Implementation and Performance Evaluation of RNS Variants of the BFV Homomorphic Encryption Scheme

Ahmad Al Badawi, Yuriy Polyakov, Khin Mi Mi Aung,
Bharadwaj Veeravalli, and Kurt Rohloff

Abstract

Homomorphic encryption provides the ability to compute on encrypted data without ever decrypting them. Potential applications include aggregating sensitive encrypted data on a cloud environment and computing on the data in the cloud without compromising data privacy. There have been several recent advances resulting in new homomorphic encryption schemes and optimized variants of existing schemes. Two efficient Residue-Number-System variants of the Brakerski-Fan-Vercauteren homomorphic encryption scheme were recently proposed: the Bajard-Eynard-Hasan-Zucca (BEHZ) variant based on integer arithmetic with auxiliary moduli, and the Halevi-Polyakov-Shoup (HPS) variant based on a combination of integer and floating-point arithmetic techniques. We implement and evaluate the performance of both variants in CPU (both single- and multi-threaded settings) and GPU. The most interesting (and also unexpected) result of our performance evaluation is that the HPS variant in practice scales significantly better (typically by 15%-30%) with increase in multiplicative depth of the computation circuit than BEHZ. This implies that the runtime performance and supported circuit depth for HPS will always be better for most practical applications. The comparison of the homomorphic multiplication runtimes for CPU and GPU demonstrates that our best GPU performance results are 3x-33x faster than our best multi-threaded results for a modern server CPU environment. For the multiplicative depth of 98, our fastest GPU implementation performs decryption in 0.5 ms and homomorphic multiplication in 51 ms for 128-bit security settings, which is already practical for cloud environments supporting GPU computations. Our best runtime results are at least two orders of magnitude faster than all previously reported results for any variant of the Brakerski-Fan-Vercauteren scheme.

Index Terms

Secure Computation, Lattice-based Cryptography, Homomorphic Encryption, Residue Number System, BFV SHE, Parallel Processing.

I. Introduction

HOMOMORPHIC encryption (HE) has been a topic of active research since the first design of a Fully Homomorphic Encryption (FHE) scheme by Gentry [16]. FHE allows performing arbitrary secure computations over encrypted data without ever decrypting them. One of the potential applications is to upload encrypted data to a public cloud computing environment and then outsource computations over these data to the cloud without sharing secret keys or compromising data privacy by decrypting the data in the cloud.

At a high level, FHE is based on a Somewhat Homomorphic Encryption (SHE) scheme that provides at least two homomorphic operations: addition and multiplication. The encryption and homomorphic operations require adding some "noise" to guarantee a certain level of security based on the underlying hardness assumption, e.g., Ring Learning With Errors. This noise grows at some controlled rate with each homomorphic operation. As long as the noise is under a certain level, the decryption returns the correct result for a given computation circuit.

The literature includes a number of FHE and SHE schemes that vary in construction, functionality and performance [32], [9], [24], [15], [17], [10]. One of the most promising schemes is the Fan-Vercauteren variant

A. Al Badawi and B.Veeravalli are with the Department of Electrical and Computer Engineering, National University of Singapore, 117576; E-mail: ahmad@u.nus.edu and elebv@nus.edu.sg

Y. Polyakov and K. Rohloff are with the Department of Computer Science and Cybersecurity Research Center, New Jersey Institute of Technology, USA 07102; E-mail: polyakov@njit.edu and rohloff@njit.edu

K. M. M. Aung is with A * STAR, Data Storage Institute (DSI), Singapore 138632; E-mail: mi_mi_aung@dsi.a-star.edu.sg

Manuscript revised June 7, 2018.

of Brakerski's scale-invariant scheme [9], [15], which we refer to as BFV in this paper. The scheme has an elegant/simple structure and provides promising practical performance [23].

One common aspect of many HE schemes is the need to manipulate large algebraic structures with multi-precision coefficients. In BFV, this multi-precision arithmetic is performed for polynomials of large degrees (several thousand) with relatively large integer coefficients (several hundred bits). Implementing the necessary multi-precision modular arithmetic is computationally expensive. A way to make these operations faster is to use the Residue Number System (RNS) to decompose large coefficients into vectors of smaller, native (machine-word-size) integers. RNS allows some arithmetic operations to be performed completely in parallel using native instructions and data types, thus potentially improving the efficiency.

Recently, two RNS variants of the BFV scheme have been proposed to reduce the computational complexity of decryption and homomorphic multiplication[1]: Bajard-Eynard-Hasan-Zucca (BEHZ) [7] and Halevi-Polyakov-Shoup [19]. Both variants share a common design but differ in some aspects such as implementation complexity, effect on noise growth, and performance. The BEHZ variant employs integer arithmetic and RNS techniques to provide an asymptotically better performance over the textbook BFV scheme [15] at the expense of higher noise growth. The HPS variant employs a combination of integer and floating-point arithmetic in addition to RNS techniques. HPS is relatively simpler to implement and has essentially the same noise growth as the BFV scheme [19].

There are multiple publications on software implementations of the BFV scheme. For instance, SEAL is an open source C++ library implementing the BFV scheme and its RNS variant BEHZ [22]. Earlier versions of SEAL used RNS for some operations of the BFV scheme but relied on multi-precision arithmetic for decryption and homomorphic multiplication. The latter require the divide-and-round and base decomposition operations that are hardly compatible with RNS. However, the recent versions of SEAL (> v2.3.0) provide a full RNS implementation of the BEHZ variant [7].

Other implementations of the BFV scheme can be found in PALISADE [26], an open source C++ lattice cryptography library that includes the implementations of multiple HE and proxy re-encryption schemes [27]; digital signature, identity-based encryption, and attribute-based encryption constructions [18], [14]; and conjunction obfuscation scheme [12]. Starting with v1.1, PALISADE provides an implementation of the HPS variant.

The BFV scheme has also become a subject for hardware acceleration studies. For instance, Al Badawi et al. [5] provide a GPU-accelerated implementation of BEHZ. Another recent effort dealt with accelerating the textbook BFV performance using FPGA [31].

In this work we evaluate and compare the practical performance of BEHZ and HPS. Although Halevi et al. provide a theoretical comparison [19], it is not clear from that analysis how the noise growth and performance compare in practical implementations. As seen later in this paper, each variant employs different elementary operations that cannot be compared easily without experiments in the same settings.

A. *Our Contributions*

In this work we implement the BEHZ variant and present an optimized implementation of the HPS scheme in PALISADE. We also implement the HPS variant in GPU.

We examine the RNS techniques of both variants and compare their computational complexity and theoretical noise growth. We also provide recommendations for achieving the best practical performance of both variants.

We evaluate the performance of both variants in CPU and GPU. Our analysis suggests that the HPS decryption and homomorphic multiplication runtimes are typically smaller (up to 30%) than those for BEHZ for most settings.

We discover from the analysis of experimental noise growth that the HPS variant scales significantly better with increase in multiplicative depth of the computation circuit than BEHZ: the depth supported by the HPS variant is typically 15%-30% larger for the same values of parameters. We provide an interpretation of the faster noise growth for the BEHZ variant.

The comparison of the homomorphic multiplication runtimes of the HPS variant for CPU and GPU demonstrates that our best GPU performance results are 3x-33x faster than our best multi-threaded results for a modern server CPU environment. This implies that we improve the prior implementation results for the HPS variant [19] by more than one order of magnitude.

[1]Homomorphic multiplication is known to be the most performance-critical primitive in HE schemes

For the multiplicative depth of 98, we are able to reduce the decryption time to 0.5 ms and homomorphic multiplication to 51 ms for 128-bit security settings, which is already practical for cloud environments supporting GPU computations. Our best runtime results are at least two orders of magnitude faster than all previous results for the BFV scheme in the literature.

B. Organization

The paper is organized as follows: Section II provides some preliminaries on the RNS tools and textbook BFV scheme. Sections III and IV review, analyze, and theoretically compare the decryption and homomorphic multiplication procedures of both RNS variants. Implementation details are discussed in Section V. In Section VI, we benchmark both variants and analyze their performance for different platforms. Finally, Section VII concludes the work and marks out the future work.

II. Preliminaries

A. Cyclotomic Rings

The BFV scheme implemented in our work employs the polynomial ring $R = \mathbb{Z}[X]/(X^n + 1)$, where the ring dimension n is a power of 2. The ring can be viewed as a set of polynomials of degree less than n. The arithmetic in R is always done modulo $(X^n + 1)$. To denote the elements of R, we use small letters, e.g., a. In some BFV primitives, the polynomials are sampled from predefined distributions. We use the symbol $a \xleftarrow{\mathcal{U}} \mathcal{S}$ to refer to uniform sampling of a from the set $\mathcal{S}$, whereas the symbol $a \xleftarrow{\mathcal{G}} \mathcal{S}$ is used to denote sampling from a Gaussian distribution.

The plaintext space in BFV is R_t, where $t \geq 2$ is an integer plaintext modulus. The polynomials in plaintext space are reduced both modulo t and $(X^n + 1)$. A plaintext is normally a single element in R_t encoding the original plaintext message. Likewise, the ciphertext space R_q has $q \gg t$ as the coefficient modulus. For practical implementations, q is usually a k-smooth number, s.t. $q = \prod_{i=1}^{k} p_i$, where p_i is a prime that fits in the underlying machine word. Similar to R_t, polynomials in R_q are reduced modulo q as well. We remark that unlike a plaintext, a ciphertext c is a pair of two elements in R_q, denoted by $(c[0], c[1])$.

B. Residue Number System and Chinese Remainder Theorem

a) Residue Number System: RNS is a non-positional numbering system in which a number is represented by a tuple of residues modulo some predefined pairwise co-prime moduli, known as the RNS base. RNS is used to distribute a computation in some relatively large domain to a set of smaller sub-domains. Computations in sub-domains are completely independent and, therefore, can be performed in parallel. Moreover, the problem size in sub-domains can be highly controlled so that the computation can be done without using multi-precision arithmetic. The results of independent computations in sub-domains can be interpolated via the Chinese Remainder Theorem (CRT) to construct a solution to the problem in the original domain. Although this approach is more complicated and less intuitive, as compared to a direct solution, it typically provides better performance [7], [19].

