$\Lambda \circ \lambda$: Functional Lattice Cryptography

Eric Crockett
University of Michigan

Chris Peikert*
University of Michigan

ABSTRACT

This work describes the design, implementation, and evaluation of $\Lambda \circ \lambda$, a general-purpose software framework for lattice-based cryptography. The $\Lambda \circ \lambda$ framework has several novel properties that distinguish it from 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 fully homomorphic encryption (FHE) 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 a collection of FHE operations 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.

*Supported by NSF CAREER Award CCF-1054495, DARPA FA8750-11-C-0096, the Sloan Foundation, and Google.

CCS'16, October 24 - 28, 2016, Vienna, Austria

ISBN 978-1-4503-4139-4/16/10…$15.00

DOI: http://dx.doi.org/10.1145/2976749.2978402

1. INTRODUCTION

Lattice-based cryptography has seen enormous growth over the past decade, and has attractive features like apparent resistance to *quantum* attacks; security under *worst-case* hardness assumptions (e.g., [1, 51]); efficiency and parallelism, especially via the use of algebraically structured lattices over polynomial *rings* (e.g., [36, 47, 42]); and powerful cryptographic constructions like identity/attribute-based and fully homomorphic encryption (e.g., [30, 27, 11, 31, 32]).

The past few years have seen a broad movement toward the *practical implementation* of lattice/ring-based schemes, with an impressive array of 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) [41], digital signatures [33, 20], key-establishment protocols [10, 2, 9], and fully homomorphic encryption (FHE) [49, 34] (using GPUs and FPGAs [56, 18]), 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 FHE functionality. (See Section 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 FHE.

1.1 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

[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 com-

$\Lambda\circ\lambda$ 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.

1.1.1 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, we implement a full-featured FHE scheme (which includes never-before-implemented functionality) in as few as 2–5 lines of code per feature. (See Sections 1.2 and 4 for details.)

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., [36, 47, 42, 43]). However, while almost all prior 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 [53, 54], 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 [42, 43] 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 [26, 16, 14], whereas theory-recommended distributions are provably immune to the same class of attacks [50].

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, $\Lambda\circ\lambda$ supports a set of homomorphic operations known as *ring-switching* [11, 28, 3], which enables efficient homomorphic evaluation of certain structured linear transforms. Ring-switching has multiple applications, such as ciphertext compression [11, 28] and asymptotically efficient "bootstrapping" algorithms for FHE [3].

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 FHE, 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.

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 [15], 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

munity's central repository, via `stack install lol`. The source repository is also available at https://github.com/cpeikert/Lol.

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

two interchangeable backends, one in the pure-Haskell Repa array library [37, 40], and one in C++.

1.1.2 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 [43] 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 the full version 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 the full version 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" [43] deals almost entirely with transforms and algorithms for a *single* cyclotomic ring, not inter-ring operations.)

FHE with ring tunneling. In Section 4, we define and implement an FHE scheme which refines a variety of techniques and features from a long series of works on Ring-LWE and FHE [42, 12, 13, 11, 28, 43, 3]. For example, it allows the plaintext and ciphertext spaces to be defined over different cyclotomic rings, which permits certain optimizations.

Using $\Lambda\circ\lambda$'s support for the cyclotomic hierarchy, we also devise and implement a more efficient variant of ring-switching for FHE, which we call *ring tunneling*. While a prior technique [3] homomorphically evaluates a linear function by "hopping" from one ring to another through a common *extension* ring, our new approach "tunnels" through a common *subring*, which makes it more efficient. 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.)

1.2 Example: FHE in $\Lambda\circ\lambda$

For illustration, here we briefly give a flavor of our FHE implementation in $\Lambda\circ\lambda$; see Figure 1 for representative code, and Section 4 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 $\Lambda\circ\lambda$'s interfaces (which include the functions `errorCoset`, `reduce`, `embed`, `twace`, `liftDec`, etc.). Indeed, the implementations of the FHE functions are often shorter than their type declarations! (For the reader who is new to Haskell, the full version 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 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.

