of at least an order of magnitude (as compared with the procedure described in [AP13]).

In a bit more detail, the relationship between [AP13] and our work is as follows. To "hop" from one hybrid ring to the next, the procedure from [AP13] embeds into their compositum ring (i.e., smallest common super-ring), and then uses ring-switching [Gen+12] to map into the target hybrid ring. Here we show how to avoid the compositum by instead decomposing elements over the largest common subring; this also leaves no explicit need for key-switching. See Figure 4.2 for a visual comparison of the two methods.

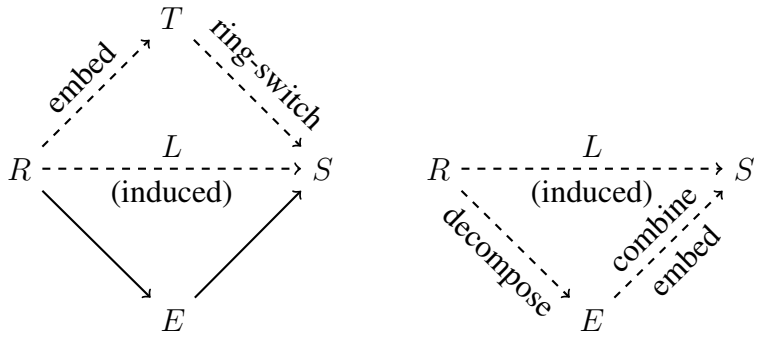

Figure 4.2: Comparison of ring hopping and ring tunneling from a ring H to a ring H'. On the left is the method from [AP13], which passes through the (large) compositum ring T. On the right is our more efficient version, which decomposes the secret key and ciphertext into E-elements, then combines them appropriately while embedding into H'.

4.2.1 Linear Functions

Here we recall the main algebraic facts needed to explain our instantiation of ring-hopping. This section relies heavily on section 2.2, especially regarding mod-p CRT sets (subsection 2.2.6). In addition, we will need some basic theory of linear functions on rings. Let E be a common subring of some rings R, S. A function $L\colon R \to S$ is *E-linear* if for all $r, r' \in R$ and $e \in E$,

$$L(r + r') = L(r) + L(r') \quad \text{and} \quad L(e \cdot r) = e \cdot L(r).$$

From this it follows that for any E-basis $\vec{b}$ of R, an E-linear function L is uniquely determined by its values $y_j = L(b_j) \in S$. Specifically, if $r = \vec{b}^t \cdot \vec{e} \in R$ for some $\vec{e}$ over E, then $L(r) = L(\vec{b})^t \cdot \vec{e} = \vec{y}^t \cdot \vec{e}$.

Extending linear functions. Now let E', R', S' respectively be cyclotomic extension rings of E, R, S satisfying certain conditions described below. As part of ring switching we will need to *extend* an E-linear function $L\colon R \to S$ to an E'-linear function $L'\colon R' \to S'$ that agrees with L on R, i.e., $L'(r) = L(r)$ for every $r \in R$. The following lemma gives a sufficient condition for when and how this is possible.

Lemma 4.2.1. *Let e,r,s,e',r',s' respectively be the indices of cyclotomic rings E,R,S,E',R',S', and suppose $e = \gcd(r, e')$, $r' = \mathrm{lcm}(r, e')$, and $\mathrm{lcm}(s, e')|s'$. Then:*

1. *The relative decoding bases $\vec{d}_{r,e}$ of R/E and $\vec{d}_{r',e'}$ of R'/E' are identical.*
2. *For any E-linear function $L\colon R \to S$, the function $L'\colon R' \to S'$ defined by $L'(\vec{d}_{r',e'}) = L(\vec{d}_{r,e})$ is E'-linear and agrees with L on R.*

Proof. First observe that L' is indeed well-defined and is E-linear, by definition of the ring operations of $R' \cong R \otimes_E E'$. Now observe that L' is in fact E'-linear: any $e' \in E'$ embeds into R' as $1 \otimes e'$, so E'-linearity follows directly from the definition of L' and the

mixed-product property. Also, any $r \in R$ embeds into R' as $r \otimes 1$, and $L'(r \otimes 1) = L(r) \cdot 1$, so L' agrees with L on R.

Finally, observe that because $R' \cong R \otimes_E E'$, the index of E is the gcd of the indices of R, E', and the index of R' is their lcm. Then by the Kronecker-product factorization of decoding bases, the relative decoding bases of R/E and of R'/E' are the Kronecker products of the exact same components, in the same order. (This can be seen by considering each prime divisor of the index of R' in turn.) □

4.2.2 Error Invariant

In cryptographic applications, error terms are combined in various ways, and thereby grow in size. To obtain the best concrete parameters and security levels, the accumulated error should be kept as small as possible. More precisely, its coefficients with respect to some choice of $\mathbb{Z}$-basis should have magnitudes that are as small as possible.

As shown in [LPR13a, Section 6], errors e whose coordinates $\sigma_i(e)$ in the canonical embedding are small and (nearly) independent have correspondingly small coefficients with respect to the *decoding* basis of $R^\vee$. In the tweaked setting, where errors e' and the decoding basis both carry an extra $t_m = \hat{m}/g_m$ factor, an equivalent hypothesis is the following, which we codify as an invariant that applications should maintain:

Invariant 4.2.2 (Error Invariant). For an error $e' \in R$, every coordinate

$$\sigma_i(e'/t_m) = \hat{m}^{-1} \cdot \sigma_i(e' \cdot g_m) \in \mathbb{C}$$

should be nearly independent (up to conjugate symmetry) and have relatively "light" (e.g., subgaussian or subexponential) tails.

As already mentioned, the invariant is satisfied for fresh errors drawn from tweaked Gaussians, as well as for small linear combinations of such terms. In general, the invariant is *not* preserved under multiplication, because the product of two tweaked error terms

$e'_i = t_m \cdot e_i$ carries a t_m^2 factor. Fortunately, this is easily fixed by introducing an extra g_m factor:

$$g_m \cdot e'_1 \cdot e'_2 = t_m \cdot (\hat{m} \cdot e_1 \cdot e_2)$$

satisfies the invariant, because multiplication is coordinate-wise under σ. We use this technique in the implementation of our SHE scheme in section 4.3.

4.2.3 Ring tunneling as key switching.

Abstractly, ring tunneling is an operation that homomorphically evaluates a desired E_p-linear function $L_p \colon R_p \to S_p$ on a plaintext, by converting its ciphertext over R'_q to one over S'_q. Operationally, it can be implemented simply as a form of key switching.

Ring tunneling involves two phases: a preprocessing phase where we use the desired linear function L_p and the secret keys to produce appropriate hints, and an online phase where we apply the tunneling operation to a given ciphertext using the hint. The preprocessing phase is as follows:

1. *Extend* L_p to an E'_p-linear function $L'_p \colon R'_p \to S'_p$ that agrees with L_p on R_p, as described above.

2. *Lift* L'_p to a "small" E'-linear function $L' \colon R' \to S'$ that induces L'_p. Specifically, define L' by $L'(\vec{d}_{r',e'}) = \vec{y}$, where $\vec{y}$ (over S') is obtained by lifting $\vec{y}_p = L'_p(\vec{d}_{r',e'})$ using the powerful basis.

   The above lifting procedure is justified by the following considerations. We want L' to map ciphertext errors in R' to errors in S', maintaining Invariant 4.2.2 in the respective rings. In the relative decoding basis $\vec{d}_{r',e'}$, ciphertext error $e = \vec{d}^t_{r',e'} \cdot \vec{e} \in R'$ has E'-coefficients $\vec{e}$ that satisfy the invariant for E', and hence for S' as well. Because we want

$$L'(e) = L'(\vec{d}^t_{r',e'} \cdot \vec{e}) = \vec{y}^t \cdot \vec{e} \in S'$$

to satisfy the invariant for S', it is therefore best to lift $\vec{y}_p$ from S'_p to S' using the powerful basis, for the same reasons that apply to modulus switching when rescaling the c_1 component of a ciphertext.[1]

3. *Prepare* an appropriate key-switching hint using keys $s_{\text{in}} \in R'$ and $s_{\text{out}} \in S'$. Let $\vec{b}$ be an arbitrary E'-basis of R' (which we also use in the online phase below). Using a gadget vector $\vec{g}$ over S'_q, generate key-switching hints H_j for the components of $L'(s_{\text{in}} \cdot \vec{b}^t)$, such that

$$(1, s_{\text{out}}) \cdot H_j \approx L'(s_{\text{in}} \cdot b_j) \cdot \vec{g}^t \pmod{qS'}. \tag{4.2.1}$$

(As usual, the approximation hides appropriate Ring-LWE errors that satisfy Invariant 4.2.2.) Recall that we can interpret the columns of H_j as linear polynomials.

The online phase proceeds as follows. As input we are given an MSD-form, linear ciphertext $c(S) = c_0 + c_1 S$ (over R'_q) with associated integer $k = 0$ and arbitrary $l \in \mathbb{Z}_p$, encrypting a message $\mu \in R_p$ under secret key s_{in}.

1. Express c_1 uniquely as $c_1 = \vec{b}^t \cdot \vec{e}$ for some $\vec{e}$ over E'_q (where $\vec{b}$ is the same E'-basis of R' used in step 3 above).

2. Compute $L'(c_0) \in S'_q$, apply the core key-switching operation to each e_j with hint H_j, and sum the results. Formally, output a ciphertext having $k = 0$, the same $l \in \mathbb{Z}_p$ as the input, and the linear polynomial

$$c'(S) = L'(c_0) + \sum_j H_j \cdot g^{-1}(e_j) \pmod{qS'}. \tag{4.2.2}$$

[1] The very observant reader may notice that because $L'_p(\vec{d}_{r',e'}) = L_p(\vec{d}_{r,e})$ is over S_p, the order in which we extend and lift does not matter.

For correctness, notice that we have

$$\begin{aligned} c_0 + s_{\text{in}} \cdot c_1 &\approx \tfrac{q}{p} \cdot l^{-1} \cdot \mu \pmod{qR'} \\ \implies L'(c_0 + s_{\text{in}} \cdot c_1) &\approx \tfrac{q}{p} \cdot l^{-1} \cdot L(\mu) \pmod{qS'}, \end{aligned} \tag{4.2.3}$$

where the error in the second approximation is L' applied to the error in the first approximation, and therefore satisfies Invariant 4.2.2 by design of L'. Then we have

$$\begin{aligned} c'(s_{\text{out}}) &\approx L'(c_0) + \sum_j L'(s_{\text{in}} \cdot b_j) \cdot \vec{g}^t \cdot g^{-1}(e_j) && \text{(Equations (4.2.2), (4.2.1))} \\ &= L'(c_0 + s_{\text{in}} \cdot \vec{b}^t \cdot \vec{e}) && (E'\text{-linearity of } L') \\ &= L'(c_0 + s_{\text{in}} \cdot c_1) && \text{(definition of } \vec{e}) \\ &\approx \tfrac{q}{p} \cdot l^{-1} \cdot L(\mu) \pmod{qS'} && \text{(Equation (4.2.3))} \end{aligned}$$

as desired, where the error in the first approximation comes from the hints H_j.

Comparison to ring hopping. We now describe the efficiency advantages of ring tunneling versus ring hopping. We analyze the most natural setting where both the input and output ciphertexts are in CRT representation; in particular, this allows the process to be iterated as in [AP13].

Both ring tunneling and ring hopping convert a ciphertext over R'_q to one over S'_q, either via the greatest common subring E'_q (in tunneling) or the compositum T'_q (in hopping). In both cases, the bottleneck is key-switching, where we compute one or more values $H \cdot g^{-1}(c)$ for some hint H and ring element c (which may be over different rings). This proceeds in two main steps:

1. We convert c from CRT to powerful representation for g^{-1}-decomposition, and then convert each entry of $g^{-1}(c)$ to CRT representation. Each such conversion takes $\Theta(n \log n) = \tilde{\Theta}(n)$ time in the dimension n of the ring that c resides in.

2. We multiply each column of H by the appropriate entry of $g^{-1}(c)$, and sum. Because both terms are in CRT representation, this takes linear $\Theta(n)$ time in the dimension n of the ring that H is over.

The total number of components of $g^{-1}(c)$ is the same in both tunneling and hopping, so we do not consider it further in this comparison.

In ring tunneling, we switch $\dim(R'/E')$ elements $e_j \in E'_q$ (see Equation (4.2.2)) using the same number of hints over S'_q. Thus the total cost is

$$\dim(R'/E') \cdot (\tilde{\Theta}(\dim(E')) + \Theta(\dim(S'))) = \tilde{\Theta}(\dim(R')) + \Theta(\dim(T')).$$

By contrast, in ring hopping we first embed the ciphertext into the compositum T'_q and key-switch there. Because the compositum has dimension $\dim(T') = \dim(R'/E') \cdot \dim(S')$, the total cost is

$$\tilde{\Theta}(\dim(T')) + \Theta(\dim(T')).$$

The second (linear) terms of the above expressions, corresponding to step 2, are essentially identical. For the first (superlinear) terms, we see that step 1 for tunneling is at least a $\dim(T'/R') = \dim(S'/E')$ factor faster than for hopping. In typical instantiations, this factor is a small prime between, say, 3 and 11, so the savings can be quite significant in practice.

4.3 Somewhat-Homomorphic Encryption in $\Lambda \circ \lambda$

In this section we describe a full-featured somewhat-homomorphic encryption scheme and its implementation in $\Lambda\circ\lambda$, using the interfaces described in chapter 3. At the mathematical level, the system refines a variety of techniques and features from a long series of works [LPR13b; BV11b; BV14a; BGV14; Gen+13; LPR13a; AP13]. In addition, we describe some important generalizations and include new operations like ring-tunneling. Along with the mathematical

description of each main component, we present the corresponding Haskell code, showing how the two forms match very closely.

Note that like all prior implementations of SHE, our implementation has a relatively low-level interface which corresponds directly to the mathematical operations described in the literature. Actually *using* this interface requires a great deal of expertise. Chapter 5 describes a compiler which drastically simplifies the use of this powerful application.

4.3.1 Keys, Plaintexts, and Ciphertexts

The cryptosystem is parameterized by two cyclotomic rings: $R = \mathcal{O}_m$ and $R' = \mathcal{O}_{m'}$ where $m|m'$, making R a subring of R'. The spaces of keys, plaintexts, and ciphertexts are derived from these rings as follows:

- A *secret key* is an element $s \in R'$. Some operations require s to be "small;" more precisely, we need $s \cdot g_{m'}$ to have small coordinates in the canonical embedding of R' (Invariant 4.2.2). Recall that this is the case for "tweaked" spherical Gaussian distributions.

- The *plaintext ring* is $R_p = R/pR$, where p is a (typically small) positive integer, e.g., $p = 2$. For technical reasons, p must be coprime with every odd prime dividing m'. A plaintext is simply an element $\mu \in R_p$.

- The *ciphertext ring* is $R'_q = R'/qR'$ for some integer modulus $q \geq p$ that is coprime with p. A ciphertext is essentially just a polynomial $c(S) \in R'_q[S]$, i.e., one with coefficients from R'_q in an indeterminant S, which represents the (unknown) secret key. We often identify $c(S)$ with its vector of coefficients $(c_0, c_1, \ldots, c_d) \in (R'_q)^{d+1}$, where d is the degree of $c(S)$.

  In addition, a ciphertext carries a nonnegative integer $k \geq 0$ and a factor $l \in \mathbb{Z}_p$ as auxiliary information. These values are affected by certain operations on ciphertexts, as described below.

Data types. Following the above definitions, our data types for plaintexts, keys, and ciphertexts as follows. The plaintext type `PT rp` is merely a synonym for its argument type `rp` representing the plaintext ring R_p.

The data type `SK` representing secret keys is defined as follows:

```
data SK r' where SK :: ToRational v => v -> r' -> SK r'
```

Notice that a value of type `SK r'` consists of an element from the secret key ring R', and in addition it carries a rational value (of "hidden" type `v`) representing the parameter $v = r^2$ for the (tweaked) Gaussian distribution from which the key was sampled. Binding the parameter to the secret key in this way allows us to automatically generate ciphertexts and other key-dependent information using consistent error distributions, thereby relieving the client of the responsibility for managing error parameters across multiple functions.

The data type `CT` representing ciphertexts is defined as follows:

```
data Encoding    = MSD | LSD
data CT m zp r'q = CT Encoding Int zp (Polynomial r'q)
```

The `CT` type is parameterized by three arguments: a cyclotomic index `m` and a $\mathbb{Z}_p$-representation `zp` defining the plaintext ring R_p, and a representation `r'q` of the ciphertext ring R'_q. A `CT` value has four components: a flag indicating the "encoding" of the ciphertext (MSD or LSD; see below); the auxiliary integer k and factor $l \in \mathbb{Z}_p$ (as mentioned above); and a polynomial $c(S)$ over R'_q.

Decryption relations. A ciphertext $c(S)$ (with auxiliary values $k \in \mathbb{Z}, l \in \mathbb{Z}_p$) encrypting a plaintext $\mu \in R_p$ under secret key $s \in R'$ satisfies the relation

$$c(s) = c_0 + c_1 s + \cdots + c_d s^d = e \pmod{qR'} \tag{4.3.1}$$

for some sufficiently "small" error term $e \in R'$ such that

$$e = l^{-1} \cdot g_{m'}^k \cdot \mu \pmod{pR'}. \tag{4.3.2}$$

By "small" we mean that the error satisfies Invariant 4.2.2, so that all the coefficients of e with respect to the decoding basis have magnitudes smaller than $q/2$. This will allow us to correctly recover $e' \in R'$ from its value modulo q, by "lifting" the latter using the decoding basis.

We say that a ciphertext satisfying Equations (4.3.1) and (4.3.2) is in "least significant digit" (LSD) form, because the message μ is encoded as the error term modulo p. An alternative form, which is more convenient for certain homomorphic operations, is the "most significant digit" (MSD) form. Here the relation is

$$c(s) \approx \tfrac{q}{p} \cdot (l^{-1} \cdot g_{m'}^k \cdot \mu) \pmod{qR'}, \tag{4.3.3}$$

where the approximation hides a small fractional error term (in $\frac{1}{p}R'$) that satisfies Invariant 4.2.2. Notice that the message is represented as a multiple of $\frac{q}{p}$ modulo q, hence the name "MSD." One can losslessly transform between LSD and MSD forms in linear time, just by multiplying by appropriate $\mathbb{Z}_q$-elements (see [AP13, Appendix A]). Each such transformation implicitly multiplies the plaintext by some fixed element of $\mathbb{Z}_p$, which is why a ciphertext carries an auxiliary factor $l \in \mathbb{Z}_p$ that must be accounted for upon decryption.

4.3.2 Encryption and Decryption

To encrypt a message $\mu \in R_p$ under a key $s \in R'$, one does the following:

1. sample an error term $e \in \mu + pR'$ (from a distribution that should be a p factor wider than that of the secret key);
2. sample a uniformly random $c_1 \leftarrow R'_q$;

3. output the LSD-form ciphertext $c(S) = (e - c_1 \cdot s) + c_1 \cdot S \in R'_q[S]$, with $k = 0, l = 1 \in \mathbb{Z}_p$.

   (Observe that $c(s) = e \pmod{qR'}$, as desired.)

This translates directly into just a few lines of Haskell code, which is monadic due to its use of randomness:

```
encrypt :: (m `Divides` m', MonadRandom rnd, ...)
           => SK (Cyc m' z)
              -> PT (Cyc m zp)
              -> rnd (CT m zp (Cyc m' zq))
encrypt (SK v s) mu = do
  e  <- errorCoset v (embed mu)  -- error from μ + pR'
  c1 <- getRandom                -- uniform from R'_q
  return $ CT LSD zero one $ fromCoeffs [reduce e - c1 * reduce s, c1]
```

To decrypt an LSD-form ciphertext $c(S) \in R'_q[S]$ under secret key $s \in R'$, we first evaluate $c(s) \in R'_q$ and then lift the result to R' (using the decoding basis) to recover the error term e, as follows:

```
errorTerm :: (Lift zq z, m `Divides` m', ...)
             => SK (Cyc m' z) -> CT m zp (Cyc m' zq) -> Cyc m' z
errorTerm (SK _ s) (CT LSD _ _ c) = liftDec (evaluate c (reduce s))
```

Following Equation (4.3.2), we then compute $l \cdot g_{m'}^{-k} \cdot e \bmod pR'$. This yields the *embedding* of the message μ into R'_p, so we finally take the twace to recover $\mu \in R_p$ itself:

```
decrypt :: (Lift zq z, Reduce z zp, ...)
           => SK (Cyc m' z) -> CT m zp (Cyc m' zq) -> PT (Cyc m zp)
decrypt sk ct@(CT LSD k l _) =
  let e = reduce (errorTerm sk ct)
  in (scalarCyc l) * twace (iterate divG e !! k)
```

4.3.3 Homomorphic Addition and Multiplication

Homomorphic addition of ciphertexts with the same values of k and l is simple: convert the ciphertexts to the same form (MSD or LSD), then add their polynomials. It is also possible adjust the values of k, l as needed by multiplying the polynomial by an appropriate factor, which only slightly enlarges the error. Accordingly, we define `CT m zp (Cyc m' zq)` to be an instance of `Additive`, for appropriate argument types.

Now consider homomorphic multiplication: suppose ciphertexts $c_1(S), c_2(S)$ encrypt messages μ_1, μ_2 in LSD form, with auxiliary values k_1, l_1 and k_2, l_2 respectively. Then

$$\begin{aligned} g_{m'} \cdot c_1(s) \cdot c_2(s) &= g_{m'} \cdot e_1 \cdot e_2 \pmod{qR'}, \\ g_{m'} \cdot e_1 \cdot e_2 &= (l_1 l_2)^{-1} \cdot g_{m'}^{k_1+k_2+1} \cdot (\mu_1 \mu_2) \pmod{pR'}, \end{aligned}$$

and the error term $e = g_{m'} \cdot e_1 \cdot e_2$ satisfies Invariant 4.2.2, because e_1, e_2 do (see subsection 4.2.2). Therefore, the LSD-form ciphertext

$$c(S) := g_{m'} \cdot c_1(S) \cdot c_2(S) \in R'_q[S]$$

encrypts $\mu_1 \mu_2 \in R_p$ with auxiliary values $k = k_1 + k_2 + 1$ and $l = l_1 l_2 \in \mathbb{Z}_p$. Notice that the degree of the output polynomial is the sum of the degrees of the input polynomials.

More generally, it turns out that we only need one of $c_1(S), c_2(S)$ to be in LSD form; the product $c(S)$ then has the same form as the other ciphertext.[2] All this translates immediately to an instance of `Ring` for `CT m zp (Cyc m' zq)`, with the interesting case of multiplication having the one-line implementation

```
(CT LSD k1 l1 c1) * (CT d2 k2 l2 c2) =
  CT d2 (k1+k2+1) (l1*l2) (mulG <$> c1 * c2)
```

[2] If both ciphertexts are in MSD form, then it is possible to use the "scale free" homomorphic multiplication method of [Bra12], but we have not implemented it because it appears to be significantly less efficient than just converting one ciphertext to LSD form.

(The other cases just swap the arguments or convert one ciphertext to LSD form, thus reducing to the case above.)

4.3.4 Modulus Switching

Switching the ciphertext modulus is a form of rescaling typically used for decreasing the modulus, which commensurately reduces the *absolute magnitude* of the error in a ciphertext—though the error *rate* relative to the modulus stays essentially the same. Because homomorphic multiplication implicitly multiplies the error terms, keeping their absolute magnitudes small can yield major benefits in controlling the error growth. Modulus switching is also sometimes useful to temporarily *increase* the modulus, as explained in the next subsection.

Modulus switching is easiest to describe and implement for ciphertexts in MSD form (Equation (4.3.3)) that have degree at most one. Suppose we have a ciphertext $c(S) = c_0 + c_1 S$ under secret key $s \in R'$, where

$$c_0 + c_1 s = d \approx \tfrac{q}{p} \cdot \gamma \pmod{qR'}$$

for $\gamma = l^{-1} \cdot g^k_{m'} \cdot \mu \in R_p$. Switching to a modulus q' is just a suitable rescaling of each $c_i \in R'_q$ to some $c'_i \in R'_{q'}$ such that $c'_i \approx (q'/q) \cdot c_i$; note that the right-hand sides here are fractional, so they need to be discretized using an appropriate basis (see the next paragraph). Observe that

$$c'_0 + c'_1 s \approx \tfrac{q'}{q}(c_0 + c_1 s) = \tfrac{q'}{q} \cdot d \approx \tfrac{q'}{p} \cdot \gamma \pmod{q'R'},$$

so the message is unchanged but the absolute error is essentially scaled by a q'/q factor.

Note that the first approximation above hides the extra discretization error $e_0 + e_1 s$ where $e_i = c'_i - \frac{q'}{q} c_i$, so the main question is what *bases* of R' to use for the discretization, to best maintain Invariant 4.2.2. We want both e_0 and $e_1 s$ to satisfy the invariant, which means

we want the entries of $\sigma(e_0 \cdot g_{m'})$ and $\sigma(e_1 s \cdot g_{m'}) = \sigma(e_1) \odot \sigma(s \cdot g_{m'})$ to be essentially independent and as small as possible; because $s \in R'$ itself satisfies the invariant (i.e., the entries of $\sigma(s \cdot g_{m'})$ are small), we want the entries of $\sigma(e_1)$ to be as small as possible. It turns out that these goals are best achieved by rescaling c_0 using the *decoding* basis $\vec{d}$, and c_1 using the *powerful* basis $\vec{p}$. This is because $g_{m'} \cdot \vec{d}$ and $\vec{p}$ respectively have nearly optimal spectral norms over all bases of $g_{m'} R'$ and R', as shown in [LPR13a].

Our Haskell implementation is therefore simply

```
rescaleLinearCT :: (Rescale zq zq', ...)
                   => CT m zp (Cyc m' zq) -> CT m zp (Cyc m' zq')
rescaleLinearCT (CT MSD k l (Poly [c0,c1])) =
  let c'0 = rescaleDec c0
      c'1 = rescalePow c1
  in CT MSD k l $ Poly [c'0, c'1]
```

4.3.5 Key Switching and Linearization

Recall that homomorphic multiplication causes the degree of the ciphertext polynomial to increase. Key switching is a technique for reducing the degree, typically back to linear. More generally, key switching is a mechanism for *proxy re-encryption*: given two secret keys s_{in} and s_{out} (which may or may not be different), one can construct a "hint" that lets an untrusted party convert an encryption under s_{in} to one under s_{out}, while preserving the secrecy of the message and the keys.

Key switching uses a gadget $\vec{g} \in (R'_q)^\ell$ and associated decomposition function $g^{-1} : R'_q \to (R')^\ell$ (both typically promoted from $\mathbb{Z}_q$; see sections 3.2.4 and 3.5.2). Recall that $g^{-1}(c)$ outputs a short vector over R' such that $\vec{g}^t \cdot g^{-1}(c) = c \pmod{qR'}$.

The core operations. Let $s_{\text{in}}, s_{\text{out}} \in R'$ denote some arbitrary secret values. A key-switching hint for s_{in} under s_{out} is a matrix $H \in (R'_q)^{2 \times \ell}$, where each column can be seen as

a linear polynomial over R'_q, such that

$$(1, s_{\text{out}}) \cdot H \approx s_{\text{in}} \cdot \vec{g}^t \pmod{qR'}. \tag{4.3.4}$$

Such an H is constructed simply by letting the columns be Ring-LWE samples with secret s_{out}, and adding $s_{\text{in}} \cdot \vec{g}^t$ to the top row. In essence, such an H is pseudorandom by the Ring-LWE assumption, and hence hides the secrets.

The core key-switching step takes a hint H and some $c \in R'_q$, and simply outputs

$$c' = H \cdot g^{-1}(c) \in (R'_q)^2, \tag{4.3.5}$$

which can be viewed as a linear polynomial $c'(S)$. Notice that by Equation (4.3.4),

$$c'(s_{\text{out}}) = (1, s_{\text{out}}) \cdot c' = ((1, s_{\text{out}}) \cdot H) \cdot g^{-1}(c) \approx s_{\text{in}} \cdot \vec{g}^t \cdot g^{-1}(c) = s_{\text{in}} \cdot c \pmod{qR'}, \tag{4.3.6}$$

where the approximation holds because $g^{-1}(c)$ is short. More precisely, because the error terms in Equation (4.3.4) satisfy Invariant 4.2.2, we want all the elements of the decomposition $g^{-1}(c)$ to have small entries in the canonical embedding, so it is best to decompose relative to the powerful basis.

Following Equation (4.3.5), our Haskell code for the core key-switching step is simply as follows (here `knapsack` computes the inner product of a list of polynomials over R'_q and a list of R'_q-elements):

```
switch :: (Decompose gad zq z, r'q ~ Cyc m' zq, ...)
          => Tagged gad [Polynomial r'q] -> r'q -> Polynomial r'q
switch hint c =
  untag $ knapsack <$> hint <*> (fmap reduce <$> decompose c)
```

Switching ciphertexts. The above tools can be used to switch MSD-form ciphertexts of degree up to d under s_{in} as follows: first publish a hint H_i for each power s_{in}^i, $i = 1, \ldots, d$, all under the same s_{out}. Then to switch a ciphertext $c(S)$:

- For each $i = 1, \ldots, d$, apply the core step to coefficient $c_i \in R'_q$ using the corresponding hint H_i, to get a linear polynomial $c'_i = H_i \cdot g^{-1}(c_i)$. Also let $c'_0 = c_0$.
- Sum the c'_i to get a linear polynomial $c'(S)$, which is the output.

Then $c'(s_{\text{out}}) \approx c(s_{\text{in}}) \pmod{qR'}$ by Equation (4.3.6) above, so the two ciphertexts encrypt the same message.

Notice that the error rate in $c'(S)$ is essentially the sum of two separate quantities: the error rate in the original $c(S)$, and the error rate in H times a factor corresponding to the norm of the output of g^{-1}. We typically set the latter error rate to be much smaller than the former, so that key-switching incurs essentially no error growth. This can be done by constructing H over a modulus $q' \gg q$, and scaling up $c(S)$ to this modulus before decomposing.

Haskell functions. Our implementation includes a variety of key-switching functions, whose types all roughly follow this general form:

```
keySwitchFoo :: (MonadRandom rnd, ...) => SK r' -> SK r'
  -> Tagged (gad, zq') (rnd (CT m zp r'q -> CT m zp r'q))
```

Unpacking this, the inputs are the two secret keys $s_{\text{out}}, s_{\text{in}} \in R'$, and the output is essentially a *re-encryption function* that maps one ciphertext to another. The extra `Tagged (gad,zq')` context indicates what gadget and modulus are used to construct the hint, while the `rnd` wrapper indicates that randomness is used in *constructing* (but not applying) the function; this is because constructing the hint requires randomness.

Outputting a re-encryption function—rather than just a hint itself, which would need to be fed into a separate function that actually does the switching—has advantages in terms

of simplicity and safety. First, it reflects the abstract re-encryption functionality provided by key switching. Second, we implement a variety of key-switching functions that each operate slightly differently, and may even involve different types of hints (e.g., see the next subsection). With our approach, the hint is abstracted away entirely, and each style of key-switching can be implemented by a single client-visible function, instead of requiring two separate functions and a specialized data type.

A prototypical implementation of a key-switching function is as follows (here `ksHint` is a function that constructs a key-switching hint for s_{in} under s_{out}, as described above):

```
-- switch a linear ciphertext from one key to another
keySwitchLinear sout sin = tag $ do  -- rnd monad
  hint :: Tagged gad [Polynomial (Cyc m' zq')] <- ksHint sout sin
  return $ \ (CT MSD k l (Poly [c0,c1])) ->
             CT MSD k l $ Poly [c0] + switch hint c1
```

4.3.6 Ring Tunneling

We provide a simple implementation of ring tunneling in $\Lambda \circ \lambda$, which to our knowledge is the first realization of ring-switching of any kind.

Linear functions. Since ring-tunneling induces a linear function on the plaintext, we introduce a useful abstract data type to represent linear functions on cyclotomic rings:

```
newtype Linear z e r s = D [Cyc s z]
```

The parameters `z` represents the base type, while the parameters `e`, `r`, `s` represent the indices of the cyclotomic rings E, R, S. For example, `Cyc s z` represents the ring S. An E-linear function L is internally represented by its list $\vec{y} = L(\vec{d}_{r,e})$ of values on the relative decoding basis $\vec{d}_{r,e}$ of R/E, hence the constructor named `D`. (We could also represent linear functions via the relative powerful basis, but so far we have not needed to do so.) Using our interface for cyclotomic rings (section 3.5), evaluating a linear function is straightforward:

```
evalLin :: (e `Divides` r, e `Divides` s, ...)
           => Linear z e r s -> Cyc r z -> Cyc s z
evalLin (D ys) r =
  dotprod ys (fmap embed (coeffsCyc Dec r :: [Cyc e z]))
```

Lemma 4.2.1 leads to the following very simple Haskell function to extend a linear function; notice that the constraints use the type-level arithmetic described in subsection 3.2.6 to enforce the hypotheses of Lemma 4.2.1.

```
extendLin :: (e ~ FGCD r e', r' ~ FLCM r e', (FLCM s e') `Divides` s')
             => Linear z e r s -> Linear z e' r' s'
extendLin (Dec ys) = Dec (fmap embed ys)
```

Tunneling. Next we give our implementation of ring tunneling.

```
tunnel f sout sin (CT MSD 0 s c) = tag $ do -- rnd monad
  hints :: [Tagged gad [Polynomial (Cyc t s' zq)]] <-
    tunnelHint f sout sin
  let f' = extendLin $ lift f :: Linear t z e' r' s'
      f'q = reduce f' :: Linear t zq' e' r' s'
      [c0,c1] = coeffs c
      -- apply E-linear function to constant term c0
      c0' = evalLin f'q c0
      -- apply E-linear function to c1 via key-switching
      c1s = coeffsPow c1 :: [Cyc t e' zq']
      c1s' = zipWith switch hints (embed <$> c1s)
      c1' = sum c1s'
  return CT MSD 0 s $ P.const c0' + c1')
```

Here, `tunnelHint` is a function that outputs the hints H_j with respect to the powerful basis as defined in subsection 4.2.3. The rest of the algorithm matches exactly with the steps

outlined in that section: we first lift and extend the linear function, compute $L'(c_0)$, and apply key switching with the appropriate hint to the powerful basis coefficients of c_1. Finally, we sum the results and produce the output ciphertext over S_q.