To use RNS, we need to first define the RNS base $B = \{m_1, \ldots, m_k\}$, where m_i is an integer and $\gcd(m_i, m_j) = 1, \forall i \neq j$. The latter condition is required to guarantee the number system is not redundant and to allow using the CRT for RNS-to-positional-number-system conversion. An integer x can be represented in RNS with base B via the tuple $x = \{x_1, \ldots, x_k\}$, where $x_i = x \pmod{m_i}$, also denoted by $[x_i]_{m_i}$. The quantity $M = \prod_{i=1}^{k} m_i$ is known as the RNS dynamic range. As long as $x < M$, or $\lceil -M/2 \rceil \leq x < \lfloor M/2 \rfloor$ as in our case, there is a unique RNS representation of x.

b) Chinese Remainder Theorem: CRT can be used to do the backward conversion, i.e., converting a number represented in RNS to its equivalent in a positional numbering system. Equation (1) can be used to perform this conversion. We remark that this procedure is not only serial, but also requires multi-precision arithmetic. Equation (1) can be reformulated into two forms as shown in equations (2) and (3). It is straightforward to show that v and v' have the upper bounds of k and $k \cdot \max\{m_i\}$, respectively.

$$[x]_M = \left[\sum_{i=1}^{k} \left[x_i \cdot \left(\frac{M}{m_i} \right)^{-1} \right]_{m_i} \cdot \frac{M}{m_i} \right]_M \tag{1}$$

$$x = \left(\sum_{i=1}^{k} \left[x_i \cdot \left(\frac{M}{m_i} \right)^{-1} \right]_{m_i} \cdot \frac{M}{m_i} \right) - v \cdot M \tag{2}$$

$$x = \left(\sum_{i=1}^{k} x_i \cdot \left[\left(\frac{M}{m_i} \right)^{-1} \right]_{m_i} \cdot \frac{M}{m_i} \right) - v' \cdot M \tag{3}$$

C. Elementary Operations

In complexity analysis, we only focus on two elementary operations: Modular Multiplication (MM) and Floating-Point (FP) operations. We compute the total number of elementary operations in core procedures. We use Single Precision (SP) to refer to elementary operations that fit in one word, and Double Precision (DP) for those that require two words.

D. Efficient RNS Base Extension

As seen later, BFV requires division and rounding in decryption and homomorphic multiplication. Since these operations are hardly compatible with RNS, they are handled by base extension techniques. In this subsection, we introduce two different techniques to extend an RNS base. Both techniques form the bases of the BFV RNS variants.

1) Base Extension via Integer Arithmetic: The first method utilizes Equation (2) as shown in Equation (4).

$$\text{FastBaseConv\_I}(x, B, B') = \left[\sum_{i=1}^{k} \left[x_i \cdot \left(\frac{M}{m_i} \right)^{-1} \right]_{m_i} \cdot \frac{M}{m_i} \bmod m'_j \right]_{m'_j \in B'} \tag{4}$$

The conversion is fast since it does not require multi-precision operations, given that the quantities $M{m_i}^{-1} \pmod{m'_j}, \forall m'_j \in B'$ are precomputed. FastBaseConv_I can be thought of as computing the CRT without reduction modulo the dynamic range M. This means that the converted value may have multiples of M as an overflow, i.e., FastBaseConv_I$(x, B, m'_j) = x + v \cdot M \pmod{m'_j}$. The overflow can be controlled and sometimes eliminated via the techniques described in later sections. FastBaseConv_I forms the basic building block of BEHZ [7].

a) Complexity: FastBaseConv_I requires for each residue two modular multiplications: one modulo $m_i \in B$ and another modulo $m'_j \in B'$. Note that the quantity $\left[x_i \cdot \left(\frac{M}{m_i} \right)^{-1} \right]_{m_i}$ can be temporarily stored and reused. Hence, the overall complexity is $(kk' + k)$ MM.

b) A Practical Note: We note that one may invoke a sequence of multiplications and additions without modulo reduction, deferring the latter to as late as possible. Eventually, a reduction should be applied. The length of the sequence can be determined if the moduli sizes are known in advance, which is typically the case. This optimization is similar to Harvey's lazy reduction [20] used to improve the computation of Number Theoretic Transform (NTT).

2) Base Extension via Integer and Floating-Point Arithmetic: The second method also utilizes Equation (2). However, instead of doing approximate conversion (conversion with overflow), it estimates v and uses the estimated value for exact conversion. One can reformulate Equation (2) to find v as follows:

$$v = \left\lfloor \sum_{i=1}^{k} \frac{1}{m_i} \left[x_i \cdot \left[\left(\frac{M}{m_i} \right)^{-1} \right]_{m_i} \right]_{m_i} \right\rceil$$

Given that v is estimated correctly, we can directly apply Equation (2) modulo m'_j to do exact conversion as follows:

$$\text{FastBaseConv\_F}(x, B, B') = \left[\left(\sum_{i=1}^{k} \left[x_i \left[\left(\frac{M}{m_i} \right)^{-1} \right]_{m_i} \right]_{m_i} \frac{M}{m_i} \right) - v \cdot [M]_{m'_j} \right]_{m'_j \in B'}. \tag{5}$$

The key issue in this method is to ensure that the estimated v is correct. Since v is estimated via floating-point operations, errors due to limited precision may occur and produce an off-by-one value of v. However, such cases can be controlled, detected, and corrected [28], [21], [19]. This base extension method forms the basic building block of HPS [19].

a) Complexity: FastBaseConv_F can be decomposed into two steps: 1) estimating v, and 2) base extension to the new RNS base. Estimating v is done once and requires k MM and $k+1$ FP. On the other hand, base conversion requires $k' \cdot (k+1)$ MM. Overall complexity is: $(kk' + k + k')\,\mathrm{MM} + (k+1)\,\mathrm{FP}$.

b) A Practical Note: While computing v, the numerator can be temporarily stored and reused in the computation of Equation (5).

c) Comparison: It is straightforward to notice that FastBaseConv_I is slightly more efficient than FastBaseConv_F. However, it introduces extra multiples of base B, therefore, a correction procedure needs to be applied afterwards. On the other hand, FastBaseConv_F does not require any correction step given that enough precision is used in estimating v. Our experiments later in this paper show that the correction techniques significantly increase the total runtime.

E. The Textbook BFV

The textbook BFV scheme, described in [15], is a tuple of 5 procedures: key generation, encryption, decryption, homomorphic addition and homomorphic multiplication. The scheme defines a set of parameters as follows:

- λ: security parameter.
- w: a decomposition base used to express a polynomial in R_q in terms of $l+1$ polynomials in base $w \in \mathbb{Z}$, where $l = \lfloor log_w^q \rfloor$.
- $\mathcal{X}_{err}$: a zero-mean discrete Gaussian distribution used to sample error polynomials. The distribution is parameterized by the standard deviation σ and error bound β_{err}.
- $t \geq 2$: a plaintext modulus that determines the plaintext ring R_t.
- $q \gg t$: a ciphertext modulus that determines the ciphertext ring R_q.

The main five procedures of the scheme are as follows:

- **KeyGen**(λ, w): The Secret Key (sk) is a ternary polynomial $sk \xleftarrow{\mathcal{U}} R_2$ taking values from the set $\{-1, 0, 1\}$. The Public Key (pk) is a pair of polynomials $(pk_0, pk_1) = (-[a \cdot sk + e]_q, a)$, where $a \xleftarrow{\mathcal{U}} R_q$ and $e \xleftarrow{\mathcal{G}} \mathcal{X}_{err}$. The Evaluation Key (evk) is a set of $(l+1)$ pairs of polynomials generated as follows: for $0 \leq i \leq l$, sample $a_i \xleftarrow{\mathcal{U}} R_q$ and $e_i \xleftarrow{\mathcal{G}} \mathcal{X}_{err}$. $evk[i] = ([w^i s^2 - (a_i \cdot sk + e_i)]_q, a_i)$. The procedure outputs the tuple: (sk, pk, evk).
- **Enc**(m, pk): takes a plaintext message $m \in R_t$, and samples $u \xleftarrow{\mathcal{U}} R_2$ and $e_1, e_2 \xleftarrow{\mathcal{G}} \mathcal{X}_{err}$. It produces the ciphertext $ct = ([\varDelta m + pk[0]u + e_1]_q, [pk[1]u + e_2]_q)$, where $\varDelta = \lfloor q/t \rfloor$.
- **Dec**(ct, sk): computes $m = \left[\left\lfloor \frac{t}{q}[ct[0] + ct[1]sk]_q \right\rceil\right]_t$.
- **EvalAdd**(ct_0, ct_1): homomorphic addition takes two ciphertexts and produces: $ct_{add} = ([ct_0[0]+ct_1[0]]_q, [ct_0[1]+ct_1[1]]_q)$.
- **EvalMul**(ct_0, ct_1, evk): homomorphic multiplication takes two ciphertexts and performs
  1) Tensoring: compute c_τ, with $\tau \in \{0, 1, 2\}$, such that:

$$c_0 = \left[\left\lfloor \frac{t}{q} ct_0[0]ct_1[0] \right\rceil\right]_q, c_2 = \left[\left\lfloor \frac{t}{q} ct_0[1]ct_1[1] \right\rceil\right]_q,$$

$$c_1 = \left[\left\lfloor \frac{t}{q} (ct_0[0]ct_1[1] + ct_0[1]ct_1[0]) \right\rceil\right]_q.$$

  2) Relinearization:
  2.1) decompose c_2 in base w as $c_2 = \sum_{i=0}^{l} c_2^{(i)} w^i$.
  2.2) return $ct_{mul}[j]$, with $j \in \{0, 1\}$, such that:

$$ct_{mul}[j] = \left[c_j + \sum_{i=0}^{l} evk[i][j] c_2^{(i)}\right]_q.$$

Key generation, encryption and homomorphic addition can be computed easily in RNS. In contrast, decryption and homomorphic multiplication are more involved as they require scaling by the ratio t/q and rounding. The second step in homomorphic multiplication, known as base decomposition, is also hardly compatible with RNS as it needs a conversion to positional representation. The variants studied in our work employ different techniques to overcome these problems.