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:

```
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 1: Representative (and approximate) code from our implementation of an FHE scheme in $\Lambda\circ\lambda$.

- 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., [42, Section 5]).

Discrete Gaussian sampling. Many lattice-based cryptosytems, such as digital signatures and identity-based or attribute-based encryption schemes following [30], 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), 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., [21, 43, 23, 22]), a complete practical solution still requires further research.

Applications. As our focus here is mainly on the $\Lambda\circ\lambda$ framework itself, we leave the implementation of additional lattice-based cryptosystems to 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 [42, 43] and NTRU-style encryption [36, 55], public-key encryption with security under chosen-ciphertext attacks [48], and pseudorandom functions (PRFs) [5, 8, 4]. It should also be possible to implement the *homomorphic evaluation* of a lattice-based PRF under our FHE scheme, in the same spirit as homomorphic evaluations of the AES block cipher [29, 17]; we have promising preliminary work in this direction.

Language layer. Rich lattice-based cryptosystems, especially homomorphic encryption, involve 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. For future work, we envision domain-specific languages (DSLs) that allow the programmer to express a *plaintext computation* at a level above the "native instruction set" of the homomorphic encryption scheme. A specialized compiler would then translate the user's description into a *homomorphic computation* (on ciphertexts) using the cryptosystem's instruction set, and possibly even instantiate secure parameters for it. Because Haskell is an excellent host language for embedded DSLs, we believe that $\Lambda\circ\lambda$ will serve as a strong foundation for such tools.

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 [36, 6], the SWIFFT hash function [41], digital signature schemes like [33] and BLISS [20], key-exchange protocols [10, 2, 9], and FHE libraries like HElib [34]. In addition, there are some high-performance backends for power-of-two cyclotomics, like NFLlib [46] and [56], which can potentially be plugged into these other systems. Also, in a Masters thesis developed concurrently with this work, Mayer [44] implemented the "toolkit" algorithms from [43] 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 [44], they do not support general cyclotomic rings using the current best theory for cryptographic purposes.

HElib. Our work compares most closely to HElib [34], which is an "assembly language" for BGV-style FHE over cyclotomic rings [11]. It holds speed records for a variety of FHE benchmarks (e.g., homomorphic AES computation [29]), and appears to be the sole public implementation of many advanced FHE features, like bootstrapping for "packed" ciphertexts [35].

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 [43], 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 (Section 5.2), 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 (FHE), 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.

1.5 Architecture and Paper Organization

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

Integer layer (Section 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 (full version): 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 [43]), and permits multiple implementations. Because the tensor layer is completely hidden from typical cryptographic applications, we defer to the full versionthe details of its design and our implementations. This material includes the definitions and analysis of several linear transforms and algorithms that, to our knowledge, have not previously appeared in the literature. Additionally, the full version describes the "sparse decomposition" DSL and compiler that underlie our pure-Haskell `Tensor` implementation.

Cyclotomic layer (Section 3): 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 (Section 4): This layer consists of implementations of cryptographic schemes. As a detailed example, we define an advanced FHE scheme that incorporates and refines a wide collection of features from a long series of works [42, 12, 13, 11, 29, 28, 43, 3]. We also show how its implementation in $\Lambda\circ\lambda$ very closely and concisely matches its mathematical definition.

Finally, in Section 5 we evaluate $\Lambda\circ\lambda$ in terms of code quality and runtime performance, and give a comparison to HElib [34].

Due to space constraints, many technical details are omitted from this extended abstract. The full, most recent version of this work is [19].

2. INTEGER & MODULAR ARITHMETIC

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, i.e., the cosets $x + q\mathbb{Z}$ with the usual addition and multiplication operations. 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.

In this section we summarize how we represent and implement these domains and operations in $\Lambda\circ\lambda$. This provides a foundation for the next section, where we show in particular how all these operations are generically "promoted" from $\mathbb{Z}$ and $\mathbb{Z}_q$ to cyclotomic rings, to support ring-based cryptosystems.

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

We exclusively use fixed-precision primitive types like `Int` and `Int64` to represent the integers $\mathbb{Z}$, and define our own specialized type `ZqBasic q z` to represent $\mathbb{Z}_q$. Here `q` 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 making `ZqBasic q z` 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 that have identical runtime representations, with no memory overhead.

