BASALISC: Programmable Asynchronous Hardware Accelerator for BGV Fully Homomorphic Encryption

Robin Geelen*
imec-COSIC KU Leuven
Leuven, Belgium
robin.geelen1@kuleuven.be

Michiel Van Beirendonck*
imec-COSIC KU Leuven
Leuven, Belgium
michiel.vanbeirendonck@kuleuven.be

Hilder V. L. Pereira
imec-COSIC KU Leuven
Leuven, Belgium
hildervitor.limapereira@kuleuven.be

Brian Huffman
Galois, Inc.
Portland, USA
huffman@galois.com

Tynan McAuley
Niobium Microsystems
Portland, USA
tynan@niobiummicrosystems.com

Ben Selfridge
Galois, Inc.
Portland, USA
benselfridge@galois.com

Daniel Wagner
Galois, Inc.
Portland, USA
dmwit@galois.com

Georgios Dimou
Niobium Microsystems
Portland, USA
georgios@niobiummicrosystems.com

Ingrid Verbauwhede
imec-COSIC KU Leuven
Leuven, Belgium
ingrid.verbauwhede@kuleuven.be

Frederik Vercauteren
imec-COSIC KU Leuven
Leuven, Belgium
frederik.vercauteren@kuleuven.be

David W. Archer
Galois, Inc.
Portland, USA
dwa@galois.com

***Abstract*—Fully Homomorphic Encryption (FHE) allows for secure computation on encrypted data. Unfortunately, huge memory size, computational cost and bandwidth requirements limit its practically. We present BASALISC, an architecture family of hardware accelerators that aims to substantially accelerate FHE computations in the cloud. BASALISC is the first to implement the BGV scheme supporting fully-packed bootstrapping – the noise removal capability necessary to support arbitrary-depth computation. We propose a generalized version of bootstrapping that can be implemented directly in our hardware, instantiated with Montgomery multipliers that save 46% in silicon area and 40% in power consumption compared to traditional approaches.**

BASALISC is a three-abstraction-layer RISC architecture, designed for a 1 GHz ASIC implementation and underway toward 150mm^2 die tape-out in a 12nm GF process. BASALISC's four-layer memory hierarchy includes a two-dimensional conflict-free inner memory layer that enables 32 Tb/s radix-256 NTT computations without pipeline stalls. Our conflict-resolution permutation hardware is generalized and re-used to compute BGV automorphisms without throughput penalty. BASALISC also has a custom multiply-accumulate unit to accelerate BGV key switching. Both BASALISC's computation units and inner memory layers are designed in asynchronous logic, allowing them to run at different speeds to optimize each function.

The BASALISC toolchain comprises a custom compiler and a joint performance and correctness simulator. To evaluate BASALISC, we study its physical realizability, emulate and formally verify its core functional units, and we study its performance on a set of benchmarks. First, we evaluate a single iteration of logistic regression training over encrypted data – an application that translates to 513 bootstraps, 900K high-level, or 27B low-level BASALISC instructions – showing that BASALISC is only 3,500× slower than an Intel Xeon-class processor running *without* data encryption. We also run an individual bootstrapping operation, for which we show a speedup of 4,000× over HElib - a popular software FHE library.

***Index Terms*—fully homomorphic encryption, Brakerski-Gentry-Vaikuntanathan, hardware accelerator, application-specific integrated circuit**

[*]R. Geelen and M. Van Beirendonck contributed equally to this research.

1. Motivation

Fully Homomorphic Encryption (FHE) [1]–[3] offers the promise of confidentiality-preserving computation over sensitive data in a variety of theoretical and practical applications, ranging from new cryptographic primitives to machine learning as a service. Unfortunately, the utility of FHE is severely limited by its high memory size, memory bandwidth and high computational overhead. The typical result - computation that runs many orders of magnitude slower than insecure computation - prevents broad adoption. Although new schemes have markedly improved FHE performance [4]–[7], and highly optimized FHE libraries [8]–[12] are now available, FHE still remains orders of magnitude beyond acceptable performance limits for most potential applications.

In other computational domains where performance on general purpose processors is problematic, innovation has turned to purpose-built *accelerators*, tuned to exploit domain-specific characteristics of computation. DSP accelerators, arguably starting with the Texas Instruments TMS320 DSP family [13] in 1983, are perhaps the first example of this approach. More recently, Graphics Processing Units (GPUs) have become popular for accelerating video stream processing and hash function computation. Our FHE accelerator, BASALISC, follows this approach in pursuit of bringing the throughput of FHE computation within an order of magnitude relative to cleartext computation [14].

We summarize the key contributions of BASALISC as follows:

- BASALISC accelerates BGV arithmetic for a large range of parameters. BASALISC is a comprehensive RISC-like architecture with a three-level instruction set architecture (ISA) that allows for reasoning at diverse levels of executive abstraction. In contrast to prior accelerators, BASALISC is the first to support and implement fully-packed BGV bootstrapping directly in hardware to enable unlimited-depth FHE computations.
- We propose a novel version of bootstrapping that is compatible with NTT-friendly primes. In contrast to prior work, BASALISC instantiates its multipliers exclusively to these NTT-friendly primes, which saves 46% logic area and 40% power consumption.
- BASALISC implements a massively parallel radix-256 NTT architecture, using a conflict-free layout, a corresponding layout permutation unit, and a twiddle factor generator. These units are deeply interleaved with the on-chip memory and provide a record 32 Tb/s NTT throughput. In addition, we show that we can efficiently generalize the required layout permutation unit to compute BGV automorphisms *without additional silicon area*.
- BASALISC adopts a four-level memory hierarchy purpose-built to address common FHE memory bottlenecks, including a mid-level 64 MB on-chip ciphertext buffer (CTB). At the lowest level, a massively parallel multiply-accumulate unit with integrated 16-entry register file allows accelerating tight BGV key switching loops, asynchronously and independently of the CTB.
- BASALISC is placed and routed with 150mm^2 die size and 1 GHz operational frequency in a 12nm low-power Global Foundries process. Critical hardware logic is emulated and formally verified for correctness. We evaluate BASALISC on a bootstrapping benchmark and a logistic regression application, showing 4,000× speedup over an HElib software reference.

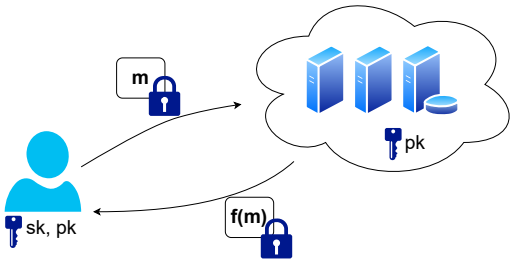

Figure 1: FHE used in a typical commercial application.

2. Preliminaries

2.1. Fully Homomorphic Encryption

Fully Homomorphic Encryption (FHE) provides a simple use model to securely outsource computation on sensitive data to a third party. Informally, the FHE model enables a user to encrypt their data m into a ciphertext $c = \text{Enc}(m)$, then send it to a third party, who can compute on c. The third party produces another ciphertext c' encrypting $f(m)$ for some desired function f. We say that f was computed homomorphically.

In FHE, the third party receives only ciphertexts and the public key, but never the secret key that allows decryption. As a result, the sensitive inputs are protected under the security of the encryption scheme. Because the result of the computation remains encrypted, the output also remains unknown to the third party: only the holder of the secret key can decrypt and access it. This scenario is illustrated in Figure 1.

To achieve security, the ciphertexts of all FHE schemes are noisy: during encryption, a small noise term is added to the input data. Decryption can still recover the correct result, provided that the noise is small enough. To evaluate a function homomorphically, we represent the function in terms of the operations provided by the scheme, typically addition and multiplication, and compute these operations on the encrypted inputs. Each operation increases the noise in the resulting ciphertext, so we can compute only a limited number of homomorphic operations before we reach the limit of decryption failure.

Because multiplications increase ciphertext noise much more than additions, we usually model noise growth by the

number of sequential multiplications only. If we compute the product $\prod_{i=1}^{L} m_i$ homomorphically, then we say that the computation requires multiplicative depth $\lceil \log_2(L) \rceil$. This is accomplished by writing the product in a tree structure, with each leaf node representing one of the factors. In general, there is a trade-off between computational cost and tolerating a larger L: we can increase the FHE parameters so that we obtain more multiplicative depth, but in doing so, we make the homomorphic operations slower and the size of ciphertexts larger.

To support the computation of functions regardless of their multiplicative depth, FHE uses *bootstrapping*. This operation reduces noise by decrypting a ciphertext homomorphically. Unfortunately, bootstrapping is very expensive, so its use is often minimized. There are several techniques in the FHE literature to slow down the noise growth, and thus delay bootstrapping. In this work, we employ *key switching* and *modulus switching* [4]. We note that bootstrapping and key switching tend to heavily dominate computation and data movement costs of an application: in a simple 1,024-point, 10-feature logistic regression, we see that these tasks account for over 95% of the computational effort and the vast majority of data movement.

The key challenges in designing an efficient FHE scheme are the high complexity of computation, the large ciphertext expansion factor (large polynomials with integer coefficients of 1000 bits or more), and the proportion of effort needed in bootstrapping (or delaying it) in sufficiently complex programs. In the remainder of this paper, we examine the magnitude of these challenges and how they impact the design of our FHE accelerator.

2.2. The BGV Cryptosystem

BASALISC targets the homomorphic encryption scheme known as BGV [4]. Plaintexts and ciphertexts are represented by elements in the ring $\mathcal{R} = \mathbb{Z}[X]/(X^N+1)$ with N a power of 2. Those elements are thus polynomials reduced modulo $X^N + 1$, and this modular reduction is implicit in our notation. BGV guarantees finite data structures by also reducing the coefficients: the plaintext space is computed modulo t (denoted $\mathcal{R}_t$), and the ciphertext space is a pair of elements modulo q (denoted $\mathcal{R}_q^2$). Reduction modulo m (with $m = t$ or q) is explicitly denoted by $[\cdot]_m$. It is always done symmetrically around 0, i.e., in the set $[-m/2, m/2) \cap \mathbb{Z}$.

As with traditional ciphers, BGV has encryption and decryption procedures to move between the *plaintext space* and the *ciphertext space*. These operations are never executed by the server doing outsourced computation and therefore not implemented by BASALISC. However, it is necessary to explain the ciphertext format in order to understand homomorphic operations. A BGV ciphertext $(\boldsymbol{c}_0, \boldsymbol{c}_1) \in \mathcal{R}_q^2$ is said to encrypt plaintext $\boldsymbol{m} \in \mathcal{R}_t$ under secret key $\boldsymbol{s}$ (which has small coefficients) if

$$\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} = \boldsymbol{m} + t\boldsymbol{e} \pmod{q} \tag{1}$$

for some element $\boldsymbol{e}$ that also has small coefficients. The term $\boldsymbol{e}$ is called the *noise*, and it determines if decryption returns the correct plaintext: as long as $\boldsymbol{e}$ has coefficients roughly smaller than $q/2t$, the expression $\boldsymbol{m} + t\boldsymbol{e}$ does not overflow modulo q. We can therefore recover the plaintext uniquely as $\boldsymbol{m} = [[\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s}]_q]_t$.

2.2.1. Basic Homomorphic Operations. Smart and Vercauteren observed that for $t = p^r$ with p an odd prime, the plaintext space $\mathcal{R}_t$ is equivalent to $\mathbb{Z}_t^\ell$ for some ℓ that divides N [15]. This technique is referred to as *packing*, and it allows us to encode ℓ numbers into one plaintext simultaneously. Addition and multiplication over tuples in $\mathbb{Z}_t^\ell$ are then performed component-wise. As a result, one ciphertext can encrypt and operate on an entire tuple, which leads to significant performance gains and memory reductions in practice.