4.4 Evaluation

Recall that $\Lambda \circ \lambda$ primarily aims to be a general, modular, and safe framework for lattice cryptography, while also achieving acceptable performance. $\Lambda \circ \lambda$ has proven to be extremely flexible and has been used (at least) for the following purposes:

- implementing advanced features of somewhat-homomorphic encryption (section 4.3);
- and creating a homomorphic compiler for homomorphic encryption (chapter 5);
- implementing the pseudorandom functions of [BPR12; BP14]; (chapter 6);
- generating RLWE/RLWR cryptanalytic challenges (chapter 7);
- exploring opportunities for parallelism of lattice cryptography using vector (SIMD) instruction sets (the C++ tensor backend), multi-core CPUs (the Repa tensor backend), and GPUs (currently in progress);
- master's thesis on FHE [Muk16];
- and implementing identity-based encryption [Ret17].

While $\Lambda \circ \lambda$'s modularity and static safety properties are demonstrated elsewhere in the paper, here we evaluate two of its lower-level characteristics: code quality and runtime performance.

For comparison, we also give a similar analysis for HElib [HS], which is $\Lambda \circ \lambda$'s closest analogue in terms of scope and features. (Recall that HElib is a leading implementation of homomorphic encryption.) We emphasize two main caveats regarding such a comparison: first, while $\Lambda \circ \lambda$ and HElib support many common operations and features, they

are not functionally equivalent—e.g., $\Lambda \circ \lambda$ supports ring-switching, error sampling, and certain gadget operations that HElib lacks, while HElib supports ring automorphisms and sophisticated plaintext "shuffling" operations that $\Lambda \circ \lambda$ lacks. Second, $\Lambda \circ \lambda$'s host language (Haskell) is somewhat higher-level than HElib's (C++), so any comparisons of code quality or performance will necessarily be "apples to oranges." Nevertheless, we believe that such a comparison is still meaningful and informative, as it quantifies the relative trade-offs of the two approaches in terms of software engineering values like simplicity, maintainability, and performance.

Summary. Our analysis shows that $\Lambda \circ \lambda$ offers high code quality, with respect to both the size and complexity. In particular, $\Lambda \circ \lambda$'s code base is about 7–8 times smaller than HElib's. Also, $\Lambda \circ \lambda$ currently offers good performance, always within an order of magnitude of HElib's, and we expect that it can substantially improve with focused optimization. Notably, $\Lambda \circ \lambda$'s C++ backend is already *faster* than HElib in Chinese Remainder Transforms for non-power-of-two cyclotomic indices with small prime divisors, due to the use of better algorithms associated with the "tensored" representations. For example, a CRT for index $m = 2^6 3^3$ (of dimension $n = 576$) takes about 99 μs in $\Lambda \circ \lambda$, and 153 μs in HElib on our benchmark machine (and the performance gap grows when more primes are included).

4.4.1 Source Code Analysis

We analyzed the source code of all "core" functions from $\Lambda \circ \lambda$ and HElib, and calculated a few metrics that are indicative of code quality and complexity: actual lines of code, number of functions, and *cyclotomatic complexity* [McC76]. "Core" functions are any that are called (directly or indirectly) by the libraries' intended public interfaces. These include, e.g., algebraic, number-theoretic, and cryptographic operations, but not unit tests, benchmarks, etc. Note that HElib relies on NTL [Sho06] for the bulk of its algebraic operations (e.g., cyclotomic and finite-field arithmetic), so to give a fair comparison we

include *only* the relevant portions of NTL with HElib, referring to their combination as HElib+NTL. Similarly, $\Lambda\circ\lambda$ includes a `Tensor` backend written in C++ (along with a pure Haskell one), which we identify separately in our analysis.

Source Lines of Code

A very basic metric of code complexity is program size as measured by *source lines of code* (SLOC). We measured SLOC for $\Lambda\circ\lambda$ and HElib+NTL using Ohcount [Bla14] for Haskell code and *metriculator* [KW11] for C/C++ code. Metriculator measures *logical* source lines of code, which approximates the number of "executable statements." By contrast, Ohcount counts *physical* lines of code. Both metrics exclude comments and empty lines, so they do not penalize for documentation or extra whitespace. While the two metrics are not identical, they provide a rough comparison between Haskell and C/C++ code.

Table 4.1 shows the SLOC counts for $\Lambda\circ\lambda$ and HElib+NTL. Overall, $\Lambda\circ\lambda$ consists of only about $5{,}000$ lines of code, or $4{,}200$ if we omit the C++ portion (whose functionality is redundant with the Haskell code). By contrast, HElib+NTL consists of about 7–8 times as much code.

Table 4.1: Source lines of code for $\Lambda\circ\lambda$ and HElib+NTL.

Codebase	SLOC		Total
$\Lambda\circ\lambda$	Haskell	C++	
	4,257	734	4,991
HElib+NTL	HElib	NTL	
	14,709	20,073	34,782

Cyclomatic Complexity and Function Count

McCabe's *cyclomatic complexity* (CC) [McC76] counts the number of "linearly independent" execution paths through a piece of code (usually, a single function), using the control-flow graph. The theory behind this metric is that smaller cyclomatic complexity typically

corresponds to simpler code that is easier to understand and test thoroughly. McCabe suggests limiting the CC of functions to ten or less.

Results. Table 4.2 gives a summary of cyclomatic complexities in $\Lambda\circ\lambda$ and HElib+NTL. A more detailed breakdown is provided in Figure 4.3. In both codebases, more than $80\,\%$ of the functions have a cyclomatic complexity of 1, corresponding to straight-line code having no control-flow statements; these are omitted from Figure 4.3.

Table 4.2: Number of functions per argon grade: cyclomatic complexities of 1–5 earn an 'A,' 6–10 a 'B,' and 11 or more a 'C.'

Codebase	A	B	C	Total
$\Lambda\circ\lambda$	1,234	14	5	1,253
HElib+NTL	6,850	159	69	7,078

Only three Haskell functions and two C++ functions in $\Lambda\circ\lambda$ received a grade of 'C.' The Haskell functions are: adding `Cyc` elements (CC=23); multiplying `Cyc` elements (CC=14); and comparing binary representations of positive integers, for promotion to the type level (CC=13). In each of these, the complexity is simply due to the many combinations of cases for the representations of the inputs (see subsection 3.5.2). The two C++ functions are the inner loops of the CRT and DFT transforms, with CC 16 and 18, respectively. This is due to a case statement that chooses the appropriate unrolled code for a particular dimension, which we do for performance reasons.

For comparison, HElib+NTL has many more functions than $\Lambda\circ\lambda$ (see Table 4.2), and those functions tend to be more complex, with 68 functions earning a grade of 'C' (i.e., CC more than 10).

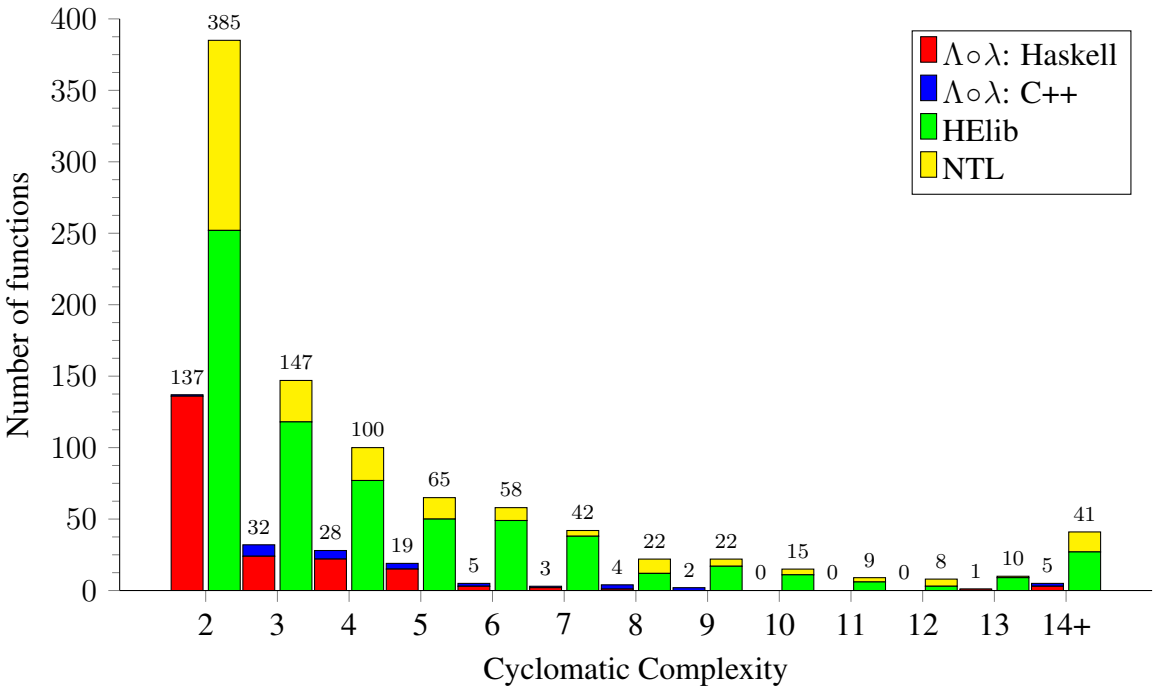

Figure 4.3: Cyclomatic complexity (CC) of functions in $\Lambda\circ\lambda$ and HElib+NTL. The case CC=1 accounts for more than 80% of the functions in each codebase, and is suppressed.

4.4.2 Performance

Here we report on the runtime performance of $\Lambda\circ\lambda$. As a general-purpose library, we do not expect it to be competitive with highly optimized (but inflexible) C implementations like SWIFFT [Lyu+08] and BLISS [Duc+13], but we aim for performance in the same league as higher-level libraries like HElib.

Here we give microbenchmark data for various common operations and parameter sets, to show that performance is reasonable and to establish a baseline for future work. All benchmarks were run by the standard Haskell benchmarking tool *criterion* [OSu14] on a mid-2012 model Asus N56V laptop with 2.3GHz Core i7-3610QM CPU and 6 GB 1600MHz DDR3 RAM, using GHC 8.0.1. All moduli in our benchmarks are smaller than 32 bits, so that all mod-q arithmetic can be performed naïvely in 64-bit registers.

We benchmarked the two Tensor backends currently included in $\Lambda\circ\lambda$: the "CT" backend is sequential and written in relatively unoptimized C++. The "RT" backend uses the Repa

array library [Kel+10; Lip+12]. For operations that $\Lambda \circ \lambda$ and HElib have in common, we also include HElib benchmarks.

Most of our optimization efforts have been devoted to the CT backend, which partially explains the poor performance of the Repa backend; we believe that similarly tuning RT could speed up benchmarks considerably. However, RT performance is currently limited by the architecture of our tensor DSL, which is blocking many compiler optimizations. Specifically, the higher-rank types that make the DSL work for arbitrary cyclotomic indices also make specialization, inlining, and fusion opportunities much more difficult for the compiler to discover. Addressing this issue to obtain a fast and general pure-Haskell implementation is an important problem for future work.

Cyclotomic Ring Operations

Table 4.3, Table 4.4, and Table 4.5 show runtimes for the main cyclotomic ring operations. We compare $\Lambda \circ \lambda$'s C++ (CT) and Repa (RT) Tensor backends, and HElib whenever it supports analogous functionality. For CT and RT, operations scale approximately linearly in the number of moduli in the RNS representation, so all the runtimes are shown for a single modulus. For a cyclotomic ring $\mathcal{O}_m$, we consider only "good" prime moduli $q = 1 \bmod m$, so that the CRT exists over $\mathbb{Z}_q$. Benchmarks are reported for the `UCyc` interface; times for analogous operations in the `Cyc` interface are essentially identical, except where noted. All times are reported in microseconds (μs).

Table 4.3: Runtimes (in microseconds) for conversion between the powerful (**P**) and CRT (**C**) bases, and between the decoding (**D**) and powerful bases (**P**). For comparison with our **P**↔**C** conversions, we include HElib's conversions between its "polynomial" and "Double CRT" (with one modulus) representations. Note that HElib is primarily used with many (small) moduli, where the conversion from Double CRT to polynomial representation is closer in speed to the other direction.

Index m	$\varphi(m)$	`UCyc P→C`			`UCyc C→P`			`UCyc D→P`		`UCyc P→D`	
		HElib	CT	RT	HElib	CT	RT	CT	RT	CT	RT
$2^{10} = 1{,}024$	512	15.9	139	2,344	38.3	142	2,623	0.7	0.02	0.7	0.02
$2^{11} = 2{,}048$	1,024	32.4	307	5,211	74.4	314	5,618	1.3	0.02	1.2	0.02
$2^6 3^3 = 1{,}728$	576	153	99	3,088	361	122	3,284	4.0	80.3	4.0	64.2
$2^6 3^4 = 5{,}184$	1,728	638	364	10,400	1,136	426	11,030	11.8	226	11.7	186
$2^6 3^2 5^2 = 14{,}400$	3,840	2,756	1,011	24,330	5,659	1,258	25,170	65.8	1,199	61.5	938

Table 4.4: Runtimes (in microseconds) for multiplication by g in the powerful (**P**) and CRT (**C**) bases, division by g in the powerful and decoding (**D**) bases, lifting from R_q to R in the powerful basis, and multiplication of ring elements in the CRT basis. (Multiplication by g in the decoding and powerful bases takes about the same amount of time, and multiplication and division by g in the CRT basis take about the same amount of time.)

Index m	(*) for `UCyc C`			(*g) for `UCyc P`		(*g) for `UCyc C`		(/g) for `UCyc P`		(/g) for `UCyc D`		`lift UCyc P`	
	HElib	CT	RT	CT	RT	CT	RT	CT	RT	CT	RT	CT	RT
1,024	1.8	7.8	73.0	0.7	0.02	5.4	72.0	5.9	56.8	5.9	56.7	1.0	39.8
2,048	4.4	15.6	142	1.2	0.02	11.4	140	11.6	110	11.6	108	2.0	77.0
1,728	2.6	9.3	82.1	10.5	107	6.1	84.0	52.6	390	33.4	385	1.2	45.8
5,184	6.2	26.3	248	30.4	333	18.1	245	155	1,148	102	1,115	3.4	128
14,400	11.6	58.9	589	134	1,515	39.6	575	663	4,679	400	5,283	13.3	297

Table 4.5: Runtimes (in microseconds) of twace and embed for UCyc. (For both CT and RT, twace UCyc D has essentially the same performance as twace UCyc P.) Due to an unresolved compiler issue, embed (in any basis) with the Cyc interface is considerably slower than the analagous UCyc operation benchmarked here.

m	m'	twace UCyc P		twace UCyc C		embed UCyc P		embed UCyc D		embed UCyc C	
		CT	RT	CT	RT	CT	RT	CT	RT	CT	RT
728	2,912	0.7	25.9	22.7	305	3.8	57.2	4.9	58.3	38.7	92.9
728	3,640	0.7	27.1	22.9	258	3.8	56.8	8.5	83.6	39.6	95.5
128	11,648	0.2	7.0	92.5	967	10.8	164	19.7	189	166	393

SHE Scheme

Table 4.6 and Table 4.7 show runtimes for certain main operations of the SHE scheme described in section 4.3. All times are reported in milliseconds (ms). We stress that unlike for our cyclotomic operations above, we have not yet designed appropriate "hints" to assist the compiler's optimizations, and we expect that performance can be significantly improved by such an effort.

Table 4.6: Runtimes (in milliseconds) for basic SHE functionality, including encrypt, decrypt, ciphertext multiplication, addPublic, and mulPublic. All ciphertext operations were performed on freshly encrypted values. The plaintext index for both parameter sets is $m = 16$. For encrypt, the bottleneck is in Gaussian sampling and randomness generation, which was done using the HashDRBG pseudorandom generator with SHA512.

m'	$\varphi(m')$	encrypt		decrypt		ciphertext (*)		addPublic		mulPublic	
		CT	RT	CT	RT	CT	RT	CT	RT	CT	RT
2,048	1,024	371	392	2.3	20.5	1.4	2.9	1.3	10.1	1.4	3.1
14,400	3,840	1,395	1,454	12.8	81.6	13.8	18.1	6.5	35.0	4.6	7.0

Table 4.7: Runtimes (in milliseconds) for SHE noise and ciphertext management operations like `rescaleCT` and `keySwitch` (relinearization) from a quadratic ciphertext, with a circular hint. The `rescaleCT` benchmark scales from (the product of) two moduli to one. The `keySwitch` benchmark uses a single ciphertext modulus and a hint with two moduli, and a two-element gadget for decomposition (subsection 3.2.4).

m'	$\varphi(m')$	`rescaleCT`		`keySwitch`	
		CT	RT	CT	RT
2,048	1,024	2.3	17.9	7.4	53.4
14,400	3,840	15.2	65.2	37.0	308

Ring Tunneling

In the ring-tunneling algorithm (subsection 4.3.6), we convert a ciphertext in a cyclotomic ring R' to one in a different cyclotomic ring S' which has the side effect of evaluating a desired E-linear function, where $E = R \cap S$ is the intersection of the corresponding plaintext rings. The performance of this algorithm depends on the dimension $\dim(R'/E')$ because the procedure performs $\dim(R'/E')$ key switches. Since ring switching can only apply an E-linear function on the plaintexts, there is a tradeoff between performance and the class of functions that can be evaluated during ring switching. In particular, when $\dim(R'/E') = \dim(R/E)$ is small, ring switching is fast but the plaintext function is highly restricted because E is large. When $\dim(R'/E')$ is large, we can apply a wider class of functions to the plaintexts, at the cost of many more (expensive) key switches. Indeed, in many applications it is convenient to switch between rings with a small common subring, e.g. $E = \mathcal{O}_1$.

As shown in [AP13], we can get both performance and a wide class of linear functions by performing a sequence of switches through adjacent hybrid rings, where the intersection between adjacent hybrid rings is large. Figure 4.4 gives a sequence of hybrid rings from $R = H_0 = \mathcal{O}_{128}$ to $S = H_5 = \mathcal{O}_{4,095}$. It also gives the corresponding ciphertext superring,

which needs to be larger than small plaintext rings for security. Such a sequence of hybrid rings could be used for bootstrapping ([AP13]) or for the homomorphic evaluation of the PRF in [BP14].

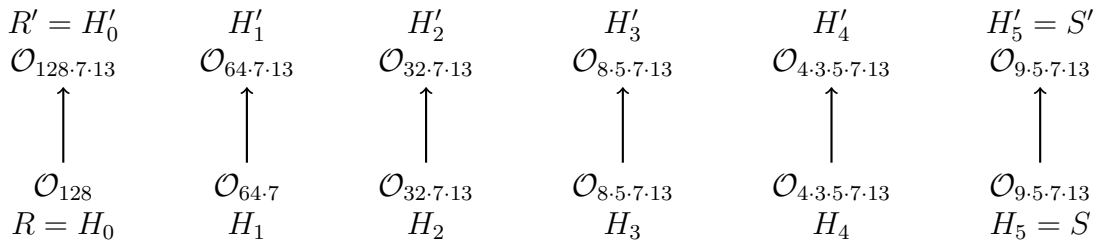

Figure 4.4: A real-world example of hybrid plaintext/ciphertext rings that could be used to efficiently tunnel from $R = \mathcal{O}_{128}$ to $S = \mathcal{O}_{4,095}$.

Table 4.8 includes timing data for each ring tunnel in Figure 4.4, using only good moduli as above. As with other operations, ring tunneling scales linearly in the number of moduli, so the numbers below are reported for a single modulus.

Table 4.8: Runtimes (in milliseconds) for ring tunneling, using one ciphertext modulus and **TrivGad** for constructing key-switch hints.

Tunnel	CT	RT
$H_0 \to H_1$	46.4	185
$H_1 \to H_2$	32.3	127
$H_2 \to H_3$	50.0	128
$H_3 \to H_4$	32.9	84.2
$H_4 \to H_5$	33.2	96.4

CHAPTER 5

ALCHEMY: A LANGUAGE AND COMPILER FOR HOMOMORPHIC ENCRYPTION MADE EASY

5.1 Introduction

The concept of homomorphic encryption was first envisioned almost 40 years ago as a powerful way to enable new privacy-aware applications. In the eight years since Gentry gave the first plausible construction [Gen09b; Gen09a], there have been a number of implementations targeting particular applications of interest (see, e.g., [GH11; NLV11; GHS12c; HS15; DM15; CLP17] and chapter 4). Unfortunately, the impact of this cryptographic "holy grail" has been tempered by the difficulty of *using* it. The primary usability challenge in all existing implementations is the level of expertise needed to satisfactorily implement a desired homomorphic computation:

1. First, one must express the "in the clear" computation (on plaintexts) in terms of the FHE scheme's native homomorphic operations, or "instruction set." This is non-trivial because the native instructions, which can vary based on the scheme, are typically algebraic operations like addition and multiplication on finite fields, and sometimes other functions like permutations on fixed-sized arrays of field elements. So one needs to "arithmetize" the desired computation in terms of these operations, as efficiently as possible for the instruction set at hand. (Moreover, the native instruction set can sometimes depend on the choice of plaintext and ciphertext rings, which also affects the third step below.)

2. Second, HE ciphertexts accumulate "errors" or "noise" under homomorphic operations, and too much noise causes the result to decrypt incorrectly—so proper noise management is essential. In addition, the ciphertext size (or degree) increases under

certain homomorphic operations, but can be brought back down via additional steps. So along with the homomorphic operations that perform "real work," one must also carefully schedule appropriate "maintenance" operations, such as *linearization* and other forms of *key switching*, and *rescaling* (also called *modulus switching*) to keep the noise as small as possible.

3. Third, one must choose appropriate parameters for the desired level of security, i.e., appropriate ciphertext dimensions and moduli relative to the noise rates at the various stages of the computation (and subject to any restrictions inherited from the first step). Importantly, the choice of parameters feeds back to affect the noise growth incurred by the homomorphic operations, so one may need many cycles of trial and error until the parameters stabilize.

4. Lastly, one also needs to generate all the needed keys and auxiliary key-switching "hints" that are needed for the maintenance operations, and to encrypt the input plaintexts under the appropriate keys.

In summary, the above process requires a great deal of expertise in both the theory of HE and the quirks of its particular implementation, in addition to a lot of manual programming and trial-and-error. Perhaps for this reason, most applications of HE to date have been ad-hoc, one-off implementations, with complex code that is hard to debug and which obscures the nature of the underlying computation.

A new approach. This work introduces ALCHEMY, a system that greatly accelerates and simplifies the implementation of homomorphic computations. [1] With ALCHEMY, one expresses a desired "in the clear" computation on plaintexts in a *domain-specific language* of simple operations, and higher-level functions built out of them. A key point is that this

[1] ALCHEMY is now publicly available under a free/open-source license at `https://github.com/cpeikert/lol/tree/alchemy-args-debruijn-monad`.

requires no detailed knowledge of the HE scheme; one simply writes (and runs, and debugs) a program that describes what should be computed on the plaintexts.

One then uses an ALCHEMY *compiler* to automatically transform the plaintext program into a corresponding homomorphic program on ciphertexts. The compiler deals with the cumbersome but rote tasks of tracking the noise and scheduling appropriate "maintenance" operations to control it, choosing (most of) the parameters, generating keys and hints, etc. In addition, compilers can be composed together to provide other useful functionality.

In short, ALCHEMY lets programmers write clear and concise code describing what they really care about—the plaintext computation—and get a corresponding homomorphic computation without needing any particular expertise in HE. ALCHEMY fits seamlessly into typical HE usage scenarios to reduce the overhead of producing homomorphic computations. In the rest of this introduction we describe the approach in more detail, and give some simple examples that show ALCHEMY's convenience and flexibility. (A richer example application and is detailed in chapter 6.)

5.1.1 Principles of ALCHEMY

ALCHEMY is a collection of *domain-specific languages* (DSLs) for expressing plaintext and (homomorphic) ciphertext computations, and *interpreters* that act on programs written in these languages. The word "interpreters" is meant broadly, and encompasses (among others) evaluators, optimizers, and, most significantly, "compilers" that transform programs from one language to another.

ALCHEMY is highly modular and extensible: the DSLs are made up of small *components* that each yield particular language features, are easy to define, and can be combined arbitrarily. Interpreters can be defined to support any subset of the language components, and can easily be extended to support new ones. In addition, ALCHEMY's interpreters can easily be composed with each other to perform a variety of different tasks on the same program, e.g., evaluating a plaintext computation "in the clear," compiling it to a

corresponding homomorphic computation, printing a representation thereof, and tracking the noise growth as it is evaluated.

Another primary goal of ALCHEMY is *safety*: only well-formed programs should be accepted, and the possibility of runtime errors or internal bugs should be minimized, or even eliminated. For these purposes, the ALCHEMY DSLs are *functional* (pure), *statically typed* with a rich type system, and have strong *type inference*.

- Purity means that a function always yields the same output when given the same inputs (no side-effects or global variables), which is a good match for the arithmetic functions and "circuits" that are common targets for homomorphic computation.

- Static typing means that every expression has a type that is known at *compile time*, and only well-formed expressions typecheck. This allows many common programming errors—in DSL code, and in ALCHEMY's own interpreters—to be caught early on. The type system is very rich, allowing many safety properties to be encoded into types and automatically verified by the type checker.

- Type inference ensures that the types of almost all DSL expressions are automatically determined by the type checker, and need not be specified by the programmer. This makes code more concise and readable, and easier to check for correctness.

We obtain all the above-described properties by defining the DSLs and interpreters in the metalanguage Haskell, from which ALCHEMY directly inherits its basic syntax, data types and structures, and safety features—with no special implementation effort or extra complexity. As its underlying HE implementation, ALCHEMY uses a BGV-style [BGV14] SHE cryptosystem as refined and implemented in $\Lambda \circ \lambda$ [CP16b]. We emphasize that much of the framework could be reused without modification for any target SHE or FHE scheme.

Languages. Domain-specific languages (DSLs) have long been appreciated as useful tools for working in a specific problem domain; e.g., LaTeX is a (Turing-complete) DSL for

typesetting documents, and MATLAB is a DSL targeted toward numerical computing and linear algebra.

ALCHEMY's first major component is a collection of modular and extensible DSLs for expressing both "in the clear" computations on plaintexts, and homomorphic computations on ciphertexts. Following the powerful "typed tagless final" approach to embedded language design [Kis10], each DSL is the union of several independent and composable *language components*. ALCHEMY provides language components that introduce the following features into the DSLs:

- data types for plaintexts rings and HE ciphertexts;
- simple data structures like tuples and lists;
- arithmetic operations like addition and multiplication, and also arbitrary linear functions between plaintext rings;
- ciphertext operations corresponding to the interface of the underlying SHE implementation;
- programmer-defined functions, including higher-order functions (i.e., those that take other functions as input);
- and even specifically limited side-effects, via monads.

It is easy to introduce additional data types and language features as needed, simply by defining more language components.

Both the plaintext and ciphertext DSLs include the generic language components for data structures, arithmetic operations, and functions. In addition, each one includes certain components that relate specifically to plaintext or ciphertext operations. Because the plaintext DSL involves relatively simple data types and operations, it is easy for the programmer to hand-write code to express a desired computation. By contrast, proper use of the ciphertext DSL is significantly more complicated—e.g., ciphertext types involve many more

parameters, and HE operations must be appropriately scheduled—so it is not intended for human use (though nothing prevents this). Instead, it is the target language for ALCHEMY's plaintext-to-ciphertext compiler. As we will see, having a dedicated ciphertext DSL allows for homomorphic computations to be operated upon in various useful ways beyond just executing them, e.g., tracking noise growth or optimizing away redundant operations.

In addition to the DSLs themselves, ALCHEMY provides a variety of useful higher-level functions and combinators that are written in the DSLs. These include "arithmetized" versions of functions that are not natively supported by HE schemes, but can be expressed relatively efficiently in terms of native operations. A particularly important example is the function that "rounds" from $\mathbb{Z}_{2^k}$, the ring of integers modulo 2^k, to $\mathbb{Z}_2$. This function is central to efficient "boostrapping" algorithms for FHE and the related Learning With Rounding problem [BPR12]. Efficient arithmetizations are given in [AP13] and a closely related algorithm from [GHS12a]). We give a different arithmetiziaton in section 6.3 (which is better for particular parameters), and provide an ALCHEMY implementation in subsection 6.6.1.

Interpreters and compilers. ALCHEMY's other main component is its collection of composable *interpreters* for programs written in the DSLs. Each interpreter defines how it acts on each relevant language component to perform a particular task. Some of the interpreters are actually *compilers* that translate programs from one collection of DSL components to another. Example interpreters in ALCHEMY include:

- a (metacircular) evaluator, which simply interprets the plaintext and ciphertext DSLs using the corresponding Haskell and $\Lambda \circ \lambda$ operations;
- various utility interprets that, e.g., "pretty print" DSL programs, or compute useful metrics like program size, multiplicative depth, etc.;
- a diagnostic compiler that modifies any ciphertext-DSL program to also log the noise rate of every ciphertext it produces;

- and most significantly, a compiler that transforms any program written in the plaintext DSL to a corresponding homomorphic computation in the ciphertext DSL.

The plaintext-to-ciphertext compiler is the most substantial and nontrivial of the interpreters, and is one of ALCHEMY's central contributions. This compiler *automatically* performs several important tasks that in all current HE systems must done manually by the programmer. In particular, it:

- generates all necessary *keys* and *auxiliary "hints"* for ciphertext operations like key-switching and ring-switching.
- properly schedules all necessary ciphertext maintenance operations like key-switching (e.g., for "linearization" after homomorphic multiplication) and modulus-switching (for noise management).
- *statically* infers, using compile-time type arithmetic, the approximate noise rates of every ciphertext, and chooses an appropriate ciphertext modulus from a provided pool. If any inferred noise rate is too small relative to the provided moduli, typechecking fails with an informative error.
- generates encrypted inputs for the resulting homomorphic computation, with appropriate noise rates to ensure correct decryption of the ultimate encrypted output.

5.1.2 Example Usage

Here we give a few concrete examples of programs in the ALCHEMY DSLs, and the various ways they can be interpreted and compiled. We start with the following very simple program:

```
ex1 = lam $ lam $ v0 *: (v0 +: v1)
```

The two calls to `lam` say that `ex1` is a function of two inputs, which respectively bind the De Bruijn-indexed variables `v1`, `v0`. (De Bruijn variables are numbered and bound from the

"inside out.") The function represents addition of `v0` and `v1` using the DSL operator `(+:)`, followed by multiplication of the result by `v0` using the DSL operator `(*:)`.

The Haskell typechecker automatically infers the full type of `ex1`, which is:

```
(Lambda expr, Mul expr a, Add expr (PreMul expr a)) =>
  expr e (PreMul expr a -> PreMul expr a -> a)
```

This type carries a great deal of important information; let us unpack its various components:

- First, the type is *polymorphic* in the type *variables* `expr`, `a`, and `e`. These type variables can represent arbitrary Haskell types...
- ...subject to the *constraints* `(Lambda expr, ...)`, which say that `expr` must be able to interpret the **`Lambda`**, **`Mul`**, and **`Add`** language components. The second arguments of **`Mul`** and **`Add`** say that multiplication can produce a value of type `a` as the product of two values of type **`PreMul`** `expr a`, and that we can add values of the latter type. (The purpose of **`PreMul`** will be explained below, when we describe the plaintext-to-ciphertext homomorphic compiler.)[2]
- Finally, `expr e (PreMul expr a -> PreMul expr a -> a)` means that `ex1` represents a *DSL function* that takes two inputs of type **`PreMul`** `expr a` and outputs a value of type `a`. The type argument `e` represents the expression's *environment*, which must hold the values of any unbound variables. Because `v0` and `v1` are bound by the two `lam`s, there are no unbound variables—the code is *closed*—so `e` is completely unconstrained.

Because `ex1` is polymorphic in `expr`, after writing it once we can interpret it in several different ways by specializing `expr` to various concrete types. One simple interpreter is the "pretty printer" **`P`**, which has easy implementations for all the requisite language components. Its public interface

[2] As the reader may have guessed, **`Lambda`** introduces programmer-defined functions and variables via `lam`, `v0`, `v1`, etc., whereas **`Mul`** and **`Add`** respectively introduce the multiplication and addition operators `*:` and `+:`.

```
pprint :: P () a -> String
```

converts any closed **P**-expression to a string representing it. Calling `pprint ex1` implicitly specializes `expr` to **P** and the environment `e` to `()`, resulting in the following:[3]

```
pprint ex1
-- "(\v0 -> (\v1 -> ((mul v1) ((add v1) v0))))"
```

Another very simple interpreter is the metacircular evaluator **E**, which interprets all of the language components using corresponding Haskell (or $\Lambda\circ\lambda$) operations. Its public interface

```
eval :: E () a -> a
```

converts any closed *DSL expression* representing a value of type `a` into a *Haskell value* of type `a`, as follows:

```
eval ex1
-- (Ring a) => a -> a -> a


eval ex1 7 11
-- 198
```

Because `eval` implicitly specializes `expr` to **E**, which itself defines `PreMul E a = a`, the call to `eval ex1` produces a polymorphic Haskell function of type `a -> a -> a`, for an arbitrary **Ring** type `a`. The **Ring** constraint arises because the **E** interpreter uses Haskell's operators `(+)` and `(*)` to interpret **Add** and **Mul**. The `eval ex1 7 11` call actually evaluates the Haskell function, producing $11 \cdot (11 + 7) = 198$.