```
reduce    :: Reduce    z  zq  => z  ->  zq
lift      :: Lift      zq z   => zq ->  z
rescale   :: Rescale   zq zq' => zq ->  zq'
gadget    :: Gadget    zq     =>       [zq]
encode    :: Gadget    zq     => zq -> [zq]
decompose :: Decompose zq z   => zq -> [z]
```

Figure 2: Methods defined by the classes (interfaces) `Reduce`, `Lift`, etc., with slightly simplified types.

CRT/RNS representation. Applications like homomorphic encryption can require moduli q that are too large for standard fixed-precision integer types. Using Haskell's unbounded `Integer` is rather slow, and the values have varying sizes, which means they cannot be stored efficiently in "unboxed" form in arrays. A standard solution is to use a Chinese Remainder Theorem (CRT), or 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/RNS representation is very natural via 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)`. Products of more than two rings are immediately supported by nesting pairs, like `((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.

2.2 Specialized Operations

For the specialized operations used across lattice cryptography, $\Lambda\circ\lambda$ defines the interfaces (called *classes*) `Reduce`, `Lift`, `Rescale`, `Gadget`, `Decompose`, and `Correct`, along with appropriate implementations (called *instances*). The methods defined by these classes are shown in Figure 2.

For the present discussion, one can read the type arguments `z` and `zq` in Figure 2 as respectively representing $\mathbb{Z}$ and $\mathbb{Z}_q$ for some modulus q. The `reduce` method represents modular reduction from $\mathbb{Z}$ to $\mathbb{Z}_q$, while the `lift` method represents lifting from $\mathbb{Z}_q$ to the set of distinguished representatives $\mathbb{Z} \cap [-\frac{q}{2}, \frac{q}{2})$. The `rescale` method represents the modular "rounding" function $\lfloor\cdot\rceil_{q'} : \mathbb{Z}_q \to \mathbb{Z}_{q'}$, defined as

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

where $\lfloor\cdot\rceil$ denotes rounding to the nearest integer.

The `gadget`, `encode`, and `decompose` methods respectively represent some "gadget" vector over $\mathbb{Z}_q$ (e.g., the powers-of-two vector $\mathbf{g} = (1, 2, 4, 8, \ldots, 2^{\ell-1})$) and efficient algorithms for the following standard tasks:

1. *Decomposition*: given $u \in \mathbb{Z}_q$, output a *short* vector $\mathbf{x}$ over $\mathbb{Z}$ 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 $s \in \mathbb{Z}_q$ and $\mathbf{e}$ is a sufficiently short error vector over $\mathbb{Z}$, output s and $\mathbf{e}$.

2.3 Type-Level Cyclotomic Indices

As discussed in Section 3 below, there is one cyclotomic ring for every positive integer m, which we call the *index*. 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 [57] and the *singletons* library [25, 24]. 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.

3. CYCLOTOMIC RINGS

In this section we summarize $\Lambda\circ\lambda$'s interfaces and implementations for cyclotomic rings. In Section 3.1 we review the relevant mathematical background. In Section 3.2 we describe the interfaces of the two data types, `Cyc` and `UCyc`, that represent cyclotomic rings. Then in Section 3.3 we describe some key aspects of the implementations.

3.1 Mathematical Background

To appreciate the material in this section, one only needs the following high-level background; see the full version and [42, 43] for many more mathematical and computational details.

3.1.1 Cyclotomic Rings

For a positive integer m, the mth *cyclotomic ring* is $R = \mathbb{Z}[\zeta_m]$, the ring extension of the integers $\mathbb{Z}$ obtained by adjoining an element ζ_m having multiplicative order m. The ring R is contained in the mth *cyclotomic number field* $K = \mathbb{Q}(\zeta_m)$, which both have degree $\deg(K/\mathbb{Q}) = \deg(R/\mathbb{Z}) = n$. We endow K, and thereby R, with a geometry via a function $\sigma : K \to \mathbb{C}^n$ called the *canonical embedding*. E.g., we define the ℓ_2 norm on K as $\|x\|_2 = \|\sigma(x)\|_2$, and define Gaussian-like distributions over R and K. The complex coordinates of σ come in conjugate pairs, and addition and multiplication are coordinate-wise under σ.