When BGV is used in conjunction with packing, we can define three basic homomorphic operations. Let $(\boldsymbol{c}_0, \boldsymbol{c}_1)$ and $(\boldsymbol{c}_0', \boldsymbol{c}_1')$ be two ciphertexts encrypting the tuples $(m_1, ..., m_\ell)$ and $(m_1', ..., m_\ell')$, then we have:

- **Addition:** we compute $([\boldsymbol{c}_0 + \boldsymbol{c}_0']_q, [\boldsymbol{c}_1 + \boldsymbol{c}_1']_q)$. The encrypted plaintext is $(m_1 + m_1', ..., m_\ell + m_\ell')$.
- **Multiplication:** we compute $([\boldsymbol{c}_0 \cdot \boldsymbol{c}_0']_q, [\boldsymbol{c}_0 \cdot \boldsymbol{c}_1' + \boldsymbol{c}_1 \cdot \boldsymbol{c}_0']_q, [\boldsymbol{c}_1 \cdot \boldsymbol{c}_1']_q)$. The resulting ciphertext is a vector of three elements, but this can be reduced back to two with a post-processing step called *key switching*. The encrypted plaintext is $(m_1 \cdot m_1', ..., m_\ell \cdot m_\ell')$.
- **Permutation:** we compute $(\phi_k(\boldsymbol{c}_0), \phi_k(\boldsymbol{c}_1))$, where the map ϕ_k is called an *automorphism*. It is parameterized by an odd integer k, and defined as $\phi_k \colon c(X) \mapsto c(X^k)$. Gentry et al. [16] show that these automorphisms induce a permutation on the elements of the encoded tuple, so the output encrypts some permutation of $(m_1, ..., m_\ell)$. Although the resulting ciphertext has only two elements, we still need post-processing by means of key switching.

The validity of these three operations can simply be verified by observing their effect on Equation 1. We refer to Zucca [17] for a more detailed analysis, including noise growth of each operation.

2.2.2. Auxiliary Homomorphic Operations. Basic homomorphic operations lead to ciphertext expansion and noise growth. Take for example a product ciphertext: it consists of three elements and it is encrypted under $(\boldsymbol{s}, \boldsymbol{s}^2)$ instead of $\boldsymbol{s}$. The same problem occurs during permutation: the automorphism ϕ_k has a side effect on the secret key, so the resulting ciphertext is encrypted under $\phi_k(\boldsymbol{s})$. Also noise growth is an issue: the noise term in a product ciphertext, for example, has increased to $t\boldsymbol{e} \cdot \boldsymbol{e}'$.

To prevent ciphertext expansion, switch between keys and slow down noise growth, BGV defines two auxiliary procedures:

- **Modulus switching:** given a ciphertext $(\boldsymbol{c}_0, \boldsymbol{c}_1) \in \mathcal{R}_q^2$ and a new modulus q', we compute a ciphertext

$(\boldsymbol{c}'_0, \boldsymbol{c}'_1) \in \mathcal{R}^2_{q'}$ that decrypts with respect to q'. Modulus switching also scales the noise by a factor of q'/q.

- **Key switching:** given a key switching matrix $(\overrightarrow{\boldsymbol{k}_0}, \overrightarrow{\boldsymbol{k}_1})$ and either a product ciphertext $(\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2) \in \mathcal{R}^3_q$ or a permuted ciphertext $(\boldsymbol{c}_0, \boldsymbol{c}_1) \in \mathcal{R}^2_q$, we compute a ciphertext $(\boldsymbol{c}'_0, \boldsymbol{c}'_1) \in \mathcal{R}^2_q$ that decrypts under Equation 1. Thus key switching brings the ciphertext back to its original format.

In summary, modulus switching is run before each multiplication to reduce the noise to its minimum level. Key switching is run after each permutation or multiplication to keep the ciphertext format consistent. Again we refer to Zucca [17] for a more detailed analysis.

2.2.3. Bootstrapping. When the entire noise budget of a ciphertext is consumed (equivalently, when the modulus q is depleted to its minimum value by successive modulus switchings), further homomorphic operations are no longer immediately possible. We can overcome this problem by means of a *bootstrapping* procedure that reduces the noise back to a lower level [2]. Bootstrapping "refreshes" a ciphertext by running decryption *homomorphically*: we first evaluate an adapted version of Equation 1, followed by coefficient-wise rounding. The currently most efficient bootstrapping technique for BGV is implemented in the HElib library [18].

2.2.4. Supported Parameter Sets. As a nod to Amdahl's Law ("make the common case fast"), hardware optimization gains throughput benefits by supporting only a limited range of commonly used parameters. We start with the realization that at least 128-bit security must be supported if BASALISC is to be interesting to real-world users. Based on this observation, we choose a range of parameters that allows for an efficient implementation, while still retaining sufficient freedom for application design.

Recall that FHE has a trade-off between implementation cost and supported complexity of computation: we can increase the multiplicative depth L and the plaintext modulus p^r by taking sufficiently high N and q, but this makes the homomorphic operations inherently slower. A typical range for the ring dimension N, still offering sufficient flexibility, is between 2^{14} and 2^{16}. BASALISC settles on a maximum value of $N = 2^{16}$, which allows us to choose ciphertext moduli up to $q = 2^{1782}$ at 128-bit security level. This results in a large number of multiplicative levels, even at a high-precision plaintext space (e.g., 31 levels at plaintext modulus $p^r = 127^3$ without bootstrapping; with bootstrapping, we get an arbitrary number of levels).

Table 1 shows the full parameter range supported by BASALISC and an example parameter set for illustration. The largest ciphertext modulus that appears during the basic homomorphic operations is denoted by Q; key switching, however, increases the modulus to QP.[1] Concretely, our largest supported modulus is $QP = 2^{1782}$ (limited by the 128-bit security target). The smallest supported modulus is $6 \cdot 2^{17} + 1$ – the smallest prime congruent to 1 modulo 2^{17} (because of the restrictions that follow in Section 7.4).

1. Extending the modulus to QP during key switching is a common trick in FHE. More information is given in Appendix A.

TABLE 1: BASALISC parameter ranges and examples.

Parameter	Range	Example
Security parameter	N/A	128 bits
Ring dimension N	$512 - 65536$	65536
Plaintext modulus p^r	> 2	127^3
Ciphertext packing ℓ	$2 - 65536$	64 slots
Max $\log_2(QP)$ for key switching	$20 - 1782$	1782 bits
Max $\log_2(Q)$ for ciphertext	$20 - 1782$	1263 bits
Max multiplicative depth L	N/A	31

3. Data Representation & Algorithms

Basic homomorphic operations use arithmetic in the ring $\mathcal{R}_q$, namely polynomial addition, multiplication and automorphism. Polynomial addition is coefficient-wise, and can be handled via vector addition directly. Multiplication and automorphism are more complicated; fortunately, they can be handled in the "frequency domain" via respectively entry-wise multiplication and entry permutation [19].

We now explain two common tricks to accelerate operations in $\mathcal{R}_q$. Section 3.1 explains how computations modulo q can be split into many smaller moduli q_i, based on the Chinese Remainder Theorem. Section 3.2 explains conversion between the polynomial and frequency domain, aiming for efficient polynomial multiplication and automorphism.

3.1. Residue Number System

Suppose that the ciphertext modulus is $q = q_1 \cdot \ldots \cdot q_k$, where the factors are pairwise coprime numbers. It is a well-known fact that computations in $\mathcal{R}_q$ can be reduced to simultaneous computations in the smaller rings $\mathcal{R}_{q_i}$. More specifically, we apply the Chinese Remainder Theorem to work modulo each q_i individually. This common optimization is referred to as a Residue Number System (RNS) [20], and brings an asymptotic speedup factor of $\mathcal{O}(k)$. It also simplifies our architecture since the size of each q_i is much smaller than q (a typical value is 32 bits for q_i versus more than 1000 bits for q).

3.2. Number-Theoretic Transform

In order to perform efficient polynomial multiplication in time $\mathcal{O}(N \log(N))$, we resort to the Number-Theoretic Transform (NTT). The NTT is a generalization of the Fast Fourier Transform (FFT) to finite fields[2], and allows us to use exact integer arithmetic, preventing round-off errors typical of real-valued FFT computations. Similar to the FFT,

2. The NTT can also be interpreted in terms of the Chinese Remainder Theorem, similarly to RNS. Hence, the combination of using RNS for fast arithmetic modulo q and the NTT for fast polynomial arithmetic modulo $X^N + 1$, is often referred to as *Double-CRT*.

the NTT can be computed with the Cooley-Tukey algorithm that recursively re-expresses an NTT of size $N = N_1 N_2$ as N_2 inner NTTs of size N_1, followed by N_1 outer NTTs of size N_2. Before the outer NTT, each output of the inner NTT is multiplied by a twiddle factor:

$$X[k_1+N_1k_2] = \sum_{n_2=0}^{N_2-1} \left(\sum_{n_1=0}^{N_1-1} x[N_2n_1+n_2]\omega_{N_1}^{n_1k_1} \right) \omega_N^{n_2k_1}\omega_{N_2}^{n_2k_2}. \tag{2}$$

By choosing $N_1 = 2$ and $N_2 = N/2$ at each recursive decomposition or vice-versa, the well-known radix-two Decimation-In-Time (DIT) and Decimation-In-Frequency (DIF) algorithms are obtained, respectively.

The NTT can be naturally used for fast cyclic convolutions (polynomial multiplication modulo X^N-1). However, BGV and other FHE schemes perform polynomial multiplication modulo X^N+1, requiring *negacyclic* convolutions. Negacyclic convolutions can still be implemented with a regular NTT, additionally requiring to pre-multiply the two input polynomials and post-multiply the output polynomial by an extra set of twiddle factors [21].

3.3. Supported Word Size

Because of algebraic constraints, the NTT is only defined for prime moduli that satisfy $q_i = 1 \pmod{2N}$. We refer to these special moduli as *NTT-friendly* primes. Since we support up to $N = 2^{16}$, this puts a lower bound of 17 bits on the size of q_i. Coupled with the requirement to have a sufficient amount of moduli to reach $\log_2(QP) = 1782$ bits, a simple analysis shows that we need q_i of at least 26 bits. In practice, however, BASALISC employs 32-bit moduli, because it gives a better utilization for the on-chip memory buffer and simplified interaction with the external memory. Furthermore, we find that both 26-bit and 32-bit moduli result in the same complement of arithmetic units within our silicon area budget. For the example parameter set of Table 1, Q is a product of 42 primes and P is a product of 14 additional primes.

3.4. Algorithmic Requirements for BASALISC

Based on the operations explained above, we conclude that BGV requires three low-level building blocks:

- Entry-wise vector addition and multiplication
- Entry permutation within a vector
- The NTT to convert between polynomial and frequency domain.

Section 7 explains how we map these three building blocks naturally to three Processing Elements (PEs): the MAC PE, Permutation PE and NTT PE, respectively. This very limited instruction set, in conjunction with the determinism of FHE programs, will make BASALISC's design much simpler than traditional CPUs.

Although we concentrated only on the basic homomorphic operations, the auxiliary ones are constructed from the same instruction set. Key-switching, for example, incorporates both the MAC and NTT unit. Still, there is a large complexity gap between the basic and auxiliary operations: in practice, key switching dominates overall computation and is more than $10\times$ as expensive as ciphertext multiplication.

4. NTT-Friendly Bootstrapping

Recall that the NTT is only defined for NTT-friendly primes of the shape $q_i = 1 \pmod{2N}$. BASALISC goes even one step further by optimizing its Montgomery multipliers [22] for NTT-friendly primes. This design choice is taken for area and power related reasons, and further explained in Section 7.4. However, restricting the moduli in this way is incompatible with all currently existing bootstrapping methods for BGV [18], [23].

Consider for example the bootstrapping routine as implemented in the HElib library [18]. Let the plaintext modulus be $t = p^r$, then bootstrapping evaluates an adapted version of Equation 1 under the ciphertext modulus $q = p^e + 1$ that is significantly smaller than Q. It also involves an *exact* division by p^r, which is implemented based on arithmetic modulo p^r. However, both $p^e + 1$ and p^r are not NTT-friendly in general, so this cannot be done with our optimized Montgomery multipliers.

We propose a generalized version of bootstrapping that works with NTT-friendly primes exclusively. Our algorithm is simpler than all current approaches, yet it can be evaluated at exactly the same computational cost. The root of our algorithm is a new decryption formula that has sufficient degrees of freedom to take q as a product of NTT-friendly primes and does not involve an exact division operation. Consider the following lemma.