We stress that `eval ex1 :: (Ring a) => a -> a -> a` is *polymorphic* in `a`, so it can be applied to elements of any plaintext ring, or even to ciphertexts from $\Lambda\circ\lambda$'s SHE scheme (both of which are **Ring** types). However, in the latter case `ex1` lacks the extra ciphertext

[3]For convenience of implementation, the pretty printer indexes variables from the "outside in," which is the reverse of De Bruijn indexing; this accounts for the swapping of `v0` and `v1` in the pretty-printed code, relative to the definition of `ex1`. Note that the two representations are equivalent.

"maintenace" operations, like relinearization and modulus-switching, that are needed in more complex homomorphic computations. For these we use ALCHEMY's homomorphic compiler, described shortly.

Ring switching. Here we exhibit a small program that illustrates another important language component, for "switching" from one cyclotomic ring to another. Ring-switching in homomorphic encryption was developed and refined in a series of works [BGV14; Gen+13; AP13; CP16b], which showed its utility for tasks like "bootstrapping" and efficiently computing a wide class of linear functions.

```
ex2 = linearCyc_ (decToCRT @F28 @F182) .:
      linearCyc_ (decToCRT @F8  @F28)
```

Here `decToCRT @F8 @F28` is a Haskell expression representing a certain linear function from the 8th to the 28th cyclotomic ring, and similarly for `decToCRT @F28 @F182`. (The exact linear functions do not matter here, and could be arbitrary.) The operator `(.:)` denotes composition of DSL functions; the type checker enforces that the output type of the first function must equal the input type of the second.

The Haskell compiler automatically infers that `ex2` has the following type (several component types and constraints have been simplified or suppressed for readability):

```
ex2 :: (LinearCyc expr cycrep, ...) =>
       expr env (cycrep F8 zp -> cycrep F182 zp)
```

This says that `ex2` is a (closed) DSL function that is well-defined for any interpreter `expr` of the `LinearCyc` language component (which introduces `linearCyc_`). Essentially, the function maps from `cycrep F8 zp`, which should represent the 8th cyclotomic ring modulo some integer p, to `cycrep F28 zp`, which should represent the 28th cyclotomic ring modulo p. The type `cycrep` is specified in the `LinearCyc expr cycrep` constraint, and could be, e.g., the concrete `Cyc` type from $\Lambda \circ \lambda$.

As with the previous example, we can pretty-print and evaluate `ex2` "in the clear." More interesting is to *homomorphically* evaluate it on HE ciphertexts, using a special form of key-switching as shown in [Gen+13] and subsection 4.3.6. For this we use ALCHEMY's homomorphic compiler.

Compiling to the ciphertext DSL. We now show how the above example programs, which should now be thought of as computations on plaintexts, can be compiled into programs that operate on HE ciphertexts to homomorphically compute the original programs on the underlying plaintexts. The compiler is given by a data type `PT2CT`, whose public interface is the Haskell function `pt2ct` (the signature is in subsection 5.5.1). In order to do its job, the compiler needs to be given *types* that specify what ciphertext rings and moduli to use. We define such types here:

```
type CTRingMap = [ (F8,F16), ... ]

type Zq1 = Zq $(mkModulus 34594561)
type Zq2 = Zq $(mkModulus 35642881)
...
type CTModuli  = [ Zq1, Zq2, ... ]
```

The type `CTRingMap` says that when the plaintext ring is the 8th cyclotomic, the ciphertext ring should be the 16th cyclotomic, etc. The type `Zq1` represents $\mathbb{Z}_{q_1}$, the ring of integers modulo $q_1 = 34594561$, and similarly for `Zq2` etc. (The macro `mkModulus` represents its argument as a *type*, which is also augmented with the number of "units of noise" the modulus can hold; see below.)

Having defined the needed types, we can now compile our plaintext-DSL programs to get new ciphertext-DSL programs, which can in turn be handled by any suitable ciphertext-DSL interpreter. One small subtlety is that because `pt2ct` automatically generates the needed *random* keys and key-switch hints, it is necessarily *monadic* (i.e., it has side effects).

We therefore use Haskell's "do notation" to invoke it with the above-defined types on plaintext-DSL program `ex1`, and then pretty-print the result:

```
do  ct1 <- pt2ct @CTRingMap @CTModuli @TrivGad ex1
    pprint ct1


-- "(\v0 -> (\v1 ->
--    (((\v2 -> (\v3 ->
--        (modSwitch
--          (keySwitchQuad <HINT>
--            (modSwitch ((mul v3) v2))))))
--      ((add v1) v0)) v1)))"
```

Despite the abundant parentheses, the structure of the program is not too hard to discern. First, because this is a program in the ciphertext DSL, we should think of all the variables as ciphertexts. In the "inner" layer, the variable `v2` is bound to `((add v1) v0)`, and `v3` is bound to `v1`.[4] These two ciphertexts are multiplied, resulting in a "quadratic" ciphertext. This is modulus-switched to match the key-switching hint, then key-switched to a "linear" ciphertext (using a "circularly" encrypted hint), then finally switched back to a modulus corresponding to its inherent noise rate.

As another example, we can compile the ring-switching program `ex2 = linearCyc_ (...) :: linearCyc_ (...)`, and print the resulting program:

```
do  ct2 <- pt2ct @CTRingMap @CTModuli @TrivGad ex2
    pprint ct2


-- "(\v0 ->
```

[4]Note that both expressions are eligible for "inlining" using β-reduction; while our compiler does not perform such optimizations at the DSL level, the Haskell compiler may do so. In any case, the performance cost of not inlining is negligible when compared with homomorphic operations. One could add this optimization by defining a new interpreter which outputs a beta-reduced expression.

```
--    ((\v0 ->
--       (modSwitch (tunnel <HINT> (modSwitch v0))))
--     ((\v0 -> (modSwitch
--                 (tunnel <HINT> (modSwitch v0))))
--      v0)))"
```

Again the structure is reasonably clear: the program takes a ciphertext as input, switches it to the modulus of the "tunneling hint" that encodes the desired linear function, then switches rings by "tunneling" with the hint (thereby homomorphically evaluating the linear function), then switches back to an appropriate ciphertext modulus. The same cycle is repeated for the next linear function. (As we shall see, some of the modulus-switches may turn out to be null operations.)

Evaluating and logging. While it is nice to be able to see a representation of ciphertext-DSL programs, we are more interested in the useful task of *evaluating* them to perform a homomorphic computation on ciphertexts. Fortunately, this is extremely simple: just replace `pprint` with `eval` in the above code! This specializes the (polymorphic) interpreter of the ciphertext-DSL program to the evaluator `E` rather than the pretty-printer `P`.

In addition, for diagnostic purposes we may wish to log the "error rates" of the ciphertexts as the homomorphic evaluation proceeds. (Recall that error rates must be kept small enough so that in the end, decryption gives the correct plaintext output.) In ALCHEMY such logging is very easy using the `ErrorRateWriter` interpreter, which transforms any ciphertext-DSL program into an equivalent one that additionally logs the error rates of all intermediate ciphertexts. The output program can then be evaluated (or printed, or sized, etc.) as usual. For example:

```
do  logct2  <- writeErrorRates ct2
    inputCT <- encrypt inputPT
    (result, log) = runWriter $ eval logct2 inputCT
```

```
    print log


-- "Error rates:
-- ("modSwitch_Q539360641*Q537264001",6.8495e-7),
-- ("tunnel_Q539360641*Q537264001",3.3651e-6),
-- ("modSwitch_Q537264001",7.3408e-6),
-- ("modSwitch_Q537264001",7.3408e-6),
-- ("tunnel_Q537264001",1.8010e-4),
-- ("modSwitch_Q537264001",1.8010e-4)"
```

The log shows the error rates of the ciphertexts produced by each ciphertext-DSL operation (which is also conveniently augmented by the ciphertext modulus of the result). We can see that the first `tunnel` operation increases the error rate by roughly 5x; the switch to the smaller modulus increases the rate by roughly 2x; and the second `tunnel` operation increases the rate by roughly 25x. (The other `modSwitch` operations do not actually change the modulus, and are therefore null.)

5.1.3 ALCHEMY In The Real World

In this section we explain how ALCHEMY can be used in the context of an actual cryptographic application. In a typical HE scenario, Alice would like Bob to perform a computation for her, but does not want Bob to learn anything about the input or output of the computation. Alice first generates appropriate symmetric keys and encrypts her data with an HE scheme, then sends the ciphertext to Bob. In order to compute the function on encrypted data, Alice must write a homomorphic version of the function that she wants Bob to compute, and send it to Bob.[5] Then Bob runs the computation on Alice's input and sends the (encrypted) result back to her.

[5] It is also possible for Alice to send the plaintext computation and let Bob turn it into computation on encrypted inputs. This requires Alice to send some additional information to Bob, but otherwise the procedure is identical.

Using ALCHEMY, many of Alice's steps are automated. Alice first writes an ALCHEMY DSL expression for the plaintext computation, and then compiles it locally to obtain a homomorphic computation. The compilation process automatically creates all necessary cryptographic keys and provides them to Alice for future use. Alice can then serialize this expression (using a serialization interpreter) and send it to Bob. Bob then uses ALCHEMY to deserialize the computation, executes it on Alice's data, and sends back the encrypted result. Alice uses they secret keys provided by the compilation process to decrypt the result.

Thus ALCHEMY simplifies the process of turning Alice's plaintext computation into a computation that Bob can actually use. All keys remain under Alice's (and only Alice's) control, and ALCHEMY is used by both Alice and Bob to respectively create and evaluate a desired homomorphic computation.

5.1.4 Related Work

As far as we are aware, there is no prior compiler for HE; all existing HE implementations require the programmer to manually call all the needed plaintext and ciphertext-maintenance operations, generate parameters and keys, etc.

For example, ALCHEMY currently targets the BGV-style SHE scheme as implemented with the $\Lambda\circ\lambda$ Haskell framework for lattice-based cryptography introduced in chapter 3.[6] This provides ALCHEMY with its underlying HE implementation, which supports advanced features like ring-switching. However, up until now those who wished to use $\Lambda\circ\lambda$ for HE still had to write code directly to its interface, which is somewhat low-level.

Probably the most well-known HE implementation is HElib [HS], an "assembly language" for homomorphic encryption, which is implemented in C++ on top of NTL [Sho06]. HElib has been used for many homomorphic computations of interest [GHS12c; HS14; HS15], but it requires quite a lot of expertise to use, because computations must be written directly in the "assembly language" itself.

[6] We emphasize that ALCHEMY contains many generic components which can be reused outside the context of $\Lambda\circ\lambda$ or BGV-style homomorphic encryption.

FHEW [DM15] is an implementation of a very fast bootstrapping algorithm for "third-generation" FHE schemes [GSW13; AP14]. However, it is not intended for general-purpose homomorphic computations, since the scheme encrypts only one bit per ciphertext.

The SEAL library [CLP17] goes one step farther than these three implementations by introducing heuristic parameter selection, an important step towards practical HE. However, users must still write homomorphic computations manually, including the ciphertext maintenance operations like (re)linearlization, and the management of cryptographic keys. These operations obscure the underlying plaintext computation and require knowledge of HE to use correctly. By contrast, the ALCHEMY compiler handles both of these components, and more, automatically.

Systems such as [JZ12] and TASTY [Hen+10] provide tools for describing interactive cryptogrphic protocols. They therefore solve a fundamentally different problem than ALCHEMY. In particular, they do not address the complexity of writing homomorphic computations. Instead, they provide tools for writing secure interactive protocols which can be instantianted with a number of different concrete cryptographic primitives like HE or MPC.

5.1.5 Chapter Organization

The rest of the chapter is organized as follows.

Section 5.2 gives the relevant background on the *(typed) tagless final* style of DSL design and implementation, and describes ALCHEMY's plaintext DSL.

Section 5.3 describes ALCHEMY's ciphertext DSL, which is the target language of the compiler.

Section 5.4 describes several simple DSL interpreters, including a transformation which logs the error rates of homomorphic computations at runtime.

Section 5.5 describes the central piece of ALCHEMY, the plaintext-to-ciphertext compiler.

We defer the evaluation of ALCHEMY to chapter 6, where we provide full-scale homomorphic application written in ALCHEMY.

5.2 ALCHEMY Domain-Specific Languages

In this section we provide the requisite background on the "typed tagless final" style of embedded DSL design and implementation, provide some simple example interpreters, and describe our plaintext DSL.

5.2.1 Typed Tagless Final Style

The elegant and powerful "typed tagless final" approach to DSLs, also called *object languages*, was introduced by Carette *et al.* [CKS09], and further explicated in the lecture notes of Kiselyov [Kis10]. The more widely known "initial" approach represents object-language terms as values of a special data type in a metalanguage, e.g., an abstract syntax tree. By contrast, the "tagless final" approach represents object-language terms as combinations of ordinary *polymorphic* terms in the metalanguage. The polymorphism allows an object-language term to be written once and interpreted in many ways, by monomorphizing it in different ways.

The tagless-final approach makes language design and interpretation highly modular, extensible, and safe: different object-language features can be defined independently and combined together arbitrarily, and interpreters can be defined to handle any subset of the available components. Interpreters can be extended to support new object-language features without changes to existing code. An interpreter is able to interpret an object-language term exactly when it is able to interpret all the language components used by the term; otherwise, type checking fails at compile time. More generally, the full strength of the metalanguage's type system, including type inference, can directly be inherited by the object language.

Here we give an introduction to the approach by providing a running example of several general-purpose language components and interpreters from [CKS09; Kis10], which are also

part ALCHEMY. In later subsections we describe more specialized language components for the plaintext and ciphertext languages.

Language components. In the tagless-final approach as realized in the metalanguage Haskell, an object-language component is defined by a *class*. A Haskell class introduces one or more polymorphic *methods*, which may be functions or just values. For example, to introduce pairs as an object-language feature, we define

```
class Pair expr where
  pair_ :: expr e (a -> b -> (a,b))
  fst_  :: expr e ((a,b) -> a)
  snd_  :: expr e ((a,b) -> b)
```

Here `pair_` is a metalanguage *value* which represents an *object-language function*. The function takes (object-language) values of type `a` and `b`, and returns an (object-language) value of the pair type `(a,b)`. Similarly, `fst_` and `snd_` represent object-language functions that respectively extract the first and second components of a pair. (By convention, names of object-language terms always end in underscore, to distinguish them from metalanguage terms.)

Notice the common form `expr e x` of the method types. Here the type `expr` is the *instance* of the `Pair` class; it serves as the *interpreter* of object-language terms involving pairs. In turn, `expr` is parameterized by an *environment* type `e` (discussed below) and a metalanguage type `x`, which serves as the type of the object-language term.

Interpreters. An interpreter of a language component is just a data type that is defined to be an instance the component's class. As running examples, we describe two simple interpreters: the metacircular evaluator `E` is defined as

```
newtype E e a = E (e -> a)
```

which says that value of type `E e a` is equivalent (isomorphic) to a function that maps e-values to a-values. The particular function will (usually) be the one that maps any e-value to the (metalanguage) value of type a represented by the object-language term. The pretty-printer `P` is defined as

```
newtype P e a = P String
```

which says that a value of type `P e a` is equivalent to a `String`. The particular string will be the printed representation of the object-language expression of type a that `P` interprets.

We make `E` and `P` interpreters of the `Pair` language component by making them instances of the `Pair` class. Observe that when `expr` is specialized to `E`, the type of `pair_` is equivalent to `e -> a -> b -> (a,b)`. We give a partial instance definition below;[7] the definition of `fst_` and `snd_` are similarly trivial:

```
instance Pair E where
  pair_ = E $ \e -> \a -> \b -> (a,b)
```

When `expr` is specialized to `P`, the type of `pair_` is equivalent to just `String`, which leads us to the easy instance definition

```
instance Pair P where
  pair_ = P $ "pair_"
```

Extending the language and interpreters. We can introduce more language features simply by defining more classes, e.g., for addition:

```
class Add expr a where
  add_ :: expr e (a -> a -> a)
  neg_ :: expr e (a -> a)
```

[7] Since `E` is an applicative functor, a shorter definition is `pair_ = E $ pure (,)`.

This says that `add_` is an object-language function that takes two values of type `a` and returns a value of type `a`. Similarly, `neg_` is an object-language function of one argument. Notice, however, that the type `a` here is *specific*, not arbitrary: it is an argument to the **`Add`** class. This means that an interpreter may support `add_` and `neg_` for certain types `a`, but not others. The (partial) instances of **`Add`** for **`E`** and **`P`** are straightforward:

```
instance Additive a => Add E a where
  add_ = E $ \e -> \x -> \y -> x+y


instance Add P a where
  add_ = P $ "add_"
```

Notice that **`E`** is an instance of **`Add`** only for types `a` that are themselves instances of the **`Additive`** class. This class defines the polymorphic addition function `(+) :: Additive a => a -> a -> a` used in the definition of `add_` for **`E`**.

Functions and environments. The above classes define object-language functions, but so far we have no way of actually applying them to arguments! Nor do we have a way to create new functions of our own in the object language. Both of these features are introduced by the **`Lambda`** class:

```
class Lambda expr where
  ($:) :: expr e (a -> b) -> expr e a -> expr e b
  lam  :: expr (e,a) b -> expr e (a -> b)
  v0   :: expr (e,a) a
  s    :: expr e a -> expr (e,x) a
```

The `($:)` operator applies an object-language function of type `a -> b` to a object-language value of type `a` to yield an object-language value of type `b`. Before describing the remaining methods, we show the easy definitions of `($:)` for the **`E`** and **`P`** interpreters:

```
(E f) $: (E a) = E $ \e -> (f e) (a e)
(P f) $: (P a) = P $ "(" ++ f ++ " " ++ a ++ ")"
```

For the first line, recall that `f` is a function of type `e -> a -> b` and `a` is a function of type `e -> a`, and we need to produce a function of type `e -> b`; the right-hand side of the definition does exactly what it should. For the second line, recall that `f` and `a` are just **`String`**s representing their respective object-language terms, so to pretty-print the function application we just separate them with a space and wrap in parentheses to avoid ambiguity.

The function `lam` denotes lambda-abstraction. Notice its use of the environment: it converts any object-language term that has type `b`, *in any environment whose "topmost" entry has type* `a`, into an object-language function of type `a -> b`. Similarly, `v0` is an object-language value of type `a` *in any environment whose topmost entry has type* `a`. Essentially, the environment can be thought of as a stack of values, and `v0` represents the value at the top. Finally, `s` "shifts" an object-language expression by pushing a value (of arbitrary type) onto the environment, so `v1 = s v0` represents the next value on the stack, `v2 = s v1` represents the next value, etc. Putting these piece together, for example, `lam v0` has type `expr e (a -> a)` and represents the identity function.

The definitions of `lam`, `v0`, and `s` are trivial for the evaluator **`E`**, and are *almost* as trivial for the pretty-printer **`P`**; however, the type just needs to be redefined to be a *function* from the "`lam` depth" to **`String`**, so that the proper variable indices can be pretty-printed. See subsection 5.4.1 for the actual ALCHEMY definition of this interpreter.

5.2.2 Generic Language Components

ALCHEMY includes two loosely defined and overlapping languages: a plaintext language for expressing "in-the-clear" computations, and a ciphertext language for computations on encrypted inputs. Each of these languages is divided into many language components, some of which are shared between the languages. In addition to the **`Pair`**, **`Add`**, and **`Lambda`**

components that we have seen in the previous section, we briefly describe the remaining language components which are common to both DSLs.

Multiplication. The `Mul` class adds multiplication to the object language:

```
class Mul expr a where
  type PreMul expr a
  mul_ :: expr e (PreMul expr a -> PreMul expr a -> a)
```

The type of `mul_` is similar to `add_`, except that the two (object-language) inputs have type `PreMul expr a` instaead of just `a`. The `PreMul` *type family* generalizes the input types to `mul_` so that they are a *function* of the output type and the interpreter. This is necessary in the compilation step (see section 5.5). In practice, `PreMul expr a` is always isomorphic to `a`, but some interpreters need to augment the input with additional information at the type level. We give the `Mul` instance for `E` below:

```
instance (Ring a) => Mul E a where
  type PreMul E a = a
  mul_ = E $ \e -> \x -> \y -> x*y
```

Similar to the `Add` instance for `E`, the `Mul` instance works for any object language type which is a `Ring`. The definition of `mul_` is defined directly in terms of Haskell's multiplcation operator `(*) :: Ring a => a -> a -> a`. This implementation in fact *determines* the definition of `PreMul` for `E`: the type of the object-language function represented by `mul_` must have the same type as `(*)`.

Since the `P` interpreter represents any object language type as a string, we are free to defined `PreMul P a` as we like. However, it turns out to be convenient to also define it simply as `a`.

Lists. Just as the `Pair` class introduces pairs into the object language, the `List` class adds support for object language lists. We only give the language component definition here as the `E` and `P` instances are trivial:

```
class List expr where
  nil_  :: expr e [a]
  cons_ :: expr e (a -> [a] -> [a])
```

The `nil_` value represents an empty object language list (of an arbitrary type), while `cons_` appends an object language value of type `a` to an object language list of the same type.

Strings. The `String` language component adds literal strings to the object language. `string_` embeds any Haskell string as a DSL expression:

```
class String expr where
  string_ :: Prelude.String -> expr e Prelude.String
```

Category Theoretical Abstractions. Haskell provides several abstractions from category theory, including *functors*, *applicatives*, and *monads*. These features are important for advanced interpreters which require effects at the object level. We emphasize that these features are used in interpreters which produce effects at the object-language *runtime* (like the error logger in subsection 5.4.4), rather than "monadic interpreters" which use effects at the object-language *compile* time (like the HE compiler in section 5.5, which uses randomness to create keys and hints).

We give the class definitions of these language features here and defer usage details to subsection 5.4.4. The following classes introduce functionality identical to the corresponding Haskell classes (without the trailing underscore).

```
class Lambda expr => Functor_ expr where
  fmap_ :: (Functor f) => expr e ((a -> b) -> f a -> f b)
```

```
class (Functor_ expr) => Applicative_ expr where
  pure_ :: (Applicative f) => expr e (a -> f a)
  ap_   :: (Applicative f) => expr e (f (a -> b) -> f a -> f b)

class (Applicative_ expr) => Monad_ expr where
  bind_ :: (Monad m) => expr e (m a -> (a -> m b) -> m b)

class (Monad_ expr) => MonadWriter_ expr where
  tell_   :: (MonadWriter w m) => expr e (w -> m ())
  listen_ :: (MonadWriter w m) => expr e (m a -> m (a,w))
```

5.2.3 Plaintext DSL

In addition to these generic language components, there are several language components which are unique to the plaintext DSL for BGV-style HE ([BGV14] and section 4.3). It is trivial to extend the plaintext language with these new features: we simply define a corresponding class and give instances for the appropriate interpreters. Note that while the generic language components can by definition be supported by any interpreter, the components which are part of the plaintext DSL but *not* the ciphertext DSL may only be supported by a subset of the interpreters. This is simple to encode: we simply omit instances of language components for interpreters that do not make sense. See section 5.5 for more details.

In this section, we give only the language definitions; their implementations for `P` and `E` are very simple.

Arithmetic with Public Values. The `AddLit` language component provides the `addLit_` operation to add a meta-language literal of type `a` to an object-language expression of type `a`:

```
class AddLit expr a where
  addLit_ :: a -> expr e (a -> a)
```

Similarly, the `MulLit` language component introduces `mulLit_`, which multiplies a public meta-language value with an object-language value. These language features are useful for performing arithmetic with known constants.

Division by two. The `Div2` language gives the `div2_` operation to divide a value that is known to be even by two, simultaneously reducing its modulus by a factor of two. Like the `Mul` class, `Div2` has an associated type family `PreDiv2` which allows the interpreter to specify how the input to the operation depends on the output. Concretely, our interpreters all require that the input have a modulus that is twice that of the output.

```
class Div2 expr a where
  type PreDiv2 expr a
  div2_ :: expr e (PreDiv2 expr a -> a)
```

Applying linear functions. The `linearCyc_` operation evaluates the given `Linear` function from a cyclotomic ring R to a cyclotomic ring S. Like `PreDiv2`, the `PreLinearCyc` type family determines the interpreter-specific input type cyclotomic representation from the output's representation. This is useful, e.g., in the `PT2CT` compiler, subsection 5.5.2. In addition, the `LinearCyc` class has an associated constraint which permits interpreters to require the types to satisfy certain relationships. This power is also available to other lanuage components like `Add` because the object language type `a` appears as a parameter to the class. Since `LinearCyc` has so many parameters, it is simpler to use a constraint synonym to achieve the same effect.

```
class LinearCyc expr rep where
  type LinearCycCtx expr rep e r s zp :: Constraint
  type PreLinearCyc expr rep :: * -> *

  linearCyc_ :: (LinearCycCtx expr rep e r s zp)
    => Linear zp e r s
    -> expr env ((PreLinearCyc expr rep) (Cyc r zp)
    -> rep (Cyc s zp))
```

Higher-level operations. Before compiling a computation into one which operates on encrypted inputs, we must first express the computation in terms of "native" HE operations. These native operations have a straightforward translation into homomrphic operations; non-native operations can be expressed in the HE scheme in many different ways, some of which are more efficient than others.

However, the set of native HE operations can be rather restrictive, and the user may want to perform a computation which includes more advanced operations. We would like to provide a way for the user to write computation using these high-level (i.e. non-native) operations.

This is easy to do using the existing design of the plaintext DSL. The basic idea is to extend the plaintext DSL with an expression which "arithmetizes" a non-native operation in terms of basic arithmetic operations which are native to the HE scheme. The expression is written entirely in terms of other plaintext DSL expressions, so the expression itself can be considered as a native extension to the plaintext DSL. The type of this expression is similar to the type of basic plaintext operations like `add_` or `div2_`, so it can be used in the same way to express computation which involve non-native HE operations. See subsection 6.6.1 for one such higher-level operation and its implementation in ALCHEMY.

5.3 Ciphertext DSL

The ciphertext DSL is composed of all of the generic language components, plus a few features which make sense for ciphertexts, but not plaintexts. For example, we can change the encryption key of a ciphertext, but this operation makes no sense on plaintexts, so this operation is not part of the plaintext DSL. In general, the ciphertext DSL for any HE scheme is closely coupled with the (implementation of the) HE scheme itself. Since ALCHEMY targets BGV-style SHE as implemented in section 4.3, the ciphertext DSL operations use types (for ciphertexts, keys, etc) from $\Lambda \circ \lambda$.

5.3.1 BGV-style SHE in $\Lambda \circ \lambda$

We give a brief overview of the relevant SHE types in $\Lambda \circ \lambda$. Some unnecessary details have been suppressed for clarity.

In this cryptosystem, a plaintext is an element of the mth *cyclotomic ring* mod p, i.e. $R_p = \mathbb{Z}_p[X]/(\Phi_m(X))$, where $\Phi_m(X)$ is the mth cyclotomic polynomial. In $\Lambda \circ \lambda$, this ring is represented with the data type `Cyc m zp`, where `m` is a (type-level) natural number representing the parameter m (known as the *cyclotomic index*) and `zp` is a type for integer arithmetic mod p (i.e., $\mathbb{Z}_p$ arithmetic).

A ciphertext is a polynomial over the m'th cyclotomic ring mod q, where the plaintext index m divides the ciphertext index m'. We denote this ring by R'_q. $\Lambda \circ \lambda$ represents ciphertexts with the type `CT m zp (Cyc m' q)`, where `m` and `zp` are the plaintext parameters, and `m'`, `q` are the ciphertext parameters.

The secret key for a ciphertext with type `CT m zp (Cyc m' zq)` is encoded as `SK (Cyc m' z)`, where `z` represents the ring of integers (not mod anything).

Arithmetic Operations. $\Lambda \circ \lambda$ defines the native SHE operations on these types. Concretely, the `CT` data type is an instance of Haskell's `Additive` and `Ring` classes, so we can use the `(+)` and `(*)` operators on ciphertexts. Note that the `Add` and `Ring` instances for `E` given

in subsection 5.2.1 and subsection 5.2.3 suffice to obtain addition and multiplication for ciphertexts, with no extra work.

In addition to adding and multiplying ciphertexts, we can also add and multiply a public plaintext value with a ciphertext. These operations are captured with the DSL expressions `addPublic_` and `mulPublic_`, which are part of the `SHE` ciphertext language component:

```
addPublic_ :: (SHE expr, ct ~ CT m zp (Cyc m' zq), ...)
  => Cyc m zp -> expr env (ct -> ct)

mulPublic_ :: (SHE expr, ct ~ CT m zp (Cyc m' zq), ...)
  => Cyc m zp -> expr env (ct -> ct)
```

Here, `Cyc m zp` represents a plaintext value in R_p. The functions take a public plaintext value and output an expression from a ciphertext encrypting an R_p value to a new ciphertext.

Rescaling Plaintexts. The SHE scheme also allows us to *rescale* an encrypted plaintext while simultaneously changing the modulus of the plaintext. The DSL operation for this is:

```
modSwitchPT_ :: (SHE expr, ...)
  => expr env (CT m zp (Cyc m' zq) -> CT m zp' (Cyc m' zq))
```

Ring switching. We can apply a linear function to an encrypted plaintext (which possibly moves the plaintext to a new ring) using the `tunnel_` operation:

```
tunnel_ :: (SHE expr, ...) => TunnelHint gad e r s e' r' s' zp zq
  -> expr env (CT r zp (Cyc r' zq) -> CT s zp (Cyc s' zq))
```

`tunnel_` corresponds to $\Lambda \circ \lambda$'s implementation of ring switching called *ring tunneling*, which moves an (encrypted) plaintext in the rth cyclotomic ring R to an encrypted plaintext in the sth cyclotomic ring S. In the process, tunneling applies any function $f : R \to S$ to the plaintext, as long as f is linear over the eth cyclotomic ring (for some e dividing both

r and s). Each of the plaintext indices e,r,s has a corresponding ciphertext index e',r',s', subject to divisibility constraints. The public `TunnelHint` input encodes the linear function f that will be applied to the plaintext. The hint data type is parameterized by the *gadget* used to make the hint, all relevant cyclotomic indices, and the plaintext and ciphertext moduli.

Ciphertext management. Many SHE operations are most convenient to perform when the ciphertext is a linear polynomial (in the secret key) over R'_q. Ciphertext multiplication produces a *quadratic* polynomial. Thus to perform more operations, we have to "linearize" the ciphertext using a (circular) key switch with `keySwitchQuadCirc`:

```
keySwitchQuad_ :: (ct ~ CT m zp (Cyc m' zq), ...)
  => KSQuadCircHint gad (Cyc m' zq) -> expr env (ct -> ct)
```

The DSL operation requires additional (public) information in the form of a *hint*. The hint data type, `KSQuadCircHint`, is parameterized by a *gadget* used to perform the key switch, as well as the type for the ciphertext ring.

Measuring and Managing Ciphertext Noise. Ciphertexts have an implicit error term which grows as homomorphic operations are performed. If this noise becomes too large, the ciphertext cannot be decrypted, and the plaintext is lost. Thus an important part of the cryptosystem is controlling the noise growth by augmenting a computation with noise management operations.

We first consider an operation which helps to control the size of the *absolute error* in a ciphertext by rescaling the *ciphertext* modulus:

```
modSwitch_ :: (SHE expr, ...)
  => expr env (CT m zp (Cyc m' zq) -> CT m zp (Cyc m' zq'))
```

See section 4.3 for details.

It is difficult to predict how the (relative) error rate changes throughout a computation. In practice, it is simplest to just decrypt the ciphertext, observing the error rate in the process. This is captured with a new ciphertext language component:

```
class ErrorRate expr where
  errorRate_ :: (...)
    => SK (Cyc m' z) -> expr e (CT m zp (Cyc m' zq) -> Double)
```

Since extracting and measuring the error term requires the decrypting the ciphertext, `errorRate_` takes the secret key under which the input (object-language) ciphertext is encrypted.

Connection to SHE implementation. Recall that the SHE language component is tightly coupled with the underlying implementation in $\Lambda \circ \lambda$. Each of the functions from the `SHE` language component correspond directly with a similarly named function from the $\Lambda \circ \lambda$ SHE interface. For example, the DSL operation `addPublic_` corresponds to `addPublic` from the SHE implementation, which has the following signature:

```
addPublic :: (AddPublicCtx m m' zp zq)
  => Cyc m zp -> CT m zp (Cyc m' zq) -> CT m zp (Cyc m' zq)
```

The `addPublic_` operation is a Haskell function which takes a value in the plaintext ring and produces an *expression* for an object language function which takes a ciphertext and produces a ciphertext. Similarly, `addPublic` takes a plaintext element and a ciphertext and produces a new ciphertext.

The coupling between the interfaces becomes even more apparent with the evaluation interpreter. To evaluate any `SHE` DSL operation, we simply call the corresponding function from $\Lambda \circ \lambda$'s SHE interface, as in this partial instance:

```
instance SHE E (CT m zp (Cyc m' zq)) =>
  addPublic_ a = E $ \e -> \ct -> addPublic a ct
```

With the exception of `errorRate_`, the other ciphertext language features described above similarly match the functionality of the corresponding function from the $\Lambda \circ \lambda$ interface, and have equally simple implementations. For `errorRate_`, we first use the $\Lambda \circ \lambda$'s SHE interface to *obtain* the error term, then compute the associated error rate.

5.4 Interpreters

We now describe a selection of the interpreters included with ALCHEMY. We describe our flagship interpreter, the homomorphic compiler, in section 5.5. We have already seen one interpreter in its entirety: the evaluation interpreter `E` described in subsection 5.2.1.