III. Efficient RNS variants of BFV decryption

A. Scaling in Decryption

Decryption includes scaling by the factor t/q followed by rounding. Both operations require special treatment so that they can be performed directly in RNS. We show below how BEHZ and HPS tackle them.

1) RNS Decryption in BEHZ: BEHZ uses an efficient simple scaling procedure to compute an approximate value of decrypted result. The approximation affects the noise threshold for successful decryption. Note that, in the textbook BFV, decryption works as long as $\|e\|_\infty < (\Delta - |q|_t)/2$, where e is a polynomial representing the noise contained in the ciphertext.

The main objective is to compute $[\lfloor t/q \cdot [x]_q \rceil]_t$, where $x = ct[0] + ct[1]sk$. In BEHZ, exact flooring is used instead of rounding as follows:

$$\left\lfloor \frac{t}{q}[x]_q \right\rfloor = \frac{t[x]_q - |t \cdot x|_q}{q}.$$

This converts the problem into integer division that is compatible with RNS. Since we are computing modulo t, the term $t[x]_q$ cancels out. Then they compute the term $-|t \cdot x|_q/q$ using FastBaseConv_I from base q to t. Since the conversion is not exact, multiples of q may be generated. The BEHZ variant introduces a redundant modulus γ to remove this overflow. Moreover, γ helps in correcting the errors due to the use of flooring instead of rounding. Algorithm 1 shows the RNS decryption function. The reader should note that the notation $|\cdot|_m$ is represented by the standard interval $\{0, \ldots, m-1\}$, whereas $[\cdot]_m$ is represented by the centered interval $\{\lceil -m/2 \rceil, \ldots, \lfloor m/2 \rfloor\}$.

Algorithm 1 Dec_{RNS}: BEHZ RNS decryption [7]

Input: ciphertext ct, secret key sk and a redundant modulus $\gamma \in \mathbb{Z}$.
Output: the plaintext message $[m]_t$.

1: **for** $j \in \{t, \gamma\}$ **do**
2: $\quad s^{(j)} \leftarrow \text{FastBaseConv\_I}(|\gamma t \cdot (ct[0] + ct[1]sk)|_q, q, j) \times |-q^{-1}|_j \pmod{j}$
3: $\tilde{s}^{(\gamma)} \leftarrow [s^{(\gamma)}]_\gamma$
4: $m^{(t)} \leftarrow [(s^{(t)} - \tilde{s}^{(\gamma)}) \times |\gamma^{-1}|_t]_t$
5: return $m^{(t)}$

a) Correctness: Algorithm 1 works as long as $\|e\|_\infty \leq \Delta(\frac{1}{2} - \frac{k}{\gamma}) - \frac{|q|_t}{2}$. To ensure that noise threshold is close to that in the textbook BFV, γ can be chosen much larger than k so that the quantity k/γ approaches 0.

b) Complexity: Algorithm 1 requires the invocation of FastBaseConv_I on two moduli t and γ. It also requires one extra modular multiplication by $|\gamma^{-1}|_t$. The overall complexity is $3k + 3$ MM for each coefficient.

2) RNS Decryption in HPS: HPS employs a simpler decryption variant that does not introduce an auxiliary modulus. It utilizes Equation (3) as follows:

$$\left\lfloor \frac{t}{q}[x]_q \right\rceil = \left[\left\lfloor \left(\sum_{i=1}^{k} x_i \cdot \left[\left(\frac{q}{q_i} \right)^{-1} \right]_{q_i} \cdot \frac{t}{q_i} \right) \right\rceil \right]_t \tag{6}$$

Note that, the term $v' \cdot t$ cancels out due to the reduction modulo t. The procedure computes the summation of residues multiplied with precomputed floating-point constants. For maximum precision, the authors suggest to split the floating-point quantities into integer and fractional parts, i.e., $\left[\left(\frac{q}{q_i} \right)^{-1} \right]_{q_i} \cdot \frac{t}{q_i} = \omega_i + \theta_i$ with $w_i \in \mathbb{Z}_t$ and $\theta_i \in [-1/2, 1/2)$.

a) Correctness: RNS decryption works as long as the rounding errors due to floating-point operations are controlled. This can be done by limiting the size of q_i's, the number of q_i's and the precision used to store θ_i's. Given that θ'_i is a limited-precision approximation of the actual value of θ_i, the approximation error is given by $\epsilon_i = \theta'_i - \theta_i$. If the total approximation error less than $1/4$, the decryption procedure works correctly. Halevi et al. provide a detailed analysis of the approximation error. We provide concrete bounds on the number and size of q_i's.

Let $y = \lfloor \sum_i x_i \theta_i \rceil$ and $y' = \lfloor \sum_i x_i(\theta_i + \epsilon_i) \rceil$, then the total error term $\epsilon = \sum_i x_i \epsilon_i$. Let ν denote the precision used to store θ_i's, i.e., $\epsilon_i < 2^{-\nu}$. We know that $|x_i|_\infty < q_i/2$, hence the total error $\|\epsilon\|_\infty < 2^{-\nu} \sum_i \frac{q_i}{2} < 2^{-\nu-1} k \cdot \max(q_i)$. As we need to ensure that $\|\epsilon\|_\infty < 1/4$, the constraint for q_i can be written as $q_i \le 2^{\nu-1}/k$ for the case when the modular arithmetic is implemented using signed integers. It can be similarly shown that in the case of an unsigned-integer implementation, the constraint changes to $q_i \le 2^{\nu-3}/k$.

There is a trade-off between ν, $\max(q_i)$, and k. A higher precision ν implies costlier computations. The values of k and $\max(q_i)$ limit the highest value of q the implementation can support. For a given ciphertext modulus q, it is better to minimize k as much as possible to reduce the number of NTT invocations. In order to optimize the performance of floating-point operations, we identify four cases, as shown in Table I.

Table I: Common C++ floating-point data types, total size in bits, floating-point precision (ν) in bits, bit size ($\log_2^{q_i}$) and maximum number (K) of supported moduli for HPS decryption (for a signed-integer implementation of modular arithmetic).

C++ data type	size	ν	$log_2^{q_i}$	K
`float`	32	24	16	128
`double`	64	53	44	64
`long double`	80	65	59	32
`double double`	128	113	107	32

We remark that `float`, `double` and `long double` are native data types in C++ 1999 standard and implemented natively (in hardware) in modern systems. On the other hand, `double double` is not supported in hardware, and is usually implemented in software. In our implementation, we utilize Shoup's NTL `quad_float` data type [30]. We also remark that our CPU implementation supports $30 \le log_2^{q_i} \le 60$ bits, and we use an unsigned-integer implementation of modular arithmetic, i.e., we employ `quad_float` only when $log_2^{q_i} > 57$ bits. Our GPU implementation, on the other hand, supports only 30-bit moduli and hence uses only `double`.

b) Complexity: The procedure requires k MM and $(k+1)$ FP operations for each coefficient.

B. Comparison and Evaluation of RNS Decryption

We have seen how each variant handles the scaling problem in decryption. The appealing feature in BEHZ is the use of integer arithmetic, therefore; one does not need to worry about rounding errors as in HPS. However, it has some drawbacks since it requires a redundant modulus which triples the computation since we work in different rings: $\mathbb{Z}_{q_i}$, $\mathbb{Z}_t$, and $\mathbb{Z}_\gamma$. It also affects the noise level for correct decryption due to the use of FastBaseConv_I and flooring instead of rounding. In contrast, HPS does not require an extra modulus nor it affects the noise level. HPS is also more efficient when native precision floating-point operations are used, as seen in Figure 1. The main drawback, however, is the need for high-precision (`double double`) floating-point operations when the moduli size is higher than 57 (59) bits. Table II shows the computational complexity of decryption for each variant.

Table II: Decryption computational complexity for BEHZ and HPS.

BEHZ	HPS
$n(3k+3)$ MM	nk MM + $n(k+1)$ FP

Now, we compare the storage complexity of precomputed constants for both RNS variants. Although this may not be important in CPU implementations, it is crucial for GPU. The reason is that we use the constant memory in GPU to store any precomputed quantities in our implementation. The GPU constant memory is very fast but limited in size. Current devices typically include 64 KB of constant memory [25]. In addition, the size of any allocated buffer in constant memory must be defined at compile time. Hence, we limit our GPU implementations to 64 CRT moduli. Table III lists the precomputed quantities and their sizes for the decryption in each RNS variant.

We see that in most cases HPS requires less memory for precomputed constants. However, if `double double` is used to store the floating-point quantities, then both procedures have the same storage complexity.