Lemma 4.1. *Let $p > 1$ be a prime number, and let $e > r \geqslant 1$ and $q = 1 \pmod{p^e}$ be sufficiently high parameters. If $(\boldsymbol{c}_0, \boldsymbol{c}_1)$ is a BGV encryption of $\boldsymbol{m}$ with plaintext modulus p^r and ciphertext modulus q, then it can be decrypted by computing*

$$\boldsymbol{c}'_i \leftarrow [p^{e-r}\boldsymbol{c}_i]_q, \quad \boldsymbol{w} \leftarrow [\boldsymbol{c}'_0+\boldsymbol{c}'_1\cdot\boldsymbol{s}]_{p^e}, \quad \boldsymbol{m} \leftarrow [\lfloor \boldsymbol{w}/p^{e-r} \rceil]_{p^r}.$$

Here we use $\lfloor\cdot\rceil$ for coefficient-wise rounding to the nearest integer.

The first and second step in Lemma 4.1 are implemented based on the techniques of Bajard et al. [20]. The third step is identical to the bootstrapping algorithm from HElib [18]. More details such as the proof and pseudocode are deferred to Appendix B.

We emphasize that NTT-friendly bootstrapping does not influence the security of BGV and is related to efficient implementation only. Since BGV is an exact homomorphic encryption scheme, security is defined in the IND-CPA model [24], which depends on key generation and encryption only. Our implementation does not change key

generation nor encryption, so security is guaranteed. The argument is simple: all key material and ciphertexts are identical in the standard and NTT-friendly scenario. Therefore, any attacker in the standard scenario could mimic the NTT-friendly scenario. So if NTT-friendly bootstrapping were insecure, the standard one would already be.

5. BASALISC Instruction Set Architecture

BASALISC is an adapted Reduced Instruction Set Computer (RISC) architecture with a three-level instruction set. This multi-level approach allows for reasoning at diverse levels of abstraction, and aids in assuring correctness of our system. Having a hierarchy of multiple intermediate representations and instruction sets, each with well-defined semantics, means that we can implement and test each stage of the compiler toolchain separately. In addition, different instruction set abstractions allow programmers to work at a higher level of abstraction while allowing compiler writers and library authors to reason about lower-level details such as scheduling and optimizations easily. For example, when writing a program to run on BASALISC, the programmer need not know about low-level data representations. Specifically, we generate and reason over three distinct levels of instruction and typesystem abstraction.

- **Macro-instructions** are at the highest level, with the largest data types and the most complex operations. Entire ciphertexts, plaintexts, and key switching matrices are treated as basic data types at this abstraction level. Operations include ciphertext addition, multiplication, modulus and key switching, automorphisms, and bootstrapping. Details about data representation and algorithms that implement those operations are opaque at this level of abstraction.
- **Mid-level instructions** expose the Double-CRT data representation. The basic data type at this level is a residue polynomial (a polynomial in RNS representation) comprising up to 2^{16} 32-bit polynomial coefficients. Basic operations on these data types include pointwise modular addition and multiplication on vectors of coefficients; automorphisms; NTTs; and multiply-accumulate iterations commonly used in key switching. Also included in this list are memory management instructions that Load and Store data to and from off-chip memory.
- **Micro-instructions** correspond very closely with the specific operations performed by the processing elements (PEs). The basic data type at this level contains as many coefficient words (1024 or 2048) as can be processed simultaneously by a PE or accessed in one on-chip memory cycle. Instructions at this level are delivered via the Peripheral Component Interconnect Express (PCIe) interface to the BASALISC processor for execution. This instruction level also includes rudimentary machine control instructions.

TABLE 2: BASALISC Example Opcodes. We omit operand specifiers.

ISA	Opcode	Semantics
Macro	LOAD	Move data from distant to near memory
	KSW	Key switch a ciphertext
	MORPH	Perform automorphism on a ciphertext
Mid	MULI	Multiply a residue polynomial by constant
	NTT	Compute NTT of residue polynomial
	FBE	Fast Base Extension
Micro	NTT1	Perform an iteration of the first pass of an NTT
	MAC	Multiply two operands, add result to accumulator

TABLE 3: BASALISC Micro-level operand addressing modes.

Mode	Definition
$XXX	address in distant memory, used only for LOAD/STORE
rXXX	address of chunk in middle memory
tXX	register number in near memory
nXXX	immediate 32-bit scalar
iXXX	index into table of prime moduli

Table 2 shows examples of operation code mnemonics (opcodes) from each of our instruction sets. Together with opcodes, BASALISC uses operand specifiers with register-like addressing modes for all levels in its ISA and memory hierarchy. Table 3 shows examples of the addressing modes for operand specifiers at the Micro-instruction level. At this level, operands are either "chunks" of 2048 32-bit coefficients within a residue polynomial, 32-bit scalar values, or natural number indices into tables of moduli.

6. BASALISC Hardware Design

Figure 2 shows a system diagram of BASALISC 1.0 – the first implementation in the BASALISC family. BASALISC 1.0 is a single-chip FHE coprocessor, designed in a 12nm Global Foundries process, with additional off-chip memory; high-speed connectivity to its host system; and extensibility via a high-speed inter-chip interconnect. For its implementation, BASALISC employs a mature asynchronous logic process controlled by standard handshaking protocols [25], allowing units to accelerate to a higher clock frequency whenever input data is available. In Section 7.3, we describe how this key design choice greatly helps to accelerate the costly BGV key-switching.

6.1. Memory Architecture

BASALISC includes four layers of memory hierarchy that exhibit diverse latencies and capacities. Table 4 depicts these four layers, where – typical of computer memory hierarchies – lower latency layers have smaller capacities. From farthest to nearest to the PEs, these four layers comprise off-chip distant memory (DRAM), a middle memory Ciphertext Buffer (CTB), as well as both a local Register File (RF) and Accumulator register (ACC) within the MAC PE.

A significant difference between typical memory hierarchies and that of BASALISC is the working set size that each layer can hold. Still, we expect capacity limits of layers

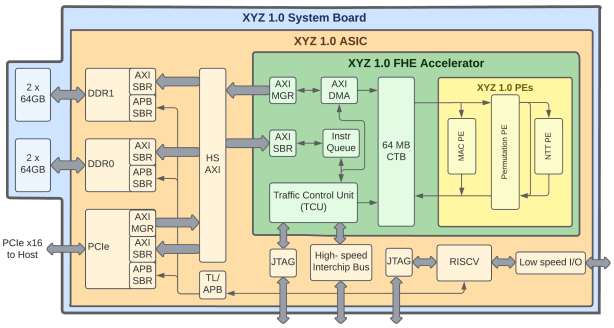

Figure 2: BASALISC 1.0 System Diagram.

in our memory hierarchy to be a major limiter of system performance. In particular, we expect minimal locality of reference for key switching matrices, each of which is larger than the entire CTB. We now describe the DRAM array and the CTB, and defer the description of the MAC memory architecture to Section 7.3.

TABLE 4: Memory hierarchy for ciphertext and key storage.

Memory	Capacity	Round-trip latency
Off-chip DRAM	256 GB	$>$ 100 ns
CTB	64 MB	$\sim$3 ns
MAC RF	128 kB	$\sim$1.25 ns
MAC ACC	8 kB	0.625 ns

6.1.1. Middle Memory – the CTB. The 64 MB CTB contains 2^{24} locations, each of which holds a 32-bit residue polynomial coefficient. In our largest supported parameter set, a single residue polynomial consists of $N = 2^{16} = 64$K coefficients and occupies one entire page of the CTB. For smaller ring dimensions, a single CTB page will contain multiple residue polynomials. The CTB is a single-port SRAM array that can either read or write 2048 32-bit residue polynomial coefficients every machine cycle, providing a total bandwidth of 8 Tb/s (at 1 GHz operation) to our set of data Processing Elements (PEs) shown in yellow.

Data-dependent control flow such as branching and iteration does not exist in FHE since variables are encrypted.[3] As an advantage of this determinism, allocation and size of all data and operands are bound at compile-time. This allows the BASALISC CTB to be structured as an addressable set of ciphertext registers, instead of requiring the complex functionality of a run-time cache memory. This set of registers is compiler-managed with a true Least-Recently Used (LRU) replacement policy. CTB bandwidth is not materially affected by concurrent transfer between distant memory and the CTB: roughly at most 0.3% of CTB access cycles are used by our total distant memory bandwidth.

6.1.2. Distant Memory – the DRAM array. The DRAM serves as the staging area for data that is scheduled for processing and for results that are ready for retrieval by the host computer. In addition, for many applications, the 64 MB CTB is too small to hold the sizeable working sets of ciphertexts and key switching matrices. The DRAM array ensures that CTB capacity misses do not have to spill to host memory.

3. Branches in FHE must typically be translated to some form of predicated execution.

6.2. System Design

In contrast to other FHE hardware accelerators [26], BASALISC 1.0 reduces cost and manufacturing risk by relying only on commercially available standard packaging, DRAM, and PCIe technologies and fits within 150mm^2 [14]. From the top down, the BASALISC system can be described in different levels of hierarchy (see Figure 2).

(Blue) The BASALISC System Board instantiates the distant DRAM memory using two DDR4 subsystems, each providing up to 128 GB of DRAM and 25.6 GB/s of bandwidth. At bottom left of the diagram is the 26 GB/s (near-peak) PCIe x16 channel that connects BASALISC to its host and carries data and instructions. We expect applications with a large working set to be performance-limited by our DDR4 bandwidth. The twin DDR4 interfaces allow us to maximize the practical throughput of the DRAM subsystem, by avoiding collisions between the PCIe-to-DRAM and FHE-to-DRAM access streams.

(Orange) The BASALISC ASIC includes JTAG I/O for testing and debuging, a RISC-V CPU to configure BASALISC at startup, and the controllers and physical interfaces (PHYs) for DDR4 and PCIe. These PHYs connect to the 512-bit wide Advanced eXtensible Interface 4 (AXI4) interconnect that transfers data between the DDR, PCIe, and the CTB. Both the AXI4 and CTB operate at a target cycle time of 1 GHz. As a result, the AXI has a peak bandwidth of 32 GB/s for each endpoint connection, all running in parallel.

(Green) The BASALISC FHE Core Processor includes the CTB, AXI4 infrastructure, and a Traffic Control Unit (TCU). The TCU maintains a batch-queue instruction buffer to manage instruction execution in the system. FHE programs are deterministic: the compiler knows in advance about the flow of instructions and data in memory. This allows the queue to be entirely compiler-managed and fairly short (128 to 1024 instructions, depending on our performance analysis). This results in a much simpler TCU design than for a CPU.

7. BASALISC Processing Elements

BASALISC 1.0's on-chip PEs and their connection to the CTB are shown on the far right in Figure 2. The BASALISC PEs that rely on the CTB for data are the Multiply-Accumulate (MAC) PE (used in ciphertext addition, multiplication, and for kernels of operations such as key switching); the Permutation PE (used to permute data into preferred orders to achieve NTT processing, and also

used for automorphisms); and the NTT PE (used to accomplish number-theoretic transforms efficiently). We describe their capabilities below.

Whereas Figure 2 shows single PE instances, their implementation is a massively multicore architecture that exploits innate parallelism in FHE ciphertext computations. FHE arithmetic in RNS representation offers four types of parallelism: *(i)* over multiple ciphertexts, *(ii)* over the polynomials within a ciphertext, *(iii)* over the residue levels of a polynomial, and *(iv)* over the coefficients of a residue polynomial. Prior work has focused on *(iii)*, instantiating multiple so-called Residue Polynomial Arithmetic Units (RPAUs) [27]–[29]. In contrast, BASALISC focuses on exploiting *(iv)*, due to two key observations. First, the number of residues decreases with the modulus level in the BGV scheme, leading to would-be idle RPAUs as the computation gets closer to bootstrapping. Second, as the lowest level of parallelism, coefficient-level parallelism offers the best opportunity to exploit locality of reference.

7.1. Number-Theoretic Transform PE

Because of the focus on coefficient-level parallelism, BASALISC implements a high-radix NTT PE. We expect that many BASLISC FHE applications will employ ring dimension $N = 65536 = 256^2$ to enable bootstrapping and thus arbitrary-depth computation. Thus, our NTT PE employs a radix-256 butterfly, allowing us to compute 65536-point NTTs with only two round trips to memory for each coefficient. NTTs of smaller sizes can be computed through shortcut paths in our NTT butterfly network.

Following the generalized Cooley-Tukey NTT description of Equation 2, a radix-256 NTT chooses $N_1 = N_2 = 256$. The main arithmetic NTT unit consists of a 256-point NTT (that computes the inner N_1-point NTT and outer N_2-point NTT) followed by 255 post-multipliers (that multiply with the twiddles $\omega_N^{n_2 k_1}$). We employ a standard DIF flow graph for the 256-point NTT, where we replace multiplications $\omega^0 = 1$ with simple pipeline balancing registers. Through this optimization, the $N = 256$-point sub-NTTs are implemented with only 769 modular multipliers, instead of $N/2\log(N) = 1024$.