5.4.1 Pretty-printer

The pretty-print interpreter turns an ALCHEMY DSL expression into a string representing the expression. This interpreter was simplified in subsection 5.2.1; we describe the actual implementation here. The pretty-printer `P` is defined as:

```
newtype P e a = P (Int -> String)


pprint :: P () a -> String -- same as previous definition
```

A pretty-print expression is thus represented by a `Int -> String` function. The argument indicates how many variables are in scope for this expression. This value is ignored for most language components like `Add`, `Mul`, `Pair`, etc:

```
instance Add P a where
  add_ = P $ \i -> "add_"
```

However, the scoping information is needed to pretty-print a lambda expression:

```
instance Lambda P where
  lam (P f) =
```

```
    P $ \i -> "(\\v" ++ show  i ++ " -> " ++ f (i+1) ++ ")"
  (P f) $: (P a) =
    P $ \i -> "("    ++ f i ++ " "    ++ a i     ++ ")"
  v0       = P $ \i -> "v" ++ show (i-1)
  s  (P v) = P $ \i -> v (i-1)
```

`lam` creates a string with a lambda for variable i, then recursively prints the rest of the expression, adding one more variable to the scope. The `($:)` operator is object-language function application, which doesn't introduce or hide any variable, so `($:)` appends the two subexpressions without changing the variable counter. Since the body of any lambda has at least one bound variable (by definition), `v0` uses $i - 1$ so that variables are zero-indexed. Finally, since `s` is used to refer to a variable in an extended scope (as opposed to the closest-bound variable), it recursively interprets the subexpression in a context with one fewer variables.

5.4.2 Expression Size

A useful metric for evaluating the complexity of an expression is its *size*, in term of the number of DSL operations used. The size interpreter `S` is defined as:

```
newtype S e a = S { size :: Int }


size :: S () a -> Int
```

Thus all expressions are simply represented by an integer, which can be extracted using `size`. This simple definition leads to equally simple instances:

```
instance AddLit S a where
  addLit_ _ = S 1


instance Lambda S where
```

```
lam (S i) = S $ i+1

(S f) $: (S a) = S $ f + a

v0 = S 1

s (S i) = S i
```

The most interesting instance is `Lambda`. Clearly `v0` should have size one. The same should be true for any other variable, so uses of `s` don't increase i. Lambdas increase the size of an expression by one, so `lam` increments i. Finally, the function application operation (`$:`) simply adds the size of the function and the size of the argument.

5.4.3 Expression Duplicator

So far, we have seen that each ALCHEMY interpreter uses a concrete representation for expressions. However, the expressions themselves are written using only DSL operations, which are abstract and not tied to any specific interpreter (i.e., ALCHEMY expressions are *polymorphic* in their interpreter). Thus expressions can be interpreted with any interpreter that supports the language components used in the expression.

Unfortunately, there is a caveat to this polymorphism: once an interpreter is chosen for a particular expression, the type of the expression is "monomorphized" for that interpreter. This limitation of Haskell's type system[8] precludes the possibility of interpreting a given expression in multiple ways, which severely restricts the flexibility of ALCHEMY expressions.

However, [Kis10] provides a simple way to work around this restriction, by first *duplicating* an expression into two new expressions which have possibly different interpreters. The duplicator interpreter `D` is defined as:

```
data Dup intp1 intp2 e a = Dup (intp1 e a) (intp2 e a)
```

[8]The functionality we seek is known as *impredicative polymorphism*, which is not available in the GHC Haskell compiler.

```
dup :: Dup intp1 intp2 e a -> (intp1 e a, intp2 e a)
```

This is the first interpreter we have seen so far which is parameterized by another interpreter. This is because rather than producing a Haskell value like an `Int` or `String`, `dup` produces *expressions* which can be further interpreted. Specifically, `dup` produces two new expressions, each of which have their own interpreter. Naturally this technique can be applied recursively to interpret an expression in arbitrarily many ways.

The instances for `D` are all very similar and mechanical. We show the `Add` instance below:

```
instance (Add intp1 a, Add intp2 a)
  => Add (Dup intp1 intp2) a where
  add_ = Dup add_ add_
  neg_ = Dup neg_ neg_
```

The constraints on the instance meant that in order to add two `Dup intp1 intp2` expressions of (object-language) type `a`, we must be able to individually add `intp1` expressions of type `a` and `intp2` expressions of type `a`. The implementation mirrors these constraints: `add_` for `Dup intp1 intp2` simply uses `add_` for the `intp1` interpreter and the `intp2` interpreter.

Using `Dup` is very simple:

```
expr = lam $ add_ $: v0 $: v0

(ex1, ex2) = dup expr

pprint ex1
-- "(\\v0 -> ((add v0) v0))"

eval ex2 3
-- 6
```

5.4.4 Logging Error Rates

HE ciphertexts include an error term which grows with homomorphic operations. If the error term gets too large (as measured by the *error rate*), the underlying plaintext is lost. The amount of noise growth depends on the homomorphic operation and the cryptosystem parameters, but it is difficult to predict in advance exactly how the parameters will affect noise growth. Yet noise growth must be taken into account when choosing parameters: if the parameters are too small, we will be unable to decrypt the ciphertext, while overly conservative parameters cause the noise to grow more quickly than necessary, limiting the homomorphic capacity of the cryptosystem.

We can iterate on an optimal parameter combination by measuring the size of the error term after each homomorphic operation, and then adjusting the parameters to increase or decrease noise growth as needed. Thus we would like a way to dynamically log the empirical error rates throughout a homomorphic computation.

In ALCHEMY, homomorphic operations happen at object-language runtime, so the logging functionality also must happen at runtime. However, the accumulation of error rates is an *effect*, so this functionality requires monads in the object language. The language components for this collection of features were given in subsection 5.2.2. Note that these features need not be used in top-level expressions; rather, we provide an *interpreter* which logs error rates by inserting monadic operations from these language components. We introduce the `ErrorRateWriter` interpreter for this purpose:

```
newtype ErrorRateWriter
  intp -- | the underlying interpreter
  k    -- | (reader) monad that supplies keys
  w    -- | (writer) monad for logging error rates
  e    -- | environment
  a    -- | represented type
```

```
    = ERW (k (intp (Monadify w e) (Monadify w a)))

type family Monadify w a where
  Monadify w (a,b) = (Monadify w a, Monadify w b)
  Monadify w (a -> b) = Monadify w a -> Monadify w b
  Monadify w a = w a

type ErrorRateLog = [(String,Double)]

writeErrorRates :: (MonadWriter ErrorRateLog w,
                    MonadReader Keys k)
  => ErrorRateWriter intp z k w e a
     -> k (intp (Monadify w e) (Monadify w a))
```

`ErrorRateWriter` represents expressions with object-language type `a` by a (monadic) sub-expression using the `intp` interpreter with a "monadified" object-language type. The `Monadify` type family pushes the writer monad `w` into the expression type, e.g., turning `a -> (b,c)` into `w a -> (w b, w c)`. If we instantiate `intp` with the evaluation interpreter `E` and run both interpreters, we are left with a monadic Haskell function which dynamically logs error rates. The log consists of a list of `(String,Double)` pairs. The `Double` is the empirical error rate for an intermediate ciphertext, and the string serves as an annotation to help identify the intermediate step within the larger expression.

To interpret a DSL operation, `ErrorRateWriter` uses the following steps:

1. Use one or more inputs to perform the operation with the inner `intp` interpreter, producing a ciphertext.

2. Obtain the secret key used to encrypt the ciphertext using the `k` reader monad.

3. Compute the error rate of the ciphertext by passing the ciphertext and secret key to `errorRate_`.

4. Append the error rate to a log using the monadic object-language features with the writer monad `w`.

These steps are clear in the **Add** instance for **ErrorRateWriter**:

```
instance (Add expr ct, ...) =>
  Add (ErrorRateWriter expr k w) (CT m zp (Cyc m' zq)) where


  add_ = ERW $ do
    Just sk <- lookupKey
    return $ lam $ lam $ tellError "add_" sk $:
      (liftA2_ $: add_ $: v1 $: v0)


tellError :: (MonadWriter_ expr, ErrorRate expr, Pair expr, ...)
  => String -> SK (Cyc m' z)
     -> expr e (CT m zp (Cyc m' zq) -> mon ())
tellError str sk =
  lam (tell_ $: (pair_ $: (LS.string_ str) $:
                          (errorRate_ sk $: v0)))
```

Notice that the **ErrorRateWriter** instance of **Add** is defined *only* when the object language type is a ciphertext. `lookupKey` uses the reader monad to obtain the correct key for this ciphertext. The key is determined by the (inferred) *type* alone, so no arguments are needed. We pass the result of adding the inputs with the `intp` interpreter to `tellError`. This helper function uses `errorRate_` to obtain the error rate of the input ciphertext `v0`, `string_` to turn the annotation into a DSL expression, and `pair_` to glue these pieces together as an

object-language pair. `tell_` logs this value using the object-language writer monad `w`. The implementation for other language components is similar.

The effect of this interpreter is best seen with a simplified example; see chapter 6 for a full example.

```
expr = lam $ add_ $: v0 $: v0


(ex1,ex) = dup expr
pprint ex1
-- "(\\v0 -> ((add v0) v0))"


(ex2,ex3) = dup $ runReader [] $ writeErrorRates ex
pprint $ ex2
-- \v0 -> bind (ap (fmap add v0) v0)
--   (\v1 -> bind (tell (pair "add_Q268440577" (errorRate v1)))
--     (\v2 -> pure v1))


(result,log) = runWriter $ eval ex3 $ encrypt 3
print log
-- [("add_",7.301429694065961e-7)]
```

This example uses the duplicator twice to get three copies of `expr` with three different interpreters. Concretely, the interpreters are `P`, `ErrorRateWriter P k w`, and `ErrorRateWriter E k w`. We first pretty-print `ex1` to show the unmodified expression. Printing the result of running `ErrorRateWriter` shows that `ex2` is a new expression which is equivalent to the original expression (note the `ap (fmap v0) v0`), but which additionally logs the error rate of the output. Evaluating `ex3` produces a list of pairs giving the error rate at each step of the computation. Note that error rates near 0.5 indicate a decryption failure, and all subsequent

ciphertexts in the computation will have a similar error rate. Thus it isn't hard to identify precisely where the parameters are invalid.

While repeatedly evaluating an expression with different parameters to find an optimal parameter combination is *possible*, it's a very tedious process. In section 5.5 we automate parts of this process by using estimates for noise growth which we obtained by using this interpreter.

5.5 Plaintext-to-Ciphertext Compiler

In this section we describe the design and implementation of a "plaintext-to-ciphertext" compiler that, given an "in the clear" program in the plaintext DSL, interprets it as a corresponding "homomorphic" program in the ciphertext DSL for BGV-style SHE. The resulting program can in turn be handled by any ciphertext-DSL interpreter, such as the evaluator, the pretty-printer, or another transformation like an optimizer or the error-rate logger described in subsection 5.4.4. The compiler automatically generates all necessary keys, hints and other auxiliary information, and input ciphertexts. And it *statically* (i.e., at compile time) infers the approximate noise rate of each ciphertext in the computation, choosing appropriate moduli based on their "noise capacity," and emitting a compile error if the programmer has not provided moduli that have enough capacity.

5.5.1 Interface

The plaintext-to-ciphertext compiler is a data type **PT2CT**, defined as follows:

```
newtype PT2CT
  m'map    -- | list of (PT index m, CT index m')
  zqs      -- | list of coprime Z_q components
  gad      -- | gadget type for key-switch hints
  ctex     -- | ciphertext-DSL interpreter
  mon      -- | monad for creating keys/noise
```

```
    e         -- | environment type
    a         -- | plaintext type
    = PC (mon (ctex (Cyc2CT m'map zqs e)
                    (Cyc2CT m'map zqs a)))
```

`PT2CT` is parameterized by several types, which are needed to transform from plaintext operations to ciphertext operations:

- `m'map` is a mapping from (the indices of) plaintext cyclotomic rings to (the indices of) their corresponding ciphertext rings;
- `zqs` is a list of types representing $\mathbb{Z}_q$-components that can be multiplied (forming product rings) to form ciphertext moduli;
- `gad` indicates what kind of decomposition "gadget" to use for creating and using key-switch hints;
- `ctex` is the target ciphertext-DSL interpreter;
- `mon` is a monad in which keys and hints can be generated and accumulated;
- `e` is the usual notion of environment; and
- `a` is the type of the plaintext DSL expression.

Based on these parameters, `PT2CT` simply "wraps" a (monadic) ciphertext-DSL expression of type `(Cyc2CT m'map zqs a)`, interpreted by `ctex`.

`Cyc2CT` is a *type family*—i.e., a function from types to types—that converts an "in the clear" plaintext type `a` to a corresponding "homomorphic" type. For example, it converts the cyclotomic ring type `(Cyc m zp)` to the type of a ciphertext over the cyclotomic ring of index `(Lookup m m'map)`, with an appropriate ciphertext modulus (as determined by the associated "noise rate;" see subsection 5.5.2 below.) Similarly, it converts the type `a -> b` of a *function* by recursing on both arguments `a`, `b`. In this way, functions on plaintexts

(even higher-order ones) correspond to functions on ciphertexts of corresponding types. We emphasize that all these type conversions occur statically at compile time, with no runtime overhead.

The public interace of **PT2CT** is the function

```
pt2ct :: (MonadRandom mon, MonadAccum Keys mon,
          MonadAccum Hints mon, ...)
      => PT2CT m'map zqs gadget ctex mon () a
      -> mon (ctex () (Cyc2CT m'map zqs a))
```

which converts any closed **PT2CT**-expression of "plaintext" type `a` to a closed `ctex`-expression of the corresponding "homomorphic" type. The **MonadAccum** constraints on `pt2ct` indicate that the compilation must take place in a context which permits the accumulation of both secret keys and key-switch hints. In subsection 5.5.3, we will see that **PT2CT** automatically generates secret keys and hints (which requires randomness), and reuses them wherever possible for efficiency. Furthermore, the accumulated keys are used to encrypt plaintext values to ciphertexts under the appropriate key, and to decrypt ciphertexts, e.g., for decrypting a result or to log an intermediate error rate.

5.5.2 Tracking Noise, Statically

Many homomorphic operations, e.g., multiplication and ring switching, introduce (additional) noise into the resulting ciphertext. The amount of noise growth is a function of the gadget, ciphertext ring, and the noise capacity of ciphertext modulus components. When chaining multiple operations together, we must track the noise growth to ensure correct homomorphic evaluation: if the noise rate grows too large, the result cannot be decrypted. For a fixed computation, we can work backwards from a target noise rate to determine the maximum noise rate permitted in the input ciphertexts.

The **PT2CT** compiler performs this analysis *statically*, so there is no runtime overhead. Given an expression and a target output noise rate (typically the maximum allowed for a

successful decryption), it *infers* the maximum allowable noise rate of a ciphertext at any step in the computation.

In order to track this information, **PT2CT** annotates each occurrence of a cyclotomic ring in the plaintext expression with a (type-level) natural number p. This value indicates that the corresponding ciphertext should have noise rate $\alpha \leq 2^{-p \cdot u}$, where u is a global rational constant representing one "unit" of noise. We introduce the data type `PNoise p a` to hold this value, where `p` represents a natural number. When `a` is a cyclotomic ring, `PNoise p a` represents a corresponding ciphertext with noise rate α satisfying $p \leq \frac{-\lg \alpha}{u}$, which explains the name "pNoise". **PT2CT** compiles expressions involving annotated cyclotomic rings[9] into a ciphertext expression which is guaranteed to satisfy the requirements on the noise rates. The compiler achieves this guarantee by statically selecting ciphertext moduli which are large enough to support the requested noise rate. For example, if a cyclotomic ring is annotated with pNoise p, the corresponding ciphertext might need to be at least $p \cdot u + c$ bits, for some constant c.

We now show how **PT2CT** calculates pNoise for two operations: `(*:)` and `linearCyc`. Their types indicate that an interpreter *may* require the input and output types of the operation to be different:

```
linearCyc :: Linear zp e r s
  -> expr env ((PreLinearCyc expr rep) (Cyc r zp))
  -> expr env rep (Cyc s zp))

(*:) :: expr e (PreMul expr a) -> expr e (PreMul expr a) -> expr e a
```

The type families **PreLinearCyc** and **PreMul** determine the input type to these operations. **PT2CT** defines them as:

[9] Since **PT2CT** requires expressions involving **PNoise**, we extend all relevant interpreters to support operations on this data type.

```
type PreLinearCyc (PT2CT m'map zqs gad z ctex mon)
       (PNoise p) = PNoise (p :+: N1)

type PreMul (PT2CT m'map zqs gadget ctex mon) (PNoise p a) =
       PNoise (Units2PNoise (TotalUnits zqs (p + 3))) a}.
```

Example Consider the example from subsection 5.1.2:

```
ex1 = lam $ lam $ v0 *: (v0 +: v1)
```

We can specialize the type of this expression to

```
PT2CT m'map zqs gadget ctex mon () (PNoise pin a -> PNoise 0 a).
```

Setting the output pNoise to zero indicates that when we homomorphically evaluate this expression, we will immediately decrypt the result without doing any further homomorphic operations. Based on the signatures of `(*:)` and `(+:)`, the compiler infers that `v0`, `v1` have type

```
PT2CT m'map zqs gadget ctex mon () (PreMul expr b).
```

Using the definition of the **PreMul** type family for **PT2CT** given above, we find that the input ciphertexts must have pNoise $pin = 0 + 3 = 3$.

Similarly, **PT2CT** gives a type family that converts annotated plaintext types into ciphertext types:

```
Cyc2CT m'map zqs (PNoise p (Cyc m zp)) =
  CT m zp (Cyc (Lookup m m'map) (ZqPairsWithUnits zqs (p + 2)))
```

which indicates that input ciphertext modulus must have at least $3 + 2 = 5$ total units, or about 31 bits.

5.5.3 Implementation

We now show how **PT2CT** implements several instructive language components. Some plaintext operations, like addition, translate directly into addition on ciphertexts. This leads to a very simple **Add** instance for **PT2CT**:

```
instance (Add ctex (Cyc2CT m'map zqs a), Applicative mon)
  => Add (PT2CT m'map zqs gad ctex mon) a where

  add_ = PC $ pure add_
  neg_ = PC $ pure neg_
```

The implementation of add_ for **PT2CT** simply embeds the (pure) function add_ *on ciphertexts* into the applicative mon.

By contrast, plaintext multiplication becomes much more involved when translated to its homomorphic counterpart:

```
instance
  (Lambda ctex, Mul ctex ctin, SHE ctex, MonadRandom mon,
   MonadAccumulator Keys mon, MonadAccumulator Hints mon, ...)
  => Mul (PT2CT m'map zqs gad ctex mon) (PNoise p (Cyc m zp)) where
  mul_ = PC $ do
    -- lookup or generate a key switch hint
    hint <- getQuadCircHint
    return $ lam $ lam $
      -- switch from the hint modulus to the output modulus
      modSwitch_ $:
        -- perform a quadratic key switch using the hint
        (keySwitchQuad_ hint $:
          -- switch the ciphertext modulus to the hint modulus
```

```
    (modSwitch_ $:
      -- multiply the input ciphertexts
      (v1 *: v0)))
```

We need the `Mul ctex ctin` constraint to multiply the `ctex` terms bound to the input variables `v0` and `v1`. The result is a *quadratic* ciphertext, which must be converted via *key switching* back to a linear ciphertext; this requires an appropriate *key-switch hint*. The function `getQuadCircHint` relies on the `MonadAccumulator Hints mon` constraint to look up an existing hint, or, failing that, to generate and store (accumulate) a random hint, where randomness generation relies on the `MonadRandom mon` constraint. Generating a hint in turn requires knowledge of the secret key under which the ciphertext is encrypted, which can be looked up (or, failing that, generated and stored) thanks to the `MonadAccumulator Keys mon` constraint.

The hint is typically generated with a larger modulus than the ciphertext modulus (the exact size depends on the choice of gadget), so that the amount of noise introduced when key-switching is small compared to the existing ciphertext noise. As a result, we must rescale the ciphertext to the key-switch modulus, perform the key switch, and then rescale the ciphertext modulus again to the final output modulus.

Another plaintext operation with a non-trivial homomorphic counterpart is `linearCyc_`:

```
instance (MonadAccumulator Keys mon, MonadRandom mon,
          SHE ctex, Lambda ctex, ...)
  => LinearCyc (PT2CT m'map zqs gad z ctex mon) (PNoise p) where
  linearCyc_ f = PC $ do
    hint <- getTunnelHint f -- generate a hint for tunneling
    return $ lam $
      modSwitch_ $:         -- scale back to the target modulus
      (tunnel_ hint $:      -- tunnel with the hint
        (modSwitch_ $: v0)) -- scale up to the hint modulus
```

We can homomorphically apply a linear function from one ring to another using a special type of ring-switching called *tunneling* (see chapter 3). This process is a special form of key-switching, which requires an appropriate hint. As with multiplication, we perform these key switches with a larger modulus to minimize noise growth, so we must switch the ciphertext modulus up before tunneling, and back down after tunneling. As before, performing a key-switch requires an appropriate hint, which depends on the secret keys. These keys are obtained or generated as above. Note that there is no `MonadAccumulatorHints mon` constraint because in typical use cases, hints for tunneling cannot be reused, so we do not attempt to explicitly store them after they are generated. (However, they *are* embedded in the ultimate compiled ciphertext-DSL term.)

5.6 Future Work

ALCHEMY represents a large step towards making practical HE a reality. We explore some areas for future improvement below.

Interpreters. It is possible to conceive of a huge number of interesting and useful interpreters. A large class of these interpreters come in the form of *optimizers*, which turn an expression (in some DSL) into an equivalent expression that is more efficient. We have already seen how an interpreter for beta-reducing expressions could be useful for simplifying expressions. Note that beta-reduced expressions are no more efficient than the original, but some interpreters benefit from this optimization. In particular, beta-reduced expressions have a much simpler representation using the `P` interpreter, and the `S` interpreter gives a more accurate estimate of the size of beta-reduced expressions.

HE Parameters. Although the `PT2CT` compiler automatically chooses ciphertext moduli at each step of the computation, users must still provide some ciphertext parameters. In particular, the user must provide a *pool* of ciphertext moduli that the compiler can choose from, as well as all ciphertext cyclotomic indices. Ideally, ALCHEMY would be a black

box that chooses all ciphertext parameters to achieve both optimal performance as well as the desired security level. However, much more theoretical work must be done before such an tool can be realized. Specifically, this functionality would require a much better understanding of the concrete security level for a given parameter combination.

Bootstrapping for Fully Homomorphic Encryption ALCHEMY currently compiles computations to a target SHE scheme. The only known way of achieving FHE is to periodically perform Gentry's *bootstrapping* procedure [Gen09a]. Like many of the steps automated by ALCHEMY, it should be relatively easy to predict when bootstrapping needs to occur. Ideally, bootstrapping would be inserted seamlessly into (arbitrary) homomorphic computations; we leave this for future work.

Meta-language Function Application. We finish this summary by noting that there is a powerful technique called *higher-order abstract syntax* (HOAS), which uses the meta-language's variable-binding and function-creation facilities for creating object-language functions, thus obviating the need for environments and De Bruijn-indexed variables as in subsection 5.2.1. However, this technique seems unsuitable for some of our advanced needs, specifically monadic interpreters. Some initial progress has recently been made on this front [KKS15], but as the authors caution, it is not yet ready for general-purpose use.

The implementation of object-language functions has far-reaching implications throughout ALCHEMY. It not only affects how easy it is to read and write plaintext expressions, but it also affects properties of the interpreters. For example, we cannot write the `Dup` or `ErrorRateWriter` interpreter using HOAS. Yet HOAS is simpler to use in expressions, and De Bruijn variables bring their own challenges. Concretely, we have been unable to obtain sharing fo subexpressions in the `ErrorRateWriter` interpreter. The only impact is that error rates for shared expressions appear multiple times in the log, however it is an indicator that De Bruijn variables do not provide all the solutions for this complex problem. It remains an active area of research in the programming language community.

CHAPTER 6

FAST HOMOMORPHIC EVALUATION OF SYMMETRIC KEY PRIMITIVES

Some recent works (e.g., [NLV11; GHS12c; WH12; Che+13]) have examined the suitability of homomorphic encryption for evaluating "non-trivial" functions that can offer practically useful functionality. In this chapter, we explore the homomorphic evaluation of *ring rounding*, which appears as the main operation in a surprising number of lattice primitives including SHE [BGV14], the RLWR problem [BPR12], and pseudorandom functions [BPR12; BP14]. The homomorphic evaluation of this operation therefore becomes an important part of the bootstrapping procedure for FHE, which evaluates the SHE decryption circuit homomorphically.

Another particularly important application, which serves as a motivating example throughout this chapter, is the homomorphic evaluation of *symmetric-key* cryptographic primitives. For instance, given a ciphertext $\text{HE}(k)$ encrypting a key k for a symmetric-key encryption scheme Enc_k, and a ciphertext $c = \text{Enc}_k(m)$ encrypting a message m, one can homomorphically compute $\text{HE}(\text{Dec}_k(c)) = \text{HE}(m)$ by homomorphically applying the function $f(x) = \text{Dec}_x(c)$ to $\text{HE}(k)$. This particular instance of homomorphic evaluation is extremely powerful, and even seems *necessary* for many practical usage scenarios of HE, as we explain in the next section.

6.1 Homomorphic Evaluation of Symmetric-Key Primitives

Perhaps the most straightforward application of the homomorphic evaluation of a symmetric-key decryption algorithm is in reducing communication and computation for a weak client (Alice) who delegates her computation to the "cloud" (Bob). The simplest form of this application is described in subsection 5.1.3, and requires Alice to encrypt her data with an HE scheme and send those ciphertexts to Bob. However, all known HE schemes that can

evaluate reasonably complex functionalities have quite large keys and ciphertexts, so a weak client like Alice may not be able to directly encrypt its data under such a scheme. Instead, it can encrypt the data under a lightweight symmetric-key scheme, and then the cloud can homomorphically transform that encrypted data into HE-encrypted data as described above, which could then be processed further (homomorphically). Since plain symmetric encryption will almost certainly remain much more time- and space-efficient than HE encryption, this approach provides major savings in communication and computation by the weak device. (The relatively heavy computation of HE(k), where k is the key of the symmetric encryption scheme, can be done once and for all in an offline preprocessing phase, and then subsequently used in all homomorphic computations.)

This example illustrates a more general template for enhancing the performance of HE-based applications, especially ones in which there is a large amount of plaintext data. For example, Wu and Haven [WH12] proposed an efficient implementation of SHE in the context of large-scale statistical analysis (e.g., linear regression on a multidimensional encrypted corpus). Since the plaintext-to-ciphertext expansion of HE schemes is large, it may be impractical to directly encrypt a large corpus under such a scheme. Instead, one could encrypt the data under a (nearly) length-preserving symmetric scheme, then homomorphically decrypt as needed. Moreover, symmetric encryption can allow for random access to the data, which is desirable if a particular homomorphic evaluation needs to use only a small part of the data.

Another appealing instantiation of this template is in the construction of a private information retrieval (PIR) scheme. As suggested by Brakerski and Vaikuntanathan [BV11a], any efficient procedure for homomorphic symmetric decryption can be efficiently converted into a single-server PIR protocol, by having the client encrypt its query under a symmetric scheme, rather than an HE. The server then homomorphically decrypts the query and then additionally evaluates an arithmetic circuit of size N and depth only $\log \log N$, where N is

the size of the database held by the server. Efficient homomorphic decryption can therefore greatly enhance the efficiency of the protocol for both the server and client.

One final application is that of key derivation in the cloud. Here a client generates a master key k, and wishes to use it for deriving a large number of pseudorandom keys $k_i = F_k(i)$ for different uses, where F_k is a pseudorandom function (PRF). The client can delegate these derivations to the cloud, and also obtain some robustness against the compromise or loss of secret keys, as follows. The client stores an encryption $c = \text{HE}(k)$ on the cloud and erases k, keeping only the HE decryption key. The client can later request that the cloud homomorphically compute $\text{HE}(F_k(i))$ for any desired i, and then decrypt the result locally. The cloud (or an attacker who compromises it) learns nothing about the master key k, whereas an attacker who compromises the client alone learns only the HE secret key. This might allow the attacker to compute several session keys, but only with the cloud's continued help, which might mitigate the damage if, e.g., the client learns of the intrusion and notifies the cloud.

6.1.1 Homomorphic Evaluation of AES

To date, the only attempts to homomorphically evaluate symmetric-key primitives [GHS12c; Che+13] have focused on the AES-128 function. The suitability of AES as a benchmark is justified by its wide deployment and extensive use in security-aware applications, as well as by its non-trivial yet manageable circuit size and depth. Moreover, the AES circuit has a regular and quite "algebraic" structure, which is very amenable to parallelism and other optimizations in the context of homomorphic evaluation. For precisely those reasons, it seemed plausible that a specially designed and optimized implementation would result in reasonable performance, and yield considerable practical utility.

Unfortunately, despite many clever optimizations and careful adaptations to the structure of AES, the best reported homomorphic evaluations of the AES function are very far from practical. As is to be expected, the inefficiency lies in the extremely high degree of the

AES function, which induces an arithmetic circuit depth of at least 50. To securely evaluate such a circuit homomorphically requires exceedingly large keys for the HE, and very large runtimes. The work of [GHS12c] reports that homomorphic evaluation of AES on a single block takes around 36 hours on a server-class machine, and uses up to 256 gigabytes of memory. "Batching," i.e., computing several hundred blocks at a time, brings the runtime down to as low as 5 minutes per block, but takes about 2.5 days to complete, and none of the results are available until the end. Similar (but slightly worse) experimental results were recently reported in [Che+13], which used a quite different underlying SHE scheme for the homomorphic computation.

6.1.2 In Search of Efficient Alternatives

Practical homomorphic evaluation of symmetric primitives seems quite far off, if the search is limited to standard candidates like AES. Motivated by this state of affairs, we consider whether different symmetric constructions can support significantly faster homomorphic evaluation. This would enable the applications mentioned above, and would undoubtedly broaden the applicability of HE in practice.

Towards this end, we consider a weak pseudorandom function $F_s\colon \{0,1\}^k \to \{0,1\}^n$ indexed by a randomly chosen key s, which is widely applicable in symmetric-key cryptography. Recall that a *weak PRF* cannot be efficiently distinguished from a uniformly random function, given polynomially many pairs of the form $(x_i, F_s(x_i))$, where the $x_i \in \{0,1\}^k$ are uniformly random and independent (not chosen by the adversary). It is well known that weak PRFs can be used in a generic manner to implement symmetric-key encryption, in the following way:

$$\text{Enc}_s(m;r) = (r, F_s(r) \oplus m), \qquad \text{Dec}_s(r,c) = F_s(r) \oplus c,$$

where $m \in \{0,1\}^n$ and $r \in \{0,1\}^k$. The scheme is IND-CPA secure provided that F_s is a weak PRF for sufficiently large input length k, and r is chosen uniformly at random in each invocation.

Notice that homomorphically computing a PRF F_s, i.e., computing HE$(F_s(r))$ from HE(s) and r, followed by a single homomorphic exclusive-or operation, corresponds to homomorphically computing HE$(\text{Dec}_s(c)) = \text{HE}(F_s(r) \oplus c)$ from HE(s) and $c = \text{Enc}_s(m; r)$. Thus, efficient homomorphic evaluation of the PRF directly translates to efficient homomorphic symmetric decryption. So for all applications described above (including homomorphic key management, which is attained just by evaluating the PRF itself), it is sufficient to focus on fast homomorphic evaluation of pseudorandom functions.

6.1.3 Our Results

Our primary technical contribution is the design and implementation for the efficient homomorphic evaluation of the non-trivial *ring rounding* operation, including a novel arithmetization of an operation for rounding integer coefficients of a ring element. As a concrete application, we use ring rounding to implement homomorphic evaluation of the weak pseudorandom function of Banerjee, Peikert, and Rosen (BPR) [BPR12]. Our experimental results show that the homomorphic evaluation of the BPR PRF is *dramatically* more efficient than the homomorphic evaluation of AES. For instance, on a standard laptop computer we can homomorphically evaluate one useful and apparently secure instantiation of the BPR weak PRF on a *single* input in ***less than 90 seconds***, and requiring less than 150 megabytes of memory. This is more than ***1,400 times faster*** (on weaker hardware) than the best reported total runtime for AES evaluation, and uses less memory by a factor of more than 1,500. Figure 6.1 gives a high-level performance comparison between our implementation and prior ones that homomorphically evaluate the AES function.

The key idea behind our design is to exploit the simple (yet still seemingly secure) algebraic structure of the BPR pseudorandom function, and its tight "algebraic fit" with

lattice-based HE constructions. Most importantly, this relationship allows us to use an HE plaintext space that perfectly coincides with the domain of the PRF key elements, and the operations that the PRF performs on them. From the perspective of homomorphic evaluation, this correspondence translates into two main advantages:

1. It yields relatively compact encryptions of the PRF key element under the HE scheme, and allows for significant savings in the amount of auxiliary data (i.e., key-switching "hints") needed for homomorphic evaluation.

2. It leads to very simple, small, and low-depth arithmetic circuits (and hence fast homomorphic evaluation) for the main operation in the PRF computation, namely, "rounding" a public multiple of the secret to a smaller modulus.