In order to show the effect of using a different floating-point precision on performance, Figure 1 shows decryption runtimes for the CPU implementations at various CRT moduli sizes. Three main ranges of CRT moduli sizes are identified for our CPU implementation of HPS: 1) < 45, 2) $45-57$, and 3) $58-60$ bits. HPS decryption is faster for

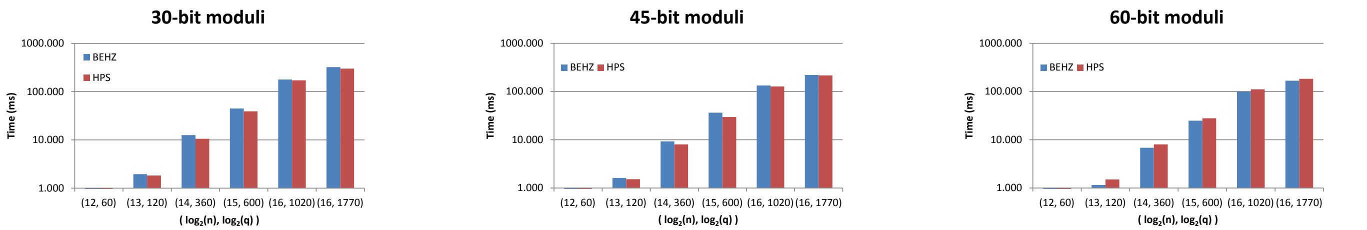

Figure 1: Effect of floating-point precision on decryption performance in PALISADE. C++ data types `double`*,* `long double`*, and NTL* `double double` *are used with 30-bit, 45-bit, and 60-bit moduli, respectively. The horizontal axis represents* $\log_2$ *of polynomial degree* (n) *and ciphertext modulus* (q)*. Note that the vertical axes are in log-scale.*

Table III: Decryption precomputed constants in both RNS variants. Size is given in terms of number of words and K *denotes the maximum number of CRT moduli supported.* $\{\cdot\}$ *denotes a fraction of* $\cdot$

Quantity	# of words	**Required by BEHZ**	**Required by HPS**
$t \cdot \gamma \cdot [(q/q_i)^{-1}]_{q_i}$	K	✓	-
$q/q_i \mod t$	K	✓	-
$q/q_i \mod \gamma$	K	✓	-
$\lfloor \frac{t \cdot [(q/q_i)^{-1}]_{q_i}}{q_i} \rfloor$	K	-	✓
$\{ \frac{t \cdot [(q/q_i)^{-1}]_{q_i}}{q_i} \}$	K	-	✓

the CRT moduli range of $30-57$ bits. However, for larger moduli sizes, the HPS performance degrades due to the use of NTL `double double`, and BEHZ becomes faster. A logical observation is that the HPS performance is faster as long as native floating-precision data types are used. We remark that the choice of parameters corresponds to at least 128 bits of security. The parameter selection is discussed in more detail in Section VI.

IV. Efficient RNS variants of BFV homomorphic multiplication

A. Scaling in Homomorphic Multiplication

Homomorphic multiplication is more complex than decryption and includes two main steps: (1) multiplication itself and (2) relinearization.

Step 1 in multiplication requires tensoring the input ciphertexts and scaling by the factor t/q, followed by rounding. Unlike decryption, the result should also be in base q, therefore, an RNS scaling algorithm is needed. Tensoring requires lifting the ciphertexts first from base q to a larger base. For that reason, an auxiliary RNS base q' with k' moduli is introduced, which is roughly as big as q. More concretely, $q \cdot q'$ should be large enough to include the largest coefficients of tensored ciphertexts without any modular reduction. Lifting can be done using the RNS base extension techniques described previously. Next, the extended tensored ciphertexts are scaled down by q, which represents a subset of the moduli in the extended RNS base $\{q \cup q'\}$. An RNS scaling algorithm can be used for this purpose. Note that the scaled-down result is represented in base q', therefore, another round of RNS base extension should be carried out to retain the result in base q.

In Step 2, relinearization is performed to reduce the ciphertext size. For relinearization, Bajard et al. [7] suggested to use the RNS representation of ciphertexts in base q instead of the standard digit decomposition procedure using the positional base ω. The idea is that q_i's and ω are of the same size in practice; therefore, they can control the noise growth (due to relinearization) similarly. As ciphertexts are already represented in the RNS base q, the decomposition is free. However, the evaluation key must be modified to be compatible with RNS relinearization. This can be done at the initialization phase without additional cost. We remark that HPS employs a similar approach for this step, but with minor optimizations.

The above logic is used as the blueprint for RNS homomorphic multiplication. Both variants follow this blueprint but rely on different RNS base extension techniques.

Input: $c_\tau = \{[c_\tau]_q, [c_\tau]_{q' \cup m_{sk}}\}$ in extended base $q \cup q' \cup m_{sk}$, and plaintext modulus t

Output: $[c^*_\tau]_{q' \cup m_{sk}}$, with $c^*_\tau = \left\lfloor \frac{t}{q} c' \right\rceil + b_\tau$ in $q' \cup m_{sk}$, where $\|b_\tau\|_\infty \leq k$, and $\tau \in \{0, 1, 2\}$

	Operations in base q		Operations in base $q' \cup m_{sk}$
0:	$[c_\tau]_q$		$[c_\tau]_{q' \cup m_{sk}}$
1:	$t \cdot [c_\tau]_q$		$t \cdot [c_\tau]_{q' \cup m_{sk}}$
2:	FastBaseConv_I($[t \cdot c_\tau]_q, q, q' \cup m_{sk}$)	$\rightarrow$	$[\lvert t \cdot c_\tau \rvert_q + v_\tau q]_{q' \cup m_{sk}}$
3:	$-$		$\dfrac{[t \cdot c_\tau]_{q' \cup m_{sk}} - (\lvert t \cdot c_\tau \rvert_q + v_\tau q)}{q}$

Figure 2: Fast RNS flooring in BEHZ

Algorithm 2 SmMRq$_{\tilde{m}}$: Small Montgomery Reduction mod q [7]

Input: polynomial $c'' = [\tilde{m}c]_q + v \cdot q$ in $q' \cup \{m_{sk}, \tilde{m}\}$

Output: c' in $q' \cup m_{sk}$, with $c' \equiv c''\tilde{m}^{-1} \pmod q$, $\|c'\|_\infty \leq \dfrac{\|c''\|_\infty}{\tilde{m}} + \dfrac{q}{2}$.

$r_{\tilde{m}} \leftarrow [-c''_{\tilde{m}} \cdot q^{-1}]_{\tilde{m}}$
for $j \in q' \cup m_{sk}$ **do**
  $c'_j \leftarrow |(c''_j + qr_{\tilde{m}})\tilde{m}^{-1}|_j$
return c' in $q' \cup m_{sk}$

1) Lift-and-Scale in BEHZ: Lift-and-scale in BEHZ is performed in 5 steps as follows:

1) Base extension from base q to base $q' \cup \{m_{sk}, \tilde{m}\}$: FastBaseConv_I($ct_i[j], q, q' \cup \{m_{sk}, \tilde{m}\}$), $\forall\ 0 \leq i, j < 2$ is used to do the conversion efficiently. Since this may generate additional multiples of q (referred to as q-overflows), namely $v \cdot q$, the redundant modulus $\tilde{m}$ is introduced for correction. Note that the correction procedure requires the input ciphertexts to be multiplied by $\tilde{m}$ before the fast base conversion.
2) q-overflow correction: This procedure is known as the small Montgomery reduction (shown in Algorithm 2). Basically, the $\tilde{m}$ residue is used to find v. Then we subtract $v \cdot q \pmod{q'_i}$ from $q' \cup m_{sk}$ residues. The result is a corrected extended RNS representation but with an extra term affecting the noise growth. Note that the choice of $\tilde{m}$ is not arbitrary. It should satisfy Equation (7). In terms of complexity, Algorithm 2 requires $2(k'+1)+1$ MM for each coefficient.
$$\tilde{m}\rho \geq 2k + 1. \tag{7}$$
We remark that in our implementation, we use a standard CRT modulus for $\tilde{m}$, i.e., similar in size to the CRT moduli in q and q'. Therefore, ρ tends to approach 0. As will be shown later, ρ is used to estimate the noise growth due to the fast conversion.
3) Computation of tensor product: At this point, we have the ciphertexts in base $\{q \cup q' \cup m_{sk}\}$. Hence we can perform polynomial multiplication in $\mathbb{Z}_{q \cdot q' \cdot m_{sk}}/(X^n+1)$. We remark that $2(k+k'+1)$ NTT and $(k+k'+1)$ INTT invocations are required for each polynomial multiplication. Namely, we need 4 polynomial multiplications but they can be reduced to 3 using the Karatsuba algorithm. Note that $\|c_\tau\|_\infty \leq \delta \dfrac{q^2}{2}(1+\rho)^2$, where $\tau \in \{0, 1, 2\}$ and $\delta = sup\|a \cdot b\|_\infty / (\|a\|_\infty \cdot \|b\|_\infty)$ is known as the ring expansion factor. Following a conservative (worst-case) approach, δ is set to n; however, in practical settings δ can often be set to $2\sqrt{n}$ if the requirements for the Central Limit Theorem are met [19].
4) Approximate rounding in base q': This step is used to scale down c_τ by the factor t/q. This gives us flooring instead of rounding, i.e., $\lfloor \dfrac{t}{q} c_\tau \rfloor$. Although this produces approximate results, Bajard et al. show that the error due to approximation is very small. This procedure works as shown in Figure 2. In terms of complexity, the procedure is invoked 3 times. In each invocation, FastBaseConv_I is invoked $(k'+1)$ times. The multiplication

by t incurs $(k + k')$ modular multiplications. Lastly, $(k' + 1)$ modular multiplications are required for the flooring. Note that the above is repeated for each coefficient.

5) Base extension from base q' to base q: This step is similar to step (1) above to convert to the original RNS base. The extra modulus m_{sk} is used to correct the overflows generated by FastBaseConv_I. This is done via Shenoy-Kumaresan exact base extension [29].

a) Remarks: The choice of $\rho, \tilde{m}, m_{sk}$, and γ is not quite straightforward. For instance, one may choose $\rho \approx 2k$ to avoid the usage of Algorithm 2. However, this can only be applied to a limited number of circuits where the multiplicative depth is small. On the other hand, higher values of ρ proportionally increase the noise growth and may reduce the multiplicative depth. Another aspect one needs to consider is that some choices of $\tilde{m}$ may require very large values of m_{sk} and γ, which may not fit in the machine word size. In our implementation, we choose $\tilde{m}$, m_{sk}, and γ as regular CRT moduli, i.e., close in size to q_i's.