As discussed in Section 3.2, additional pre- and post-multiplication steps are required to construct negacyclic forward and inverse NTTs from regular NTTs. Because a radix-256 butterfly already includes an array of 255 post-multipliers, it suffices to add 255 pre-multipliers to efficiently support negacyclic NTTs. The result is a 3-stage NTT architecture, as illustrated in Figure 3 for a scaled-down radix-4 unit. In Figure 4, we illustrate how a radix-4 unit is composed to compute the full NTT flow graph in two passes that each take 4 chunks. In between the passes is an implicit memory transposition that we enable with a *conflict-free CTB design*.

Our NTT PE instantiates four parallel 3-stage NTT units. Each unit is deeply pipelined with 40 pipeline stages in order to run at 2 GHz. Together, these four parallel pipes consume 1024 32-bit residue polynomial coefficients at that 2 GHz rate – sufficient to consume all available data bandwidth from the CTB.

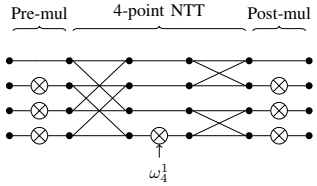

Figure 3: Radix-4 negacyclic NTT unit with pre- and post-multiplier arrays.

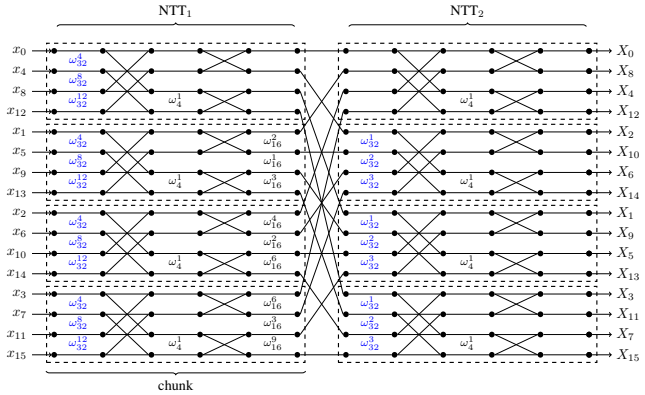

Figure 4: 16-point radix-4 negacyclic NTT flow graph. Extra negacyclic twiddles (in blue) are decomposed into two pre-multiply passes.

7.1.1. Conflict-Free Schedule. A well-known performance inhibitor for NTTs is that successive NTT passes access coefficients at different memory strides, introducing access conflicts in memory. Prior NTT accelerators present custom access patterns and reordering techniques that only work for small-radix NTT architectures [30], [31] or require expensive in-memory transpositions [32]. BASALISC avoids reinventing the wheel, instead building upon years of DSP literature [33]–[35]. The most high-performance FFT accelerators present *conflict-free schedules* [36]–[38] to tackle this exact issue.

Conceptually, a $N = N_1 N_2 = 256^2$-point radix-256 NTT can be represented as a two-dimensional NTT, where the data is laid-out with $N_1 = 256$ rows and $N_2 = 256$ columns. In this format, the inner N_1-point NTT requires coefficients in column-major order, whereas the outer N_2-point NTT requires data in row-major order. The crux of building conflict-free NTT schedules is to structure the data so that it can be read out in either order without bank conflicts. This requires a minimum of 256 independently addressable banks, each containing 2^{16} bank addresses (for a total CTB size of 2^{24} values).

We employ a conflict-free layout based on XOR-permutations [38], as illustrated in Figure 5. In this layout, data with logical address $\{row, col\}$ is stored at $bank = row \oplus col$. This layout ensures that each unique index for every element in every row and column corresponds to a

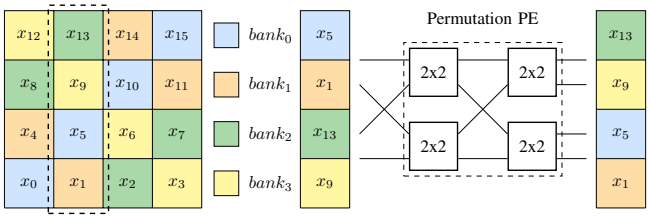

Figure 5: Example conflict-free CTB layout for a 16-point radix-4 NTT. Data is striped using the equation $bank = row \oplus col$, which ensures that both entire columns or entire rows can be read out without bank conflicts. The on-the-fly Permutation PE maps values from *bank order* into *natural order*, as illustrated for access to the second column.

unique physically accessible bank of CTB SRAM.

When reading rows or columns from the CTB, values come out of memory in *bank order*, one value for each bank from bank 0 to 255. However, operations like NTT require values in *natural order*: when accessing a row, we need values sorted by column from 0 to 255, and when accessing a column, we need values sorted by row from 0 to 255. Thus, when accessing row r, we must map bank i to index $i \oplus r$. Likewise, when accessing column c, we must map bank i to index $i \oplus c$.

We build a custom "on-the-fly" Permutation PE to compute these XOR-based permutations as data moves to or from the other PEs. Furthermore, we are the first to observe a remarkable optimization opportunity for this unit. By implementing a slightly more general permutation PE that supports permutations of the form $i \mapsto (i \cdot a + b) \oplus c$, we can not only use the Permutation PE to implement conflict-free XOR permutations, but also any BGV ring automorphism *without additional hardware*. The Permutation PE is described in more detail in Section 7.2.

7.1.2. Twiddle Factor Factory. Similarly to polynomial residue coefficients, twiddle factors in BASALISC are 32-bit integers. There are N twiddle factors for each residue for both forward and inverse NTT, and a maximum of 56 residues at max-capacity key switching, together requiring ∼29.4 MB of twiddle factor material in a naive implementation. Moreover, our four NTT units have 5116 multipliers total that must be fed each cycle with twiddles, requiring massively parallel access into this storage memory. BASALISC prevents this storage requirement in two ways. First, we contribute new insights and a twiddle decomposition method, that reduces the required parallel number of distinct twiddle accesses. Second, we develop a custom *twiddle factor factory* that drastically reduces the number of twiddles stored. In the remainder, we analyze only the forward NTT, but note that identical optimizations apply to the inverted twiddles for the inverse NTT.

For a forward negacyclic NTT, each input x_i is pre-multiplied by the twiddle $\phi^i = \omega_{2N}^i$. Using techniques from the DSP literature [39], we propose to decompose the additional negacyclic twiddles to extract a regular pattern, and to distribute them evenly between the two NTT passes in the flow graph. This is illustrated in Figure 4 by the extra twiddles present in blue. The benefit of this technique is twofold. Firstly, it can be seen that the pre-multiplications become identical for each chunk in both passes. This allows the four NTT units to share the same pre-multiply twiddles, and drastically reduces the total number of pre-multiply twiddles from $N = 256^2$ to $2 \cdot \sqrt{N}$, easily fitting in a small SRAM. Second, the internal butterfly twiddles (powers of ω_{256}) are now a strict subset of the pre-multiply twiddle in the first pass (powers of ω_{512}). Both can therefore be routed from the same small SRAM.

The remaining twiddle factor complexity sits in the post-multiply twiddles. For each chunk k, there are 255 twiddles $\omega_{256^2}^{ik}$. An SRAM storing vectors of 255 twiddles with depth 255 for each residue is still much too large. We propose a technique to reduce the *width* of this SRAM. It can be coupled with techniques that reduce the *depth* of this SRAM, such as On-the-fly-Twiddling (OT) [40]. To reduce the width, we propose a *power generator circuit* that trades SRAM storage for multipliers. The main idea is as follows. By using the identity $\omega_{256^2}^{ik} = \omega_{256^2/k}^{i}$, it can be observed that the required twiddles for chunk k are always the 255 consecutive powers of a *seed* value $\omega = \omega_{256^2/k}$. Using only ω, we can compute its successive powers in a number of multiply layers. The first layer computes ω^2 from ω, with a single multiplier. The second layer takes ω^2 and ω to compute ω^4 and ω^3, and so forth. Every multiplier in the circuit produces a unique value that is used as an output, so the number of multipliers to generate 255 powers from ω is simply 254. Using this technique, instead of storing vectors of 255 twiddles with depth 255 for each residue, it suffices to store the single seeds with depth 255.

7.2. Permutation PE

A pair of Permutation PEs forms the interface between the CTB and the other PEs. Our original contribution is a slightly more generalized Permutation PE that can support both conflict-free schedules required by NTT operations, as well as BGV automorphisms *with the same hardware*. In order to do so, the Permutation PE is generalized to compute permutations of the form $i \mapsto (i \cdot a + b) \oplus c$. Each permutation unit reorders an array of input coefficients to produce a permuted output array of the same length.

The *Read Permutation PE* unscrambles data in conflict-free CTB bank ordering in order to pass it to the other PEs expecting natural ordering. It is a specialized instance of the more general Permutation PE that only implements permutations $i \mapsto i \oplus c$, requiring values $a = 1$ and $b = 0$. The *Write Permutation PE* passes data in the opposite direction. It implements the general permutation $i \mapsto (i \cdot a + b) \oplus c$ in order to re-scramble the data into its conflict-free layout, or to compute ring automorphisms. In the latter case, the output of the Read Permutation PE is fed directly into the input of the Write Permutation PE to achieve the complete operation of the automorphism.

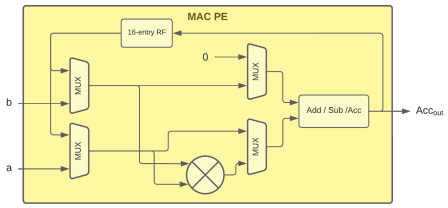

Figure 6: Simplified Diagram of MAC PE architecture.

Each Permutation PE itself is split into two portions. Firstly, the data-permutation portion of the logic is implemented using 2x2 switch nodes placed using an Omega-Network topology. Secondly, a configuration portion takes constants a, b, and c in order to generate the routing pattern for the switches in the network. The configuration portion of the logic attaches the routing pattern to the data and the combined payload word is sent through the network. The switch nodes forward the data according to the least significant bit of the pattern part of the payload data, which is also removed before forwarding. Thus the message is reduced by one bit at each stage of the network, and at the end, the payload only contains the data portion.

7.3. Multiply-Accumulate PE

We realize modular addition and multiplication for FHE in the Multiply-Accumulate (MAC) PE, shown in Figure 6. This pipelined unit can start 2048 32-bit modular addition or modular multiply operations each cycle, if data is available. Because the MAC PE is built with asynchronous logic, it free-runs at 1.6 GHz when not accessing the 1 GHz CTB.

Therefore, operations that read and/or write to the CTB are limited by the 1 GHz CTB bottleneck, while other operations that operate on local data (accumulator register or register file) can accelerate to 1.6 GHz, without using any additional logic. The asynchronous logic provides significant area and latency savings over implementing wide Clock Domain Crossings (CDCs) instead to achieve this 60% performance using a clocked approach. At left in the figure, the 2048 a inputs, each 32-bit in size, come from the CTB. The b inputs are replicated copies of a 32-bit constant from the instruction stream. The MAC includes a 16-entry Register File (RF), shown at top in the figure. In addition, there is a single accumulator register at the output of the adder/subtractor/accumulator unit, shown at right in the figure.

Using the multiplexers shown in the figure, this arrangement can accomplish a variety of functions. Residue chunk multiplication by or addition of a constant to each coefficient can be accomplished at full rate: 2048 32-bit operations per cycle. Multiplication or addition of chunks when both are sourced directly from the CTB can be accomplished at half-rate, using a register to buffer one operand from the CTB, and directly feeding the second operand into the operation from the CTB in a second read cycle. Acceleration of tight kernels that repeatedly process the same chunks can be achieved by storing up to 16 different chunks in the RF and then operating on them at full rate. Finally, the MAC has a multiply-add capability similar to that often found in digital signal processors, allowing double-rate processing: a multiply and accumulate in every cycle. The above possibilities are impacted by the write bandwidth needed to the CTB for results. Write operations might occur as often as for every chunk result, or much less often when the local RF or the accumulator are used to store results during tight kernel operations.

TABLE 5: Area and power comparison of NTT single-butterfly unit with original and optimized Montgomery multipliers at 1 GHz.