Table 6.1: Performance comparison with prior homomorphic evaluations of AES [GHS12c; Che+13].

	total runtime (sec)	time/block (sec)	memory
AES-128, 54-block batch	130,000	2,400	256 GB
AES-128, 720-block batch	216,000	300	256 GB
BPR weak PRF	90	90	160 MB

To date, a large roadblock for implementing homomorphic computations like pseudorandom functions has been the complexity of using existing HE implementations. We give a concise implementation using ALCHEMY (chapter 5), which automatically handles complexities such as parameter generation, key/hint management, and noise management operations. We compare our ALCHEMY implementation with a reference implementation using the interface from section 4.3, and find that ALCHEMY greatly reduces the implementation burden with no loss in performance.

Organization. The rest of the chapter is organized as follows.

Section 6.2 defines the important *ring-rounding* operation and our method for evaluating it homomorphically.

Section 6.3 gives our novel arithmetization (in terms of operations natively supported by our target SHE scheme) of the *integer* rounding operation that is central to the homomoprhic evaluation of ring-rounding. This arithmetization is more suitable for rounding small moduli like those used in our PRF instantiation.

Section 6.4 introduces the BPR weak PRF, our concrete instantiation, and the homomorphic evaluation fo the PRF.

Section 6.5 analyzes the concrete security of our BPR instantiation against known classes of attacks.

Section 6.6 describes the implementation of homomorphic ring-rounding and PRF evaluation using ALCHEMY.

Section 6.7 quantitatively measures the savings of using ALCHEMY compared to the current method of hand-writing homomorphic computations. Note that the PRF application is primarily a tool for evaluating ALCHEMY; for evaluating the BPR PRF implementation, see the discussion above this paragraph which compares it to prior homomorphic evaluation of symmetric-key primitives.

6.2 Homomorphic Computation of Ring Rounding

We start by describing the ring rounding function and the technical ideas underlying its homomorphic evaluation. Let R be a cyclotomic ring of arbitrary index m, and let $n = \varphi(m)$. The ring rounding function is $\lfloor \cdot \rceil_q \colon R_p \to R_q$. This operation is highly non-linear (which leads to its usefulness in lattice primitives), so it is not obvious how to efficiently evaluate in homomorphically.

We call upon the literature on *bootstrapping*, which is Gentry's technique [Gen09b; Gen09a] for transforming an SHE into an FHE by homomorphically evaluating the SHE's decryption function. Since the rounding function $\lfloor \cdot \rceil_q \colon R_p \to R_q$ is essentially the same nonlinear step performed in the decryption algorithm of lattice-based cryptosystems (but for a much smaller modulus p), bootstrapping techniques from several prior works provide exactly what we need here. In more detail, the rounding step proceeds in two phases:

- **Ring-switch:** First, we homomorphically move the $\mathbb{Z}_p$-coefficients of the input into separate plaintext "slots" of a different plaintext ring S_p, using the ring-tunneling technique from section 4.2.
- **Batch-round:** Then, we apply the *integer* rounding function $\lfloor \cdot \rceil_q \colon \mathbb{Z}_p \to \mathbb{Z}_q$ *in batch* to all the slots at once, at the cost of just one homomorphic evaluation of the integer rounding function.

Starting with the latter step, Smart and Vercauteren [SV11] first proposed the idea of batched (or SIMD) homomorphic operations. There are several known arithmetizations of the integer rounding step in the special case where $p = 2^k$ is a power of two and $q = 2$. Gentry, Halevi and Smart [GHS12a] described a simple arithmetic circuit for these parameters (slightly improved in [AP13]) which has depth exactly $\log(p/2)$ and performs about $\log^2(p)/2$ multiplications and additions. In section 6.3 we give a quite different circuit for the same specialized parameters, having the same $\log(p/2)$ depth, which can be evaluated using exactly $p/4$ multiplications (and no additions). This is asymptotically worse but *concretely* better than $\log^2(p)/2$ when $p \leq 32$, which is the case in our implementation. We emphasize that all of these parameters are restricted to the case where $p = 2^k$ and $q = 2$; an arithmetization for somewhat more general parameters is given in [HS15].

Moving coefficients into separate slots is more involved. Gentry *et al.* [GHS12a] gave a procedure for doing this, but it requires working in more complex cyclotomic rings than are convenient for our PRF, and it appears very difficult to implement and inefficient. In particular, it relies on a general-purpose circuit compiler for HE [GHS12b],

and seems primarily of theoretical interest. Instead we rely on the ring-tunneling technique given in chapter 4, which improves upon the work of Alperin-Sheriff and Peikert [AP13]. Tunneling gives a simple linear procedure for transferring the coefficients of an R_p-element into the plaintext slots of a *different* ring S_p, in which we can batch-round and finally decrypt the resulting bits. To our knowledge, our weak PRF provides the first implementation of a batched rounding circuit.

Altogether, for the full evaluation of ring rounding, we obtain a very simple and regular arithmetic circuit, consisting of: (1) a sequence of (at most) $\log n$ ring-tunnels (which, despite being a linear operation, performs operations and induces noise growth roughly matching those of a homomorphic multiplication for each tunnel), and (2) a complete binary tree of multiplications for the (batched) integer rounding. The total effective multiplicative depth is therefore bounded by $\log n + \log(p/2)$.

6.3 Rounding Circuit for Small Moduli

In this section we describe a simple arithmetic circuit that for any $p = 2^\ell$ computes the rounding function $\lfloor \cdot \rceil_2 \colon \mathbb{Z}_p \to \mathbb{Z}_2$, i.e., it returns the bit indicating whether the input is closer (modulo p) to 0, or to $p/2$.[1] This operation is useful in a variety of contexts: it is an important part of the bootstrapping step for FHE and is also the central component needed for the RLWR problem and the strong and weak PRFs given in [BPR12], as well as for the strong PRF in [BP14]. While this operation is easy to implement in-the-clear, it is not a "native" operation for our SHE scheme.

Gentry, Halevi, and Smart [GHS12a] described an algebraic procedure (slightly improved in [AP13, Appendix B]) that can be used to (homomorphically) compute the rounding function in $\log_2(p/2)$ multiplicative depth, using a total of about $\log_2^2(p)/2$ (homomorphic) multiplications and additions each. Here we describe a very different procedure that computes the function in $\log_2(p/2)$ multiplicative depth, exactly $p/4$ homomorphic multiplica-

[1] We thank Jacob Alperin-Sheriff (personal communication) for important observations that contributed to the results of this section.

tions, and no homomorphic additions.[2] While our procedure is clearly worse *asymptotically*, it actually performs fewer operations in the same depth when $p \leq 32$, which is the case for our PRF instantiation. The procedure is also very simple to implement, especially with the help of ALCHEMY (see chapter 5).

For $i \in [\ell]$, define functions $f_i \colon \mathbb{Z}_p \to \mathbb{Z}_{p/2^i}$ recursively as follows: let $f_0(x) = x$ be the identity function, and for $1 \leq i \leq \ell - 1$ define

$$f_{i+1}(x) = \frac{f_i(x) \cdot f_i(x - 2^i)}{2} \bmod p/2^{i+1}. \tag{6.3.1}$$

Note that due to the division by two in Equation (6.3.1), in order for f_{i+1} to be well defined, at least one of $f_i(x), f_i(x - 2^i)$ must be even for all $x \in \mathbb{Z}_p$. The following lemma (for the special case $k = 1$) proves this fact in a more general form, which we will need for our final claim.

Lemma 6.3.1. *Let $0 \leq i \leq \ell$ and $0 \leq k \leq \ell - i$, and let $x \in \mathbb{Z}_p$ be arbitrary. Then over all $j \in [2^k]$, exactly one of $f_i(x - j \cdot 2^i) \in \mathbb{Z}_{p/2^i}$ is divisible by 2^k, namely, the one for which $j = \lfloor x/2^i \rfloor \pmod{2^k}$.*

Proof. We proceed by induction on i. First consider the base case $i = 0$, where $f_0(x) = x$ is the identity function. Since $2^k \leq 2^\ell \leq p$, the 2^k consecutive residue classes $x, x-1, \ldots, x-(2^k-1) \in \mathbb{Z}_p$ are all distinct, and clearly, $x - j$ for $j = x \pmod{2^k}$ is the only one divisible by 2^k.

To prove the lemma for positive $i \leq \ell$ and any $k \leq \ell - i$, assume that it holds for $i - 1$ and any $k \leq \ell - i + 1$. By definition of f_i, for any $j \in \mathbb{Z}$ we have

$$f_i\big(x - j \cdot 2^i\big) = \frac{f_{i-1}(x - (2j) \cdot 2^{i-1}) \cdot f_{i-1}(x - (2j+1) \cdot 2^{i-1})}{2} \bmod p/2^i.$$

[2] Our procedure also adds several fixed constants to a ciphertext, but these steps take essentially no time, and incur no growth in the ciphertext noise.

By the inductive hypothesis applied with $x - (2j) \cdot 2^{i-1}$ and $k = 1$, exactly one of the two terms in the numerator is even, and so the largest power of two that divides $f_i(x - j \cdot 2^i)$ is exactly half that of the even term. In addition, over all $j \in [2^k]$, each $f_{i-1}(x - j' \cdot 2^{i-1})$ for $j' \in [2^{k+1}]$ appears in the numerator exactly once. By the inductive hypothesis, exactly one of those terms is divisible by 2^{k+1}, so exactly one of $f_i(x - j \cdot 2^i)$ is divisible by 2^k. Specifically, it is the one for which $j = \lfloor j'/2 \rfloor$, where $j' = \lfloor x/2^{i-1} \rfloor \pmod{2^{k+1}}$ by the inductive hypothesis. Therefore, $j = \lfloor x/2^i \rfloor \pmod{2^k}$, as claimed. □

Corollary 6.3.2. *The function* $f_{\ell-1} \colon \mathbb{Z}_p \to \mathbb{Z}_2$ *is* $f_{\ell-1}(x) = \mathrm{msb}_p(x) = \lfloor x/2^{\ell-1} \rfloor$.

Proof. Letting $i = \ell - 1$ and $k = 1$ in Lemma 6.3.1, we have that $f_{\ell-1}(x) \in \mathbb{Z}_2$ is even (i.e., equals 0) exactly when $\lfloor x/2^{\ell-1} \rfloor = 0 \pmod 2$, i.e., when $x \in \{0, \ldots, p/2 - 1\} \pmod p$.□

By fully expanding $f_{\ell-1}(x + p/4)$ in terms of f_0 using Equation (6.3.1), we see that the rounding function $\lfloor x \rceil_2$ can be expressed as a complete binary tree with $p/2$ leaf nodes and depth $\log_2(p/2) = \ell - 1$, where the leaf nodes hold the terms $x - j$ for $j \in \{-p/4, \ldots, p/4 - 1\}$, and the internal nodes are all "multiply-and-divide-by-two" arithmetic gates. Given an encryption $c = (c_0, c_1)$ of x, we can trivially get an encryption of each $x - j$ by just subtracting j from the constant term c_0. We can then homomorphically compute $\lfloor x \rceil_2$ by evaluating the gates of the tree, which takes exactly $p/2 - 1$ homomorphic multiplications (and no additions).

Finally, the above method can be improved to require only $p/4$ multiplications, thus halving the total work. The idea is to restructure the tree so that leaves $(x - j), (x - (-j - 1))$ for $j \in \{-p/4, \ldots, -1\}$ are paired as siblings, and more generally, every internal node at level $i = 1, 2, \ldots, \ell$ (where level 0 is the leaf level) has one descendant leaf from each residue class modulo 2^i. It is straightforward to generalize the proof of Lemma 6.3.1 to show that any such tree correctly computes the rounding function. With these pairings, the nodes at level 1 are encryptions of $(x - j)(x - (-j - 1))/2 = (x^2 + x - (j^2 + j))/2$, which just differ by known constants. Therefore, all the encryptions at level 1 can be computed

using just one homomorphic multiplication, then adjusting its constant term. It is tempting to think that this trick could be generalized to reduce the number of multiplications further (perhaps to only $\log_2(p)$), by efficiently deriving many of the level-2 ciphertexts from just a few others, but so far we have not found a way to do this. In any case, the rounding function is not the main bottleneck in our implementations.

6.4 Homomorphic Computation of the BPR Weak PRF

In this section, we use the homomorphic evaluation of ring rounding as a building block for the homomorphic evaluation of the BPR weak pseudorandom function [BPR12]. We give concrete parameters for our instantiation and a security analysis against known attacks on the PRFs.

6.4.1 BPR Weak PRF

Let R be a cyclotomic ring of arbitrary index m, and $n = \varphi(m))$ be the dimension of the ring over the integers. For $p = 2^k$ a power of two, the BPR family of weak pseudorandom functions is the set of functions $f_s : R_p \to \{0,1\}^n$, indexed by a ring element $s \in R_p$, and defined as the "rounded product"

$$f_s(a) := \lfloor a \cdot s \rceil_2.$$

Here $\lfloor \cdot \rceil_2 \colon R_p \to R_2$ denotes the "rounding function" that maps each of its input polynomial's n coefficients to $\mathbb{Z}_2 = \{0,1\}$ depending on whether the coefficient is closer (modulo p) to 0 or to $p/2$. (Formally, the integer rounding function maps $a \in \mathbb{Z}_p$ to $\lfloor \frac{2}{p} \cdot a \rceil \in \mathbb{Z}_2$.) The resulting polynomial is interpreted as an n-bit string simply by reading off its coefficients in order.

It is proved in [BPR12] that when $s \in R_p$ is drawn from an appropriate distribution, and p is sufficiently large, the above function family is a weak PRF family—or equivalently,

that the *ring-Learning With Rounding* (ring-LWR) problem is hard—assuming that the ring-LWE problem [LPR13b] is hard in R_p. This proof provides strong evidence that the family has a sound design and is indeed a secure weak PRF, at least in an asymptotic sense. The intuition behind the proof is that the rounding function destroys all but the most-significant bits of the product $a \cdot s$, and that the round-off term can be seen as a kind of "small" error, though one that is generated deterministically from $a \cdot s$ rather than as an independent random variable (as in the LWE problem).

We note that the *known proofs* of security (under ring-LWE) require p to be super-polynomial in n. (More precisely, p has to be lower bounded by the total number of samples observed by the adversary, times a $\mathrm{poly}(n)$ factor [Alw+13]). However, as discussed in [BPR12], the family may not *require* such large parameters for concrete security. Indeed, even for rather small values of n and p—much smaller than those typically required for *public-key* schemes—the family appears to be secure against all classes of attacks that are usually employed against lattice-based cryptography. (See section 6.5 for further details.)

6.4.2 PRF Instantiation

We instantiate this PRF with $m = 128$ (corresponding to $n = 64$) and $p = 32$. Our implementation uses the arithmetization given in section 6.3, which is the most efficient for this choice of p. We emphasize that these parameters are *substantially more aggressive* than those that have been *proven* secure based on LWE and worst-case lattice problems [BPR12; Alw+13]. However, as we show in section 6.5, they still appear to provide *more than 100 bits of security against all known attacks*. This state of affairs may be explained by the fact that the known proofs of security (which only provide *lower* bounds on security) appear quite loose in terms of parameters.

Here we summarize how the parameters n and p affect the security of our PRF instantiation and the efficiency of its homomorphic evaluation.

- The ring dimension n of R is the primary security parameter of the PRF, i.e., security grows exponentially with n (for large enough p). Since secret-key elements from R_p are encrypted under the SHE, this ring will be the initial SHE plaintext space; however, for security it must be embedded in a much larger ciphertext ring (see section 4.3). Therefore, n turns out to have almost no effect on the efficiency of the homomorphic (subset-)product. However, it does moderately affect the efficiency of the rounding step, because we need to switch to a different ring S (via the tunneling procedure in section 4.2; see below) having at least n CRT slots.

- The weak PRF is evaluated homomorphically by composing the ring-tunneling procedure with the (batch) integer rounding procedure. The former operation is linear in the plaintext, but the computation and noise growth of each of the (at most) $\log(n)$ ring switches is roughly comparable to that of a homomorphic multiplication. The latter procedure has multiplicative depth $\log(p/2)$.

Since the efficiency of homomorphic encryption schemes degrades primarily with the multiplicative depth supported, for efficiency we want to minimize n and p while ensuring that the weak PRF is secure. In section 6.5 we argue that $n \geq 64, p \geq 32$ suffices against all known attacks.

The full evaluation of the BPR weak PRF has the same multiplicative depth as the ring rounding evaluation, namely, $\log n + \log(p/2)$, which for our choices of parameters ranges between 8 and 10, and the total number of homomorphic multiplications (of two ciphertexts) is only $p/4$. We note that while an arithmetic depth of 8 might initially seem a bit worrisome in terms of security (certainly compared to the depth of AES, say), the operations performed at each level of the circuit are much more complex than simple binary logic gates, since they correspond to arithmetic operations in complex rings.

6.4.3 Homomorphic Evaluation

Given an encryption of $s \in R_p$ and an input $a \in R_p$ (in the clear), the homomorphic evaluation of $f_s(a)$ proceeds in two steps:

- **Multiply:** The (encrypted) key element s and the input a are homomorphically multiplied to obtain the (encrypted) element $a \cdot s \in R_p$. Since a is public, the product is cheaply computed as a "scalar" multiplication with the encryption of s (i.e., no key-switching or degree/modulus reduction is required, and there is little noise growth.)[3]

- **Round:** The coefficients of the product $a \cdot s \in R_p$ are homomorphically rounded, resulting in an element of the quotient ring R_2 (representing the n-bit output). This step uses the homomorphic evaluation of ring rounding described in the previous section.

See subsection 6.5.1 for a security analysis of the PRF instantiation, and subsection 6.5.2 for details about the parameters used for homomorphic evaluation, as well as a security analysis of the SHE instantiation.

We note that for fast evaluation "in the clear," it is best if the modulus p is a prime congruent to 1 modulo 128, so that efficient Chinese remaindering techniques can be used. But for such moduli, it is somewhat cumbersome to round in a way that produces unbiased output bits. In our setting, we can conveniently set p to be a power of 2, thus ensuring unbiased rounding, while using Chinese remaindering on the HE *ciphertexts* to speed up computation.

[3] If a were also encrypted, then the product could still be computed using a "true" homomorphic multiplication, but at greater expense.

6.5 Security of the PRF Instantiation

In this section we analyze the security of our BPR instantiation from section 6.4, and its homomorphic evaluation, against known classes of attacks. To summarize:

- The security of the weak PRF is syntactically equivalent to the hardness of the corresponding ring-LWR problem. The best known attacks against ring-LWR are those against the corresponding ring-LWE problem, where the round-off term is viewed as the error.
- Our parameters are such that the corresponding ring-LWR/LWE problem enjoys more than 100 bits of security against all known attacks.
- We choose conservative parameters for our SHE scheme, which should offer at least 128 bits of security.

6.5.1 Security of PRF

We briefly point out that the input space of the weak PRF is p^n, which for our parameters is more than enough to defeat birthday attacks on the standard weak-PRF encryption scheme.

PRF Attacks as Learning Problems Breaking the weak PRF is syntactically equivalent to the ring-$\mathsf{LWR}_{R,p,2}$ problem, which is to distinguish between uniformly random pairs in $R_p \times R_2$, and pairs of the form $(a \leftarrow R_p, b = \lfloor a \cdot s \rceil_2)$ for some unknown $s \in R_p$. By scaling b up by a factor of $p/2$, we can equivalently interpret the latter pairs as ring-LWE pairs $(a, \frac{p}{2} \cdot b = a \cdot s + e) \in R_p \times R_p$, where $e \in R$ is the uniquely determined "small" error term with coefficients in $[-\frac{p}{4}, \frac{p}{4}) \cap \mathbb{Z}$ that makes $(a \cdot s + e)$ a multiple of $p/2$. Note that if $s \in R_p^*$ (i.e., it is a unit), then e is uniformly random in its domain, over the random choice of a. Therefore, the LWR problem can be modelled as LWE with uniformly random error of rate $1/2$, i.e., the range of the error term's coefficients covers half of $\mathbb{Z}_p$ (although unlike in LWE, the error is not independent of a).

Hardness of the Learning Problems

For relatively small values of n and p—quite a bit smaller than those typically required for *public-key* lattice cryptography—the ring-$\mathsf{LWR}_{R,p,2}$ problem appears to be secure against all attacks that are usually employed against lattice-based cryptography and related learning problems. This is primarily because the $1/2$ error rate is *much* larger than the inverse-polynomial (or smaller) rates required in public-key cryptography.

The main classes of attacks against noisy learning problems like LWR and LWE are: (1) brute-force attacks on the secret, (2) combinatorial attacks [BKW03; Wag02; MR09], (3) lattice attacks, and (4) algebraic attacks [AG11]. We consider each of these in turn.

Brute-force and combinatorial attacks. A brute-force attack on the weak PRF involves searching for the secret $s \in R_p$, or for the error terms in enough samples to uniquely determine s. The secret and rounding errors come from sets of size at least $(p/2)^n$, which is prohibitively large for all our parameters. Combinatorial (or "generalized birthday") attacks [BKW03; Wag02] work by drawing an exponential number of samples (a_i, b_i) and finding (via birthday collisions) a small combination of the a_i that sums to zero, then testing whether the same combination applied to the b_i is small, or noticeable non-uniform. This works for small error rates because the combination of the b_i is exactly the combination of their error terms. However, because our error terms are so large, even an optimally small combination does not yield a small value when applied to the b_i, nor is the value statistically biased in any way that is efficiently exploitable. Therefore, combinatorial attacks do not appear to work at all in this setting.

Lattice attacks. Lattice attacks on (ring-)LWE/LWR typically work by casting it as a bounded-distance decoding (BDD) problem on a lattice (see, e.g., [MR09; LP11; LN13; PS13b]). At a high level, the attack draws a sufficiently large number L of samples $(a_i, b_i) \in R_p \times R_p$, so that the secret (in the LWE case) is uniquely determined with good

probability. With error rate $1/2$, we need $L \geq \log(p/2)$ by a simple information-theoretic argument. The attack collects the samples into vectors $\vec{a}, \vec{b} \in R_p^L$, and considers the "$p$-ary" lattice $\mathcal{L}$ of dimension $N = nL$ (over $\mathbb{Z}$) corresponding to the set of vectors $s \cdot \vec{a} \in R_p^L$ for all $s \in R_p$. It then attempts to determine whether $\vec{b}$ is sufficiently close to $\mathcal{L}$, which corresponds to whether (a_i, b_i) are LWE samples or uniform. In our setting, because the error rate $1/2$ is so large, the distance from $\vec{b}$ to $\mathcal{L}$ (in the LWE case) is nearly the minimum distance of the lattice, up to a constant factor no larger than four (this is a conservative bound). Therefore, for the attack to succeed it needs to solve BDD (or the shortest vector problem SVP) on $\mathcal{L}$ to within an very small constant approximation factor. For the parameters in our instantiations, the lattice dimension is at least $N \geq n \log(p/2) \geq 256$ (and likely more). For this setting, the state of the art in BDD and SVP algorithms [CN11; LN13; MV10b], take time at least 2^{120}, and likely more. Moreover, the SVP algorithm of [MV10b], which appears to provide the best heuristic runtime in this setting, as a most conservative estimate requires space at least $2^{0.18N} \geq 2^{46}$.

Algebraic attacks. Finally, the algebraic "linearization" attack of Arora and Ge [AG11] yields a lower bound on p for security. The attack is applicable when every coefficient of every error term is guaranteed to belong to a known set of size d; in our setting, $d = p/2$. The attack requires at least N/n ring-LWE samples to set up and solve a dense linear system of dimension N, where

$$N = \binom{n+d}{n} \approx 2^{(n+d) \cdot H(n/(n+d))}$$

and $H(\delta) = -\delta \log(\delta) - (1-\delta)\log(1-\delta)$ is the binary entropy function for $\delta \in (0,1)$. Therefore, the attack requires time and space at least N^2, which is at least 2^{109} for all our parameters.

Table 6.2: Sequence of plaintext (PT) and ciphertext (CT) cyclotomic ring indices used for ring tunneling from $R = \mathcal{O}_{128}$ to $S = \mathcal{O}_{7,680}$.

PT index	CT index	CT dim
128	$128 \cdot 243$	10,368
$64 \cdot 17$	$64 \cdot 27 \cdot 17$	9,216
$16 \cdot 13 \cdot 17$	$16 \cdot 9 \cdot 13 \cdot 17$	9,216
$4 \cdot 5 \cdot 13 \cdot 17$	$4 \cdot 5 \cdot 7 \cdot 13 \cdot 17$	9,216
$3 \cdot 5 \cdot 13 \cdot 17$	$3 \cdot 5 \cdot 13 \cdot 17$	7,680

6.5.2 Security of Homomorphic Evaluation

We use what we believe to be quite conservative parameters in our SHE scheme, i.e., large dimensions for the noise rates in our SHE ciphertexts and key-switching hints. Following the methodology of [MR09] for estimating the security of LWE-based encryption, in order to break (ring-)LWE according to lattice attacks it is necessary to have $2^{2\sqrt{n \log Q \log \delta}} < Q$, where n is the dimension of the problem, Q is the largest modulus ever used, and $\delta \geq 1$ is the parameter that the lattice reduction algorithm can obtain. This means that breaking the SHE scheme at a minimum requires obtaining $\delta < 2^{\log(Q)/(4n)}$. Obtaining $\delta \leq 1.005$ is considered completely out of reach, offering at least 128 bits of security [CN11; LN13].

Concretely, we implement the weak PRF with $p = 32$, $n = 64$. The homomorphic rounding step switches from the 128th cyclotomic ring (corresponding to $n = 64$) to a ring S with cyclotomic index 7,680, which contains (at least) $n = 64$ $\mathbb{Z}_p$ slots. The ring switch for proceeds using a sequence of tunneling operations which moves the ring element through a series of hybrid cyclotomic rings; the full schedule is given in Table 6.2.

In order to support correct evaluation of the weak PRF, we use moduli Q no larger than 2^{152} in our ciphertexts and key-switch hints, with error terms having Gaussian coefficients with parameter at least 5 (times p, the plaintext modulus, but we do not use this factor in evaluating security). Except for the final ring of dimension 7,680 (in which the noise rate is

very large), the minimal ring dimension across all rings we use in the evaluation is 9,216. This means that breaking the scheme requires obtaining $\delta < 1.0035$, which is a very large security margin (see Table 7.3 for reference.)

6.6 ALCHEMY Implementation

In this section we describe our implementation of the BPR weak PRF using ALCHEMY.

6.6.1 Integer Rounding Circuit

Here we show how to implement the rounding function $\lfloor \cdot \rceil_2 : \mathbb{Z}_p \to \mathbb{Z}_2$ for $p = 2^k \geq 4$, using the arithmetization given in section 6.3. We note that it would be simple to define multiple arithmetizations (like those given in [GHS12a; AP13]) of the same operation by defining multiple expressions. We use this expression in the next subsection as a building block for the implementation of the BPR weak PRF.

The implementation given below works for any input modulus $p = 2^k$, though the arithmetization from [AP13] is more efficient for $p > 32$. For type safety, we must compute the *input* type from the output type (i.e., R_2) and k using the following type family:

```
type family PreRescalePTPow2 intp k r2 where
  PreRescalePTPow2 intp 1     r2 = r2
  PreRescalePTPow2 intp (k+1) r2 =
    PreMul intp (PreDiv2 intp (PreRescalePTPow2 intp k r2))
```

recalling from section 6.3 that each additional power of two adds another layer to the rounding tree, and computing the next layer (for a pair of inputs) involves a single multiplication followed by a division. This order is reversed above since we compute the type starting from the output. The Haskell compiler uses the type family to infer the object-language input type. of the the main interface to ring rounding:

```
rescalePTPow2 :: (Lambda intp, k > 1, ...)
```

```
  => Tagged k (expr e (PreRescalePTPow2 expr (k+1) r2 -> r2))
rescalePTPow2 = tag $ lam $
  let v'    = v0 *: (one >+: v0)
      kval  = proxy value (Proxy::Proxy k) :: Int
      pDiv4 = 2^(kval-2)
  in let_ v' $ treeMul (Proxy::Proxy k) $
       map ((div2_ $:) . (>+: v0)) $ take pDiv4 $
         [fromInteger $ y * (-y+1) | y <- [1..]]
```

The tag `k` is a positive natural number represting the power of two associated with $p = 2^k$ and is constrained to be at least 2, i.e., $p >= 4$. `rescalePTPow2` is a (tagged) DSL expression representing a function from `PreRescalePTPow2 expr k r2` to `r2`, where `r2` represents the integers mod 2 (or, in the next subsection, a cyclotomic ring with $\mathbb{Z}_2$ slots via the Chinese remainder theorem). The code above implements the optimization in the first level of the rounding tree explained at the end of section 6.3. We first compute $x \cdot (x+1)$ and *share* it using `let_`. We then add constant offsets of the form $i \cdot (-i+1)$ to create the first level of leaf nodes. These are passed to `treeMul`, which handles the main recursive algorithm.

We emphasize that this expression is *not* a language component, but rather a higher-level expression written in terms of existing language components. However, the programmer uses this expression like they would any other language component.

Below we give a small example which shows the implementation when $p = 4$:

```
-- expr :: (Lambda intp, AddLit intp (PreMul intp (PreDiv2 intp z2)),
--          Mul intp (PreDiv2 intp z2), Div2 intp z2, ...)
--  => intp e (PreMul intp (PreDiv2 intp z2) -> z2)
expr = untag $ rescalePTPow2 @2 -- set k=2 => p=4

pprint expr
-- "(\\v0 -> (div2 ((mul v0) (addLit (Scalar ZqB 1) v0))))"
```

Note that the type of `expr` (including the complex type of the input to the object-language function) is inferred by the Haskell compiler.

6.6.2 Ring Rounding

Our goal is to round the coefficients of an encrypted cyclotomic ring element. Recall from subsection 5.3.1 that the mth cyclotomic ring is isomorphic to $\mathbb{Z}[X]/(\Phi_m(X))$, so elements can be represented as a list of coefficients with respect to some fixed basis. For a cyclotomic ring R, moduli $p = 2^k$, ring rounding proceeds by first moving the $\mathbb{Z}_p$ coefficients of the input into "CRT slots" of a different ring S_p using ring switching. Once the coefficients are in slots, we can apply the $\mathbb{Z}_p$ rounding function `rescalePTPow2` to the entire ring element, which induces the operation on each coefficient. If desired, we can use ring switching again to move the rounded coefficients back to the ring $\mathbb{R}_q$. The exact number of ring switches needed to move the coefficients into slots depends on the particular choice of parameters. The following example implements a variant of $\lfloor \cdot \rceil_p$ that moves the coefficients of a ring `H0` into the slots of the ring `H2` via the intermediate ring `H1`. Since it does not switch back, the output ring element is in S_2:

```
roundCycCoeffs = do
  rescalePT <- rescalePTPow2 @(outputPNoise (Cyc t h5 z2))
  return $ rescalePT .:
   linearCyc_ (decToCRT @H1 @H2) .:
    linearCyc_ (decToCRT @H0 @H1)
```

Notice that `rescalePTPow2` is a higher-level DSL feature, but it is used in exactly the same way as any other DSL operation.

6.6.3 BPR PRF

All of the ALCHEMY expressions in this section so far are plaintext expressions, and do not use any details of the HE scheme. We now demonstrate a full-strength, real-world

example with ALCHEMY by implementing the BPR weak PRF [BPR12] and evaluating in *homomorphically* with concrete parameters. The core step uses the ring rounding expression from subsection 6.6.2. Recall the the BPR PRF is indexed by a secret key and maps an R_p element to a rounded product.

We remark that our implementation leaves the rounded coefficients in the CRT slots, which seems like the most useful option. For example, a symmetric ciphertext can be homomorphically decrypted by placing its bits in the slots and xoring with the encrypted bits. Then the plaintext data bits are in slots, which allows SIMD computations to be performed on them. At amy rate, it is not too much more work to tunnel back to R_2 after rounding in S.

First we must specify the concrete types for the cyclotomic rings and available moduli:

```
-- uses Factored types from Λ∘λ
-- plaintext rings
type H0  = 128
type H1  = 64 * 7
type H2  = 32 * 7 * 13
-- ciphertext rings
type H0' = H0 * 7 * 13
type H1' = H1 *F13
type H2' = H2

-- creates (ciphertext) moduli which are annotated with
-- their noise capacity
type Zq1 = Zq $(mkTLNatNat 1520064001)
type Zq2 = Zq $(mkTLNatNat 3144961)
type Zq3 = Zq $(mkTLNatNat 5241601)
```

```
-- the PRF output ring
type S2 = PNoise 0 (Cyc CT H2 (Zq 2))
```

The rings `H0` $= R$, `H1`, and `H2` $= S$ are the plaintext rings used for tunneling. The corresponding ciphertext rings are `H0'`, `H1'`, and `H2'`. Recall that we specify a collection of moduli from which the compiler automatically assigns valid moduli at each step of the computation. The PRF output ring is the cyclotomic index with index `H2` and over the integers mod two, augmented with `PNoise 0` to indicate that we don't need to do any further homomorphic operations.

Next we use these concrete types to instantiate the compiler and produce an expression which homomorphically evaluates the PRF:

```
-- takes a Haskell value s ∈ R_p
homomPRF s = do
  -- random value in R_p
  a <- getRandom
  -- get the ring rounding circuit for p = 2^5
  let round = proxy roundCycCoeffs (Proxy::Proxy 5)
  -- the in-the-clear PRF
      prf = lam $ round $: (mulPublic_ a v0)
  withKeys $ do
          -- homomorphic version of the PRF
          hprf <- pt2ct
      @[ (H0, H0'), (H1,H1'), (H2,H2') ] -- m'map
      @[ Zq1, Zq2, Zq3 ]                 -- zqs
      @TrivGad                           -- gadget from Lol
      (prf @S2)
         -- DSL expression for encryption of s
    sct <- encrypt s
```

```
-- apply the compiled PRF to an (encrypted) secret key
let prfeval = hprf $: sct
-- interpret the expression
(result,rates) <- eval <$> writeErrorRates prfeval
-- print the error rates
print rates
-- the PRF output
clearResult <- decrypt result
print clearResult
```

The call to `withKeys` creates an environment where the compiler *creates* keys, and `encrypt` and **`ErrorRateWriter`** use the same keys to encrypt inputs and collect runtime statistics, respectively. `homomPRF` prints the intermediate error rates from the homomorphic computation. `result` is the output of the homomorphic PRF evaluation, i.e. an encrypted PRF output, so we print its decryption, i.e. the in-the-clear PRF output.