For cryptographic purposes, there are two particularly important $\mathbb{Z}$-bases of R: the *powerful* basis $\vec{p}_m \in R^n$ and the *decoding* basis $\vec{d}_m \in R^n$. That is, every $r \in R$ can be uniquely represented as $r = \langle \vec{p}_m, \mathbf{r} \rangle$ for some integral vector $\mathbf{r} \in \mathbb{Z}^n$, and similarly for $\vec{d}_m$. In particular, there are $\mathbb{Z}$-linear transformations that switch from one representation to the other. For geometric reasons, certain cryptographic operations are best defined in terms of a particular one of

these bases, e.g., decryption uses the decoding basis, whereas decomposition uses the powerful basis.

There are special ring elements $g_m, t_m \in R$ whose product is $g_m \cdot t_m = \hat{m}$, which is defined as $\hat{m} := m/2$ when m is even, and $\hat{m} := m$ otherwise. The elements g_m, t_m are used in the generation and management of error terms in cryptographic applications, as described below.

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 $\zeta_{m'}^{m'/m}$. In the reverse direction, we can *twace* from R' to R, which is a certain R-linear function that fixes R pointwise. (The name is short for "tweaked trace," because the function is a variant of the true *trace* function to our "tweaked" setting, described next.)

3.1.2 (Tweaked) Ring-LWE and Error Distributions

Ring-LWE is a family of computational problems that was defined and analyzed in [42, 43]. Those works deal with a form of Ring-LWE that involves a special (fractional) ideal $R^\vee$, which is "dual" to R, and an error distribution ψ that corresponds to a *spherical* Gaussian D_r of some parameter r (e.g., $r = 2$ or $r \approx n^{1/4}$) in the canonical embedding. Such spherical distributions play an important role in the worst-case hardness proofs for Ring-LWE, and also behave very well in cryptographic applications.

In cryptosystems, it can be convenient to use a form of Ring-LWE that does not involve $R^\vee$. As first suggested in [3], this can be done by working with an equivalent "tweaked" form of the problem, which is obtained by multiplying Ring-LWE samples by a "tweak" factor $t = t_m = \hat{m}/g_m \in R$, which satisfies $t \cdot R^\vee = R$. This yields "noisy products"

$$(a_i \in R_q\,,\, b_i = a_i \cdot s + e_i \bmod qR),$$

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

3.1.3 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 "small." 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 [43, 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 both errors and the decoding basis carry an extra $t_m = \hat{m}/g_m$ factor, an equivalent hypothesis is the following:

INVARIANT 1. *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" tails.

This 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 our FHE scheme of Section 4.

3.2 Cyclotomic Types: `Cyc` and `UCyc`

In this subsection we summarize 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`. 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 3.2.1 and 3.3.3 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 Section 3.3 for details).

3.2.1 Instances

`Cyc` and `UCyc` 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.

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, the `Functor` instance for `f` $=$ `UCyc t m rep` defines `fmap :: (r -> r') -> f r -> f r'` to apply the `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` could be properly defined.

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 Section 3.3.3 for further details.

3.2.2 Functions