Multiplier Design	Area	TDP @0.72V, 125C
Unoptimized	3768 μm^2	7.2 μW
Optimized	2052 μm^2	4.3 μW

A particularly important example of kernel acceleration in the MAC is key switching from Appendix A. Key switching typically makes up the large majority of the workload of an FHE program. The inner loop of our key switching algorithm is the "fast base extension" subroutine from Algorithm 1 that pre-computes a table of about 12 residue polynomials, and then computes many (around 40) different weighted sums of those twelve values, with constant weights. Use of the local registers in the MAC PE and the compound multiply-accumulate function realizes a $44\times$ improvement compared to a naive design. In addition, this approach reduces the use of the CTB during fast base extensions to 10.6% versus nearly 100%, saving 90% of the CTB for use by the other PEs.

7.4. Modular Multiplier Arithmetic Optimization

Both the MAC and NTT Butterfly units use Montgomery modular arithmetic, optimized for NTT-friendly primes [41], and matched to our novel bootstrapping approach. Specifically, instead of supporting the full 32-bit prime value, the multiplier is optimized to only support a subset compatible with our approach, where the lower 17 bits of the prime are fixed (bits 16:1 are tied to 0 and bit 0 is tied to 1). This optimization of the Montgomery modular arithmetic saves 46% in area and 40% in power consumption compared to a generic multiplier that can support all moduli. The results are summarized in Table 5.

8. BASALISC Compilation and Simulation Tools

Figure 7 shows the main components, languages, and intermediate data representations in the BASALISC software toolchain. The dashed boxes in the figure represent our two main software tools: Artemidorus is our compiler, which takes input programs written in a Domain-Specific Language (DSL) and outputs one of our three distinct instruction sets. Simba is our simulator, which takes instruction traces as

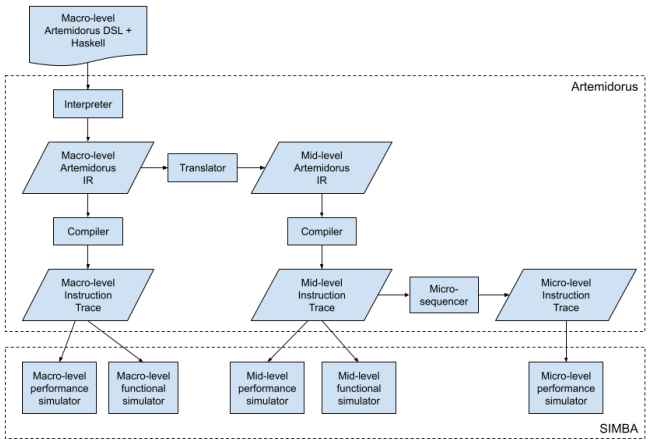

Figure 7: BASALISC Software Toolchain.

input, and produces either a performance report or concrete result values as output.

8.1. Artemidorus

As shown at top left in the figure, our toolchain begins with high-level DSL that allows programmers to create FHE applications for BASALISC to execute, and which features data types including fixed-point numbers, vectors, and matrices. The program passes through several stages in Artemidorus. Bootstrap operations, vector operations, and matrix operations are expanded into BGV primitives; key switching operations are inferred; each operation is tagged with the length of its modulus chain (i.e., the number of prime factors in q), and then expanded into primitive operations on individual residue polynomials for each factor in the modulus; and finally memory regions and registers are allocated and instructions are scheduled.

Artemidorus produces instruction traces at our three levels of the ISA, which pass to Simba for performance or correctness simulation. Especially *Simba-micro*, the micro-level performance simulator shown on the bottom right, presents an integral part of evaluating BASALISC at this point in the design stage. We now describe it in detail.

8.2. Simba-micro

Our micro-level performance simulator employs a step-based operational semantics to model the execution of the BASALISC coprocessor. There are five basic operational components: the CTB, and the four PEs (MAC, Read Permutation, NTT, and Write Permutation). Each of these components operates at a different internal frequency (Table 6). The simulator models a micro-instruction's life cycle from instruction dispatch, to data transfer from CTB to the appropriate functional unit, to proceeding down the pipeline, to the "writeback" phase.

In order to account for the different clock rates of the different components, we use a global "micro-clock" which operates at 6 GHz as the time increment for the model's step function. We made the simplifying assumption that the MAC operates at 1.5 GHz. In this way, we were able to model each PE's progress by causing the CTB to be accessible every 6 micro-cycles, the MAC every 4 micro-cycles, and the permutation/NTT units every 3 micro-cycles. This behavior is modeled by supplying each component with a wait counter which is reset to these values every time it is accessed; the component is only accessible if the counter is 0. If a component is accessible but has no work to do in the given micro-cycle, it simply waits.

Each individual PE is modeled as a pipeline with a certain number of stages and "stage capacity" (number of coefficients that fit in each pipeline stage). The MAC's stage capacity is 2048 coefficients, while the other three have a capacity of 1024. Every time the given PE is enabled (its wait counter is 0), the pipeline advances. When there is a write at the end of the pipeline, it stays at the end of the pipeline until the CTB is available for writing.

Oftentimes, the CTB can be used for either reading data or writing data in a given micro-cycle. This occurs whenever the next instruction reads from the CTB, while there is a write "waiting" at the end of either the MAC or write permutation pipeline (or both). In this scenario, we opted to always favor reads over writes; therefore in our simulations, pipelines tend to fill up. Once a pipeline is full, instruction dispatch is no longer possible to that pipeline, and the control mechanism allows the pending writes to occur. After execution, the following data is reported by Simba-micro: number of CTB cycles (i.e., 6 micro-cycles) of execution, overall CTB utilization (percentage of time spent reading/writing/stalling), and utilization of each PE (how "full" the pipelines are, broken down by % of time).

9. Evaluation of BASALISC

BASALISC is an architecture with an implementation-in-progress but not delivered to silicon yet. We evaluate the architecture and design of BASALISC in diverse ways at this point in the design cycle.

9.1. Physical Realizability

One way to evaluate the design of BASALISC 1.0 and the BASALISC architecture is by a physical design implementation in a practical semiconductor process, with a reasonable target die size and operational frequency target. One resulting evaluation criterion that can be objectively measured using this approach is timing closure – the verification that, with placement and routing of key blocks complete, and using industry best practice estimation of inter-block wire delays based on a mature floorplan, the design achieves a target operating frequency that yields useful levels of performance. In the case of BASALISC 1.0, we completed placement and routing of the novel circuitry - our PEs - and the CTB RAM block. Our operational frequency target was a minimum of 1.0 GHz at the standard "slow-slow" (SS) process corner and a supply voltage of 0.72V in the 12nm low-power Global Foundries process.

TABLE 6: Performance characteristics of BASALISC hardware elements.

Component	f_{max}	Area	TDP @0.72V	Throughput
CTB	1.0 GHz	77.9 mm^2	9 W	2 × 32 Tb/s
MAC PE†	1.6 GHz	7.17 mm^2	18.6 W	102 Tb/s
NTT PE	2.0 GHz	16 mm^2	24.6 W	32 Tb/s
Permuation PE	2.0 GHz	0.16 mm^2	∼0 W	32 Tb/s
PCIe	500 MHz	12 mm^2	5 W	26 GB/s
DDR	800 MHz	18.2 mm^2		51 GB/s
Overall	>1.0 GHz	150 mm^2	57.5 - 115 W	N/A

† Operation of PEs above the frequency of the CTB is advantageous when they can run independently of the CTB.

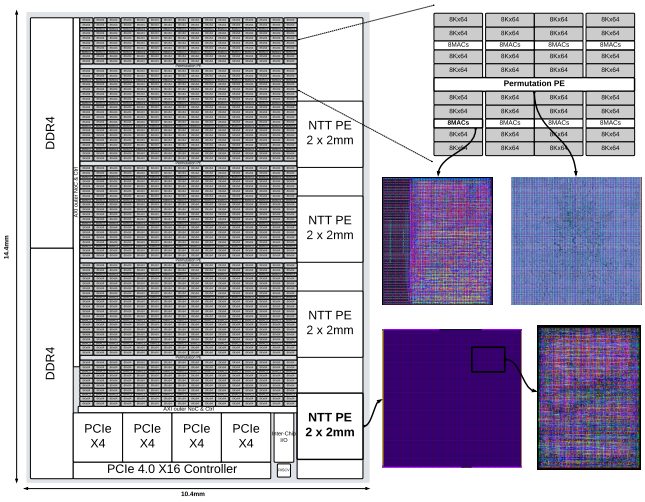

Figure 8: Floorplan of BASALISC 1.0 with all cells placed and intra-block routing complete. Note that the MAC PE and Permutation PE are interleaved within the CTB.

We achieved timing closure for the diverse functional units at the frequencies given in Table 6.

Our floorplan shown in Figure 8 uses actual IP block sizes for DDR4 DRAM, PCIe, our RAM array, and placed and routed PEs. I/O pads are part of the DDR4/PCIe macros, including bumps for power and ground. Interface clock trees run along provisioned routing channels; other big clock trees are avoided through asynchronous design. Everything is drawn assuming 75% density, with a 200μm peripheral gap accounting for process DRC rules, including crackstop, corners, and ESD protection. Our target die size is constrained to 150mm^2 with an aspect ratio of $2:1$ or less [14], which we achieve with a 14.4mm × 10.4mm layout that satisfies both of those constraints.

9.2. Logic Emulation & Formal Verification

A commonplace verification step prior to ASIC manufacturing provides yet another evaluation criterion: successful *hardware emulation* of critical logic in the design. In the case of BASALISC 1.0, that critical logic is the set of processing elements (MAC, permutation, and NNT PEs). We successfully emulated each PE in full, using test vectors extracted from our Verilog testbenches and our formal models of each PE. Each PE passed its emulation test vector suite.

In addition to hardware emulation, BASALISC employs formal methods with two primary goals: first, that the design be proven mathematically correct, and second, that the design be proven consistent at every intermediate representation by demonstrating proof of equivalence. For both the mathematical and consistency proofs, BASALISC employs the Cryptol language [42] and related tools.

In order to satisfy mathematical correctness, top-level FHE algorithms are expressed as a mathematical model in Cryptol. Subsequently, using Cryptol's proof capabilities, the mathematical model is proven to sustain a set of separately-developed correctness definitions. Proof of equivalence is provided through a two-step approach. First, formal equivalence is proven between the high-level mathematical Cryptol model and a low-level logic-oriented Cryptol description using SAW [43]. Next, the low-level Cryptol is converted to Verilog that we prove equivalent to the optimized implementation-Verilog using the commercial Synopsys Formality tool.

TABLE 7: Performance comparison of HElib and BASALISC.

Operation	HElib	BASALISC	Speedup
NTT	27 ms	11 μs	$2.5 \cdot 10^3\times$
Add/Sub	4 ms	8 μs	$5.0 \cdot 10^2\times$
Plaintext mul	159 ms	5 μs	$3.2 \cdot 10^4\times$
Mul (no key switch)	531 ms	20 μs	$2.7 \cdot 10^4\times$
Permutation (no key switch)	12 ms	11 μs	$1.1 \cdot 10^3\times$
Key switching	580 ms	292 μs	$2.0 \cdot 10^3\times$
Bootstrapping	160 s	40 ms	$4.0 \cdot 10^3\times$
Logistic regression	N/A	40.5 s	N/A

9.3. Benchmark Performance Simulation

We evaluate BASALISC 1.0 performance on a set of benchmarks: six micro-benchmarks covering the basic and auxiliary homomorphic operations; and two macro-benchmarks that are respectively a bootstrapping operation and a single iteration of logistic regression training over encrypted data. All results are generated using the example parameter set from Table 1, resulting in ciphertext size 21 MB and key switching size 84 MB. Everything is summarized in Table 7.

9.3.1. Micro-Benchmarks. The first part of Table 7 compares BASALISC performance to HElib for a ciphertext NTT, and some basic and auxiliary homomorphic operations. Each operand is a freshly encrypted ciphertext.

We achieve major speedups for all homomorphic operations. In particular, we accelerate key switching - the most time-intensive kernel - by a factor of $2.0 \cdot 10^3\times$.

9.3.2. Macro-Benchmarks. We estimate execution time for respectively a bootstrapping operation and a single iteration of logistic regression training over encrypted data. Both benchmarks require bootstrapping: we rely on HElib's thin

bootstrapping procedure [18] for encryption of tuples in $\mathbb{Z}_t^\ell$, but adapted to our NTT-friendly approach. The comparison to HElib is summarized in the second part of Table 7.