6.7 ALCHEMY Evaluation

In this section we use the example of homomorphic PRF evaluation to quantify the advantages of using ALCHEMY. For the evaluation, we compute various metrics on the `homomPRF` expression given above, but using parameters corresponding to a cryptographically secure instantiation. Specifically, we instantiate `homomPRF` with $k = 5$ (corresponding to $p = 32$) and we use a sequence of five tunnels rather than two. These secure parameters allow us evaluate the savings that users of ALCHEMY are likely to obtain in the real-world.

The main goal of ALCHEMY is to reduce the complexity of writing homomorphic computations. We can measure these savings by calculating size of an alchemy expression, in terms of source lines of code and by counting the number of DSL expressions. Of course ALCHEMY will not be used if the compiled expression is much more inefficient than hand-

written homomorphic code for the same operation, so we also evaluate the performance of compiled expressions.

Expression Size. We can compare the size of the user-written plaintext expression with the corresponding homomorphic expression produced by the compiler. We measure expression size with the `S` interpreter, which counts the number of individual DSL operations that make up the epxression. Since the compiler introduces new operations, the difference in the size of the DSL expressions is a rough measure of the work done by the compiler, which corresponds to reduced complexity for the author of the plaintext expression.

The size of the in-the-clear `homomRoundCycCoeffs` expression is 39, while the size of the compiled expression increases to 87. Thus the homomorphic computation has about 48 more DSL operations than the plaintext computation, a considerable savings for the user.

Unfortunately, this measure both overstates and understates ALCHEMY's contribution. First, the compiler misses many opportunities for *beta reduction*, which corresponds to inlining certain function arguments. A fully beta-reduced expression would be much smaller (as measured with `S`) because we could elimiate many `lam` nodes in the expression. Thus the compiled expression size could have many fewer than 87 DSL operations. On the other hand, DSL expression size greatly understates the compiler's work because it does not account for the knowledge required for the user to manually insert the extra DSL expressions and to choose ciphertext moduli.

Comparison to Hand-written Applications. All existing HE implementations require users to write homomorphic computations using a low-level HE interface. Thus another way to measure ALCHEMY's contribution is to compare the number of source lines of code needed to hand-write a particular application using the HE interface directly with the total number of lines to write and compile the corresponding ALCHEMY expression on plaintexts.

In order to compare ALCHEMY, we used $\Lambda\circ\lambda$'s SHE interface to write code which computes the same function as `homomPRF`. This hand-written implementation uses the SHE

interface directly, meaning the author must understand all SHE operations and interfaces and manually choose appropriate parameters throughout the computation.

The hand-written implementation is about 225 lines of Haskell code, whereas our ALCHEMY implementation is about five lines of code for the (in-the-clear) implementation of `roundCycCoeffs`, three more for the PRF, and about five lines to invoke the **PT2CT** compiler and interpret the result (as in `homomPRF`). Thus ALCHEMY resulted in about 32x less code for *much* more functionality: the hand-written code can only be evaluated, while we can interpret the ALCHEMY expression in many interesting ways.

Runtime Performance. Finally, we compare the performance of the optimized hand-written computation with the compiled ALCHEMY expression. The runtime of the hand-written homomorphic computation is about 44 seconds, while the compiled ALCHEMY expression can be evaluated in 41 seconds. Thus ALCHEMY expressions incurs no runtime overhead compared to hand-tuned code, but are much easier to write and more flexible to use.

Homomorphic Encryption for Non-experts. We emphasize that although the above metrics show that ALCHEMY allows homomorphic computations to be expressed with moderately *less* code (with no performance loss), they do not capture how much *simpler* the plaintext expressions are compared to their homomorphic counterpart. Homomomorphic expressions in ALCHEMY can be written with *no* knowledge of the HE scheme, and compiled with only general knowledge. Concretely, compared to the ALCHEMY expression, the hand-written homomorphic computation required knowledge of where to place maintenance operations, explicit management of moduli at every step of the computation, and the manual generation of secret keys and key switch hints. Although this simplicity cannot be captured with simple numbers, we believe it is the most significant contribution of this work.

CHAPTER 7

CHALLENGES FOR RING-LWE

As lattice cryptography begins a transition to widespread deployment (see, e.g., [Ste14; LS16; Bra16b]), there is a pressing need for increased cryptanalytic effort and higher-confidence hardness estimates for its underlying computational problems. Of particular interest is a class of problems used in many recent implementations (e.g., [HS; GLP12; Duc+13; Bos+15; Alk+16; Bos+16a] and $\Lambda\circ\lambda$ [CP16b]), namely:

- Learning With Errors (LWE) [Reg09],
- its more efficient ring-based variant Ring-LWE [LPR13b], and
- their "deterministic error" counterparts Learning With Rounding (LWR) and Ring-LWR [BPR12].

Informally, the *search* version of the Ring-LWE problem is to find a secret ring element s given multiple random "noisy ring products" with s, while the *decision* version is to distinguish such noisy products from uniformly random ring elements. More precisely, Ring-LWE is actually a *family* of problems, with a concrete *instantiation* given by the following parameters:[1]

1. a *ring* R, which can often (but not always) be represented as a polynomial quotient ring $R = \mathbb{Z}[X]/(f(X))$ for some irreducible $f(X)$, e.g., $f(X) = X^{2^k} + 1$ or another cyclotomic polynomial;
2. a positive integer *modulus* q defining the quotient ring $R_q := R/qR = \mathbb{Z}_q[X]/(f(X))$;
3. an *error distribution* χ over R, which is typically concentrated on "short" elements (for an appropriate meaning of "short");

[1] This actually describes the "tweaked," discretized form of Ring-LWE, which for convenience avoids a special ideal denoted $R^\vee$. This form is equivalent to the original "untweaked" form under a suitable change to the error distribution; see subsection 2.2.7 for details.

4. a *number of samples* provided to the attacker.

The Ring-LWE search problem is to find a uniformly random secret $s \in R_q$, given independent samples of the form

$$(a_i\,,\, b_i = s \cdot a_i + e_i) \in R_q \times R_q,$$

where each $a_i \in R_q$ is uniformly random and each $e_i \leftarrow \chi$ is drawn from the error distribution. The decision problem is to distinguish samples of the above form from uniformly random samples over $R_q \times R_q$.

Ring-LWR is a "derandomized" variant of Ring-LWE in which the random errors are replaced by deterministic "rounding" to a smaller modulus $p < q$. Specifically, the search problem is to find a random secret $s \in R_q$ given independent samples

$$(a_i\,,\, b_i = \lfloor s \cdot a_i \rceil_p) \in R_q \times R_p,$$

where each $a_i \in R_q$ is uniformly random, and $\lfloor \cdot \rceil_p \colon R_q \to R_p$ denotes the function that rounds each coefficient $c_j \in \mathbb{Z}_q$ of the input (with respect to an appropriate basis) to $\lfloor \frac{p}{q} \cdot c_j \rceil \in \mathbb{Z}_p$. The decision problem is to distinguish such samples from $(a_i, \lfloor u_i \rceil_p)$, where $a_i, u_i \in R_q$ are uniformly random and independent. (Notice that $\lfloor u_i \rceil_p \in R_p$ itself is uniformly random when p divides q, but otherwise is biased.)

Hardness. A main attraction of Ring-LWE (and Ring-LWR) is their *worst-case hardness* theorems, also known as *worst-case to average-case reductions*. Essentially, these say that solving certain instantiations is at least as hard as quantumly solving a corresponding approximate Shortest Vector Problem (approx-SVP) on *any* "ideal lattice," i.e., a lattice corresponding to an ideal of the ring. (Interestingly, the converse is unclear: it is unknown how to solve Ring-LWE using an oracle for even exact-SVP on any ideal lattice of the ring.) See [LPR13b; PRS17] and [BPR12] for precise theorem statements, subsection 7.1.1 below

for further discussion, and [Cra+16; CDW17] for the status of approx-SVP on ideal lattices for quantum algorithms.[2]

As long as the underlying approx-SVP problem is actually hard in the worst case, the above-described theorems give strong evidence of cryptographic hardness, at least asymptotically (i.e., for large enough n). For practical purposes, though, the following property of (Ring-)LWE and related problems has been noticed, studied, and exploited for many years (see, e.g., [Lyu+08; MR09; Lyu09; LP11; Ban+14; HKM15]): even instantiations that are *not* supported by known worst-case hardness theorems, or that have too-small dimensions n to draw any meaningful conclusions from them, *can still appear very hard*—as measured against all known classes of attack. Indeed, almost every implementation of lattice cryptography to date has used considerably smaller dimensions and errors than what worst-case hardness theorems alone would recommend. However, care is needed in following this approach: e.g., some instantiations involving especially small errors turn out to be broken or seriously weakened by various attacks (see, e.g., [AG11; CLS15; Pei16]).

Given this state of affairs, and especially the common usage in practice of parameters that lack much (if any) theoretical support, we believe that a deeper understanding of how the different aspects of Ring-LWE affect concrete hardness is a critically important direction of research.

7.1 Contributions

This work provides a broad collection of cryptanalytic challenges for concrete instantiations of the search-Ring-LWE/LWR problems over *cyclotomic* rings, which are the most widely used and studied class of rings in this context. Our challenges cover a wide variety

[2] In brief: the fastest known quantum algorithms for the $\mathrm{poly}(n)$-approx-SVP problems underlying many cryptographic constructions, in any class of rings covered by the hardness theorems, perform essentially no better than algorithms for arbitrary lattices of the same dimension n, and take at least exponential $2^{\Omega(n)}$ time. Under plausible number-theoretic conjectures, $2^{O(\sqrt{n \log n})}$-approx-SVP is solvable in quantum polynomial time in certain rings, such as prime-power cyclotomics and their maximal totally real subrings [Cra+16; CDW17]; however, the main algorithmic technique used in these works meets a barrier at $2^{\Omega(\sqrt{n}/\log n)}$-factor approximations [Cra+16, Section 6].

of parameterizations and conjectured security levels, ranging from "toy" to "very hard" (see subsection 7.1.1 for details). We hope that these challenges will provide a focal point for theoretical and practical cryptanalytic effort on Ring-LWE/LWR, and will help to more precisely quantify the concrete security of their instantiations.[3]

A central issue in the creation of challenges for problems like (Ring-)LWE is that a dishonest challenger can publish instances that are much harder to solve than honestly generated ones—or even impossible. This is because (properly instantiated) Ring-LWE is conjectured to be pseudorandom, so it is difficult to distinguish between a correctly generated challenge and a harder one with much larger errors, or even a uniformly random one, which has no solution. A dishonest challenger could therefore publish unsolvable challenges, and point to the absence of breaks as bogus evidence of hardness.[4]

To deal with this issue, we design and implement a simple, non-interactive, and publicly verifiable "cut-and-choose" protocol that gives reasonably convincing evidence that the challenge instances are properly distributed, or at least not much harder than claimed. In short, for each Ring-LWE/LWR instantiation the challenger announces many timestamped instances. At a later time, the challenger reveals the secrets for all but a *random one* of the instances, as determined by a publicly verifiable source of randomness. (Concretely, we use the NIST randomness beacon [11].) Anyone can then verify that all the revealed instances look "proper," which makes it likely that the remaining instance is proper as well. Otherwise, the challenger would have had been caught with rather larger probability—assuming, of course, that it cannot predict or influence the randomness source. See section 7.2 for further details and discussion of some potential alternatives, which turn out *not* to give the kind

[3]The challenges and their parameters can be obtained via the Ring-LWE challenges website [16]. The archive `rlwe-challenges-v1.tar.gz` contains challenges for 516 different instantiations, and has a SHA-256 hash value `07cd f744 5c9d 178c 8b13 5a42 47ca a143 5320 c104 8ee8 c634 8914 a915 5757 dcef`. All our challenge-related archives are digitally signed under the PGP/GPG public key having ID `b8b2 45f5`, which has fingerprint `8126 1e02 fc1a 11c9 631a 65be b5b3 1682 b8b2 45f5`.

[4]This appears qualitatively different from problems like integer factorization and discrete logarithms, where deviating from the prescribed distributions seems like it can only make challenges *easier* to solve, or at least no harder.

of guarantees we desire. See subsection 7.1.2 for discussion of a recent approach to LWE challenges that aims for different goals.

Search versus decision. We stress that our challenges are for *search* versions of Ring-LWE/LWR, whereas many cryptographic applications rely on the conjectured hardness of solving *decision* with noticeable advantage. Unfortunately, it appears impractical to give meaningful challenges for the latter regime. This is because detecting a tiny advantage requires a very large number of instances, and a corresponding increase in effort by the attacker. And even for relatively large advantages, the naïve method of confirming the solutions would require the challenger to retain the correct answers and honestly compare them to the attacker's, because the attacker cannot confirm its own answers (unlike with the search problem, where it can).[5]

Nevertheless, we gain confidence in the usefulness of search challenges from the fact that the known classes of attack against decision either proceed by directly solving search, or can be adapted to do so with relatively little or no extra overhead. (See [LP11; LN13; Alk+16].) In addition, there are search-to-decision reductions [LPR13b, Section 5] which provide evidence that decision cannot be much easier than search (though the known reductions incur some as-yet unoptimized overhead). Finally, we note that practical constructions of, e.g., key exchange as in [Bos+16a] can use "hashed" variants, for which hardness of search can be sufficient for a reductionist security analysis in the random oracle model.

Implementation. Our free and open-source challenge generator and verifier are implemented using $\Lambda\circ\lambda$. We rely on its support for arbitrary cyclotomics and sampling from the theory-recommended Ring-LWE distributions that are needed for our instantiations (see subsection 7.1.1 for details). To encourage participation, we stress that all the challenge

[5] We considered more sophisticated non-interactive methods for confirming answers, like using a "fuzzy extractor" [Dod+08] to encrypt a secret that can only be recovered by solving a large enough fraction of decision challenges. Such methods seem tantalizing, but are complex to implement and bandwidth-intensive in our setting, so we leave this direction to future work.

data is formatted using Google's platform- and language-neutral *protocol buffers* (protobuf) framework [Goo08]. This allows the challenges to be read using most popular programming languages, via parsers that are automatically generated from our protobuf message specifications. The Ring-LWE challenges website [16] contains auto-generated parsers, and simple examples demonstrating their use, in C++, Java, Python, and Haskell. (The protobuf specifications can be found in [CP16a], and with the challenges themselves.) In addition, $\Lambda \circ \lambda$ includes C++ code for cyclotomic ring operations, which can be used by alternative implementations written in other languages.

7.1.1 Challenge Instantiations

Our challenge instantiations cover a wide range of parameters for several aspects of the Ring-LWE/LWR problems, including: size and form of the cyclotomic *index* and corresponding dimension; *width* of the error distribution; size and arithmetic form of the *modulus*; and number of *samples*. Each of these parameters has some degree of influence on the conjectured hardness of a Ring-LWE instantiation, as we discuss below.

For each challenge instantiation we give a qualitative hardness estimate, ranging from "toy" and "easy" to "very hard," along with an approximate block size that should allow the Block Korkin-Zolotarev (BKZ) basis-reduction algorithm to solve the instantiation. (See section 7.4.) We intentionally do *not* estimate concrete "bits of security" (though BKZ block size is a useful proxy), since any such estimates would necessarily be very imprecise. We hope that real-world efforts to break the challenges will provide more precision.

The easier categories represent instantiations that should be breakable using standard lattice algorithms on desktop-class machines in somewhere between a few minutes and a few months, whereas the hardest category should be out of reach even for nation-state adversaries—based on the current state of public cryptanalysis, at least. We deduce our hardness estimates by approximating the Hermite factors and BKZ block sizes needed to

solve the instantiations via lattice attacks, which usually represent the most practically efficient attacks against Ring-LWE/LWR. See section 7.4 for further details.

Cyclotomic Ring

A primary parameter influencing Ring-LWE's conjectured hardness is the *degree* (or dimension) of the ring R, which in the cyclotomic case is the totient $n = \varphi(m)$ of the *index* (or conductor) m. Thus far, most implementations have used *two-power* cyclotomic rings, because they have the computationally and analytically simplest form $R \cong \mathbb{Z}[X]/(X^n + 1)$, where n is a power of two. Moreover, sampling from a spherical Gaussian in their "canonical" geometry is equivalent to sampling independent identically distributed Gaussian coefficients for the powers of X.

We believe that Ring-LWE over non-two-power cyclotomics is deserving of more cryptanalytic effort. First, powers of two are rather sparse, especially in the relevant range of n in the several hundreds or more. In addition, two-power cyclotomics are incompatible with some advanced features of homomorphic encryption schemes, such as "plaintext packing" [SV14] and asymptotically efficient "bootstrapping" algorithms [GHS12a; AP13] for characteristic-two plaintext rings like $\mathbb{F}_{2^k}$. Finally, non-two-power cyclotomic rings lack orthogonal bases (in the canonical geometry), so sampling from recommended error distributions and error management are more subtle [LPR13a], and it is interesting to consider what effect (if any) this has on concrete hardness.

Our challenges are weighted toward the popular two-power case, but they also include indices of a variety of other forms, including powers of other small primes, those that are divisible by many small primes, and moderately large primes. We are particularly interested in whether there are any cryptanalytic attacks that can take special advantage of any of these forms. Our choices of indices m correspond to dimensions n ranging from 128 to $4{,}096$ for Ring-LWE, and from 16 to 162 for Ring-LWR.

Error Width

The *absolute* error of a (Ring-)LWE instantiation is, very informally, the "width" of the coefficients of the error distribution, with respect to an appropriate choice of basis. The main worst-case hardness theorems for (Ring-)LWE (e.g., [Reg09; Pei09; LPR13b]) apply to Gaussian-like error distributions whose widths exceed certain $\Omega(\sqrt{n})$ bounds. Conversely, there are algebraic attacks that can exploit significantly narrower errors, if enough samples are available (see, e.g., [AG11; Alb+14; EHL14; CLS15; CLS16; Pei16]). However, there is still a poorly understood gap between the theoretical bounds and parameters that plausibly fall to such attacks, especially in the low-sample regime (see Figure 7.1.1 below for further details).

Following the original definition and recommended usage of Ring-LWE [LPR13b; LPR13a], our challenge instantiations use the "dual" form involving the fractional ideal $R^\vee$ of the ring R, with *Gaussian* error that is *spherical* in the canonical embedding. More specifically, the products $s \cdot a_i$ reside in the quotient group $R^\vee/qR^\vee$, and we add error whose canonical embedding is distributed as a continuous Gaussian D_r of some parameter $r > 0$ (with optional discretization to $R^\vee$). In comparison to plain LWE, we emphasize that $R^\vee$ in the canonical embedding is a much denser lattice than $\mathbb{Z}^n$; in particular, errors drawn from D_r have (not necessarily independent) Gaussian coefficients of width $r\sqrt{n}$ with respect to the so-called "decoding" $\mathbb{Z}$-basis of $R^\vee$ [LPR13a]. (See Figure 7.1 and section 2.2 for further details.) Therefore, our parameterization is closely analogous to plain LWE with Gaussian error of parameter $r\sqrt{n}$.

Our challenge instantiations use four qualitative categories of error parameter r:

Trenta corresponds to a bound from the main "worst-case hardness of decision-Ring-LWE" theorem [LPR13b, Theorem 3.6], namely, $r \geq (n\ell/\ln(n\ell))^{1/4} \cdot \sqrt{\ln(2n/\varepsilon)/\pi}$, where ℓ is the number of revealed samples and (say) $\varepsilon \approx 2^{-80}$ is a bound on the

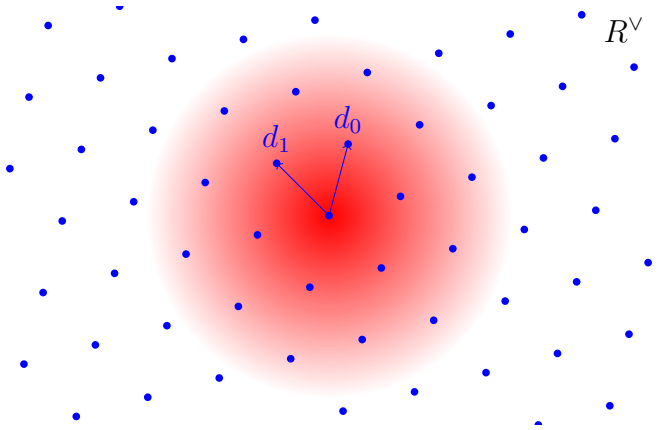

Figure 7.1: The canonical embedding of: (in dark blue) the dual ideal $R^\vee$ of the 3rd cyclotomic ring $R = \mathbb{Z}[\zeta_3]$, (in light blue) its "decoding" $\mathbb{Z}$-basis $\{d_0, d_1\}$, and (in red) the continuous spherical Gaussian D_r of parameter $r = \sqrt{2}$.

---

statistical distance in the reduction.[6] We pose this class of challenges to give some insight into instantiations that conform to the error bounds from known worst-case hardness theorems (though not necessarily for large enough dimensions n to obtain meaningful hardness guarantees via the reductions alone).

Grande corresponds to some $r \geq c = \Theta(1)$ (i.e., coefficients of width $c\sqrt{n}$) that satisfies the lower bound from Regev's worst-case hardness theorem [Reg09] for *plain* LWE, and that also suffices for provable immunity to the class of "ring homomorphism" attacks defined in [EHL14; Eli+15; CLS15; CLS16], as shown in [Pei16, Section 5]. We note that while the theorems from [Reg09] and [Pei16] are stated for $c = 2$, an inspection of the proofs and tighter analysis reveal that the constant can be improved to nearly $1/(2\sqrt{\pi}) \approx 0.282$ in the former case [Reg16], and to $c = \sqrt{8/(\pi e)} \approx 0.968$ or better in the latter case, depending on the dimension and desired time/advantage lower bound (see subsection 7.3.1 for details). We pose this class of challenges to give instantiations which *might* someday conform to significantly improved worst-case

[6] It is very likely that the bound can be improved by a small constant factor within the same proof framework; in addition, the $(n\ell/\ln(n\ell))^{1/4}$ factor might be an artifact of the proof. However, we use the bound as stated for our challenges.

hardness theorems for Ring-LWE, and which in any case satisfy the bounds from known hardness theorems in the absence of ring structure.

Tall corresponds to $r \in \{6, 9\}/\sqrt{n}$, i.e., error coefficients of width 6 or 9. Errors of roughly this size have been used in prior concrete analyses of LWE instantiations (e.g., [MR09; LP11]) and in practical implementations of (Ring-)LWE cryptography (e.g., [Alk+16; Bos+16a]).

Short corresponds to $r \in \{1, 2\}/\sqrt{n}$, i.e., error coefficients of width 1 or 2. In light of the above-mentioned small-error and homomorphism attacks, we consider such parameters to be riskier, at least when a large number of Ring-LWE samples are available. But at present it is unclear whether the attacks are feasible when only a small or moderate number of samples are available, as is the case in our challenges and in many applications (see Figure 7.1.1 below for further discussion).

Finally, for each setting of the error parameter we give challenges for both *continuous* error and its corresponding *discretized* version, where each real coefficient (with respect to the decoding basis) is rounded off to the nearest integer. Cryptographic applications almost always use discrete forms of Ring-LWE, but continuous forms are also cryptanalytically interesting. In particular, rounding yields a tight reduction from any continuous form to its corresponding discrete form, i.e., the latter is at least as hard as the former.

Modulus

Another main quantity that strongly influences Ring-LWE's apparent hardness is the *error rate*, which is, informally, the ratio of the (absolute) error width to the modulus q. There is much theoretical and practical cryptanalytic evidence that, all else being equal, Ring-LWE becomes harder as the error rate increases. E.g., there are tight reductions from smaller to larger rates; worst-case hardness theorems yield stronger conclusions for larger error rates; and lattice-based attacks perform worse in practice. Therefore, cryptographic applications

typically aim to use the smallest possible modulus that can accomodate the accumulated error terms without mod-q "wraparound" (so as to avoid, e.g., incorrect decryption). However, other considerations can introduce additional subtleties in the choice of modulus.

The initial worst-case hardness theorem for *search*-Ring-LWE [LPR13b, Theorem 4.1] applies to any sufficiently large modulus q and absolute error. However, the search-to-decision reduction [LPR13b, Theorems 5.1 and 5.2] requires q to be a prime integer that "splits well" in R, i.e., the ideal qR factors into distinct prime ideals of small norm.[7] Subsequent work [BV14a; Bra+13] used the "modulus switching" technique to obtain a reduction for essentially any modulus, at the cost of an increase in the error rate. Finally, recent work [PRS17] gave a worst-case hardness theorem for *decision*-Ring-LWE for *any* modulus, which either matches or improves upon the just-described results in terms of parameters. On the cryptanalytic side, the above-mentioned homomorphism attacks of [EHL14; Eli+15; CLS15; CLS16] can take advantage of moduli q for which the ideal qR has small-norm ideal divisors, but only when the error is insufficiently "well spread" relative to those ideals. (See [Pei16] for further details.)

With these considerations in mind, our challenge instantiations include moduli of a variety of sizes and arithmetic forms. We include moduli that split completely, others that split very poorly, and some that "ramify" (e.g., two-power moduli for two-power cyclotomics). Each instantiation uses a modulus that is large enough, relative to the absolute error, to yield correct decryption with high probability in public-key encryption and key-exchange protocols following the template from [LPR13b; Pei14]. See subsection 7.3.2 for further details.

Number of Samples

Finally, each of our challenge instantiations consist of either a small or moderate number of samples (specifically, three or 100) for Ring-LWE, and 500 samples for Ring-LWR.

[7] Such moduli also enable FFT-like algorithms over $\mathbb{Z}_q$, also called Chinese Remainder Transforms, which yield fast multiplication algorithms for R/qR using just $\mathbb{Z}_q$ operations.

These choices are motivated by the following considerations: while simple cryptographic constructions like key exchange and digital signatures reveal only a few samples (per fresh secret) to the adversary, other constructions like homomorphic encryption, identity/attribute-based encryption, and pseudorandom functions can reveal a much larger (possibly even adversary-determined) number of samples.

Clearly, revealing more samples cannot increase the hardness of an instantiation, because the attacker can just ignore some of them. There is also evidence that in certain parameter regimes, such as small bounded errors, increasing the number of samples can significantly reduce concrete hardness [AG11; Alb+14]. At the same time, the main worst-case hardness theorems for Ring-LWE place mild or no conditions at all on the number of samples [LPR13b, Theorem 3.6], and the same goes for plain LWE [Reg09; Pei09; Bra+13]. (Worst-case hardness theorems for less-standard LWE instantiations [MP13], and for (Ring-)LWR [BPR12; Alw+13; Bog+16; AA16], do have a strong dependence on the number of samples, however.) There are also standard techniques to generate fresh (Ring-)LWE samples from a fixed number of given ones, though at a cost in the error rate of the new samples [Lyu05; GPV08; App+09].

In summary, the practical effect of the number of samples on concrete hardness is unclear, and seems to depend heavily on the other parameters of the instantiation. Therefore, we separately consider both the small- and moderate-sample regime for our challenge instantiations.

7.1.2 Other Related Work

In a recent concurrent and independent work, Buchmann *et al.* [Buc+16] describe a method and implementation for creating challenges for LWE (but not Ring-LWE). Both their work and ours encounter a common issue—that naïve methods of generating challenges require knowing the solutions—but their main goal is quite different from ours: to prevent the solutions from existing in any one place, so that nobody is excluded from participating in

the subsequent cryptanalysis. They accomplish this by generating the challenges using a multi-party computation protocol, so that the solutions never reside with any single party. (Their implementation uses three parties, although this is not inherent to the approach.)

The protocol of [Buc+16] also allows for retroactively auditing that the parties honestly executed the protocol as implemented, *but only after a challenge has been solved*. This is a substantially weaker verifiability property than we obtain, for at least three reasons:

1. First, just half of the parties can undetectably create harder-than-expected or even unsolvable instances, which would never have the chance to be audited at all. To achieve the same end in our system, the challenger and the randomness beacon (e.g., NIST) would have to collude.
2. Second, auditing the MPC protocol requires the parties to retain their secret input seeds in perpetuity, and to reveal them when challenge solutions are found. If any of the seeds are lost, then so is verifiability. In our system, once the cut-and-choose protocol completes, the challenges are self-contained and verifiable with no external help.
3. Third, even if the parties do run the MPC protocol of [Buc+16] *as implemented*, one still needs to carefully audit the code to conclude that the resulting challenges actually have solutions. In fact, due to an bug, the first set of published challenges had no solutions! In our system, one does not need to trust or audit code, but only check that the "spoiled" instances have proper-looking errors.

Over the years there have been many analyses of various LWE parameterizations, in both the asymptotic and concrete settings, against various kinds of attacks, e.g., [MR09; LP11; AFG13; Alb+14; Alb+15; APS15; HKM15]. All of these apply equally well to Ring-LWE, which can be viewed as a specialized form of LWE, although they do not attempt to exploit the ring structure.

Cryptanalytic challenges have been provided for many other kinds of problems and cryptosystems, including integer factorization [91], discrete logarithm on elliptic curve groups [97], short-vector problems on ad-hoc distributions of ideal lattices [PS13a], the NTRU cryptosystem [15], and multivariate cryptosystems [Yas+15].

7.1.3 Organization

The remainder of the paper uses the background material from chapter 2 and is organized as follows:

Section 7.2 describes our non-interactive, publicly verifiable "cut-and-choose" protocol for giving evidence that the challenge instances are properly distributed.

Section 7.3 gives further details on how we choose our instantiations' parameters, specifically their Gaussian widths and moduli.

Section 7.4 describes how we obtain approximate hardness estimates for our challenge instantiations.

Section 7.5 gives some lower-level technical details about our implementation and the operational security measures we used while creating the challenges.

Acknowledgments. We thank Oded Regev for helpful discussions, and for initially suggesting the idea of publishing Ring-LWE challenges.

7.2 Cut-and-Choose Protocol

A central issue in the creation of challenges for LWE-like problems is that a dishonest challenger could publish improperly generated instances that are much harder than honestly generated ones, or even impossible to solve, because they have larger error than claimed or are even uniformly random. Because both the proper and improper distributions are conjectured to be pseudorandom, such misbehavior would be very difficult to detect. This

stands in contrast to other types of cryptographic challenges for, e.g., the factoring or discrete logarithm problems, where improper distributions like unbalanced factors or non-uniform exponents seem like they can only make the instances *easier* to solve (or at least no harder), so the challenger has no incentive to use them.

To deal with this issue, we use a simple, non-interactive, publicly verifiable "cut-and-choose" protocol to give reasonably convincing evidence that the challenge instances are properly distributed, or at least not much harder than claimed. The protocol uses a *timestamp service* and a *randomness beacon*. The former allows anyone to verify that a given piece of data was generated and submitted to the service before a certain point in time. The latter is a source of public, timestamped, truly random bits. Concretely, for timestamps we use the Bitcoin blockchain via the OriginStamp service [GB14], and for randomness we use the NIST beacon [11].

The use of a centralized beacon means that a verifier must trust that the challenger cannot predict or influence the beacon values, e.g., by collusion. This is obviously not entirely ideal from a security standpoint. Unfortunately, at the time we released our challenges we knew of no decentralized and practically usable alternatives that met our needs. For example, while the Bitcoin blockchain has been proposed and analyzed as a source of randomness, it turns out to be relatively easy and inexpensive to introduce significant bias [BCG15; PW16]. Similarly, the "unicorn" protocol [LW15] is trivial to bias completely, unless the time window for public contribution is smaller than the (fastest possible) computation time for a "slow" hash function, which is impractical for our purposes: we would need a large time window to ensure sufficient participation. Lastly, a proposal based on multi-national lotteries [Bai+15] does not come with a practically usable implementation, and requires the verifier to manually obtain past lottery numbers from many different countries.

7.2.1 Protocol Description and Properties

At a high level, our protocol proceeds as follows:

1. For each challenge instantiation (i.e., type of problem and concrete parameter set), the challenger *commits* by generating and publishing a moderately large number N (e.g., $N = 32$) of independent *instances*, along with a distinct *beacon address* indicating a time in the near future, e.g., a few days later. The challenger also *timestamps* the commitment.[8]

2. At the announced time, the challenger obtains from the beacon a random value $i \in \{0, \ldots, N-1\}$.

3. The challenger then publicly *reveals* the secrets (which also implicitly reveals the errors) underlying all the instances except for the ith one. The one unrevealed instance is then considered the "official" challenge instance for its instantiation, and the others are considered "spoiled."

4. Anyone who wishes to *verify* the challenge checks that:

   (a) the original commitment was timestamped sufficiently in advance of the beacon address (and all beacon addresses across multiple challenges are distinct);

   (b) secrets for the appropriate instances were revealed, as indicated by the beacon value; and

   (c) the revealed secrets appear "proper." For Ring-LWE, one checks that the errors are short enough, potentially along with other statistical tests, e.g., on the errors' covariance. For Ring-LWR one recomputes the rounded products with the revealed secret and compares them to the challenge instance.

Importantly, a verifier does not need to witness the challenger's initial commitment firsthand, because it can just check the timestamp. In addition, the beacon's random outputs are cryptographically signed, and can be downloaded and verified at any time, or even provided by the challenger in the reveal step (which is what our implementation does).

[8] All the challenger's public messages are cryptographically signed under a known public key. This is for the challenger's protection, so that other parties cannot publish bogus data in its name.