2) Lift-and-Scale in HPS: HPS lift-and-scale follows the same blueprint as BEHZ. However, it is much simpler and requires no correction tools. The reason is that base extension is exact and does not generate extra multiples of the dynamic range. Therefore, the small Montgomery reduction algorithm is not required. Likewise, the Shenoy-Kumaresan CRT extension is not necessary in step 5. One only needs to worry about the rounding errors due to floating-point operations. Below we provide an overview of the lift-and-scale in HPS.

Lift-and-scale in HPS is performed in 4 steps as follows:

1) Base extension from base q to q': FastBaseConv_F $(ct_i[j], q, q'), \forall\ 0 \leq i, j < 2$ is used to lift the ciphertexts to a larger RNS base q'.
2) Computation of tensor product: This step is similar to step 3 in BEHZ. We remark that the number of NTTs in this step is less by 7, as compared to BEHZ, since there is no redundant modulus m_{sk}.
3) Exact rounding in base q': In this step, c_τ is scaled down by the factor t/q. This can be done by a procedure similar to the one used in HPS decryption. c_τ is bounded by $qq'/2t^2$. The scaling is done by replacing t and q in Equation (6) by tq' and qq' respectively. Applying Equation (6) gives us $[\lfloor tq'/qq' \cdot c_\tau \rceil]_{q'}$. Since c_τ is bounded by $qq'/2t$, then $\lfloor tq'/qq' \cdot c_\tau \rceil \in [-q'/2, q'/2)$. The complete procedure is given in Equation (8), where x is a single coefficient in c_τ. Note that the quantity $\dfrac{1}{q_i} \cdot tq' \big[\big(\dfrac{qq'}{q_i}\big)^{-1} \big]_{q_i}$ is precomputed and broken into integral and fractional parts. The same applies to $\left[t\big(\dfrac{qq'}{q'_j}\big)^{-1} \cdot \dfrac{q'}{q'_j} \right]_{q'_j}$. In terms of complexity, this procedure requires $(k$ + 1) FP + $k'(k + 1)$ MM for each coefficient.

$$[\lfloor t/q \cdot x \rceil]_{q'_j} = \left[\left\lfloor \sum_{i=1}^{k} x_i \cdot \frac{1}{q_i} \cdot tq' \big[\big(\frac{qq'}{q_i}\big)^{-1} \big]_{q_i} \right\rceil + x'_j \cdot \left[t\big(\frac{qq'}{q'_j}\big)^{-1} \cdot \frac{q'}{q'_j} \right]_{q'_j} \right]_{q'_j} \tag{8}$$

4) Exact base extension from base q' to base q using FastBaseConv_F. Note that there is no extra correction step here.

B. Comparison and Evaluation of RNS Lift-and-Scale

a) Complexity: It is not straightforward to compare the computational complexity between the variants. The employed tools use different elementary operations. However, we provide below an estimated analysis of the computational complexity, leaving the more precise empirical analysis for Section VI.

We only evaluate the lift-and-scale complexity as both variants use the same relinearization procedure. Table IV summarizes the computational complexity of the RNS and tensoring operations in the lift-and-scale procedure for BEHZ and HPS. We assume that the Karatsuba multiplication algorithm is used in tensoring, hence the factor 3 instead of 4 in tensoring, RNS rounding, and FastBaseConv(x', q', q). Note that FastBaseConv(x', q', q) in BEHZ includes the computational complexity of the Shenoy-Kumaresan algorithm as well.

Table IV suggests that the number of modular multiplications for HPS is less by $n(14k + 10k' + 24)$ and the number of NTTs is less by 7. However, HPS has a floating-point cost of $n(7k + 3k' + 10)$ operations. We discuss the effect of these differences on homomorphic multiplication runtime in Section VI.

[2]Note that t in the denominator is due to plaintext scaling in encryption.

Table IV: The computational complexity of the RNS tools in lift-and-scale (assuming the Karatsuba multiplication technique is used in tensoring).

Proc	BEHZ	HPS
FastBaseConv(x, q, q')	$4nk(k'+2)$ MM	$4n(k(k'+1)+k')$ MM + $4n(k+1)$ FP
SmMRq$_{\tilde{m}}$	$4n(2(k'+1)+k+1)$ MM	–
Tensoring	$7(k+k'+1)$ NTT + $3n(k+k'+1)$ MM	$7(k+k')$ NTT + $3n(k+k')$ MM
RNS Rounding	$3n(k'(k+2)+2(k+1))$ MM	$3nk'(k+1)$ MM + $3n(k+1)$ FP
FastBaseConv(x', q', q)	$3n(k(k'+1)+2k'+1)$ MM	$3n(k'(1+k)+k)$ MM + $3n(k'+1)$ FP
Total	$n(10k'k+24k+23k'+24)$ MM + $7(k+k'+1)$ NTT	$n(10k'k+10k+13k')$ MM + $7(k+k')$ NTT + $n(7k+3k'+10)$ FP

b) Effect on Noise Growth: Noise growth can be analyzed using the same logic as applied to the textbook BFV in [23] and YASHE in [8]. Both the BEHZ and HPS papers provide a detailed noise analysis. We only review some closed-form solutions that describe the noise growth in each variant.

Noise growth in BEHZ [7]: Initial noise in a fresh ciphertext is $V = \beta_{err}(1+2\delta\beta_{key})$. To ensure the correctness of a depth-L binary tree multiplication, the maximum noise $C_1^L V + LC_1^{L-1}C_2$ must be less than $\frac{q}{t}(\frac{1}{2}-\frac{k}{\gamma}) - \frac{|q|_t}{2}$, where

$$
\begin{aligned}
C_1 &= \delta^2 t(1+\rho)\beta_{key} + \delta t(4+\rho) + \frac{\delta}{2},\\
C_2 &= (1+\delta\beta_{key})(\delta t|q|_t\frac{(1+\rho)}{2} + \delta\beta_{key}(k+\frac{1}{2}))+\\
&2\delta t|q|_t + k(2\delta\beta_{err}l_{\omega,2^\nu}\omega+1) + \frac{1}{2}(3+|q|_t).
\end{aligned}
$$

Here, $l_{\omega,2^\nu}$ is the number of base-w digits in a CRT modulus q_i (for the second level of decomposition in relinearization).

Noise growth in HPS [19]: To guarantee the correctness of a depth-L binary tree multiplication for HPS, the maximum noise ${C'}_1^L V + L{C'}_1^{L-1}{C'}_2$ must be less than $(\Delta - r_t(q))/4$, where $r_t(q) = t\,(q/t - \Delta)$ and

$$
\begin{aligned}
{C'}_1 &= (1+\frac{5}{\delta\beta_{key}})\delta^2 t\beta_{key},\\
{C'}_2 &= \delta^2\beta_{key}((1+2||\epsilon_s||_\infty)\beta_{key}+t^2) + \delta\beta_{err}l_{\omega,2^\nu}\omega k.
\end{aligned}
$$

This constraint is similar to the textbook BFV case [23]; HPS adds at most two extra bits to the textbook BFV constraint, as shown in [19].

Noise Growth Comparison: The most significant quantity in noise growth is C_1 (or C'_1 for HPS). In BEHZ, an extra factor of $(1+\rho)$ is introduced, which can be minimized if ρ is small. This can be controlled by choosing a large $\tilde{m}$ using Equation (7). Note that $\beta_{key} = 1$. We use the noise growth bounds above to find the maximum multiplicative depth $L_\circ$ supported under a given parameter set as shown in Table V. Note that in this experiment, we set $\delta = n$, which corresponds to the worst-case analysis. It can be seen that the BEHZ and HPS RNS techniques have almost no effect on $L_\circ$ for these settings. Note that the behavior in BEHZ is attributed to our choice of the redundant moduli, which is different from the parameter selection in [7]. We use standard CRT moduli for all redundant moduli to minimize the effect of the moduli on noise growth. We remark that the practical noise growths observed in our experiments are significantly different, as discussed later in this paper.

c) Precomputed Constants: The number of precomputed constants for homomorphic multiplication is quite large. Due to space constraints, we only list the numbers of vectors and matrices required by each variant in Table VI. Note that our GPU implementations include other precomputed constants related to CRT and NTT computation. We maximized K to use the largest amount of the GPU constant memory. It can be seen that BEHZ includes a larger number of parameters but requires less storage.

Table V: *Parameters of the BFV scheme and its RNS variants with plaintext modulus t, and maximum theoretical multiplicative depth $L_\circ$ for each variant. Parameters are generated to provide at least 128 bits of security for 30-bit moduli, with the worst-case bound for the expansion factor $\delta = n$.*

log_2^n	$\lceil log_2^q \rceil$	t	$L_\circ$		
			BEHZ	HPS	Textbook
12	60	2 65537	1 0	1 0	1 0
13	120	2 65537	3 2	3 2	3 2
14	360	2 65537	11 7	11 7	11 7
15	600	2 65537	18 12	18 12	18 12
16	1020	2 65537	29 20	29 20	29 20
16	1770	2 65537	52 36	52 36	52 36

V. Implementation

A. CPU Implementation

We implemented the BEHZ variant in the PALISADE library, which already provides an implementation of HPS (starting with version 1.1). We added the parameter generation, decryption, and homomorphic multiplication and RNS tools for the BEHZ variant. Other primitives, such as key generation, encryption, and homomorphic addition, were borrowed from the existing HPS implementation. The details of the HPS implementation are provided in [19]. Our implementation of the BEHZ variant is publicly accessible (included in PALISADE starting with version 1.2).