```
scalarCyc   ::                      r -> Cyc t m r
mulG        ::              Cyc t m r -> Cyc t m r
divG        ::              Cyc t m r -> Maybe (Cyc t m r)
liftPow/Dec :: Lift b a => Cyc t m b -> Cyc t m a

-- error sampling
tGaussian    :: MonadRandom rnd =>   v -> rnd (Cyc t m q)
errorRounded :: MonadRandom rnd =>   v -> rnd (Cyc t m z)
errorCoset   :: ... => v -> Cyc t m zp -> rnd (Cyc t m z)

-- inter-ring operations
embed                :: Cyc t m  r ->  Cyc t m' r
twace                :: Cyc t m' r ->  Cyc t m  r
coeffsPow, coeffsDec :: Cyc t m' r -> [Cyc t m  r]
powBasis             ::      Tagged m [Cyc t m' r]
crtSet               ::      Tagged m [Cyc t m' r]
```

Figure 3: Representative functions for the `Cyc` data type. For brevity, we omit most constraints, including `Tensor t`, `Ring r`, `Fact m`, and `m 'Divides' m'`.

We briefly mention a few of the remaining functions that define the interface for `Cyc`; see Figure 3 for their type signatures. (`UCyc` admits a very similar collection of functions, which we omit from the discussion.)

`mulG`, `divG` respectively multiply and divide by the special element g_m. These are commonly used in applications, and have especially fast algorithms in all our representations, which is why we define them as special functions. Note that because the input may not always be divisible by g_m, the output type of `divG` is a `Maybe`.

`tGaussian` samples an element of the number field K from the "tweaked" continuous Gaussian distribution $t \cdot D_r$, given $v = r^2$. Because the output is a random variable, its type must be monadic. (The functions `errorRounded` and `errorCoset` work similarly.)

The following functions involve `Cyc` data for two indices `m|m'` (so the `m`th cyclotomic is a subring of the `m'`th one). The type signatures express the divisibility constraint as `m 'Divides' m'`, which is statically checked at compile time.

`embed`, `twace` are respectively the embedding and "tweaked trace" functions.

`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 the full version for its formal definition and a novel algorithm for computing it.)

3.3 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 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.

3.3.1 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: while cryptosystems 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 Section 3.3.2 below.

`UCyc` represents a cyclotomic ring element by its coefficient 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 $\mathbb{C}$.

3.3.2 Operations

Most of the `Cyc` functions shown in Figure 3 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 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 subrings of indices `k1`, `k2`|`m` are converted to the *compositum* of the two subrings, i.e., the cyclotomic of index `k` $=$ lcm(`k1`, `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 `embed` for `Cyc` 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. Otherwise, the input is in the `k'`th cyclotomic for some `k'`|`m'`, in which case we apply `twace` from index `k'` to index `k` = gcd(`m`, `k'`), which is the smallest index where the result is guaranteed to reside, and store the result as a subring element.

3.3.3 Promoting Base-Ring Operations

Many cryptographic operations on cyclotomic rings are defined as working entry-wise on the ring element's coefficient tensor with respect to some basis. 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` etc.

As a brief example, consider the `Functor` class, which introduces `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 2. For example, `Reduce z zq` can be promoted to `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.

The full version of the paper additionally describes how we promote the "gadget" operations `decompose` and `correct`, which requires the richer category-theoretic class `Traversable`. It also describes how we implement a specialized, more efficient algorithm for rescaling a product ring $R_q = R_{q_1} \times R_{q_2}$ to R_{q_1} (where the moduli q_i may themselves be products).

4. FHE IN $\Lambda\circ\lambda$

In this section we describe a full-featured fully homomorphic encryption and its implementation in $\Lambda\circ\lambda$, using the interfaces described in the previous sections. At the mathematical level, the system refines a variety of techniques and features from a long series of works [42, 12, 13, 11, 28, 43, 3]. In addition, we describe some novel generalizations and operations, such as "ring-tunneling." Due to space restrictions, most of the mathematical operations and their implementations are deferred to the full version (but see Figure 1 for representative code).

4.1 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'. 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' (see Invariant 1). 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.

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 "most significant digit" (MSD) relation

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