Thin bootstrapping. We evaluate thin bootstrapping, where we set the parameter from Section 4 to $e = 4$. Using our Simba-micro simulator, bootstrapping consumes only 40ms of execution time. In comparison, HElib takes 160s to bootstrap a single ciphertext with comparable parameters on an Intel Xeon E5-2630 v2 CPU at 2.6 GHz running a single thread. Hence, we achieve a speedup of 4,000 times.

Logistic regression. We estimate execution time on one iteration of secure logistic regression training. We use the algorithm from Chen et al. [44] on a 1,024-sample, 10-feature infant mortality data set from the US Centers for Disease Control. We apply three changes to the original algorithm: we replace the FV scheme by the BGV scheme; we replace sign extraction by an improved variant that has higher precision; and we replace the sigmoid function by the piece-wise linear approximation over $[-63, 64]$ that is shown in Figure 9. These three changes were made for conformity with the DARPA DPRIVE program [14].

The single iteration of logistic regression training includes 513 bootstrapping operations. However, we point out that this application requires a variant of bootstrapping that is heavier than our micro-benchmark, and adapted to fixed-point arithmetic [44]. As a BASALISC instruction trace, the logistic regression is composed of 900K high-level, 800M mid-level, and 27B micro-level instructions. Table 8 shows how the sigmoid is broken down into mid-level BASALISC instructions. Each sigmoid accounts for 2 of the 513 bootstrappings in the entire instruction trace.

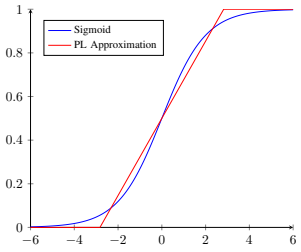

Figure 9: Sigmoid and PL approximation.

Again, we evaluate the resulting micro-level trace using our cycle-accurate simulator Simba-micro. The trace consumes 40.5s of simulated execution time: 3,491 times slower than running the same algorithm on a single core Intel Xeon Silver 4210R CPU at 2.4 GHz *without* using FHE. Comparison to HElib is difficult, because this application uses the fixed-point variant of bootstrapping that is not present in HElib. However, since the benchmark is dominated by bootstrapping operations, we expect a similar speedup of around 4,000 times over software implementations.

TABLE 8: Sigmoid mid-level instructions.

ADD	218,650
SUB	150
MUL	120,948
MORPH	3192
NTT	114,389
INNT	17,954
CONVERT	56,204
FBE	1,787

9.4. Related Work

As part of BASALISC's evaluation, we attempt a comparison to prior and concurrent FHE accelerators. This comparison is complicated, because many architectures do not report bootstrapping benchmarks or simply do not support it [29], [31], [45]–[51]. These architectures support only unrealistically small parameter sets, often allowing them to fully compute on-chip. Furthermore, not all accelerators implement full FHE computations, but rather individual computations such as the NTT. As one outcome, these other approaches require frequent interaction with a host processor to sequence operations and combine results. In this category, HEAWS [51] reports a 5.5× speedup over a software reference for a low-complexity neural network with multiplicative depth 4. HEAX [31] achieves more significant acceleration numbers, in the order of 200× for high-level operations such as key switching, compared to SEAL [8]. Another recent accelerator, Medha [29], shows a 130× speedup for ciphertext multiplication.

Other accelerators that support bootstrapping implement the homomorphic scheme CKKS, and are again complicated to compare with ours. Although CKKS and BGV are very similar at first glance, there are some important lower-level differences. Therefore, an accelerator for one scheme may not necessarily support the other one.

BTS [52] is specially tailored to implement CKKS with a large 374 mm^2 area budget – 2.5× of ours. BTS uses a grid-based microarchitecture that lays out 2,048 PEs as 32 by 64. Conceptually, this architecture is much more complex than the simple vector architecture of BASALISC. Each PE unit has a local SRAM memory, an NTT unit, a "base extension" unit, adders, and multipliers. This incurs a lot of communication between the PEs, so to simplify the data movement management, they treat the entire output of each PE as a "package of coefficients". This restricts the automorphisms that BTS can evaluate to a subset of only half the size of the automorphism group. While sufficient for CKKS, BTS cannot compute all automorphisms used in BGV bootstrapping. We note that BTS uses On-the-fly Twiddling (OT) [40] to store twiddle factors. As we mentioned in Section 7.1.2, this technique can be further combined with our more efficient twiddle factor factory, to drastically reduce the requirements of twiddle factor storage even more. Even at the larger area budget, BTS reports similar speedups to BASALISC. Respectively, in a first benchmark involving ciphertext multiplication and bootstrapping, and a second logistic regression training benchmark, BTS outperforms the Lattigo software library by 2,237× and 1,306×.

The architecture that is closest to ours is F1 [32]. F1 is an ASIC architecture targeting the same die size (150mm^2), technology node (12nm GF), clock frequency (1 GHz), and it also implements bootstrapping. Whereas our macro-benchmark shows that BGV bootstrapping takes 40ms in BASALISC 1.0, F1 [32] reports a bootstrapping time of only 2.4ms. However, these numbers are not comparable: F1 provides lower security (4× smaller ring dimension N)

and supports a plaintext space of only 1 bit with no packing. BASALISC supports plaintext modulus 127^3 with packing capability. We stress that essentially all applications of FHE use packing to speed up the homomorphic evaluation, therefore, not supporting packed bootstrapping severely limits the use cases of F1. BASALISC is the first accelerator to implement packed bootstrapping for BGV directly in hardware. F1 also scales poorly to larger parameter sets: it is optimized for simple BV key-switching, which is less efficient than our hybrid key-switching for high-depth computations [53]. Our asynchronous MAC PE with local RF is key to accelerating hybrid key-switching, resulting in a 44× improvement over running hybrid key-switching on F1's set of PEs.

A key novel aspect of our accelerator is the conflict-free CTB and NTT, with the corresponding Permutation PE that reuses the same hardware for NTT and automorphism. Compared to F1's FFT algorithm, we avoid an expensive matrix transpose unit, by computing the same transpose directly within the CTB with our conflict-free schedule. F1's matrix transpose unit must buffer entire ciphertext polynomials within the NTT PE, which we found prohibitive for our parameter set. Apart from a simpler and more efficient NTT algorithm, BASALISC also features a more performant NTT butterfly. F1 reports 2.27mm^2 for their radix-128 NTT unit at 1 GHz. A single XYZ radix-256 unit measures 4mm^2 – scaling down to 1.75mm^2 for radix-128 – and, thanks to our asynchronous design, runs at 2 GHz. We expect further savings due to our new twiddle factor factory; F1 does not describe how they implement their large twiddle factor SRAM.

Finally, a recent study by De Castro et al. highlighted the memory bottleneck of FHE acceleration [54]. The starting point for their analysis is a CPU-like architecture, where ciphertexts do not fit in the Last-Level Cache (LLC). However, BASALISC's compiler-managed on-chip CTB is very different from a typical LLC. Moreover, at 64 MB, we are able to fit several ciphertexts within the CTB. Nevertheless, we also observe in BASALISC that data movement of *key switching matrices* will often be the practical bottleneck of FHE applications.

10. Conclusion

FHE enables new privacy-preserving applications, but its adoption is limited because of high computational costs. BASALISC accelerates FHE computations by more than three orders of magnitude over CPU performance, and thereby takes a step toward practical feasibility.

In contrast to prior works, BASALISC supports all BGV operations, including fully-packed bootstrapping, in a single ASIC architecture. Our design includes a complete memory hierarchy, and an ISA that supports different levels of abstraction. Moreover, we propose several new hardware improvements: we implement a 32 Tb/s NTT architecture, and show that its permutation unit can be generalized to compute BGV automorphisms without additional area. We also save over 40% in area and power consumption by restricting our multipliers to NTT-friendly primes. We show that this optimization still allows BGV bootstrapping, and therefore does not compromise the generality of our design.

We evaluate the design of BASALISC for correctness and performance. Our functional units are emulated and formally verified to meet their specification. We also simulate performance on two FHE benchmarks, showing at least 4,000 times speedup compared to classical software implementations. BASALISC has been selected as a candidate for future fabrication of an IC that can be applied in real-world applications.

Acknowledgment

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-21-C-0034. The views, opinions, and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. This work was additionally supported in part by CyberSecurity Research Flanders with reference number VR20192203 and the Research Council KU Leuven (C16/15/058). Michiel Van Beirendonck is funded by Research Foundation – Flanders (FWO) as Strategic Basic (SB) PhD fellow (project number 1SD5621N).

References

[1] R. L. Rivest, L. Adleman, M. L. Dertouzos *et al.*, "On data banks and privacy homomorphisms," *Foundations of secure computation*, vol. 4, no. 11, pp. 169–180, 1978.

[2] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, M. Mitzenmacher, Ed. ACM, 2009, pp. 169–178. [Online]. Available: https://doi.org/10.1145/1536414.1536440

[3] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan, "Homomorphic encryption security standard," HomomorphicEncryption.org, Toronto, Canada, Tech. Rep., November 2018.

[4] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

[5] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption over the torus," vol. 33, no. 1, pp. 34–91, Jan. 2020.

[6] C. Bonte, I. Iliashenko, J. Park, H. V. L. Pereira, and N. P. Smart, "Final: Faster fhe instantiated with ntru and lwe," Cryptology ePrint Archive, Report 2022/074, 2022, https://ia.cr/2022/074.

[7] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology – ASIACRYPT 2017*, T. Takagi and T. Peyrin, Eds. Cham: Springer International Publishing, 2017, pp. 409–437.

[8] "Microsoft SEAL (release 4.0)," https://github.com/Microsoft/SEAL, Mar. 2022, microsoft Research, Redmond, WA.

[9] S. Halevi and V. Shoup, "Algorithms in helib," in *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, ser. Lecture Notes in Computer Science, J. A. Garay and R. Gennaro, Eds., vol. 8616. Springer, 2014, pp. 554–571. [Online]. Available: https://doi.org/10.1007/978-3-662-44371-2_31

[10] "Lattigo v3," Online: https://github.com/tuneinsight/lattigo, Apr. 2022, ePFL-LDS, Tune Insight SA.

[11] I. Chillotti, M. Joye, D. Ligier, J.-B. Orfila, and S. Tap, "Concrete: Concrete operates on ciphertexts rapidly by extending tfhe," in *WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, vol. 15, 2020.

[12] Y. Polyakov, K. Rohloff, and G. W. Ryan, "Palisade lattice cryptography library user manual," 2017.

[13] K.-S. Lin, G. Frantz, and R. Simar, "The tms320 family of digital signal processors," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1143–1159, 1987.

[14] T. Rondeau, "Data protection in virtual environments (DPRIVE)," 2020.

[15] N. P. Smart and F. Vercauteren, "Fully homomorphic simd operations," *Designs, codes and cryptography*, vol. 71, no. 1, pp. 57–81, 2014.

[16] C. Gentry, S. Halevi, and N. P. Smart, "Fully homomorphic encryption with polylog overhead," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2012, pp. 465–482.

[17] V. Zucca, "Towards efficient arithmetic for ring-lwe based homomorphic encryption," Ph.D. dissertation, Sorbonne université, 2018.

[18] S. Halevi and V. Shoup, "Bootstrapping for helib," *Journal of Cryptology*, vol. 34, no. 1, pp. 1–44, 2021.

[19] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the aes circuit," in *Annual Cryptology Conference*. Springer, 2012, pp. 850–867.

[20] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, "A full rns variant of fv like somewhat homomorphic encryption schemes," in *International Conference on Selected Areas in Cryptography*. Springer, 2016, pp. 423–442.

[21] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[22] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[23] C. Gentry, S. Halevi, and N. P. Smart, "Better bootstrapping in fully homomorphic encryption," in *International Workshop on Public Key Cryptography*. Springer, 2012, pp. 1–16.

[24] B. Li and D. Micciancio, "On the security of homomorphic encryption on approximate numbers," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2021, pp. 648–677.

[25] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, 2018.

