An Improved RNS Variant of the BFV Homomorphic Encryption Scheme

Shai Halevi[⋆1], Yuriy Polyakov[⋆⋆2], and Victor Shoup[3]

[1] IBM Research
[2] NJIT Cybersecurity Research Center
[3] NYU

Abstract. We present an optimized implementation of the Fan-Vercauteren variant of the scale-invariant homomorphic encryption scheme of Brakerski. Our algorithmic improvements focus on optimizing decryption and homomorphic multiplication in the Residue Number System (RNS), using the Chinese Remainder Theorem (CRT) to represent and manipulate the large coefficients in the ciphertext polynomials. In particular, we propose efficient procedures for scaling and CRT basis extension that do not require translating the numbers to standard (positional) representation. Compared to the previously proposed RNS design due to Bajard et al. [3], our procedures are simpler and they do not introduce any additional noise. We implement our optimizations in the PALISADE library and evaluate the runtime performance for the range of multiplicative depths from 1 to 50. Our results show that the homomorphic multiplication for the depth-20 setting can be executed in 63 ms on a modern server system, which is already practical for some outsourced-computing applications. Our algorithmic improvements can also be applied to other scale-invariant homomorphic encryption schemes, such as YASHE.

Keywords: Lattice-Based Cryptography · Homomorphic Encryption · Scale-Invariant Scheme · Residue Number Systems · Software Implementation

1 Introduction

Homomorphic encryption has been an area of active research since the first design of a Fully Homomorphic Encryption (FHE) scheme by Gentry [9]. FHE allows performing arbitrary secure computations over encrypted sensitive data without ever decrypting them. One of the potential applications is to outsource computations to a public cloud without compromising data privacy.

A salient property of contemporary FHE schemes is that ciphertexts are "noisy", where the noise increases with every homomorphic operation, and decryption starts failing once the noise becomes too large. This is addressed by

⋆ Supported by the Defense Advanced Research Projects Agency (DARPA) and Army Research Office (ARO) under Contract No. W911NF-15-C-0236.

⋆⋆ Supported in part by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA).

setting the parameters large enough to accommodate some level of noise, and using Gentry's "bootstrapping" technique to reduce the noise once it gets too close to the decryption-error level. However, the large parameters make homomorphic computations quite slow, and so significant effort was devoted to constructing more efficient schemes. Two of the the most promising schemes in terms of practical performance have been the BGV scheme of Brakerski, Gentry and Vaikuntanathan [6], and the Fan-Vercauteren variant of Brakerski's scale-invariant scheme [5,8], which we call here the BFV scheme. Both of these schemes rely on the hardness of the Ring Learning With Errors (RLWE) problem.

Both schemes manipulate elements in large cyclotomic rings, modulo integers with many hundreds of bits. Implementing the necessary multi-precision modular arithmetic is expensive, and one way of making it faster is to use a "Residue Number System" (RNS) to represent the big integers. Namely, the big modulus q is chosen as a smooth integer, $q = \prod_i q_i$, where the factors q_i are same-size, pairwise coprime, single-precision integers (typically of size 40-60 bits). Using the Chinese Remainders Theorem (CRT), an integer $x \in \mathbb{Z}_q$ can be represented by its CRT components $\{x_i = x \bmod q_i \in \mathbb{Z}_{q_i}\}_i$, and operations on x in $\mathbb{Z}_q$ can be implemented by applying the same operations to each CRT component x_i in its own ring $\mathbb{Z}_{q_i}$.

Unfortunately, both BGV and BFV feature some scaling operations that cannot be directly implemented on the CRT components. In both schemes there is sometimes a need to interpret $x \in \mathbb{Z}_q$ as a rational number (say in the interval $[-q/2, q/2)$) and then either lift x to a larger ring $\mathbb{Z}_Q$ for $Q > q$, or to scale it down and round to get $y = \lceil \delta x \rfloor \in \mathbb{Z}_t$ (for some $\delta \ll 1$ and accordingly $t \ll q$). These operations seem to require that x be translated from its CRT representation back to standard "positional" representation, but computing these translations back and forth will negate the gains from using RNS to begin with.