Table VI: *Memory requirement for precomputed constants used in homomorphic multiplication. K denotes the largest number of CRT moduli supported by the GPU implementation. Memory size is given in KB.*

Item	BEHZ	HPS
K	64	61
# of vectors size	12 3 KB	7 1.67 KB
# of matrices size	2 32 KB	3 43.61 KB
total size	35 KB	45.27 KB

Multi-threading in our CPU implementations is achieved via OpenMP[3]. The loop parallelization in the scaling and RNS base extension operations is applied at the level of single-precision polynomial coefficients (w.r.t. n). The loop parallelization for NTT and component-wise vector multiplications (polynomial multiplication in the evaluation representation) is applied at the level of CRT moduli (w.r.t. k).

We also added a new optimization in this work to improve the performance of both BEHZ and HPS in PALISADE:

a) Lazy Reduction: Lazy reduction can be used to reduce the number of modular reductions in a sum of products. Suppose we have a summation $\sum_i a \cdot b \pmod p$. If we can determine the upper bound of $a \cdot b$, we may use regular multiplications and additions as long as the bound is not reached. As soon as we approach the bound, we can apply one modular reduction modulo p and continue. In PALISADE, the largest efficient CRT modulus is 60 bits long, i.e., a and b are also 60-bit numbers. If we use a 128-bit multiplier, we can do 128 multiply-add (MULADD) operations before we overflow a 128-bit register. Thus, a single modular reduction after 128 MULADDs is sufficient to compute the sum. To use lazy reduction, we extended PALISADE by adding two

[3]http://www.openmp.org/

Algorithm 3 128-bit Barrett reduction modulo 60-bit integer.

Input: 128-bit $a \in \mathbb{Z}^+$, 60-bit $p \in \mathbb{Z}^+$, and $\mu = \lfloor 2^{128}/p \rfloor$.
Output: $r = a \pmod p$.

l_{hi} = Mul128$(a_{lo}, \mu_{lo}) \gg 64$
$m =$ Mul128(a_{lo}, μ_{hi})
$(s_0, c_{out}) =$ AddwCarry(m_{lo}, l_{hi}) ▷ addition with carry
$s_1 = m_{hi} + c_{out}$
$m =$ Mul128(a_{hi}, μ_{lo})
$(NULL, c_{out}) =$ AddwCarry(m_{lo}, s_0) ▷ check carry
$l_{hi} = m_{hi} + c_{out}$
$s_0 = a_{hi} * \mu_{hi} + s_1 + l_{hi}$
$r = a_{lo} - s_0 * p$
while $(r >= p)$ **do**
    r -= p
return r

procedures: a 64-bit–by–64-bit multiplier to produce a 128-bit result, and a 128-bit modular reducer using Barrett algorithm. Although we replace a 64-bit modular MULADD with a 128-bit MULADD, experiments show that the latter is more efficient. Algorithm 3 shows our 128-bit Barrett reduction algorithm.

B. GPU Implementation

The GPU library, referred to as DSI_BFV, already includes a full implementation of BEHZ. We report here some of the recent features added to DSI_BFV and details of our implementation of HPS. For further details, the reader is referred to [5].

DSI_BFV includes two main components: 1) lattice cryptography library and 2) implementation of BEHZ. We should note that a key feature in DSI_BFV is that it executes the entire BFV computation on GPU. The CPU is merely responsible for launching kernels and memory allocations/deallocations. Even the cryptographic keys are computed on GPU. This is slightly different from the normal processor-coprocessor model where only intensive tasks are offloaded to coprocessor. As we saw previously, BFV and its RNS variants include a large level of parallelism that is suitable for vector processors such as GPUs. We avoid frequent costly memory copying between CPU and GPU by performing the entire computation on GPU. This paradigm is also suitable for cluster GPUs where the CPU may distribute a large homomorphic circuit to multiple GPUs.

The lattice library includes a set of tools described as follows.

a) CRT/RNS: Currently, for CRT/RNS, DSI_BFV includes 30-bit moduli generated in a special form to support lazy reductions in NTT computations [20]. It also employs fixed-size primes generated according to the design in NFLlib [3]. For CRT reconstruction, DSI_BFV uses Garner's mixed radix algorithm since it requires less memory for precomputed constants and is faster than the classic CRT reconstruction [5]. For RNS arithmetic, we launch $k \cdot n$ threads over 2D thread blocks to exploit the maximum parallelism provided by RNS, which is a scalable solution that can benefit from as many computational cores as are available in the GPU.

b) Discrete Galois Transform (DGT): The DGT is used for efficient polynomial multiplication using negacyclic convolution. It was found suitable for GPUs and memory-bound platforms as it cuts the transform length into half and requires less amount of memory for precomputed twiddle factors. The DGT algorithm was originally proposed by Crandall [13] for fast negacyclic convolution. It works in the field $GF(p^2)$ where p is a Gaussian prime, i.e., $p \equiv 3 \pmod 4$. Unfortunately, NFLlib primes are non-Gaussian and hence they are not compatible with vanilla DGT. However, Al Badawi et al. [4] showed how to extend Crandall's DGT algorithm to work with non-Gaussian primes as well. We remark that the current version of DSI_BFV includes the NTT optimizations from David Harvey [20] adapted for DGT computations.

c) Uniform and Gaussian Random Samplers: Random polynomials are generated on GPU using CUDA cuRAND. We need to sample polynomials from three random distributions: 1) $\mathcal{X}_2$, 2) $\mathcal{X}_q$, and 3) $\mathcal{X}_{err}$. The first two are uniform distributions while the third is a discrete Gaussian. DSI_BFV includes efficient parallel implementations for these distributions.

d) Memory Pool: DSI_BFV includes a GPU memory pool. Memory allocations are done once when needed, and the used memory is deallocated only when the program terminates. This is important for performance since we are dealing with large polynomials that require substantial amounts of memory. Frequent memory allocations and deallocations on GPU are costly and should be avoided whenever possible.

e) Implementation of HPS: The main component of HPS is FastBaseConv_F. For a polynomial of degree n, we launch n threads to extend a polynomial coefficient in base B to base B'. The required constants are precomputed and stored in the GPU constant memory. We note here that although HPS requires less precomputed parameters, their size is larger than that required by BEHZ. We had to reduce K from 64 to 61 in order to fit the precomputed parameters in the GPU constant memory. Since we are only dealing with 61 30-bit CRT moduli, we use the native C++ double for floating-point operations. This provides enough precision as shown in Table I. Our FastBaseConv_F kernel launches n threads, one for each coefficient residue. We also apply the lazy reduction technique similar to the CPU implementation. The only difference is that we work with 30-bit moduli; therefore, the GPU Barrett reduction algorithm is simpler than Algorithm 3. Key generation, encryption, homomorphic addition, and relinearization procedures are borrowed from the implementation of BEHZ that is already included in DSI_BFV.

VI. Performance Evaluation

In this section, we evaluate the performance of the BFV RNS variants on different platforms: CPU Single-Threaded (CPU$_{ST}$), CPU Multi-Threaded (CPU$_{MT}$), and two GPU cards: Tesla K80 and Tesla V100.

A. Methodology

We report the execution times for decryption and homomorphic multiplication. On CPU, we measure the time via the C++ library chrono [1]. On GPU, CUDA events are used instead.

We perform experiments in different platform settings. For CPUs, we use single-threaded and multi-threaded (OpenMP) settings. For GPUs, we run our experiments on one GPU card.

B. Experimental Setup

Our GPU implementation was developed via CUDA 9.0 toolkit on a 64-bit server equipped with 2 sockets, 26 cores per socket and 2 logical CPUs per core, i.e., 104 CPU threads in total. The machine also hosts two NVIDIA cards: 1) Tesla K0 and 2) V100-PCIe. Table VII describes the hardware configuration of both CPU and GPU.

The OS was ArchLinux version (4.15.13-1-ARCH), and the compilers were g++ (GCC) 7.3.1 and nvcc (8.0.61). We ran our CPU and GPU implementations on the same machine. We disabled the PALISADE library OpenMP support in single-threaded experiments and used 26 threads in multi-threaded experiments.

Table VII: CPU and GPU hardware configurations

Feature	**CPU**	**GPU**	
		K80	V100
Model	Intel(R) Xeon(R) Platinum	K80	V100-PCIe
# Cores	104	2496	5120
Frequency	2.10 GHz	0.82 GHz	1.380 GHz
RAM	187.5 GB	12 GB	16 GB

C. Parameter Selection

To choose the ring dimension n, we ran the Learning With Errors security estimator[4] (commit f59326c) [6] to find the lowest security levels for the uSVP, decoding, and dual attacks following the standard homomorphic encryption security recommendations [11]. We selected the least value of the number of security bits λ for all 3 attacks on classical computers based on the estimates for the BKZ sieve reduction cost model.

Table VIII: Decryption latency in (milliseconds) of BFV RNS variants for single- and muli-threaded CPU settings, and GPUs with different moduli sizes (ν).

Configuration	Variant	($\log n$, $\log q$)					
		(12,60)	(13,120)	(14,360)	(15,600)	(16,1020)	(16,1770)
Single-threaded CPU ($\nu = 30$)	BEHZ	0.535	2.079	13.330	48.547	196.266	336.617
	HPS	**0.472**	**1.857**	**11.374**	**43.334**	**172.916**	**318.942**
Multi-threaded CPU ($\nu = 30$)	BEHZ	**0.298**	0.831	3.799	12.397	49.850	88.845
	HPS	0.302	**0.819**	**3.607**	**12.042**	**49.741**	**87.774**
Single-threaded CPU ($\nu = 60$)	BEHZ	**0.401**	**1.153**	**7.379**	**26.389**	**103.034**	**173.313**
	HPS	0.504	1.632	8.735	30.314	120.576	195.398
Multi-threaded CPU ($\nu = 60$)	BEHZ	0.278	**0.64**	**2.311**	**7.189**	**24.204**	**46.708**
	HPS	**0.273**	0.669	2.392	7.475	25.811	48.195
K80 GPU ($\nu = 30$)	BEHZ	0.115	0.139	0.304	0.558	1.564	2.630
	HPS	**0.111**	**0.123**	**0.235**	**0.455**	**1.207**	**2.025**
V100 GPU ($\nu = 30$)	BEHZ	0.057	0.063	0.101	0.134	0.329	0.516
	HPS	**0.054**	**0.059**	**0.087**	**0.111**	**0.298**	**0.457**

Table IX: Homomorphic multiplication (including relinearization) latency in (milliseconds) of BFV RNS variants in single- and muli-threaded CPU settings, and GPUs with different moduli sizes (ν).