[26] A. Feldmann, N. Samardzic, A. Krastev, S. Devadas, R. Dreslinski, K. Eldefrawy, N. Genise, C. Peikert, and D. Sanchez, "F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption (Extended Version)," *arXiv:2109.05371 [cs]*, Sep. 2021, 13 citations (Semantic Scholar/arXiv) [2022-04-29] arXiv: 2109.05371. [Online]. Available: http://arxiv.org/abs/2109.05371

[27] F. Turan, S. S. Roy, and I. Verbauwhede, "HEAWS: an accelerator for homomorphic encryption on the amazon AWS FPGA," *IEEE Trans. Computers*, vol. 69, no. 8, pp. 1185–1196, 2020. [Online]. Available: https://doi.org/10.1109/TC.2020.2988765

[28] S. S. Roy, A. C. Mert, Aikata, S. Kwon, Y. Shin, and D. Yoo, "Accelerator for computing on encrypted data," *IACR Cryptol. ePrint Arch.*, p. 1555, 2021. [Online]. Available: https://eprint.iacr.org/2021/1555

[29] A. C. Mert, Aikata, S. Kwon, Y. Shin, D. Yoo, Y. Lee, and S. S. Roy, "Medha: Microcoded hardware accelerator for computing on encrypted data," Cryptology ePrint Archive, Report 2022/480, 2022, https://ia.cr/2022/480.

[30] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, "Compact ring-lwe cryptoprocessor," in *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, ser. Lecture Notes in Computer Science, L. Batina and M. Robshaw, Eds., vol. 8731. Springer, 2014, pp. 371–391. [Online]. Available: https://doi.org/10.1007/978-3-662-44709-3_21

[31] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: an architecture for computing on encrypted data," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, J. R. Larus, L. Ceze, and K. Strauss, Eds. ACM, 2020, pp. 1295–1309. [Online]. Available: https://doi.org/10.1145/3373376.3378523

[32] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 238–252.

[33] D. Cohen, "Simplified control of fft hardware," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 24, no. 6, pp. 577–579, 1976.

[34] L. Johnson, "Conflict free memory addressing for dedicated fft hardware," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 39, no. 5, pp. 312–316, 1992.

[35] Y. Ma, "An effective memory addressing scheme for FFT processors," *IEEE Trans. Signal Process.*, vol. 47, no. 3, pp. 907–911, 1999. [Online]. Available: https://doi.org/10.1109/78.747802

[36] P. Tsai and C. Lin, "A generalized conflict-free memory addressing scheme for continuous-flow parallel-processing FFT processors with rescheduling," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 19, no. 12, pp. 2290–2302, 2011. [Online]. Available: https://doi.org/10.1109/TVLSI.2010.2077314

[37] D. I. Reisis and N. Vlassopoulos, "Conflict-free parallel memory accessing techniques for FFT architectures," *IEEE Trans. Circuits Syst. I Regul. Pap.*, vol. 55-I, no. 11, pp. 3438–3447, 2008. [Online]. Available: https://doi.org/10.1109/TCSI.2008.924889

[38] S. Richardson, D. Marković, A. Danowitz, J. Brunhaver, and M. Horowitz, "Building conflict-free fft schedules," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 62, no. 4, pp. 1146–1155, 2015.

[39] M. Garrido, "A new representation of fft algorithms using triangular matrices," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 63, no. 10, pp. 1737–1745, 2016.

[40] S. Kim, W. Jung, J. Park, and J. H. Ahn, "Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus," in *IEEE International Symposium on Workload Characterization, IISWC 2020, Beijing, China, October 27-30, 2020*. IEEE, 2020, pp. 264–275. [Online]. Available: https://doi.org/10.1109/IISWC50251.2020.00033

[41] A. C. Mert, E. Öztürk, and E. Savas, "Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture," in *22nd Euromicro Conference on Digital System Design, DSD 2019, Kallithea, Greece, August 28-30, 2019*. IEEE, 2019, pp. 253–260. [Online]. Available: https://doi.org/10.1109/DSD.2019.00045

[42] J. R. Lewis and B. Martin, "Cryptol: High assurance, retargetable crypto development and validation," in *IEEE Military Communications Conference, 2003. MILCOM 2003.*, vol. 2. IEEE, 2003, pp. 820–825.

[43] K. Carter, A. Foltzer, J. Hendrix, B. Huffman, and A. Tomb, "Saw: The software analysis workbench," in *Proceedings of the 2013 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 15–18. [Online]. Available: https://doi-org.kuleuven.e-bronnen.be/10.1145/2527269.2527277

[44] H. Chen, R. Gilad-Bachrach, K. Han, Z. Huang, A. Jalali, K. Laine, and K. Lauter, "Logistic regression over encrypted data from fully homomorphic encryption," *BMC medical genomics*, vol. 11, no. 4, pp. 3–12, 2018.

[45] T. Pöppelmann, M. Naehrig, A. Putnam, and A. Macias, "Accelerating Homomorphic Evaluation on Reconfigurable Hardware," in *Cryptographic Hardware and Embedded Systems – CHES 2015*, ser. Lecture Notes in Computer Science, T. Güneysu and H. Handschuh, Eds. Berlin, Heidelberg: Springer, 2015, pp. 143–163, 57 citations (Semantic Scholar/DOI) [2022-04-29].

[46] D. D. Chen, N. Mentens, F. Vercauteren, S. S. Roy, R. C. C. Cheung, D. Pao, and I. Verbauwhede, "High-Speed Polynomial Multiplication Architecture for Ring-LWE and SHE Cryptosystems," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 62, no. 1, pp. 157–166, Jan. 2015, 100 citations (Semantic Scholar/DOI) [2022-04-29].

[47] Y. Doröz, E. Öztürk, and B. Sunar, "Accelerating Fully Homomorphic Encryption in Hardware," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1509–1521, Jun. 2015, 72 citations (Semantic Scholar/DOI) [2022-04-29].

[48] S. Sinha Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2019, pp. 387–398, 50 citations (Semantic Scholar/-DOI) [2022-04-29].

[49] S. Sinha Roy, K. Järvinen, J. Vliegen, F. Vercauteren, and I. Verbauwhede, "HEPCloud: An FPGA-Based Multicore Processor for FV Somewhat Homomorphic Function Evaluation," *IEEE Transactions on Computers*, vol. 67, no. 11, pp. 1637–1650, Nov. 2018, 25 citations (Semantic Scholar/DOI) [2022-04-29].

[50] S. Sinha Roy, K. Järvinen, F. Vercauteren, V. Dimitrov, and I. Verbauwhede, "Modular Hardware Architecture for Somewhat Homomorphic Function Evaluation," in *Cryptographic Hardware and Embedded Systems – CHES 2015*, ser. Lecture Notes in Computer Science, T. Güneysu and H. Handschuh, Eds. Berlin, Heidelberg: Springer, 2015, pp. 164–184, 40 citations (Semantic Scholar/DOI) [2022-04-29].

[51] F. Turan, S. S. Roy, and I. Verbauwhede, "HEAWS: An Accelerator for Homomorphic Encryption on the Amazon AWS FPGA," *IEEE Transactions on Computers*, vol. 69, no. 8, pp. 1185–1196, Aug. 2020, 13 citations (Semantic Scholar/DOI) [2022-04-29].

[52] S. Kim, J. Kim, M. J. Kim, W. Jung, M. Rhu, J. Kim, and J. H. Ahn, "BTS: An Accelerator for Bootstrappable Fully Homomorphic Encryption," *arXiv:2112.15479 [cs]*, Dec. 2021, 1 citations (Semantic Scholar/arXiv) [2022-04-29] arXiv: 2112.15479. [Online]. Available: http://arxiv.org/abs/2112.15479

[53] A. Kim, Y. Polyakov, and V. Zucca, "Revisiting homomorphic encryption schemes for finite fields," in *Advances in Cryptology – ASIACRYPT 2021*, M. Tibouchi and H. Wang, Eds. Cham: Springer International Publishing, 2021, pp. 608–639.

[54] L. de Castro, R. Agrawal, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, C. Juvekar, and A. Joshi, "Does Fully Homomorphic Encryption Need Compute Acceleration?" *arXiv:2112.06396 [cs]*, Dec. 2021, 0 citations (Semantic Scholar/arXiv) [2022-04-29] arXiv: 2112.06396. [Online]. Available: http://arxiv.org/abs/2112.06396

Appendix A. Key Switching Procedure

Key switching transforms a ciphertext ct encrypting a plaintext $\boldsymbol{m}$ under a key $\boldsymbol{s}$, into a ciphertext ct' encrypting the same plaintext under a different key $\boldsymbol{s}'$. In practice, we usually switch from $\boldsymbol{s}^2$ (for multiplication) or from $\phi_k(\boldsymbol{s})$ (for automorphism) to the original secret key. We need an auxiliary data structure called *key switching matrix*, which is essentially a set of encryptions of $\boldsymbol{s}$ under $\boldsymbol{s}'$. A high level description of various key switching methods is given in Appendix B by Kim et al. [53]. BASALISC implements the *hybrid key switching* method from Appendix B.2.3, which we review here.

A.1. High-Level Overview

The idea is to provide a key switching matrix that encrypts multiples of $\boldsymbol{s}$ under $\boldsymbol{s}'$. Then given a ciphertext

$$\mathsf{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1) = (-\boldsymbol{c}_1 \cdot \boldsymbol{s} + t\boldsymbol{e} + \boldsymbol{m}, \boldsymbol{c}_1) \in \mathcal{R}_Q^2,$$

we can decompose $\boldsymbol{c}_1$ into digits and multiply by the key switching matrix to obtain an encryption of $\boldsymbol{c}_1 \cdot \boldsymbol{s}$ under $\boldsymbol{s}'$. We add the result to $\boldsymbol{c}_0$ to remove the term $-\boldsymbol{c}_1 \cdot \boldsymbol{s}$, and obtain $\boldsymbol{c}_0' = -\boldsymbol{c}_1' \cdot \boldsymbol{s}' + t\boldsymbol{e}' + \boldsymbol{m}$.

For reasons related to noise growth, key switching is done with respect to the modulus QP, which is larger than the original ciphertext modulus Q. That is, we consider two coprime moduli

$$Q = \prod_{i=1}^{\ell} q_i \qquad \text{and} \qquad P = \prod_{i=\ell+1}^{\ell+k} q_i. \tag{3}$$

Since the input ciphertext is only defined modulo Q, we need a procedure to extend the ciphertext residues from Q to QP. This procedure is known as *fast base extension*, and explained in the next section.

A.2. Fast Base Extension

Algorithm 1 shows the procedure for fast base extension from Q to QP. Given an element $\boldsymbol{a} \in \mathcal{R}_Q$ in polynomial representation, we extend it as follows:

$$\texttt{FastBaseExt}(\boldsymbol{a}, Q, P) = \left(\left[\sum_{i=1}^{\ell} \boldsymbol{b}_i \cdot \frac{Q}{q_i} \right]_{q_j} \right)_{j=\ell+1}^{\ell+k},$$

where

$$\boldsymbol{b}_i = \left[\boldsymbol{a} \cdot \left(\frac{Q}{q_i} \right)^{-1} \right]_{q_i}.$$

The algorithm assumes that

$$Q_i = \frac{Q}{q_i} \qquad \text{and} \qquad Q_i^{-1} = \left(\frac{Q}{q_i} \right)^{-1} \pmod{q_i}$$

are given in precomputed format. Note that the residue of $\boldsymbol{a}$ moduli q_i is denoted by $\boldsymbol{a}^{(i)}$, and that the accumulator on line 5 is implicitly initialized to 0.

Next to the MAC unit, fast base extension also relies on the NTT unit to convert between polynomial and Double-CRT representation. This is necessary because we mix residues defined with respect to distinct moduli, which can only be computed in polynomial format.

Algorithm 1 Fast base extension

Input: $\boldsymbol{a} \in \mathcal{R}_Q$, Q and P
Output: $\boldsymbol{a} \in \mathcal{R}_{QP}$
1: **function** FASTBASEEXT($\boldsymbol{a}, Q, P$)
2: **for** $i \in \{1, \ldots, \ell\}$ **do**
3: $\boldsymbol{r} \leftarrow [\boldsymbol{a}^{(i)} \cdot Q_i^{-1}]_{q_i}$
4: **for** $j \in \{\ell+1, \ldots, \ell+k\}$ **do**
5: $\boldsymbol{a}^{(j)} \leftarrow [\boldsymbol{a}^{(j)} + \boldsymbol{r} \cdot Q_i]_{q_j}$ ▷ MAC unit
6: **end for**
7: **end for**
8: **return** $(\boldsymbol{a}^{(1)}, \ldots, \boldsymbol{a}^{(\ell+k)})$.
9: **end function**

Appendix B. Bootstrapping Details

This appendix includes extra details about our NTT-friendly bootstrapping algorithm. We first give the proof of Lemma 4.1, presented in Section 4. Then we give pseudocode for our new bootstrapping technique. Finally, we explain how the parameters of our method can be chosen in practice.