where the approximation hides a small fractional error term (in $\frac{1}{p}R'$) that satisfies Invariant 1. (The full version also describes an alternative "least significant digit" (LSD) relation; it is possible to losslessly convert between the two forms.)

Due to space restrictions, we describe the algorithms and implementations for encryption, decryption, homomorphic addition and multiplication, modulus switching, and key-switching/linearization in the full version.

4.2 Ring Tunneling

The term "ring switching" encompasses a collection of techniques, introduced in [11, 28, 3], 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.

Here we describe a novel method of ring switching, which we call *ring tunneling*, that is more efficient than the functionally equivalent method of [3], which for comparison we call *ring hopping*. The difference between the two methods is that hopping goes "up and then down" through the *compositum* of the source and target rings, while tunneling goes "down and then up" through their *intersection* (the largest common subring). Essentially, tunneling is more efficient because it uses an intermediate ring that is smaller than, instead of larger than, the source and target rings. In addition, we show how the linear function that is homomorphically applied to the plaintext can be integrated into the key-switching hint, thus combining two separate steps into a simpler and more efficient operation overall. 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. 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 $L(r) = L(\vec{b})^t \cdot \vec{e} = \vec{y}^t \cdot \vec{e}$.

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. (See the full version for a proof.)

LEMMA 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.*

Ring tunneling as key switching. Abstractly, ring tunneling 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. We now show that it can be implemented 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 1 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.

3. *Prepare* an appropriate key-switching hint using keys $s_{\mathrm{in}} \in R'$ and $s_{\mathrm{out}} \in S'$. 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 $L'(s_{\mathrm{in}} \cdot \vec{b}^t)$, such that

   $$(1, s_{\mathrm{out}}) \cdot H_j \approx L'(s_{\mathrm{in}} \cdot b_j) \cdot \vec{g}^t \pmod{qS'}. \quad (4.1)$$

   (As usual, the approximation hides appropriate Ring-LWE errors that satisfy Invariant 1.) Note 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

   $$c'(S) = L'(c_0) + \sum_j H_j \cdot g^{-1}(e_j) \pmod{qS'}. \quad (4.2)$$

For correctness, notice that we have

$$c_0 + s_{\mathrm{in}} \cdot c_1 \approx \tfrac{q}{p} \cdot l^{-1} \cdot \mu \pmod{qR'}$$
$$\implies L'(c_0 + s_{\mathrm{in}} \cdot c_1) \approx \tfrac{q}{p} \cdot l^{-1} \cdot L(\mu) \pmod{qS'}, \quad (4.3)$$

where the error in the second approximation is L' applied to the error in the first approximation, and therefore satisfies Invariant 1 by design of L'. Then by Equations (4.2), (4.1), E'-linearity of L', the definition of $\vec{e}$, and Equation (4.3),

$$\begin{aligned} c'(s_{\mathrm{out}}) &\approx L'(c_0) + \sum_j L'(s_{\mathrm{in}} \cdot b_j) \cdot \vec{g}^t \cdot g^{-1}(e_j) \\ &= L'(c_0 + s_{\mathrm{in}} \cdot \vec{b}^t \cdot \vec{e}) \\ &= L'(c_0 + s_{\mathrm{in}} \cdot c_1) \\ &\approx \tfrac{q}{p} \cdot l^{-1} \cdot L(\mu) \pmod{qS'} \end{aligned}$$

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 [3].

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)) 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.

5. EVALUATION

Recall that $\Lambda\circ\lambda$ primarily aims to be a general, modular, and safe framework for lattice cryptography, while also achieving acceptable performance. While $\Lambda\circ\lambda$'s modularity and static safety properties are described in the other sections of 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 [34], which is $\Lambda\circ\lambda$'s closest analogue in terms of scope and features. (Recall that HElib is a leading implementation of fully 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.

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).

Due to space restrictions, in this section we present summary statistics; detailed results can be found in the full version.

5.1 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 *cyclomatic complexity* [45]. "Core" functions are any that are called (directly or indirectly) by the libraries' public interfaces, such as algebraic, number-theoretic, and cryptographic operations, but not unit tests, benchmarks, etc. Note that HElib relies on NTL [52] 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 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 [7] for Haskell code and *metriculator* [38] for C/C++ code. Metriculator measures *logical* source lines of code, which approximates the number of "executable statements." By contrast, Ohcount counts "*physical*" (or actual) 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 equivalent, they provide a rough comparison between the two code bases.

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

Figure 4: Source lines of code for $\Lambda\circ\lambda$ and HElib+NTL.