Configuration	Variant	($\log n$, $\log q$)					
		(12,60)	(13,120)	(14,360)	(15,600)	(16,1020)	(16,1770)
Single-threaded CPU ($\nu = 30$)	BEHZ	10.157	**40.675**	**382.149**	1899.136	11761.077	35173.622
	HPS	**10.065**	41.417	385.057	**1873.935**	**10815.283**	**27229.210**
Multi-threaded CPU ($\nu = 30$)	BEHZ	4.270	10.838	74.716	351.296	1986.586	5697.640
	HPS	**4.054**	**10.179**	**74.420**	**351.140**	**1984.211**	**5553.058**
Single-threaded CPU ($\nu = 60$)	BEHZ	6.952	23.300	155.365	**670.946**	3526.113	9260.904
	HPS	**6.326**	**22.088**	**154.350**	673.132	**3464.492**	**8605.612**
Multi-threaded CPU ($\nu = 60$)	BEHZ	3.343	7.325	32.244	**124.117**	**585.080**	**1653.966**
	HPS	**2.844**	**6.834**	**31.744**	126.945	603.554	1667.919
K80 GPU ($\nu = 30$)	BEHZ	2.166	2.754	9.603	25.885	112.062	307.531
	HPS	**1.977**	**2.517**	**7.834**	**21.924**	**96.151**	**255.502**
V100 GPU ($\nu = 30$)	BEHZ	0.997	1.178	2.412	5.705	22.848	59.473
	HPS	**0.859**	**1.012**	**2.010**	**4.826**	**18.725**	**50.779**

The secret-key polynomials were generated using discrete ternary uniform distribution over $\{-1, 0, 1\}^n$. In all of our experiments, we selected the minimum ciphertext modulus bitwidth that satisfied the correctness constraint for the lowest ring dimension n corresponding to the security level $\lambda \geq 128$.

We set the Gaussian distribution parameter σ to $8/\sqrt{2\pi}$ [11], the error bound B_e to 6σ, and the lower bound for q' to $2tnq$. In the relinearization procedure, we utilized only the CRT decomposition (and did not use the second-level digit decomposition of residues).

[4]https://bitbucket.org/malb/lwe-estimator

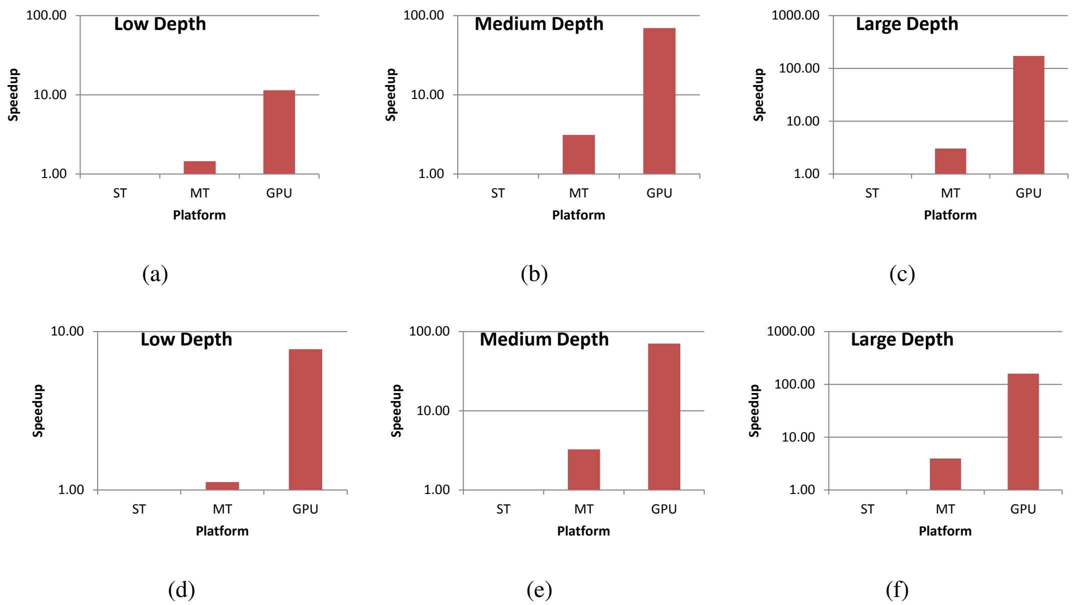

Figure 3: Speedup factors of homomorphic multiplication on different platforms (CPU single-threaded (ST), CPU multi-threaded (MT), and GPU) for different parameter sets: $(\log_2 n, \log_2 q) = (12, 60)$ *[low]*, $(14, 360)$ *[medium]*, $(16, 1770)$ *[large]. Upper three are for BEHZ, while the lower three are for HPS. Note that vertical axes are in log-scale.*

D. Benchmarking

Tables VIII and IX show the results for decryption and homomorphic multiplication, respectively, in BEHZ and HPS. As PALISADE supports CRT moduli sizes $\nu \in \{30, \ldots, 60\}$ bits, we include the runtimes for $\nu = 30$ and $\nu = 60$. Note that the GPU library only supports 30-bit moduli.

It can be clearly seen that HPS outperforms BEHZ in decryption for $\nu = 30$ bits on all platforms. When $\nu = 60$ bits, HPS decryption performance degrades due to the use of quad-float (`double double`) floating-point arithmetic. Given that HPS performs faster in three different platforms when $\nu = 30$ bits, it can be concluded that HPS's decryption function requires less computational overhead compared to BEHZ (unless quad-float or higher-precision floating-point arithmetic is needed). More concretely, HPS decryption can achieve the following speedup factors: 1.06 to 1.17 ($\mathrm{CPU_{ST}}, \nu = 30$), 0.71 to 0.89 ($\mathrm{CPU_{ST}}, \nu = 60$), 0.99 to 1.05 ($\mathrm{CPU_{MT}}$, $\nu = 30$), 0.94 to 1.02 ($\mathrm{CPU_{MT}}, \nu = 60$), 1.04 to 1.30 ($\mathrm{GPU_{K80}}, \nu = 30$), and 1.06 to 1.21 ($\mathrm{GPU_{V100}}, \nu = 30$), as compared to BEHZ.

For homomorphic multiplication, HPS typically outperforms BEHZ in all platforms regardless of ν. Significant improvements can be noticed for large parameters. For instance, in $\mathrm{CPU_{ST}}$ and $\nu = 30$ bits, a 7.95-second difference is recorded. More concretely, HPS homomorphic multiplication can achieve the following speedup factors: 0.98 to 1.29 ($\mathrm{CPU_{ST}}, \nu = 30$), 1.00 to 1.10 ($\mathrm{CPU_{ST}}, \nu = 60$), 1.00 to 1.06 ($\mathrm{CPU_{MT}}$, $\nu = 30$), 0.97 to 1.18 ($\mathrm{CPU_{MT}}, \nu = 60$), 1.09 to 1.23 ($\mathrm{GPU_{K80}}, \nu = 30$), and 1.16 to 1.22 ($\mathrm{GPU_{V100}}, \nu = 30$), as compared to BEHZ. The fluctuations in the runtimes for the multi-threaded CPU experiments can be attributed to specifics of the OpenMP setup in our benchmarking environment.

Tables VIII and IX suggest that the use of $\nu = 60$ bits provides roughly 0.9x to 2x (resp. 1.5x to 3.9x) improvements for decryption (resp. homomorphic multiplication) on 64-bit machines. This is logical since these machines perform 64-bit operations natively. It can also be noticed that P100 outperforms K80 by 2x to 5x for both decryption and homomorphic multiplication. This can mainly be attributed to the number of cores P100 includes (5120) compared to (2496) for K80. It should be noted that K80 and P100 were released by NVIDIA in November 2014 and June 2017, respectively [2].

To examine the improvements that can be achieved from parallel implementations on different platforms, Figure 3 shows the speedup factors computed as the ratio of $\mathrm{CPU_{ST}}$ and either $\mathrm{CPU_{MT}}$ or GPU. We use the best performance between $\nu = 30$ bits and $\nu = 60$ bits for multi-threaded speedups and the best between K80 and P100 for GPU speedups. We observe that GPU can improve the performance by one to two orders of magnitudes whereas a multi-threaded CPU implementation can hardly achieve one order of magnitude.

We remark that our best GPU results, namely the homomorphic multiplication runtime of 51 ms for $n = 2^{16}$ and $\log_2 q = 1,770$ and 18.7 ms for $n = 2^{16}$ and $\log_2 q = 1,020$, **are more than two orders of magnitude faster than best previously reported runtimes for the BFV scheme**. For instance, the FPGA-based implementation HEPCloud in [31] of the textbook BFV scheme computed a homomorphic multiplication for $n = 2^{15}$ and $\log_2 q = 1,228$ in 26.67 seconds (with 3.36 seconds spent on the computation and the rest on the off-chip memory access). The BEHZ variant NFLlib CPU implementation in [7] ran a homomorphic multiplication for $n = 2^{15}$ and $\log_2 q = 1,590$ in 4.9 seconds.

E. Practical Noise Growth

We showed previously that BEHZ and HPS can theoretically achieve the same multiplicative depth (see Table V) as the textbook BFV scheme. The noise analysis provided for both variants was conservative (worst-case) and used $\delta = n$ to estimate the noise growth. However, in practice it was shown [19] that a lower value of δ can be used, specifically, $\delta = 2\sqrt{n}$. The authors used the Central Limit Theorem (CLT) to derive a practical heuristic estimate of noise growth. We ran an experiment to verify this analysis and measure the maximum multiplicative depth each variant can achieve.