B.1. Proof of Lemma 4.1

Lemma 4.1. *Let $p > 1$ be a prime number, and let $e > r \geqslant 1$ and $q = 1 \pmod{p^e}$ be sufficiently high parameters. If $(\boldsymbol{c}_0, \boldsymbol{c}_1)$ is a BGV encryption of $\boldsymbol{m}$ with plaintext modulus p^r and ciphertext modulus q, then it can be decrypted by computing*

$$\boldsymbol{c}'_i \leftarrow [p^{e-r}\boldsymbol{c}_i]_q, \quad \boldsymbol{w} \leftarrow [\boldsymbol{c}'_0 + \boldsymbol{c}'_1 \cdot \boldsymbol{s}]_{p^e}, \quad \boldsymbol{m} \leftarrow [\lfloor \boldsymbol{w}/p^{e-r} \rceil]_{p^r}.$$

Here we use $\lfloor \cdot \rceil$ for coefficient-wise rounding to the nearest integer.

Proof. Let $\boldsymbol{u} = \boldsymbol{c}'_0 + \boldsymbol{c}'_1 \cdot \boldsymbol{s}$, then it follows that

$$\boldsymbol{u} = p^{e-r}(\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s}) = p^{e-r}(\boldsymbol{m} + p^r \boldsymbol{e}) \pmod{q},$$

where we have used the definition of $\boldsymbol{c}'_i$ and Equation 1. Now we make the reduction modulo q explicit and write

$$\boldsymbol{u} = p^{e-r}(\boldsymbol{m} + p^r \boldsymbol{e}) + q\boldsymbol{r} \tag{4}$$

for some $\boldsymbol{r} \in \mathcal{R}$. Following the decryption procedure, we have

$$\boldsymbol{w} = [\boldsymbol{u}]_{p^e} = [p^{e-r}(\boldsymbol{m} + p^r \boldsymbol{e}) + q\boldsymbol{r}]_{p^e} = [p^{e-r}\boldsymbol{m} + \boldsymbol{r}]_{p^e},$$

where we have used $q = 1 \pmod{p^e}$. Now we make the reduction modulo p^e explicit and write

$$\boldsymbol{w} = p^{e-r}\boldsymbol{m} + \boldsymbol{r} + p^e \boldsymbol{t}$$

for some $\boldsymbol{t} \in \mathcal{R}$. Again following the decryption procedure, we have

$$[\lfloor \boldsymbol{w}/p^{e-r} \rceil]_{p^r} = [\boldsymbol{m} + \lfloor \boldsymbol{r}/p^{e-r} \rceil]_{p^r} = \boldsymbol{m}$$

where the last equation is correct if the coefficients of $\boldsymbol{r}$ are appropriately upper bounded. Formally, we write this requirement as $||\boldsymbol{r}||_\infty < p^{e-r}/2$, where $||\boldsymbol{r}||_\infty$ denotes the uniform norm on the coefficients of $\boldsymbol{r}$. So we need to find parameters e and q that satisfy this requirement.

Applying the triangle inequality on Equation 4, we have

$$\begin{aligned} ||\boldsymbol{r}||_\infty &\leqslant ||\boldsymbol{u}/q||_\infty + ||p^{e-r}(\boldsymbol{m} + p^r \boldsymbol{e})/q||_\infty \\ &\leqslant ||(\boldsymbol{c}'_0 + \boldsymbol{c}'_1 \cdot \boldsymbol{s})/q||_\infty + ||p^{e-r}\boldsymbol{m}/q||_\infty + \\ &\quad ||p^e \boldsymbol{e}/q||_\infty. \end{aligned} \tag{5}$$

The first term on the right-hand side of Equation 5 can easily be upper bounded, depending on the magnitude of the secret key coefficients. The second term can be made arbitrarily small by choosing q sufficiently large. The third term depends on the remaining noise budget of $(\boldsymbol{c}_0, \boldsymbol{c}_1)$, and can be controlled by invoking bootstrapping early enough. Finally, we choose e as the smallest value such that $p^{e-r}/2$ is larger than the sum of these three upper bounds.

□

B.2. Small Montgomery Reduction

Our NTT-friendly bootstrapping needs one more subroutine that is known as small Montgomery reduction.[4] It was introduced by Bajard et al. [20] and repeated here in Algorithm 2. It takes as input an element $\boldsymbol{a} \in \mathcal{R}_{QP}$ in polynomial representation, and outputs $\boldsymbol{a} \cdot P^{-1} \in \mathcal{R}_Q$ with coefficients reduced modulo a given parameter m. Specifically, the coefficients will be upper bounded by $(1+\epsilon)m/2$ for some $\epsilon \ll 1$ that is not further specified here. We need this subroutine for reduction modulo q and p^e in Lemma 4.1. Note that the algorithm is defined with respect to Q and P as in Equation 3, and that the residue of $\boldsymbol{a}$ moduli q_i is denoted by $\boldsymbol{a}^{(i)}$.

4. This small Montgomery reduction is not directly related to the fact that we use Montgomery multipliers. In fact, we have even specified Algorithm 1 and Algorithm 2 assuming a standard reduction technique. When instantiating either algorithm with Montgomery multipliers, we need to convert all residues out and in Montgomery format whenever we reinterpret a variable modulo q_i as a variable modulo q_j. In both algorithms, this happens on the fifth line.

Algorithm 2 Small Montgomery reduction

Input: $\boldsymbol{a} \in \mathcal{R}_{QP}$, Q, P and m s.t. $||\boldsymbol{a}||_\infty \ll P \cdot m$
Output: $\boldsymbol{b} \in \mathcal{R}_Q$ s.t. $\boldsymbol{b} = \boldsymbol{a} \cdot P^{-1} \pmod m$ and $||\boldsymbol{b}||_\infty \leqslant (1+\epsilon)m/2$

1: **function** SMALLMONT($\boldsymbol{a}$, Q, P, m)
2: **for** $i \in \{\ell + k, \ldots, \ell + 1\}$ **do** ▷ Reverse direction
3: $\boldsymbol{r} \leftarrow [-\boldsymbol{a}^{(i)} \cdot m^{-1}]_{q_i}$
4: **for** $j \in \{1, \ldots, i-1\}$ **do**
5: $\boldsymbol{a}^{(j)} \leftarrow [(\boldsymbol{a}^{(j)} + m \cdot \boldsymbol{r}) \cdot q_i^{-1}]_{q_j}$ ▷ MAC unit
6: **end for**
7: **end for**
8: **return** $(\boldsymbol{a}^{(1)}, \ldots, \boldsymbol{a}^{(\ell)})$
9: **end function**

B.3. NTT-Friendly Bootstrapping

Algorithm 3 gives the pseudocode for our NTT-friendly bootstrapping. This is a direct translation of Lemma 4.1 to the homomorphic domain, including the following steps:

- Line 4 multiplies the input ciphertext by p^{e-r} and an auxiliary modulus b. The auxiliary modulus is introduced for compensation on line 7.
- Line 6 extends the ciphertext modulus from q to $Q \cdot q \cdot b$ using fast base extension from Algorithm 1. This procedure can lead to undesired overflows modulo q, which causes $||\boldsymbol{d}_i||_\infty$ to be greater than $q/2$. This increases the tightness of the bound in Equation 5, but fortunately, the overflows can be compensated in the next step. Finally, note that this step assumes that Q, q and b are pairwise coprime.
- Line 7 compensates for possible overflows modulo q introduced on line 6. The small Montgomery reduction has two side effects: it decreases the modulus from $Q \cdot q \cdot b$ to $Q \cdot q$, and the result gets an additional factor $b^{-1} \pmod q$. The latter was already compensated by the factor b on line 4.
- Line 8 reduces the result modulo p^e and further decreases the modulus from $Q \cdot q$ to q. Now the result gets no additional factor since $q = 1 \pmod{p^e}$. Note that this step is necessary to minimize the noise growth in the next step.
- Line 10 takes the inner product between the ciphertext and the secret key. The secret key is processed homomorphically in the form of a bootstrapping key.
- Line 11 performs homomorphic coefficient-wise rounding. This functionality is the same as in HElib, so we can reuse its implementation. Note that this step dominates execution time in practice.

Finally, we remark that bootstrapping comes in two variants: the ciphertext either encrypts an element from $\mathcal{R}_t$, or a tuple $(m_1, ..., m_\ell) \in \mathbb{Z}_t^\ell$ as explained in Section 2.2.1. The second variant is known as *thin bootstrapping* [18]. Although Algorithm 3 describes the first variant, it can easily be ported to thin bootstrapping. We omit further pseudocode.

Algorithm 3 NTT-friendly bootstrapping

Input: $\mathsf{ct} \in \mathcal{R}_q^2$ and $\mathsf{bsk} \in \mathcal{R}_Q^2$ s.t. $q = 1 \pmod{p^e}$
Output: $\mathsf{ct}' \in \mathcal{R}_Q^2$ s.t. $\mathrm{Dec}(\mathsf{ct}') = \mathrm{Dec}(\mathsf{ct})$

1: **function** BOOTSTRAP($\mathsf{ct}, \mathsf{bsk}$)
2: **for** $i \in \{0, 1\}$ **do** ▷ $\mathsf{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1)$
3: **for** $j \in \{\ell + 1, \ldots, \ell + k\}$ **do**
4: $\boldsymbol{d}_i^{(j)} \leftarrow [\boldsymbol{c}_i^{(j)} \cdot p^{e-r} \cdot b]_{q_j}$
5: **end for**
6: $\boldsymbol{d}_i \leftarrow$ FASTBASEEXT($\boldsymbol{d}_i, q, Q \cdot b$)
7: $\boldsymbol{c}'_i \leftarrow$ SMALLMONT($\boldsymbol{d}_i, Q \cdot q, b, q$) ▷ Mod q
8: $\boldsymbol{c}'_i \leftarrow$ SMALLMONT($\boldsymbol{c}'_i, Q, q, p^e$) ▷ Mod p^e
9: **end for**
10: $\mathsf{ct}' \leftarrow$ ADD(MUL($\mathsf{bsk}, \boldsymbol{c}'_1$), $\boldsymbol{c}'_0$)
11: **return** $\lfloor \mathsf{ct}'/p^{e-r} \rceil$ ▷ Same as in HElib
12: **end function**

B.4. Choice of Parameters

We note a few things about the concrete choice of the parameters e and q:

- The constraint from Equation 5 is in practice determined by the first term. The reason is that we can make the contribution of the second and third term significantly smaller, respectively by taking q sufficiently high and preventing the noise from growing to its maximum level. Hence the concrete values of e and q mainly depend on the secret key distribution and the ring dimension N.
- Our proof describes a very conservative way to choose e and q. However, the complexity of bootstrapping increases heavily with the magnitude of e, so it is beneficial to take it as low as possible. Halevi and Shoup [18] have therefore proposed a statistical analysis on the first term of Equation 5. Leveraging their approach, we can choose e based on a trade-off between time complexity and probability of a bootstrapping failure.
- For a plaintext modulus of 15 bits or less, we can directly take q as an NTT-friendly prime that satisfies the extra constraint $q = 1 \pmod{p^e}$ of Lemma 4.1. For higher precision plaintext spaces, this is not directly possible anymore since the native word size of BASALISC is 32 bits, and the requirement of q to be NTT-friendly would already consume 17 bits. Hence we must apply a brute force or meet-in-the-middle search for an appropriate q that factors into 32-bit NTT-friendly primes. This is a tedious procedure, and better practice is to weaken the constraint of Lemma 4.1 to $\gcd(q, p^e) = 1$. However, then we need to change the last equation in Lemma 4.1 to

$$\boldsymbol{m} \leftarrow [q \cdot \lfloor q^{-1} \cdot \boldsymbol{w}/p^{e-r} \rceil]_{p^r}$$

for $q^{-1} \cdot q = 1 \pmod{p^e}$. We omit the adapted proof and pseudocode.