Under the reasonable assumptions that the challenger cannot backdate timestamps, nor predict or influence the output of the randomness beacon, the above protocol provides the following guarantee: if one or more of the instances in a particular challenge are "improper," i.e., they lack a secret that would convince the verifier, then the challenger has probability at most $1/N$ of convincing the verifier. (Moreover, if two or more of the instances are improper, then the challenger can never succeed.)

Potential cheats and countermeasures. It is important to notice that as described, the protocol does not prove that the instances were *correctly sampled* according to the claimed Ring-LWE distribution, only that the revealed errors satisfy the statistical tests (i.e., they are short enough, etc.). Below in subsection 7.2.2 we describe a supplementary (but platform- and implementation-specific) test, which we also include in our implementation, that gives a stronger assurance of correct sampling. However, the above protocol already seems adequate for practical purposes, because there does not appear to be any significant advantage to the challenger in choosing non-uniform $a_i \in R_q$ or $s \in R_q^\vee$, nor in deviating from spherical Gaussian errors within the required error bound. In particular, spherical Gaussians are rotationally invariant, and have maximal entropy over all distributions bounded by a given covariance.

Another way the challenger might try to cheat is a variant of the "perfect prediction" stock market scam: the challenger could prepare and timestamp a large number of different initial commitments (step 1) containing various invalid instances. The challenger's goal is for at least one of these commitments to be successfully revealable once the beacon values become available; the challenger would then publish only that (timestamped) commitment as the "official" one, and discard the rest. The more commitments it prepares in advance, the more invalid (but unrevealed) instances it can hope to sneak past the verifier. However, the number of commitments it must prepare grows exponentially with the number of invalid instances.

In order to rule out this kind of misbehavior, we prove that there is a *single* commitment by widely announcing it (or its hash value under a conjectured collision-resistant hash function) *before* the beacon values become available, in several venues where it would be hard or impossible to make multiple announcements or suppress them at a later time. For example, on the IACR ePrint archive we have created one dated submission for this paper, every version of which contains the same hash value of the commitment (in section 3). Also, we announced the hash value at the IACR Crypto 2016 Rump Session, which was streamed live on the Internet and is available for replay on YouTube.[9]

7.2.2 Alternative Protocols

Here we describe some potential alternative approaches for validating Ring-LWE challenges, and analyze their strengths and drawbacks.

Publishing PRG seeds. As noted above, revealing the secrets and errors does not actually prove that the instances were sampled from the claimed Ring-LWE distribution. To address this concern, the challenger could generate each instance *deterministically*, making its random choices using the output of a cryptographically secure pseudorandom generator (PRG) on a short truly random seed. Then to reveal an instance, the challenger would simply reveal the corresponding seed, which the verifier would use to regenerate the instance and check that it matches the original one. We caution that this method still does not *guarantee* that the instances are properly sampled, because the challenger could still introduce some bias by generating many instances and suppressing ones it does not like, or even choosing seeds maliciously. However, publishing PRG seeds seems to significantly constrain a dishonest challenger's options for misbehavior. (Using a public randomness beacon is not an option, because some of the PRG seeds must remain secret.)

There are a few significant practical drawbacks to this approach. First, establishing any reasonable level of assurance requires the verifier to understand and run the challenger-

[9]The announcement can be viewed at `https://youtu.be/FpdoPcThsU0?t=24m37s`.

provided code of the instance generator, rather than just checking that its outputs appear "proper," as the above protocol does. This also makes it difficult to write an alternative verification program (e.g., in a different programming language) without specifying exactly how the PRG output bits are consumed by the instance generator, which is cumbersome for continuous distributions like Gaussians. Second, even the provided verification code might be platform-specific: using different compiler versions or CPUs could result in different outputs on the same seed, due to differences in how the PRG output bits are consumed.[10]

Despite the above drawbacks, however, using and revealing PRG seeds does not need to *replace* the above protocol, but can instead *supplement* it to provide an extra layer of assurance. Therefore, our challenger and verifier also implement this method (and allow for very small $\leq 2^{-20}$ differences in floating-point values, to account for compiler differences). A failed match does not necessarily indicate misbehavior on the challenger's part, but is output as a warning by the verifier.

Zero-knowledge proofs. Another possibility is to view a Ring-LWE instance as a Bounded Distance Decoding (BDD) problem on a lattice, and have the challenger give a non-interactive zero-knowledge proof that it knows a solution within a given error bound. This can be done reasonably efficiently via, e.g., the public-coin protocol of [MV03] or Stern-style protocols for LWE-like problems [Lin+13], using a randomness beacon to provide the public coins. While at first glance this appears to provide exactly what we need, it turns out *not to give any useful guarantee*, due to the *approximation gap* between the completeness and soundness properties.

In more detail, for a BDD error bound B, an honest prover can always succeed in convincing the verifier that the error is at most B. However, the soundness guarantees only prevent a dishonest prover from succeeding when the BDD error is significantly larger

[10]We actually witnessed this phenomenon during development: different compilers yielded very small differences in the floating-point values of our continuous Ring-LWE instances, but not our discrete ones. We attribute this to the compilers producing different orders of instructions, and the non-associativity/commutativity of floating-point arithmetic.

than B. Specifically, the protocol from [MV03] has a bound of $\approx B\sqrt{d}$ where d is the lattice dimension, and the protocol from [Lin+13] only proves that the largest *coefficient* (in some basis) of the error is bounded. For our Gaussian error distributions, this bound would need to be about 2–3 times larger than the size of a typical coefficient. In summary, these protocols can only guarantee that the error is bounded by (say) $2B$, which can correspond to a much harder Ring-LWE instance than one with error bound B. By contrast, our protocol has a gap of only 10-15%, as shown next.

7.2.3 Verifier and Error Bounds

Here we describe our verifier in more detail, including some relevant aspects of its implementation, and describe how we compute rather sharp error bounds for our Ring-LWE instantiations.

Recall that each of our Ring-LWE instantiations is parameterized by a cyclotomic index m defining the mth cyclotomic number field K and cyclotomic ring R, which have degree $n = \varphi(m)$; a positive integer modulus q defining $R_q := R/qR$ and $R_q^\vee := R^\vee/qR^\vee$; and a Gaussian error parameter $r > 0$. (The number of samples is also a parameter, but it plays no role in the bounds.)

Verification. To verify a (continuous) Ring-LWE instance consisting of samples $(a \in R_q, b \in K/qR^\vee)$ for a purported secret $s \in R_q^\vee$ and given error bound B, one does the following for each sample:

1. compute $\bar{e} := b - s \cdot a \in K/qR^\vee$,
2. express $\bar{e}$ with respect to the decoding basis $\vec{d} = (d_j)$ of $R^\vee$, as $\bar{e} = \sum_j \bar{e}_j d_j$ where each $\bar{e}_j \in \mathbb{Q}/q\mathbb{Z}$.
3. "lift" $\bar{e} \in K/qR^\vee$ to a representative $e \in K$, defined as $e = \sum_j e_j d_j$ where each $e_j \in \mathbb{Q} \cap [-\frac{q}{2}, \frac{q}{2})$ is the distinguished representative of $\bar{e}_j$.

4. check that $\|e\| \leq B$ (where recall that $\|e\| := \|\sigma(e)\|$, the length of the canonical embedding of e).

For a discrete instance one does the same, but with K replaced by $R^\vee$ and $\mathbb{Q}$ replaced by $\mathbb{Z}$. In either case, properly generated Ring-LWE samples for our instantiations will correctly verify (with high probability) because the original errors $e \in K$ have coefficients of magnitude smaller than $q/2$ with respect to the decoding basis, hence they are correctly recovered from $b - s \cdot a = e \bmod qR^\vee$. Moreover, we show below that they have Euclidean norms below the error bound B with high probability.

Implementation. $\Lambda \circ \lambda$ (and hence the challenges themselves) actually uses the "tweaked" form of Ring-LWE as described in subsection 2.2.7, in which $R^\vee$ is replaced by R by implicitly multiplying each b component, and thereby the secret s and each error term e, by the "tweak" factor t (where $tR^\vee = R$). Correspondingly, the basis $t \cdot \vec{d}$ is referred to as the decoding basis of R. Therefore, we use an equivalent verification procedure to the one above, which simply replaces $R^\vee, \vec{d}$ with $R, t \cdot \vec{d}$, and the test $\|e\| \leq B$ with $\|g \cdot e\| \leq \hat{m}B$, where $g \in R$ is the special element such that $g \cdot t = \hat{m}$. (Recall that $\hat{m} = m/2$ when m is even, and $\hat{m} = m$ otherwise.)

The $\Lambda \circ \lambda$ framework provides operations for efficiently "lifting" elements of K/qR or R/qR to K or R (respectively) using the decoding basis of R, and for computing $\hat{m}^{-1} \cdot \|g \cdot e\|^2$ (see subsection 3.5.1). Thus our verifier actually checks the equivalent condition $\hat{m}^{-1} \cdot \|g \cdot e\|^2 \leq \hat{m}B^2$. For convenience, we also include the bound $\hat{m}B^2$ with the challenges, see [CP16a] for details.

Continuous error bound. For continuous Ring-LWE instantiations with spherical Gaussian error D_r over K, we use Lemma 2.1.1 and Corollary 2.1.2 to get rather sharp tail bounds on the Euclidean norm of the error. In our actual challenge instances, the error bound we use was typically within a factor of ≈ 1.10 of the largest error in each instance, so it gives little room for misbehavior relative to the correct error distribution.

The bound is obtained as follows. For an appropriate small $\varepsilon > 0$ we compute the minimal $c > 1/\sqrt{2\pi}$ (up to $\approx 10^{-4}$ precision) such that

$$\pi c^2 - \ln c \geq \frac{1}{n}\ln(1/\varepsilon) + \frac{1}{2}\ln(2\pi e).$$

Then by Corollary 2.1.2, we have $\Pr_{x\sim D_r}[\|x\| > B] < \varepsilon$, where $B := cr\sqrt{n}$. Concretely, we set $\varepsilon = 2^{-25}$ to get a rather strict bound that is still not too likely to be violated over the tens of thousands of error terms across all the instances.

Discrete error bound. For Ring-LWE instantiations with spherical Gaussian error D_r over K, discretized (i.e., rounded off) to $R^\vee$ using the decoding basis $\vec{d}$, we need to use a high-probability bound on the norm of the discretized error. For this we use a combination of Corollary 2.1.2 and a (partially heuristic) analysis of the round-off term. In our actual challenge instances, the ultimate bound was typically within a factor of ≈ 1.15 of the largest error in each instance.

Our discrete bound is obtained as follows. We first compute the same bound $B = cr\sqrt{n}$ on D_r as above. Now, because D_r is above or near the "smoothing parameter" of $R^\vee$, the fractional part $\mathbf{f} \in [-\frac{1}{2}, \frac{1}{2})^n$ of its coefficient vector with respect to $\vec{d}$ is close to uniformly random; henceforth we model it as such. The discretization error is $f = \langle \vec{d}, \mathbf{f} \rangle \in K$, which corresponds to $\mathbf{Df}$ in the canonical embedding, where $\mathbf{D} = \sigma(\vec{d}) = (\sigma_i(d_j))_{i,j}$. Observe that

$$\|f\|^2 = \langle \mathbf{Df}, \mathbf{Df} \rangle = \mathbf{f}^t \mathbf{G} \mathbf{f},$$

where $\mathbf{G} = \mathbf{D}^* \cdot \mathbf{D}$ is the positive definite Gram matrix of $\mathbf{D}$.

We now analyze the trace $\mathrm{Tr}(\mathbf{G})$, and use this to obtain a high-probability tail bound on $\|f\|$. Note that by definition of the decoding basis, $\mathbf{G} = \mathbf{H}^{-1}$ is the inverse of the Gram matrix $\mathbf{H}$ of the powerful basis $\vec{p}$. When m is a prime p, the proof of [LPR13a, Lemma 4.3] shows that $\mathbf{H} = p\mathbf{I}_{p-1} - \mathbf{1}$, so $\mathbf{G} = p^{-1}(\mathbf{I}_{p-1} + \mathbf{1})$, which has trace $\mathrm{Tr}(\mathbf{G}) = 2(p-1)/p =$

$2n/m$. By the tensorial decomposition of the powerful and decoding bases, this immediately generalizes for arbitrary m to

$$\mathrm{Tr}(\mathbf{G}) = \frac{2^k n}{m},$$

where k is the number of distinct primes dividing m.

Recalling that we model $\mathbf{f} \in [-\frac{1}{2}, \frac{1}{2})^n$ as uniformly random, by independence of f_i, f_j for $i \neq j$ and linearity of expectation we have

$$\mathop{\mathbb{E}}_{f}[\|f\|^2] = \mathop{\mathbb{E}}_{\mathbf{f}}[\mathbf{f}^t \mathbf{G} \mathbf{f}] = \frac{1}{12}\mathrm{Tr}(\mathbf{G}) = \frac{2^k n}{12m}.$$

We heuristically assume that $\sigma(f) = \mathbf{D}\mathbf{f}$ obeys essentially the same concentration bound (Lemma 2.1.1) as a spherical Gaussian having the above expected squared norm, times a small constant factor to account for the somewhat heavier tails (due to the non-spherical, non-Gaussian distribution). Our ultimate bound is $\sqrt{B^2 + F^2}$, where $B = cr\sqrt{n}$ and $F = c\sqrt{2^k n/m}$ are the high-probability bounds on the norms of D_r and the rounding term f, respectively.

7.3 Parameters

Here we give further details on how we choose the parameters of our instantiations, particularly the Gaussian error parameters r (subsection 7.3.1) and modulus q (subsection 7.3.2).

7.3.1 Error Parameter

As already mentioned in subsection 7.1.1, we consider four categories of parameter r for the Gaussian error distribution D_r over K: "Trenta," "Grande," "Tall," and "Short." For all categories except Grande, the descriptions in subsection 7.1.1 give the exact Gaussian parameter, or range of parameters, that we use in our instantiations.

For the Grande category, we use parameters that in particular have provable immunity to the "homomorphism" attack explored in [EHL14; Eli+15; CLS15; CLS16]. In [Pei16] it was shown that $r \geq 2$ is a sufficient condition for such immunity (in rings of cryptographically relevant dimensions). Here we generalize and tighten the analysis to obtain better bounds, which we use in our Grande instantiations.

The homomorphism attack on the original (non-"tweaked") definition of decision-Ring-LWE is as follows. (This is for the continuous form; it adapts immediately to the discrete form by replacing K with $R^\vee$.) Let ψ be an arbitrary error distribution over K, and let $\mathcal{I} \subseteq R$ be any ideal divisor of qR. We are given independent samples $(a_i, b_i) \in R_q \times K/qR^\vee$, which are distributed either uniformly or according to the Ring-LWE distribution for some secret $s \in R_q^\vee$. We first reduce the samples to

$$(a'_i = a_i \bmod \mathcal{I}\,,\, b'_i = b_i \bmod \mathcal{I}R^\vee) \in R/\mathcal{I} \times K/(\mathcal{I}R^\vee).$$

Then for each of the $\mathrm{N}(\mathcal{I})$ candidate (reduced) secrets $s' \in R^\vee/\mathcal{I}R^\vee$, we try to distinguish the $d'_i := b'_i - s' \cdot a'_i \in K/\mathcal{I}R^\vee$ from uniform. (How this is done does not matter for the present discussion.) Observe that if the samples come from the Ring-LWE distribution, i.e., $b_i = s \cdot a_i + e_i \bmod qR^\vee$ for $e_i \leftarrow \psi$, then for the correct candidate $s' = s \bmod \mathcal{I}R^\vee$ we have $d'_i = e_i \bmod \mathcal{I}R^\vee$.

Observe that the above attack takes time at least $\mathrm{N}(\mathcal{I})$ times the number of samples consumed, and that it can work *only if* the reduced error distribution $\psi \bmod \mathcal{I}R^\vee$ has noticeable statistical distance from uniform over $K/\mathcal{I}R^\vee$. Otherwise, the d'_i are statistically indistinguishable from uniform for any candidate s', regardless of the form of the original samples (uniform or Ring-LWE), and the attack fails.

Immunity to homomorphism attack. The following lemma gives a sufficient condition on the parameter of Gaussian error $\psi = D_r$ to ensure that the homomorphism attack has exponentially large time/advantage ratio t^n, for any desired $t > 1$. (Note that the proof never

uses the fact that $\mathcal{I}$ divides qR.) For simplicity, in our Grande instantiations we always use $t = 2$ and hence $r = \sqrt{8/(\pi e)} \approx 0.968$. For dimensions (say) $n > 256$ one could take $t = 2^{256/n}$ to obtain an even smaller r.

Lemma 7.3.1. *For any $n \geq 17$, $t > 1$, and $r \geq t\sqrt{2/(\pi e)} \approx 0.484t$, the time/advantage ratio of the homomorphism attack (for any choice of the ideal $\mathcal{I}$) is at least t^n.*

Proof. Let $s = \mathrm{N}(\mathcal{I})^{1/n}$, and note that the running time of the attack is at least $\mathrm{N}(\mathcal{I}) = s^n$, so we may assume without loss of generality that $s \leq t$.

The dual ideal of $\mathcal{I}R^\vee$ is $(\mathcal{I}R^\vee)^{-1} \cdot R^\vee = \mathcal{I}^{-1}$, which has norm $\mathrm{N}(\mathcal{I})^{-1}$, so by Lemma 2.2.1 its minimum distance is $\lambda_1(\mathcal{I}^{-1}) \geq \sqrt{n}/s$. Letting $f(x) = \sqrt{2\pi e} \cdot x \cdot \exp(\pi x^2)$ be as in Equation (2.1.1), define

$$\begin{aligned} c &:= \frac{r\lambda_1(\mathcal{I}^{-1})}{\sqrt{n}} \geq \frac{r}{s} \geq \frac{r}{t} \geq \sqrt{2/(\pi e)} > 1/\sqrt{2\pi}, \\ C &:= f(c) \leq 2\exp(-2/e) < 2^{-1/17}, \end{aligned}$$

where the penultimate inequality follows by $c \geq \sqrt{2/(\pi e)}$ and the fact that f is decreasing for $x \geq 1/\sqrt{2\pi}$.

By Lemma 2.1.3, the statistical distance between $D_r \bmod \mathcal{I}R^\vee$ and the uniform distribution over $K/\mathcal{I}R^\vee$ is at most $\frac{1}{2}C^n/(1 - C^n)$. Then because $n \geq 17$, the time/advantage ratio of the attack is

$$\frac{2(1 - C^n)\,\mathrm{N}(\mathcal{I})}{C^n} \geq \frac{\mathrm{N}(\mathcal{I})}{C^n} = (s/C)^n,$$

so it remains to show that $s/C \geq t$. By the previous observation on $f(x)$ and the fact that $c \geq r/s > 1/\sqrt{2\pi}$,

$$s/C = s/f(c) \geq s/f(r/s) = \frac{r}{\sqrt{2\pi e} \cdot (r/s)^2 \cdot \exp(-\pi(r/s)^2)}.$$

A straightforward calculation shows that the denominator (as a function of s) has a global maximum when $r/s = 1/\sqrt{\pi}$, so as desired, $s/C \geq r\sqrt{\pi e/2} \geq t$. □

7.3.2 Modulus

For a given Gaussian error parameter r, we choose moduli q to reflect a typical Ring-LWE public-key encryption or key-exchange application following the basic template from [LPR13b; Pei14]. Essentially, this means that q must be large enough to accomodate the ultimate error term, which is a combination of the original errors, without any "wraparound." A bit more precisely, we need that with sufficiently high probability, the ultimate error has coefficients (with respect to an appropriate choice of basis) in the interval $(-\frac{q}{4}, \frac{q}{4})$. The precise meaning of "high probability" depends on the low-level details of the application. For example, wraparound of a few coefficients might be acceptable if error-correcting codes are used, or a final key-confirmation step may handle the rare case when wraparound does occur.

The Ring-LWE "toolkit" [LPR13a] provides general techniques and reasonably sharp concentration bounds for analyzing the coefficients of sums and products of (discretized) error terms in arbitrary cyclotomics (see, e.g., [LPR13a, Lemma 6.6]). However, their generality makes them a bit pessimistic, so they do not capture the strongest possible concentration properties for concrete cases of interest.

In this work we take a combined empirical and theoretical approach to more tightly bound the ultimate error in encryption/key-exchange applications, and thereby obtain smaller values of the modulus and larger error rates. Our empirical approach is as follows:

1. We simulate thousands of ultimate error terms $E := \hat{m}(e \cdot e' + f \cdot f') \in R^\vee$, where $e, e', f, f' \in R^\vee$ are independent samples from D_r, discretized to $R^\vee$ using the decoding basis.[11]

2. We compute the largest magnitude B among all the coefficients of all the Es (again with respect to the decoding basis), and use $4B$ as a heuristic "very high probability" bound on the coefficients.

[11] Depending on the primes dividing the cyclotomic index m, replacing the $\hat{m}$ factor by t in the expression for E can sometimes yield smaller coefficients. We use the best of the two choices in our simulation.

3. Using $4B$ as a lower bound on $q/4$, we choose moduli q of different arithmetic forms (e.g., completely split, power of two, ramified) that all conform to this bound.

The theoretical (though heuristic) basis for this approach is as follows: in the canonical embedding, the coordinates of D_r are i.i.d. Gaussians over $\mathbb{C}$ (up to conjugate symmetry), and the same *nearly* holds for the discretization to $R^\vee$ when D_r is "well-spread" relative to $R^\vee$ (as it is in our instantiations). Because multiplication is coordinate-wise in the canonical embedding, the products $e \cdot e'$, $f \cdot f'$ have nearly i.i.d. subexponential coordinates. (The multiplication by $\hat{m}$ simply scales them all by the same factor.) Finally, each coefficient of E with respect to the decoding basis is by definition the inner product of $\sigma(E)$ with a vector consisting of various roots of unity. Bernstein's inequality says that such inner products have subgaussian $\exp(-\Theta(k^2))$ tail probabilities in the "near zone," which in our setting goes all the way out to $k = O(\sqrt{n})$ standard deviations. In the "far zone" beyond that, the tails are still subexponential $\exp(-\Theta(k))$.

Because the near zone is so wide, the largest coefficient among the tens or hundreds of thousands in our simulation should be not much smaller than a true high-probability bound. Concretely, the largest empirical coefficient B should have a tail probability of no more than, say, 2^{-13}. Under the subgaussian model, the probability of obtaining a coefficient of magnitude more than $4B$ is therefore less than $(2^{-13})^{4^2} = 2^{-208}$. Even under the weaker subexponential model, the probability is at most $(2^{-13})^4 = 2^{-52}$.

7.4 Hardness Estimates

In this section we describe how we obtain hardness estimates for our challenges. There are many different algorithmic approaches for attacking lattice problems like the approximate Shortest Vector Problem (SVP) and the Bounded Distance Decoding (BDD) problem, of which Ring-LWE/LWR are special cases. These include lattice-basis reduction (e.g., [LLL82; Sch87; GNR10; CN11; MW16]), exponential-time and -space sieving or Voronoi-based algorithms (e.g., [AKS01; NV08; MV10b; MV10a; Laa15; Agg+15]),

combinatorial and algebraic attacks [BKW03; AG11; Alb+14], and combinations thereof (e.g., [How07]).

Because all the above approaches represent active areas of research and can be difficult to compare directly—especially because some require enormous memory—we do not attempt to give precise estimates of "bits of security." Instead, we follow the analysis approach of [MR09; LP11; LN13; Alk+16] for (Ring-)LWE to derive two kinds of hardness estimates. First, we give the approximate *root-Hermite factor* $\delta > 1$ needed to solve each challenge via lattice attacks. We use δ to classify each challenge into one of a few broad categories, ranging from "toy" (very easy) to "very hard"(likely out of reach for nation-state attackers using the best publicly known algorithms). Second, we estimate the smallest *block size* that is sufficient to solve the challenge using the BKZ algorithm [SE94; CN11].

In figures 7.1 and 7.2, we give a sample of the hardness estimates for our Ring-LWE/LWR challenges, using the methods described below (specifically, Equations (7.4.1) and (7.4.2)). The estimates for the complete list of challenges can be found in [CP16a].

7.4.1 Ring-LWE/LWR as BDD

A standard attack on Ring-LWE casts it as a Bounded Distance Decoding (BDD) problem on a random lattice from a certain class. For a collection of ℓ Ring-LWE samples $(a_i \in R_q, b_i = s \cdot a_i + e_i \bmod qR^\vee)$ defining $\vec{a} = (a_1, \ldots, a_\ell)$, we consider the corresponding "q-ary" lattice

$$\mathcal{L}(\vec{a}) := \{\vec{v} \in (R^\vee)^\ell : \exists\, z \in R^\vee \text{ such that } \vec{v} = z \cdot \vec{a} \,(\bmod\ qR^\vee)\}.$$

The vector $\vec{b} = (b_1, \ldots, b_\ell) \approx s \cdot \vec{a} \bmod qR^\vee$ is then a BDD target that is close to an element of $\mathcal{L}(\vec{a})$, and the BDD error is $\vec{e} = (e_1, \ldots, e_\ell)$, where each e_i is distributed as the spherical Gaussian D_r.

Table 7.1: Hardness estimates for a selection of our *continuous* Ring-LWE challenges, in terms of approximate root-Hermite factors and smallest BKZ block size required to solve them: r' is the rescaled error parameter (subsection 7.4.1), δ is the root-Hermite factor (subsection 7.4.2), and κ is the GSA factor (subsection 7.4.3). Hardness estimates for our *discrete* Ring-LWE challenges (odd challenge IDs, with parameters identical to the preceding even challenge ID) are essentially the same, but may be slightly larger due to the extra round-off error.

					Hermite Factor		BKZ		
ID	m	$\varphi(m)$	r'	q	δ	Qualitative	κ	Dimension d	Block size
432	500	200	177.953	8,791,500	1.0104	easy	1.0098	343	89
434	500	200	383.329	37,996,001	1.0107	easy	1.0100	349	84
436	1,155	480	266.103	41,817,931	1.0048	very hard	1.0049	777	291
438	1,155	480	579.489	212,466,871	1.0050	very hard	1.0051	810	276
440	179	178	176.904	8,382,929	1.0116	toy	1.0108	325	71
442	179	178	176.904	8,388,608	1.0116	toy	1.0108	325	71
444	179	178	176.904	8,382,033	1.0116	toy	1.0108	325	71
446	179	178	380.444	37,250,617	1.0120	toy	1.0111	316	66
448	257	256	230.425	15,802,417	1.0083	moderate	1.0080	428	131
450	257	256	230.425	15,792,907	1.0083	moderate	1.0080	428	131
452	257	256	498.003	72,720,721	1.0086	moderate	1.0083	457	123
454	797	796	1,152.130	741,587,779	1.0030	very hard	1.0033	1,360	527

Table 7.2: Hardness estimates for a selection of our Ring-LWR challenges, in terms of approximate root-Hermite factors and smallest BKZ block size required to solve them: δ is the root-Hermite factor (subsection 7.4.2), and κ is the GSA factor (subsection 7.4.3).

					Hermite Factor		BKZ		
ID	m	$\varphi(m)$	q	p	δ	Qualitative	κ	Dimension d	Block size
456	32	16	97	2	1.0100	easy	1.0081	75	$\leq$ 30
457	32	16	32	2	1.0133	toy	1.0092	60	101
458	32	16	105	7	1.0299	toy	1.0124	33	$\leq$ 30
459	64	32	193	2	1.0043	very hard	1.0053	141	263
460	64	32	16	2	1.0083	moderate	1.0075	72	150
461	64	32	105	7	1.0148	toy	1.0108	82	71
462	128	64	257	2	1.0021	very hard	1.0034	250	497

The difficulty of BDD is primarily determined by the lattice dimension, and the width of the error relative to the (dimension-normalized) lattice determinant. Because $R^\vee$ is isomorphic as a group to $\mathbb{Z}^n$, we have that $\mathcal{L}(\vec{a})$ is an ℓn-dimensional lattice; however, by ignoring some coordinates we can view it as a d-dimensional lattice for any desired $d \in [n, \ell n]$. In order to most easily adapt the prior analyses for attacks on (Ring-)LWE, we also implicitly rescale the canonical embedding (thereby rescaling both the lattice and the error) by a factor of $\delta_R := \mathrm{vol}(\sigma(R))^{1/n}$, so that the rescaled $R^\vee$ has unit volume, just like $\mathbb{Z}^n$. The determinant of the lattice is then q^{d-n}—the same as for a d-dimensional LWE lattice—and the error is distributed as a spherical Gaussian of parameter $r' := \delta_R \cdot r$.

For Ring-LWR we proceed similarly, but because the rounding is done with respect the decoding basis of $R^\vee$—which in general is not orthogonal in the canonical embedding—we instead use the geometry given by identifying the decoding basis with the standard basis of $\mathbb{Z}^n$, and we model the rounding error in each coordinate as uniform in the interval $(-\frac{q}{2p}, \frac{q}{2p})$. This makes the rounding error isotropic and gives $R^\vee$ unit volume, and therefore yields the smallest ratio of error width to dimension-normalized determinant. Specifically,

the lattice determinant is again q^{d-n}, and the error has standard deviation $\frac{q}{p}/\sqrt{12}$ in each coordinate, so we heuristically model it as a spherical Gaussian with parameter $r' := \frac{q}{p}\sqrt{\pi/6}$.

7.4.2 Root-Hermite Factor

The quality of lattice vectors, and the concrete hardness of obtaining them, is often measured by the *Hermite factor*: for a d-dimensional lattice $\mathcal{L}$, vector $\mathbf{v} \in \mathcal{L}$ has Hermite factor δ^d given by $\|\mathbf{v}\| = \delta^d \cdot \mathrm{vol}(\mathcal{L})^{1/d}$; we call δ the *root-Hermite factor*. Experiments on random lattices indicate that δ is a very good indicator of hardness in cryptographically relevant dimensions. For example, $\delta \approx 1.022$ and $\delta \approx 1.011$ are efficiently obtainable by the LLL and BKZ-28 algorithms (respectively) [GN08], whereas $\delta = 1.005$ is considered far out of practical reach for $d \geq 500$ [CN11]. To our knowledge, the best publicly demonstrated root-Hermite factors for cryptographic dimensions are $\delta \approx 1.00955$ or more, on the Darmstadt lattice challenges [Lin+10].

Assuming that the error is sufficiently "smooth" over the integers, which is the case for all our challenges, the analyses of [MR09; LP11; LN13] show that one can solve LWE/BDD with some not-too-small probability by obtaining a root-Hermite factor δ given by

$$\lg \delta = \frac{\lg^2(Cq/r')}{4n \lg q}. \tag{7.4.1}$$

Here the factor C influences the success probability: larger values correspond to smaller chance of success. For example, extrapolating from [LN13, Table 2] for $n \leq 256$, taking $C \in [1.7, 2.5]$ can yield probability ≈ 1 (depending on the exact dimension); $C \approx 3.0$ corresponds to probability $\approx 2^{-32}$; and $C \approx 4.0$ corresponds to probability $\approx 2^{-64}$. (These are only rough estimates, and can be affected by the number of iterations, choice of pruning strategy, etc.) In our estimates, for simplicity we always use $C = 2.0$.

We use our root-Hermite factor estimates to classify each challenge into one of several qualitative hardness categories. The category thresholds are given in Table 7.3.

Table 7.3: Root-Hermite factor thresholds for our qualitative hardness estimates. Each challenge is classified according the largest applicable threshold (i.e., the weakest category.)

Class	$\delta >$
Toy	1.011
Easy	1.0095
Moderate	1.0075
Hard	1.005
Very Hard	1.0

7.4.3 BKZ Block Size

Another very good indication of hardness for a BDD instance is the smallest *block size* needed for the success of the BKZ lattice-basis reduction algorithm [SE94; CN11]. This parameter is a useful proxy for hardness because the runtime for BKZ is at least exponential in the block size.

Heuristic algorithms exist to approximate the runtime of BKZ [CN11; Che13], but they focus on the runtime of an SVP subroutine. This subroutine is called many times by the BKZ algorithm, but there are no precise estimates for the number of calls, and hence no very precise estimates for the total runtime of BKZ. Furthermore, the heuristic estimates are for sufficiently large block sizes in high dimensions, while some of our challenges have low dimension or can be attacked with a relatively small block size. Therefore, rather than provide an imprecise "bits of security" estimate, we instead give the approximate block size needed for the BKZ algorithm to successfully solve each challenge.

The "primal" form of the BKZ attack on LWE/BDD is most easily explained using Kannan's embedding technique, which converts a d-dimensional BDD instance with error $\vec{e}$

to a $(d+1)$-dimensional SVP instance with a "planted" shortest vector $(\vec{e}, 1)$.[12] When BKZ is run with a large enough block size b, it successfully finds the planted shortest vector. More specifically, by modeling the behavior of BKZ using the geometric series assumption (GSA) [Sch03], and assuming the error is Gaussian with parameter r', the analysis of [Alk+16] shows that the attack succeeds when

$$r'\sqrt{b/(2\pi)} \leq \kappa^{2b-d-1} \cdot q^{1-n/d}, \tag{7.4.2}$$

where $\kappa = ((\pi b)^{1/b} \cdot b/(2\pi e))^{1/(2b-2)}$ is the GSA factor. We optimize our choice of $d \in [n, \ell n]$ to minimize the block size needed for each challenge.

7.5 Implementation Notes

In this section we describe some of the lower-level technical details of our challenges, and the operational security measures we used when generating them.

Beacon addresses. Every 60 seconds the NIST randomness beacon [11] announces a 512-bit string, which is identified by the corresponding *(Unix) epoch*, i.e., the number of seconds elapsed since 1 January 1970 00:00:00 UTC. (The beacon epochs are always divisible by 60.) For our cut-and-choose protocol, a *beacon address* is a pair (s, i) consisting of an epoch s and a zero-indexed offset $i \in \{0, \ldots, 63 = 512/8 - 1\}$, which indexes the ith byte of the beacon's output string for epoch s.