Function count and cyclomatic complexity. McCabe's *cyclomatic complexity* (CC) [45] 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.

Figure 5 gives a summary of cyclomatic complexities in $\Lambda\circ\lambda$ and HElib+NTL, as measured by argon [39] for Haskell code and metriculator [38] for C/C++ code. A more detailed histogram is provided in the full version. 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), but at higher complexities, $\Lambda\circ\lambda$ has many fewer functions in both an absolute and relative sense.

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

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

Only three Haskell functions and two C++ functions in $\Lambda\circ\lambda$ received a grade of 'C;' in each of these, the complexity is simply due to the many combinations of cases for the representations of the inputs (see Section 3.3.2). The two C++ functions are the inner loops of the CRT and DFT transforms, due to a case statement that chooses the appropriate unrolled code for a particular dimension, which we do for performance reasons.

5.2 Performance

As a general-purpose library, we do not expect $\Lambda \circ \lambda$'s performance to be competitive with highly optimized (but inflexible) C implementations like SWIFFT [41] and BLISS [20], but we aim for performance in the same league as higher-level libraries like HElib. In the full version 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.

In summary, the benchmarks show that our pure-Haskell `Tensor` implementation (RT) based on Repa [37] tends to be about 2–40 times slower than our C++ backend (CT). However, CT itself is within a factor of 4–10 of HElib on bottleneck operations like CRTs and pointwise multiplications, and is actually *faster* for CRTs on non-power-of-two indices, because it uses a more efficient algorithm stemming from the "tensored" representations. (See Figure 6.)

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.

Index m	$\varphi(m)$	HElib	CT	RT
$2^{10} = 1{,}024$	512	16	139	2,344
$2^{11} = 2{,}048$	1,024	32	307	5,211
$2^6 3^3 = 1{,}728$	576	153	99	3,088
$2^6 3^4 = 5{,}184$	1,728	638	364	10,400
$2^6 3^2 5^2 = 14{,}400$	3,840	2,756	1,011	24,330

Figure 6: Runtimes (in microseconds) for the Chinese Remainder Transform from the powerful (`P`) to CRT (`C`) basis, for a single modulus. For comparison, we include HElib's analogous transformation from its "polynomial" to "Double CRT" representation, also with one modulus. All benchmarks were performed on a mid-2012 laptop with a Core i7-3610QM processor and 6 GB RAM.

6. ACKNOWLEDGMENTS

We thank Tancrède Lepoint for providing HElib benchmark code, Victor Shoup for helpful discussions regarding HElib performance, and the anonymous CCS'16 reviewers for many useful comments.

7. REFERENCES

[1] M. Ajtai. Generating hard instances of lattice problems. *Quaderni di Matematica*, 13:1–32, 2004. Preliminary version in STOC 1996.

[2] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. Post-quantum key exchange - a new hope. In *USENIX Security Symposium*, pages ??–??, 2016.

[3] J. Alperin-Sheriff and C. Peikert. Practical bootstrapping in quasilinear time. In *CRYPTO*, pages 1–20, 2013.

[4] A. Banerjee and C. Peikert. New and improved key-homomorphic pseudorandom functions. In *CRYPTO*, pages 353–370, 2014.

[5] A. Banerjee, C. Peikert, and A. Rosen. Pseudorandom functions and lattices. In *EUROCRYPT*, pages 719–737, 2012.

[6] D. J. Bernstein, C. Chuengsatiansup, T. Lange, and C. van Vredendaal. NTRU prime. Cryptology ePrint Archive, Report 2016/461, 2016. http://eprint.iacr.org/2016/461.

[7] Black Duck Software. Ohcount, 2014. https://github.com/blackducksoftware/ohcount, last retrieved May 2016.

[8] D. Boneh, K. Lewi, H. W. Montgomery, and A. Raghunathan. Key homomorphic PRFs and their applications. In *CRYPTO*, pages 410–428, 2013.

[9] J. Bos, C. Costello, L. Ducas, I. Mironov, M. Naehrig, V. Nikolaenko, A. Raghunathan, and D. Stebila. Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE. Cryptology ePrint Archive, Report 2016/659, 2016. http://eprint.iacr.org/2016/659.

[10] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *IEEE Symposium on Security and Privacy*, pages 553–570, 2015.

[11] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. *TOCT*, 6(3):13, 2014. Preliminary version in ITCS 2012.

[12] Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from Ring-LWE and security for key dependent messages. In *CRYPTO*, pages 505–524, 2011.

[13] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. *SIAM J. Comput.*, 43(2):831–871, 2014. Preliminary version in FOCS 2011.

[14] W. Castryck, I. Iliashenko, and F. Vercauteren. Provably weak instances of Ring-LWE revisited. In *EUROCRYPT*, pages 147–167, 2016.

[15] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore GPUs. In *DAMP 2011*, pages 3–14, 2011.

[16] H. Chen, K. Lauter, and K. E. Stange. Attacks on search RLWE. Cryptology ePrint Archive, Report 2015/971, 2015. http://eprint.iacr.org/.

[17] J. Coron, T. Lepoint, and M. Tibouchi. Scale-invariant fully homomorphic encryption over the integers. In *PKC*, pages 311–328, 2014.