Table X: Maximum multiplicative depth $L_\circ$ experimentally observed for each RNS variant (30-bit moduli and $\lambda_\circ \geq 128$) vs. maximum heuristic estimate for $\delta = 2\sqrt{n}$ for the same lattice parameters.

log_2^n	$\lceil log_2^q \rceil$	t	Max. Est. $L_\circ$	Max. Exp. $L_\circ$	
				BEHZ	HPS
12	60	2	1	2	2
		65537	0	0	1
13	120	2	5	5	6
		65537	2	2	3
14	360	2	18	16	21
		65537	10	9	10
15	600	2	31	26	35
		65537	16	15	19
16	1020	2	51	43	56
		65537	28	25	30
16	1770	2	90	77	98
		65537	50	44	52

We wrote a simple procedure that encrypts a plaintext message μ and iteratively multiplies it with an encryption of 1. In each iteration, we decrypt the product and check if it is equal to μ counting the number of sequential multiplications. Table X shows the lowest value of the maximum multiplicative depth that can be reached without decryption failures by each variant in this experiment vs. the maximum depth estimated using the heuristic CLT-based technique of [19] for the same parameters. It suggests that HPS can achieve a higher multiplicative depth than BEHZ for almost all parameters. The HPS multiplicative depth conforms to the heuristic noise analysis having $\delta = O(\sqrt{n})$, where the constant is always less than two (which corresponds to the column "Max. Est. $L_\circ$" in Table X), whereas the BEHZ depth requires a higher value for δ. We found by fitting the experimental results to the noise expression that $\delta \approx O(n^{0.7})$ provides an adequate estimate of practical noise growth in BEHZ.

Our interpretation of this behavior is that the RNS techniques used in the BEHZ variant transform the ciphertexts in such a way that CLT is no longer valid, i.e., we no longer deal with sums of zero-centered independent random variables in certain polynomial multiplications. We claim that the deviation from uncorrelated zero-centered random distribution is introduced by the step described in (Lemma 4 in [7]), i.e., by the small Montgomery representation/reduction shown in Algorithm 2. In the textbook BFV and HPS variants, ciphertexts are uncorrelated and zero-centered (with respect to the interval $\lceil -q/2 \rceil \leq x < \lfloor q/2 \rfloor$). When we introduce the overflow $(q \cdot u)$ term, where $\|u\|_\infty \leq k$, in fast base conversion, we get ciphertexts that are no longer zero-centered but are biased (correlated). Therefore, the noise growth is faster than what CLT predicts (except for the case of $k = 2$ at $t = 2$ in Table X, when the overflow term contribution is the smallest). In contrast to the HPS case, we cannot use heuristic (average-case) estimates for BEHZ and can guarantee the correctness only with worst-case estimates, i.e., $\delta = n$.

This implies there is a major practical difference between BEHZ and HPS, which is far more significant than the incremental performance improvements we observed when comparing the runtimes for same lattice parameters. Although the effect of ρ is small for large $\tilde{m}$ (for the worst-case analysis), the deviation from zero-centered random distribution has a more profound effect in practice. We note that this behavior has been observed in both our CPU and GPU implementations of BEHZ. We also observed a similar noise growth behavior in the SEAL implementation of BEHZ (SEAL version 2.3.0-4).

VII. Conclusion

Our work presents the implementation and performance evaluation of two RNS variants (BEHZ [7] and HPS [19]) of the BFV SHE scheme. We have shown that both variants follow the same blueprint to handle polynomial lifting and scaling but use different RNS tools. We have analyzed the performance of both variants theoretically and experimentally using several flavors of implementations (CPU single- and multi-threaded, and GPU). Our analysis shows that HPS outperforms BEHZ in almost all settings on different platforms (for same values of lattice parameters). However, HPS decryption is outperformed by BEHZ when the moduli size is 60 bits, which is due to the multi-precision (`double double`) floating-point operations in HPS.

Our experiments show that our multi-threaded CPU implementation using OpenMP can hardly attain a one-order-of-magnitude improvement over the single-threaded setting, whereas GPUs can achieve up to two orders of magnitude.

We have also demonstrated that the practical noise growth in BEHZ is much faster than that of HPS, which can significantly increase the runtime and storage requirements, and also limit the maximum multiplicative depth supported by BEHZ. We provide a possible explanation for this behavior; however, a more careful analysis of BEHZ noise growth would be needed to characterize this behavior formally.

VIII. Acknowledgment

This work was supported by A * STAR Data Storage Institute and the National University of Singapore. Polyakov and Rohloff's work was supported in part by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA). The views expressed are those of the authors and do not necessarily reflect the official policy or position of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

References

[1] C++ chrono time library. Available at: http://en.cppreference.com/w/cpp/chrono. Accessed 2018.

[2] List of nvidia graphics processing units. Available at: https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units. Accessed May 26th 2018.

[3] Carlos Aguilar-Melchor, Joris Barrier, Serge Guelton, Adrien Guinet, Marc-Olivier Killijian, and Tancrede Lepoint. Nfllib: Ntt-based fast lattice library. In *Cryptographers? Track at the RSA Conference*, pages 341–356. Springer, 2016.

[4] Ahmad Al Badawi, Veeravalli Bharadwaj, and Khin Mi Mi Aung. Efficient polynomial multiplication via modified discrete galois transform and negacyclic convolution. In *Future of Information and Communications Conference (FICC)*. IEEE, 2018.

[5] Ahmad Al Badawi, Bharadwaj Veeravalli, and Mi Mi Khin Aung. High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2), 2018.

[6] Martin Albrecht, Samuel Scott, and Rachel Player. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 10 2015.

[7] Jean-Claude Bajard, Julien Eynard, M Anwar Hasan, and Vincent Zucca. A full rns variant of fv like somewhat homomorphic encryption schemes. In *International Conference on Selected Areas in Cryptography*, pages 423–442. Springer, 2016.

[8] Joppe W. Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In *IMACC 2013*, pages 45–64, 2013.

[9] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Advances in cryptology–crypto 2012*, pages 868–886. Springer, 2012.

[10] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):13, 2014.

[11] Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Jeffrey Hoffstein, Kristin Lauter, Satya Lokam, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Security of homomorphic encryption. Technical report, HomomorphicEncryption.org, Redmond WA, July 2017.

[12] D. B. Cousins, G. Di Crescenzo, K. D. Gür, K. King, Y. Polyakov, K. Rohloff, G. W. Ryan, and E. Savaş. Implementing conjunction obfuscation under entropic ring lwe. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 68–85.

[13] Richard E Crandall. Integer convolution via split-radix fast galois transform. *Center for Advanced Computation Reed College*, 1999.

[14] W. Dai, Y. Doröz, Y. Polyakov, K. Rohloff, H. Sajjadpour, E. Savaş, and B. Sunar. Implementation and evaluation of a lattice-based key-policy abe scheme. *IEEE Transactions on Information Forensics and Security*, 13(5):1169–1184, May 2018.
[15] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.
[16] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.
[17] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology–CRYPTO 2013*, pages 75–92. Springer, 2013.
[18] Kamil Doruk Gür, Yuriy Polyakov, Kurt Rohloff, Gerard W. Ryan, Hadi Sajjadpour, and Erkay Savaş. Practical applications of improved gaussian sampling for trapdoor lattices. Cryptology ePrint Archive, Report 2017/1254, 2017. https://eprint.iacr.org/2017/1254.
[19] Shai Halevi, Yuriy Polyakov, and Victor Shoup. An improved rns variant of the bfv homomorphic encryption scheme. https://eprint.iacr.org/2018/117.
[20] David Harvey. Faster arithmetic for number-theoretic transforms. *Journal of Symbolic Computation*, 60:113–119, 2014.
[21] Shinichi Kawamura, Masanobu Koike, Fumihiko Sano, and Atsushi Shimbo. Cox-rower architecture for fast parallel montgomery multiplication. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 523–538. Springer, 2000.
[22] Hao Chen Kim Laine and Rachel Player. Simple encrypted arithmetic library-seal (v2.1). Technical report, Technical report, Microsoft Research, 2016.
[23] Tancrede Lepoint and Michael Naehrig. A comparison of the homomorphic encryption schemes fv and yashe. In *International Conference on Cryptology in Africa*, pages 318–335. Springer, 2014.
[24] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 1219–1234. ACM, 2012.
[25] CUDA Nvidia. Toolkit documentation. *NVIDIA CUDA Getting Started Guide for Linux*, 2014.
[26] Yuriy Polyakov, Kurt Rohloff, and Gerard W. Ryan. PALISADE lattice cryptography library. https://git.njit.edu/palisade/PALISADE, Accessed June 2018.
[27] Yuriy Polyakov, Kurt Rohloff, Gyana Sahu, and Vinod Vaikuntanathan. Fast proxy re-encryption for publish/subscribe systems. *ACM Trans. Priv. Secur.*, 20(4):14:1–14:31, 2017.
[28] Karl C Posch and Reinhard Posch. Modulo reduction in residue number systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):449–454, 1995.
[29] AP Shenoy and Ramdas Kumaresan. Fast base extension using a redundant modulus in rns. *IEEE Transactions on Computers*, 38(2):292–297, 1989.
[30] Victor Shoup et al. Ntl: A library for doing number theory, 2001.
[31] Sujoy Sinha Roy, Kimmo Jarvinen, Ingrid Verbauwhede, Frederik Vercauteren, and Jo Vliegen. Hepcloud: An fpga-based multicore processor for fv somewhat homomorphic function evaluation. *IEEE Transactions on Computers*, 2017.
[32] Marten Van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 24–43. Springer, 2010.