Each of our challenges is associated with a distinct beacon address, which is used to determine which of its $N = 32$ instances will become the "official" one; the remainder will have their secrets revealed in the cut-and-choose protocol (see section 7.2 for details). A beacon address of (s, i) means that the official instance will be the one indexed by the ith

[12] Alkim *et al.* [Alk+16], found that by adjusting the parameters appropriately, the best "dual" attack required an almost identical block size as the primal attack, so we do not consider it here.

byte of the beacon value for epoch s, interpreted as an unsigned 8-bit integer and reduced modulo 32. That is, we use the least-significant 5 bits of the ith byte, and ignore the rest.

To ensure distinct beacon addresses, we generated our challenges to have sequentially increasing addresses starting from epoch $1{,}471{,}449{,}600$ (corresponding to 17 August 2016 12:00:00 EDT) and index zero. "Sequentially increasing" means that the index increments from 0 to 63, after which the epoch increments (by 60) and the index is reset to zero.[13]

Randomness. As the source of randomness for generating each instance of our challenges, we used the Haskell `DRBG` implementation [DuB15] of the NIST standard CTR-DRBG-AES-128 [BK15] pseudorandom generator, with a 256-bit seed ("input entropy"). The seeds themselves were derived using the Hash-DRBG-SHA-512 generator [BK15], seeded with 512 bits of system entropy. We would have preferred to use Hash-DRBG-SHA-512 for all pseudorandomness, but its implementation in `DRBG` is much slower, and pseudorandom bit generation is currently the main bottleneck in our implementation.

Operational security. A primary goal when generating our challenges and executing the cut-and-choose protocol was to reduce the risk of unauthorized exfiltration of the underlying secrets, e.g., by malware or hacking.

We generated the challenges on a 2010 MacBook Pro laptop with a freshly installed operating system, which was never connected to any network and had all network interfaces disabled. We exclusively used write-once CD and DVD media for copying the challenge-generator executable to the laptop, and the challenges and revealed secrets from the laptop.[14]

[13] Actually, there are two non-sequential "jumps" in the beacon addresses of our challenges, corresponding to batches we created with different runs of the generator. However, all beacon addresses are distinct across all our challenges.

[14] Because our executable requires compilers and external libraries to build, it was produced on a networked machine. It is conceivable, but seems highly unlikely, that the resulting executable could contain malicious code that manages to exfiltrate secrets via the external media when we export the challenges and revealed secrets. Unfortunately, this risk is inherent to our setup, because we must copy data from the laptop at some point.

We enabled FileVault encryption for the user account storage. As an extra layer of protection, we also created and stored the challenges and their secrets in a separately encrypted volume (within user storage), which was kept unmounted except when the challenges were being created or operated upon. The random passphrases for the user account and encrypted volume were generated and stored non-electronically, and were destroyed with fire once the cut-and-choose protocol was completed. Finally, we wiped the storage media with all-zeros. Therefore, we believe that the non-revealed secrets should be completely unrecoverable (even by us), except by solving the corresponding challenges.

REFERENCES

[11] *NIST randomness beacon*. http://www.nist.gov/itl/csd/ct/nist_beacon.cfm, last retrieved Aug 2016. Sept. 2011.

[15] *NTRU challenge*. https://www.securityinnovation.com/products/ntru-crypto/ntru-challenge, last retrieved Aug 2016. 2015.

[16] *Ring-LWE challenges website*. https://web.eecs.umich.edu/~cpeikert/rlwe-challenges. 2016.

[91] *RSA factoring challenge*. http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-factoring-challenge-faq.htm, last retrieved Aug 2016. Mar. 1991.

[97] *Certicom ECC challenge*. https://www.certicom.com/images/pdfs/challenge-2009.pdf, last retrieved Aug 2016. Nov. 1997.

[AA16] Jacob Alperin-Sheriff and Daniel Apon. *Dimension-Preserving Reductions from LWE to LWR*. Cryptology ePrint Archive, Report 2016/589. http://eprint.iacr.org/2016/589. 2016.

[AD97] Miklós Ajtai and Cynthia Dwork. "A Public-Key Cryptosystem with Worst-Case/Average-Case Equivalence". In: *STOC*. 1997, pp. 284–293.

[AFG13] Martin R. Albrecht, Robert Fitzpatrick, and Florian Göpfert. "On the Efficacy of Solving LWE by Reduction to Unique-SVP". In: *ICISC*. 2013, pp. 293–310.

[AG11] Sanjeev Arora and Rong Ge. "New Algorithms for Learning in Presence of Errors". In: *ICALP (1)*. 2011, pp. 403–415.

[Age15] National Security Agency. *Commercial National Security Algorithm Suite*. https://www.iad.gov/iad/programs/iad-initiatives/cnsa-suite.cfm. Blog. 2015.

[Agg+15] Divesh Aggarwal et al. "Solving the Shortest Vector Problem in 2^n Time Using Discrete Gaussian Sampling". In: *STOC*. 2015, pp. 733–742.

[Ajt04] Miklós Ajtai. "Generating Hard Instances of Lattice Problems". In: *Quaderni di Matematica* 13 (2004). Preliminary version in STOC 1996, pp. 1–32.

[AKS01] Miklós Ajtai, Ravi Kumar, and D. Sivakumar. “A sieve algorithm for the shortest lattice vector problem”. In: *STOC*. 2001, pp. 601–610.

[Alb+14] Martin R. Albrecht et al. *Algebraic Algorithms for LWE*. Cryptology ePrint Archive, Report 2014/1018. http://eprint.iacr.org/2014/1018. 2014.

[Alb+15] Martin R. Albrecht et al. “On the complexity of the BKW algorithm on LWE”. In: *Designs, Codes and Cryptography* 74.2 (2015), pp. 325–354.

[Alk+16] Erdem Alkim et al. “Post-quantum Key Exchange - A New Hope”. In: *USENIX Security Symposium*. 2016, pp. 327–343.

[Alw+13] Joël Alwen et al. “Learning with Rounding, Revisited - New Reduction, Properties and Applications”. In: *CRYPTO*. 2013, pp. 57–74.

[AP13] Jacob Alperin-Sheriff and Chris Peikert. “Practical Bootstrapping in Quasilinear Time”. In: *CRYPTO*. 2013, pp. 1–20.

[AP14] Jacob Alperin-Sheriff and Chris Peikert. “Faster Bootstrapping with Polynomial Error”. In: *CRYPTO*. 2014, pp. 297–314.

[App+09] Benny Applebaum et al. “Fast Cryptographic Primitives and Circular-Secure Encryption Based on Hard Learning Problems”. In: *CRYPTO*. 2009, pp. 595–618.

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. “On the concrete hardness of Learning with Errors”. In: *J. Mathematical Cryptology* 9.3 (2015), pp. 169–203.

[Bai+15] Thomas Baignères et al. *Trap Me If You Can – Million Dollar Curve*. Cryptology ePrint Archive, Report 2015/1249. http://eprint.iacr.org/2015/1249. 2015.

[Ban+14] Abhishek Banerjee et al. “SPRING: Fast Pseudorandom Functions from Rounded Ring Products”. In: *FSE*. 2014, pp. 38–57.

[Ban93] Wojciech Banaszczyk. “New bounds in some transference theorems in the geometry of numbers”. In: *Mathematische Annalen* 296.4 (1993), pp. 625–635.

[BCG15] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. *On Bitcoin as a public randomness source*. Cryptology ePrint Archive, Report 2015/1015. http://eprint.iacr.org/2015/1015. 2015.

[Ber+16] Daniel J. Bernstein et al. *NTRU Prime*. Cryptology ePrint Archive, Report 2016/461. http://eprint.iacr.org/2016/461. 2016.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) Fully Homomorphic Encryption without Bootstrapping". In: *TOCT* 6.3 (2014). Preliminary version in ITCS 2012, p. 13.

[BK15] Elaine Barker and John Kelsey. *Recommendation for random number generation using deterministic random bit generators*. NIST Special Publication 800-90A, revision 1. June 2015.

[BKW03] Avrim Blum, Adam Kalai, and Hal Wasserman. "Noise-tolerant learning, the parity problem, and the statistical query model". In: *J. ACM* 50.4 (2003), pp. 506–519.

[Bla14] Black Duck Software. *Ohcount*. `https://github.com/blackducksoftware/ohcount`, last retrieved May 2016. 2014.

[Bog+16] Andrej Bogdanov et al. "On the Hardness of Learning with Rounding over Small Modulus". In: *TCC*. 2016, pp. 209–224.

[Bon+13] Dan Boneh et al. "Key Homomorphic PRFs and Their Applications". In: *CRYPTO*. 2013, pp. 410–428.

[Bos+15] Joppe W. Bos et al. "Post-Quantum Key Exchange for the TLS Protocol from the Ring Learning with Errors Problem". In: *IEEE Symposium on Security and Privacy*. 2015, pp. 553–570.

[Bos+16a] Joppe W. Bos et al. "Frodo: Take off the Ring! Practical, Quantum-Secure Key Exchange from LWE". In: *CCS*. 2016, pp. 1006–1018.

[Bos+16b] Joppe Bos et al. *Frodo: Take off the ring! Practical, Quantum-Secure Key Exchange from LWE*. Cryptology ePrint Archive, Report 2016/659. `http://eprint.iacr.org/2016/659`. 2016.

[Bou+17] Charles Bouillaguet et al. "Fast Lattice-Based Encryption: Stretching SPRING". In: To appear in PQCrypto 2017. 2017.

[BP14] Abhishek Banerjee and Chris Peikert. "New and Improved Key-Homomorphic Pseudorandom Functions". In: *CRYPTO*. 2014, pp. 353–370.

[BPR12] Abhishek Banerjee, Chris Peikert, and Alon Rosen. "Pseudorandom Functions and Lattices". In: *EUROCRYPT*. 2012, pp. 719–737.

[Bra+13] Zvika Brakerski et al. "Classical hardness of learning with errors". In: *STOC*. 2013, pp. 575–584.

[Bra12] Zvika Brakerski. “Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP”. In: *CRYPTO*. 2012, pp. 868–886.

[Bra16a] Matt Braithwaite. *Experimenting with Post-Quantum Cryptography*. https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html. Blog. 2016.

[Bra16b] Matt Braithwaite. *Experimenting with Post-Quantum Cryptography*. https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html, last retrieved Aug 2016. July 2016.

[Buc+16] Johannes A. Buchmann et al. “Creating Cryptographic Challenges Using Multi-Party Computation: The LWE Challenge”. In: *AsiaPKC*. 2016, pp. 11–20.

[BV11a] Zvika Brakerski and Vinod Vaikuntanathan. “Efficient Fully Homomorphic Encryption from (Standard) LWE”. In: *FOCS*. 2011, pp. 97–106.

[BV11b] Zvika Brakerski and Vinod Vaikuntanathan. “Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages”. In: *CRYPTO*. 2011, pp. 505–524.

[BV14a] Zvika Brakerski and Vinod Vaikuntanathan. “Efficient Fully Homomorphic Encryption from (Standard) LWE”. In: *SIAM J. Comput.* 43.2 (2014). Preliminary version in FOCS 2011, pp. 831–871.

[BV14b] Zvika Brakerski and Vinod Vaikuntanathan. “Lattice-Based FHE as Secure as PKE”. In: *ITCS*. 2014, pp. 1–12.

[CDW17] Ronald Cramer, Léo Ducas, and Benjamin Wesolowski. “Short Stickelberger Class Relations and application to Ideal-SVP”. In: *EUROCRYPT*. To appear. 2017.

[Cha+11] Manuel M. T. Chakravarty et al. “Accelerating Haskell array codes with multicore GPUs”. In: *DAMP 2011*. 2011, pp. 3–14.

[Che+13] Jung Hee Cheon et al. “Batch Fully Homomorphic Encryption over the Integers”. In: *EUROCRYPT*. 2013, pp. 315–335.

[Che13] Yuanmi Chen. “Lattice Reduction and Concrete Security of Fully Homomorphic Encryption”. PhD thesis. Paris Diderot University, 2013.

[CIV16] Wouter Castryck, Ilia Iliashenko, and Frederik Vercauteren. “Provably Weak Instances of Ring-LWE Revisited”. In: *EUROCRYPT*. 2016, pp. 147–167.

[CKS09] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. “Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages”. In: *J. Funct. Program.* 19.5 (2009), pp. 509–543.

[CLP17] Hao Chen, Kim Laine, and Rachel Player. *Simple Encrypted Arithmetic Library - SEAL v2.1*. Cryptology ePrint Archive, Report 2017/224. `http://eprint.iacr.org/2017/224`. 2017.

[CLS15] Hao Chen, Kristin Lauter, and Katherine E. Stange. *Attacks on Search RLWE*. Cryptology ePrint Archive, Report 2015/971. `http://eprint.iacr.org/`. 2015.

[CLS16] Hao Chen, Kristin Lauter, and Katherine E. Stange. *Vulnerable Galois RLWE Families and Improved Attacks*. Cryptology ePrint Archive, Report 2016/193. `http://eprint.iacr.org/`. 2016.

[CN11] Yuanmi Chen and Phong Q. Nguyen. “BKZ 2.0: Better Lattice Security Estimates”. In: *ASIACRYPT*. 2011, pp. 1–20.

[CNT12] Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. “Public Key Compression and Modulus Switching for Fully Homomorphic Encryption over the Integers”. In: *EUROCRYPT*. 2012, pp. 446–464.

[Con09] Keith Conrad. *The Different Ideal*. Available at `http://www.math.uconn.edu/~kconrad/blurbs/`, last accessed 12 Oct 2009. 2009.

[Cor+11] Jean-Sébastien Coron et al. “Fully Homomorphic Encryption over the Integers with Shorter Public Keys”. In: *CRYPTO*. 2011, pp. 487–504.

[Cou+14] David Bruce Cousins et al. “An FPGA co-processor implementation of Homomorphic Encryption”. In: *HPEC 2014*. 2014, pp. 1–6.

[CP16a] Eric Crockett and Chris Peikert. *Challenges for Ring-LWE*. Cryptology ePrint Archive, Report 2016/782. `http://eprint.iacr.org/2016/782` and `https://web.eecs.umich.edu/~cpeikert/rlwe-challenges`. 2016.

[CP16b] Eric Crockett and Chris Peikert. “$\Lambda \circ \lambda$: Functional Lattice Cryptography”. In: *ACM CCS*. Full version at `http://eprint.iacr.org/2015/1134`. 2016, pp. 993–1005.

[Cra+16] Ronald Cramer et al. “Recovering Short Generators of Principal Ideals in Cyclotomic Rings”. In: *EUROCRYPT*. 2016, pp. 559–585.

[DH76] Whitfield Diffie and Martin E. Hellman. “New Directions in Cryptography”. In: *IEEE Transactions on Information Theory* IT-22.6 (Nov. 1976), pp. 644–654.

[Dij+10] Marten van Dijk et al. “Fully Homomorphic Encryption over the Integers”. In: *EUROCRYPT*. 2010, pp. 24–43.

[DM15] Léo Ducas and Daniele Micciancio. “FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second”. In: *EUROCRYPT*. 2015, pp. 617–640.

[DN12] Léo Ducas and Phong Q. Nguyen. “Faster Gaussian Lattice Sampling Using Lazy Floating-Point Arithmetic”. In: *ASIACRYPT*. 2012, pp. 415–432.

[Dod+08] Yevgeniy Dodis et al. “Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data”. In: *SIAM J. Comput.* 38.1 (2008). Preliminary version in EUROCRYPT 2004, pp. 97–139.

[DP15a] Léo Ducas and Thomas Prest. *A Hybrid Gaussian Sampler for Lattices over Rings*. Cryptology ePrint Archive, Report 2015/660. `http://eprint.iacr.org/`. 2015.

[DP15b] Léo Ducas and Thomas Prest. *Fast Fourier Orthogonalization*. Cryptology ePrint Archive, Report 2015/1014. `http://eprint.iacr.org/`. 2015.

[DuB15] Thomas DuBuisson. *DRBG Haskell package*. `https://hackage.haskell.org/package/DRBG`. Nov. 2015.

[Duc+13] Léo Ducas et al. “Lattice Signatures and Bimodal Gaussians”. In: *CRYPTO*. 2013, pp. 40–56.

[EHL14] Kirsten Eisenträger, Sean Hallgren, and Kristin E. Lauter. “Weak Instances of PLWE”. In: *SAC*. 2014, pp. 183–194.

[Eli+15] Yara Elias et al. “Provably Weak Instances of Ring-LWE”. In: *CRYPTO*. 2015, pp. 63–92.

[ES14] Richard A. Eisenberg and Jan Stolarek. “Promoting functions to type families in Haskell”. In: *Haskell 2014*. 2014, pp. 95–106.

[EW12] Richard A. Eisenberg and Stephanie Weirich. “Dependently typed programming with singletons”. In: *Haskell 2012*. 2012, pp. 117–130.

[GB14] André Gernandt and Bela Bipp. *OriginStamp*. `https://www.originstamp.org/`, last retrieved Aug 2016. 2014.

[Gen+12] Craig Gentry et al. “Ring Switching in BGV-Style Homomorphic Encryption”. In: *SCN*. Full version at `http://eprint.iacr.org/2012/240`. 2012, pp. 19–37.

[Gen+13] Craig Gentry et al. “Field switching in BGV-style homomorphic encryption”. In: *Journal of Computer Security* 21.5 (2013). Preliminary version in SCN 2012, pp. 663–684.

[Gen09a] Craig Gentry. “A fully homomorphic encryption scheme”. `http://crypto.stanford.edu/craig`. PhD thesis. Stanford University, 2009.

[Gen09b] Craig Gentry. “Fully homomorphic encryption using ideal lattices”. In: *STOC*. 2009, pp. 169–178.

[GGH97] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. “Public-Key Cryptosystems from Lattice Reduction Problems”. In: *CRYPTO*. 1997, pp. 112–131.

[GH11] Craig Gentry and Shai Halevi. “Implementing Gentry’s Fully-Homomorphic Encryption Scheme”. In: *EUROCRYPT*. 2011, pp. 129–148.

[GHS12a] Craig Gentry, Shai Halevi, and Nigel P. Smart. “Better Bootstrapping in Fully Homomorphic Encryption”. In: *Public Key Cryptography*. 2012, pp. 1–16.

[GHS12b] Craig Gentry, Shai Halevi, and Nigel P. Smart. “Fully Homomorphic Encryption with Polylog Overhead”. In: *EUROCRYPT*. 2012, pp. 465–482.

[GHS12c] Craig Gentry, Shai Halevi, and Nigel P. Smart. “Homomorphic Evaluation of the AES Circuit”. In: *CRYPTO*. 2012, pp. 850–867.

[GLP12] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. “Practical Lattice-Based Cryptography: A Signature Scheme for Embedded Systems”. In: *CHES*. 2012, pp. 530–547.

[GM84] Shafi Goldwasser and Silvio Micali. “Probabilistic encryption”. In: *Journal of Computer and System Sciences* 28.2 (1984), pp. 270–299.

[GN08] Nicolas Gama and Phong Q. Nguyen. “Predicting Lattice Reduction”. In: *EUROCRYPT*. 2008, pp. 31–51.

[GNR10] Nicolas Gama, Phong Q. Nguyen, and Oded Regev. “Lattice Enumeration Using Extreme Pruning”. In: *EUROCRYPT*. 2010, pp. 257–278.

[Goo08] Google. *Protocol Buffers (version 2)*. `https://developers.google.com/protocol-buffers/`, last retrieved Aug 2016. July 2008.

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. “Trapdoors for hard lattices and new cryptographic constructions”. In: *STOC*. 2008, pp. 197–206.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. “Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based”. In: *CRYPTO*. 2013, pp. 75–92.

[GVW13] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. “Attribute-based encryption for circuits”. In: *STOC*. 2013, pp. 545–554.

[Hen+10] Wilko Henecka et al. *TASTY: Tool for Automating Secure Two-partY computations*. Cryptology ePrint Archive, Report 2010/365. `http://eprint.iacr.org/2010/365`. 2010.

[HKM15] Gottfried Herold, Elena Kirshanova, and Alexander May. *On the Asymptotic Complexity of Solving LWE*. Cryptology ePrint Archive, Report 2015/1222. `http://eprint.iacr.org/2015/1222`. 2015.

[How07] Nick Howgrave-Graham. “A Hybrid Lattice-Reduction and Meet-in-the-Middle Attack Against NTRU”. In: *CRYPTO*. 2007, pp. 150–169.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. “NTRU: A Ring-Based Public Key Cryptosystem”. In: *ANTS*. 1998, pp. 267–288.

[HS] Shai Halevi and Victor Shoup. *HElib: an implementation of homomorphic encryption*. `https://github.com/shaih/HElib`, last retrieved August 2016.

[HS14] Shai Halevi and Victor Shoup. “Algorithms in HElib”. In: *CRYPTO*. 2014, pp. 554–571.

[HS15] Shai Halevi and Victor Shoup. “Bootstrapping for HElib”. In: *EUROCRYPT*. 2015, pp. 641–670.

[JZ12] D. Stefan J.C. Mitchell R. Sharma and J. Zimmerman. *Information-flow control for programming on encrypted data*. Cryptology ePrint Archive, Report 2012/205. `http://eprint.iacr.org/2012/205`. 2012.

[Kel+10] Gabriele Keller et al. “Regular, shape-polymorphic, parallel arrays in Haskell”. In: *ICFP 2010*. 2010, pp. 261–272.

[Kis10] Oleg Kiselyov. “Typed Tagless Final Interpreters”. In: *Generic and Indexed Programming - International Spring School, SSGIP 2010*. `http://okmij.org/ftp/tagless-final/`. 2010, pp. 130–174.

[KKS15] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. “Combinators for impure yet hygienic code generation”. In: *Sci. Comput. Program.* 112 (2015), pp. 120–144.

[KW11] Ueli Kunz and Julius Weder. *Metriculator*. https://github.com/ideadapt/metriculator. version 0.0.1.201310061341. 2011.

[Laa15] Thijs Laarhoven. "Sieving for Shortest Vectors in Lattices Using Angular Locality-Sensitive Hashing". In: *CRYPTO*. 2015, pp. 3–22.

[LCP17] Kim Laine, Hao Chen, and Rachel Player. *Simple Encrypted Arithmetic Library - SEAL v2.2*. Tech. rep. June 2017.

[Lin+10] Richard Lindner et al. *TU Darmstadt Lattice Challenge*. https://www.latticechallenge.org/. 2010.

[Lin+13] San Ling et al. "Improved Zero-Knowledge Proofs of Knowledge for the ISIS Problem, and Applications". In: *PKC*. 2013, pp. 107–124.

[Lip+12] Ben Lippmeier et al. "Guiding parallel array fusion with indexed types". In: *Haskell 2012*. 2012, pp. 25–36.

[Lip11] Miran Lipovača. *Learn You a Haskell for Great Good!* Available free online at http://learnyouahaskell.com/. No Starch Press, 2011.

[LLL82] Arjen K. Lenstra, Hendrik W. Lenstra Jr., and László Lovász. "Factoring polynomials with rational coefficients". In: *Mathematische Annalen* 261.4 (Dec. 1982), pp. 515–534.

[LN13] Mingjie Liu and Phong Q. Nguyen. "Solving BDD by Enumeration: An Update". In: *CT-RSA*. 2013, pp. 293–309.

[LP11] Richard Lindner and Chris Peikert. "Better Key Sizes (and Attacks) for LWE-Based Encryption". In: *CT-RSA*. 2011, pp. 319–339.

[LPR13a] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "A Toolkit for Ring-LWE Cryptography". In: *EUROCRYPT*. 2013, pp. 35–54.

[LPR13b] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On Ideal Lattices and Learning with Errors Over Rings". In: *Journal of the ACM* 60.6 (Nov. 2013). Preliminary version in Eurocrypt 2010, 43:1–43:35.

[LS16] Isis Lovecruft and Peter Schwabe. *RebelAlliance: A Post-Quantum Secure Hybrid Handshake Based on NewHope*. https://gitweb.torproject.org/user/isis/torspec.git/tree/proposals/XXX-newhope-hybrid-handshake.txt?h=draft/newhope, last retrieved Aug 2016. May 2016.

[LW15] Arjen K. Lenstra and Benjamin Wesolowski. *A random zoo: sloth, unicorn, and trx*. Cryptology ePrint Archive, Report 2015/366. `http://eprint.iacr.org/2015/366`. 2015.

[Lyu+08] Vadim Lyubashevsky et al. "SWIFFT: A Modest Proposal for FFT Hashing". In: *FSE*. 2008, pp. 54–72.

[Lyu05] Vadim Lyubashevsky. "The Parity Problem in the Presence of Noise, Decoding Random Linear Codes, and the Subset Sum Problem". In: *APPROX-RANDOM*. 2005, pp. 378–389.

[Lyu09] Vadim Lyubashevsky. "Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures". In: *ASIACRYPT*. 2009, pp. 598–616.

[May16] Christoph M. Mayer. *Implementing a Toolkit for Ring-LWE Based Cryptography in Arbitrary Cyclotomic Number Fields*. Cryptology ePrint Archive, Report 2016/049. `http://eprint.iacr.org/2016/049`. 2016.

[McC76] T. J. McCabe. "A Complexity Measure". In: *IEEE Trans. Softw. Eng.* 2.4 (July 1976), pp. 308–320.

[McE78] Robert J. McEliece. *A public-key cryptosystem based on algebraic coding theory*. DSN Progress Report 42-44. Jet Propulsion Laboratory, 1978, pp. 114–116.

[Mel+16] Carlos Aguilar Melchor et al. "NFLlib: NTT-Based Fast Lattice Library". In: *CT-RSA*. 2016, pp. 341–356.

[Mic07] Daniele Micciancio. "Generalized Compact Knapsacks, Cyclic Lattices, and Efficient One-Way Functions". In: *Computational Complexity* 16.4 (2007). Preliminary version in FOCS 2002, pp. 365–411.

[MP12] Daniele Micciancio and Chris Peikert. "Trapdoors for Lattices: Simpler, Tighter, Faster, Smaller". In: *EUROCRYPT*. 2012, pp. 700–718.

[MP13] Daniele Micciancio and Chris Peikert. "Hardness of SIS and LWE with Small Parameters". In: *CRYPTO*. 2013, pp. 21–39.

[MR07] Daniele Micciancio and Oded Regev. "Worst-Case to Average-Case Reductions Based on Gaussian Measures." In: *SIAM J. Comput.* 37.1 (2007). Preliminary version in FOCS 2004, pp. 267–302.

[MR09] Daniele Micciancio and Oded Regev. "Lattice-based Cryptography". In: *Post Quantum Cryptography*. Springer, Feb. 2009, pp. 147–191.

[Muk16] Tamalika Mukherjee. “Cyclotomic-Polynomials in Ring-LWE Homomorphic Encryption Schemes”. http://scholarworks.rit.edu/theses/9142/. MA thesis. Rochester Institute of Technology, June 2016.

[MV03] Daniele Micciancio and Salil P. Vadhan. “Statistical Zero-Knowledge Proofs with Efficient Provers: Lattice Problems and More”. In: *CRYPTO*. 2003, pp. 282–298.

[MV10a] Daniele Micciancio and Panagiotis Voulgaris. “A deterministic single exponential time algorithm for most lattice problems based on Voronoi cell computations”. In: *STOC*. 2010, pp. 351–358.

[MV10b] Daniele Micciancio and Panagiotis Voulgaris. “Faster Exponential Time Algorithms for the Shortest Vector Problem”. In: *SODA*. 2010, pp. 1468–1480.

[MW15] Pratyay Mukherjee and Daniel Wichs. *Two Round Multiparty Computation via Multi-Key FHE*. Cryptology ePrint Archive, Report 2015/345. http://eprint.iacr.org/2015/345. 2015.

[MW16] Daniele Micciancio and Michael Walter. “Practical, Predictable Lattice Basis Reduction”. In: *EUROCRYPT*. 2016, pp. 820–849.

[NLV11] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. “Can homomorphic encryption be practical?” In: *CCSW*. 2011, pp. 113–124.

[NV08] Phong Q. Nguyen and Thomas Vidick. “Sieve algorithms for the shortest vector problem are practical”. In: *J. Mathematical Cryptology* 2.2 (2008), pp. 181–207.

[OSu14] Bryan O’Sullivan. *Criterion*. https://hackage.haskell.org/package/criterion, version 1.1.1.0. 2014.

[Pei09] Chris Peikert. “Public-key cryptosystems from the worst-case shortest vector problem”. In: *STOC*. 2009, pp. 333–342.

[Pei14] Chris Peikert. “Lattice Cryptography for the Internet”. In: *PQCrypto*. 2014, pp. 197–219.

[Pei16] Chris Peikert. “How (Not) to Instantiate Ring-LWE”. In: *SCN*. 2016, pp. 411–430.

[PRS17] Chris Peikert, Oded Regev, and Noah Stephens-Davidowitz. “Pseudorandomness of Ring-LWE for Any Ring and Modulus”. In: *STOC*. To appear. 2017.

[PS13a] Thomas Plantard and Michael Schneider. *Creating a Challenge for Ideal Lattices*. Cryptology ePrint Archive, Report 2013/039. http://eprint.iacr.org/2013/039. 2013.

[PS13b] Joop van de Pol and Nigel P. Smart. *Estimating Key Sizes For High Dimensional Lattice Based Systems*. Cryptology ePrint Archive, Report 2013/630. http://eprint.iacr.org/. 2013.

[PW16] Cecile Pierrot and Benjamin Wesolowski. *Malleability of the blockchain's entropy*. Cryptology ePrint Archive, Report 2016/370. http://eprint.iacr.org/2016/370. 2016.

[Rab79] Michael O. Rabin. *Digitalized signatures and public-key functions as intractable as factorization*. Tech. rep. MIT/LCS/TR-212. MIT Laboratory for Computer Science, 1979.

[RAD78] Ronald L. Rivest, Len Adleman, and Michael L. Dertouzos. "On Data Banks and Privacy Homomorphisms". In: *Foundations of secure computation* 4.11 (1978), pp. 169–180.

[Reg09] Oded Regev. "On lattices, learning with errors, random linear codes, and cryptography". In: *J. ACM* 56.6 (2009). Preliminary version in STOC 2005, pp. 1–40.

[Reg16] Oded Regev. Personal communication. Mar. 2016.

[Ret17] Diego Retana. personal communication. Apr. 2017.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Commun. ACM* 21.2 (1978), pp. 120–126.

[Sch03] Claus-Peter Schnorr. "Lattice Reduction by Random Sampling and Birthday Methods". In: *STACS*. 2003, pp. 145–156.

[Sch16] Bruce Schneier. *Cryptography Is Harder Than It Looks*. https://www.schneier.com/blog/archives/2016/03/cryptography_is.html. Blog. 2016.

[Sch87] Claus-Peter Schnorr. "A Hierarchy of Polynomial Time Lattice Basis Reduction Algorithms". In: *Theor. Comput. Sci.* 53 (1987), pp. 201–224.

[SE94] Claus-Peter Schnorr and M. Euchner. "Lattice basis reduction: Improved practical algorithms and solving subset sum problems". In: *Mathmatical Programming* 66 (1994), pp. 181–199.

[Sho06] Victor Shoup. *A library for doing number theory*. http://www.shoup.net/ntl/, version 9.8.1. 2006.

[Sho97] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: *SIAM J. Comput.* 26.5 (Oct. 1997), pp. 1484–1509.

[SS11] Damien Stehlé and Ron Steinfeld. "Making NTRU as Secure as Worst-Case Problems over Ideal Lattices". In: *EUROCRYPT*. 2011, pp. 27–47.

[Ste14] Andreas Steffen. *strongSwan BLISS implementation*. https://wiki.strongswan.org/projects/strongswan/wiki/BLISS, updated Nov 2015, last retrieved Aug 2016. Dec. 2014.

[SV10] Nigel P. Smart and Frederik Vercauteren. "Fully Homomorphic Encryption with Relatively Small Key and Ciphertext Sizes". In: *Public Key Cryptography*. 2010, pp. 420–443.

[SV11] N.P. Smart and F. Vercauteren. *Fully Homomorphic SIMD Operations*. Cryptology ePrint Archive, Report 2011/133. http://eprint.iacr.org/. 2011.

[SV14] Nigel P. Smart and Frederik Vercauteren. "Fully homomorphic SIMD operations". In: *Designs, Codes and Cryptography* 71.1 (2014). Preliminary version in ePrint Report 2011/133, pp. 57–81.

[TTJ15] Dylan Thurston, Henning Thielemann, and Mikael Johansson. *Haskell numeric prelude*. https://hackage.haskell.org/package/numeric-prelude. 2015.

[Wag02] David Wagner. "A Generalized Birthday Problem". In: *CRYPTO*. 2002, pp. 288–303.

[Wan+12] Wei Wang et al. "Accelerating fully homomorphic encryption using GPU". In: *HPEC 2012*. 2012, pp. 1–5.

[WH12] David Wu and Jacob Haven. *Using Homomorphic Encryption for Large Scale Statistical Analysis*. Stanford CURIS 2012, http://www.stanford.edu/~dwu4/CURISPoster.pdf. 2012.

[Yas+15] Takanori Yasuda et al. *MQ Challenge: Hardness Evaluation of Solving Multivariate Quadratic Problems*. Cryptology ePrint Archive, Report 2015/275. http://eprint.iacr.org/2015/275. 2015.

[Yi+13] X. Yi et al. “Single-Database Private Information Retrieval from Fully Homomorphic Encryption”. In: *IEEE Transactions on Knowledge and Data Engineering* 25.5 (May 2013), pp. 1125–1134.

[Yor+12] Brent A. Yorgey et al. “Giving Haskell a promotion”. In: *TLDI 2012*. 2012, pp. 53–66.