[18] D. B. Cousins, J. Golusky, K. Rohloff, and D. Sumorok. An FPGA co-processor implementation of homomorphic encryption. In *HPEC 2014*, pages 1–6, 2014.

[19] E. Crockett and C. Peikert. $\Lambda \circ \lambda$: Functional lattice cryptography. In *ACM CCS*, pages ??–??, 2016. Full version at http://eprint.iacr.org/2015/1134.

[20] L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky. Lattice signatures and bimodal gaussians. In *CRYPTO*, pages 40–56, 2013.

[21] L. Ducas and P. Q. Nguyen. Faster Gaussian lattice sampling using lazy floating-point arithmetic. In *ASIACRYPT*, pages 415–432, 2012.

[22] L. Ducas and T. Prest. Fast fourier orthogonalization. Cryptology ePrint Archive, Report 2015/1014, 2015. http://eprint.iacr.org/.

[23] L. Ducas and T. Prest. A hybrid Gaussian sampler for lattices over rings. Cryptology ePrint Archive, Report 2015/660, 2015. http://eprint.iacr.org/.

[24] R. A. Eisenberg and J. Stolarek. Promoting functions to type families in haskell. In *Haskell 2014*, pages 95–106, 2014.

[25] R. A. Eisenberg and S. Weirich. Dependently typed programming with singletons. In *Haskell 2012*, pages 117–130, 2012.

[26] Y. Elias, K. E. Lauter, E. Ozman, and K. E. Stange. Provably weak instances of Ring-LWE. In *CRYPTO*, pages 63–92, 2015.

[27] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

[28] C. Gentry, S. Halevi, C. Peikert, and N. P. Smart. Field switching in BGV-style homomorphic encryption. *Journal of Computer Security*, 21(5):663–684, 2013. Preliminary version in SCN 2012.

[29] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In *CRYPTO*, pages 850–867, 2012.

[30] C. Gentry, C. Peikert, and V. Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *STOC*, pages 197–206, 2008.

[31] C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *CRYPTO*, pages 75–92, 2013.

[32] S. Gorbunov, V. Vaikuntanathan, and H. Wee. Attribute-based encryption for circuits. In *STOC*, pages 545–554, 2013.

[33] T. Güneysu, V. Lyubashevsky, and T. Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In *CHES*, pages 530–547, 2012.

[34] S. Halevi and V. Shoup. HElib: an implementation of homomorphic encryption. https://github.com/shaih/HElib, last retrieved August 2016.

[35] S. Halevi and V. Shoup. Bootstrapping for HElib. In *EUROCRYPT*, pages 641–670, 2015.

[36] J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: A ring-based public key cryptosystem. In *ANTS*, pages 267–288, 1998.

[37] G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, S. L. P. Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *ICFP 2010*, pages 261–272, 2010.

[38] U. Kunz and J. Weder. Metriculator. https://github.com/ideadapt/metriculator, 2011. version 0.0.1.201310061341.

[39] M. Lacchia. Argon, 2015. https://hackage.haskell.org/package/argon, version 0.4.1.0.

[40] B. Lippmeier, M. M. T. Chakravarty, G. Keller, and S. L. P. Jones. Guiding parallel array fusion with indexed types. In *Haskell 2012*, pages 25–36, 2012.

[41] V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen. SWIFFT: A modest proposal for FFT hashing. In *FSE*, pages 54–72, 2008.

[42] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. *Journal of the ACM*, 60(6):43:1–43:35, November 2013. Preliminary version in Eurocrypt 2010.

[43] V. Lyubashevsky, C. Peikert, and O. Regev. A toolkit for ring-LWE cryptography. In *EUROCRYPT*, pages 35–54, 2013.

[44] C. M. Mayer. Implementing a toolkit for ring-lwe based cryptography in arbitrary cyclotomic number fields. Cryptology ePrint Archive, Report 2016/049, 2016. http://eprint.iacr.org/2016/049.

[45] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, July 1976.

[46] C. A. Melchor, J. Barrier, S. Guelton, A. Guinet, M. Killijian, and T. Lepoint. NFLlib: NTT-based fast lattice library. In *CT-RSA*, pages 341–356, 2016.

[47] D. Micciancio. Generalized compact knapsacks, cyclic lattices, and efficient one-way functions. *Computational Complexity*, 16(4):365–411, 2007. Preliminary version in FOCS 2002.

[48] D. Micciancio and C. Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In *EUROCRYPT*, pages 700–718, 2012.

[49] M. Naehrig, K. Lauter, and V. Vaikuntanathan. Can homomorphic encryption be practical? In *CCSW*, pages 113–124, 2011.

[50] C. Peikert. How (not) to instantiate Ring-LWE. In *SCN*, pages ??–??, 2016.

[51] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6):1–40, 2009. Preliminary version in STOC 2005.

[52] V. Shoup. A library for doing number theory, 2006. http://www.shoup.net/ntl/, version 9.8.1.

[53] N. P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography*, pages 420–443, 2010.

[54] N. P. Smart and F. Vercauteren. Fully homomorphic SIMD operations. *Designs, Codes and Cryptography*, 71(1):57–81, 2014. Preliminary version in ePrint Report 2011/133.

[55] D. Stehlé and R. Steinfeld. Making NTRU as secure as worst-case problems over ideal lattices. In *EUROCRYPT*, pages 27–47, 2011.

[56] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar. Accelerating fully homomorphic encryption using GPU. In *HPEC 2012*, pages 1–5, 2012.

[57] B. A. Yorgey, S. Weirich, J. Cretin, S. L. P. Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *TLDI 2012*, pages 53–66, 2012.