While implementations of the BGV scheme using CRT representation are known (e.g., [10, 11]), implementing BFV in this manner seems harder. One difference is that BFV features more of these scaling operations than BGV. Another is that in BGV the scaling factors are usually just single-precision numbers, while in BFV these factors are often big, of order similar to the multi-precision modulus q. An implementation of the BFV scheme using CRT representation was recently reported by Bajard et al. [3], featuring significant speedup as compared to earlier implementations such as in [14]. This implementation, however, uses somewhat complex procedures, and moreover these procedures incur an increase in the ciphertext noise.

In the current work we simplify the CRT-based scaling and lifting procedures of [3]. Our procedures are all quite elementary, and while they are not much faster than the ones of Bajard et al., they are simpler to implement and do not introduce additional noise. Hence the noise behavior of our implementation is the same as in the original BFV scheme. The same techniques are also applicable to other scale-invariant homomorphic encryption schemes, such as YASHE and YASHE' [4].

We implemented our procedures in the PALISADE library [16]. We evaluate the runtime performance of decryption and homomorphic multiplication in the range of multiplicative depths from 1 to 50. For example, the runtimes for depth-20 decryption and homomorphic multiplication are 3.0 and 63 ms, respectively, which can already support outsourced-computing applications with latencies up to few seconds, even without bootstrapping.

Our implementation pre-computes some tables, consisting of single-precision integers as well as floating-point numbers, then uses these pre-computed values to implement fast operations. When using the IEEE 754 double-precision floating-point format (with 53 bits of precision), the limited accuracy entails a bound on the size of the moduli q_i that we can use, limiting them to around 47 bits. We note that the moduli size can be easily increased by storing the floating-point numbers with higher precision (e.g., storing two double-floats for each number). The floating-point operations take only a minuscule fraction of the computation time, hence such modification will not slow down the computation.

2 Notations and Basic Procedures

For an integer $n \geq 2$, we identify below the ring $\mathbb{Z}_n$ with its representation in the symmetric interval $\mathbb{Z} \cap [-n/2, n/2)$. For an arbitrary rational number x, we denote by $[x]_n$ the reduction of x into that interval (namely the real number $x' \in [-n/2, n/2)$ such that $x' - x$ is an integer divisible by n). We also denote by $\lfloor x \rfloor$, $\lceil x \rceil$, and $\lceil x \rfloor$ the rounding of x to an integer down, up, and to the nearest integer, respectively. We denote vectors by boldface letters, and extend the notations $\lfloor x \rfloor$, $\lceil x \rceil$, $\lceil x \rfloor$ to vectors element-wise.

Throughout this note we fix a set of k co-prime moduli $q_1, \ldots, q_k$ (all integers larger than 1), and let their product be $q = \prod_{i=1}^{k} q_i$. For all $i \in \{1, ..., k\}$, we also denote

$$q_i^* = q/q_i \in \mathbb{Z} \text{ and } \tilde{q}_i = q_i^{*-1} \pmod{q_i} \in \mathbb{Z}_{q_i}, \tag{1}$$

namely, $\tilde{q}_i \in \left[-\frac{q_i}{2}, \frac{q_i}{2}\right)$ and $q_i^* \cdot \tilde{q}_i = 1 \pmod{q_i}$.

Complexity measures. In our setting we always assume that the moduli q_i are single-precision integers (i.e. $|q_i| < 2^{63}$), and that operations modulo q_i are inexpensive. We assign unit cost to mod-q_i multiplication and ignore additions, and analyze the complexity of our routines just by counting the number of the multiplications. Our procedures also include floating-point operations, and here too we assign unit cost to floating-point multiplications and divisions (in "double float" format as per IEEE 754) and ignore additions.

2.1 CRT Representation

We denote the CRT representation of an integer $x \in \mathbb{Z}_q$ relative to the CRT basis $\{q_1, \ldots, q_k\}$ by $x \sim (x_1, \ldots, x_k)$ with $x_i = [x]_{q_i} \in \mathbb{Z}_{q_i}$. The formula expressing x in terms of the x_i's is $x = \sum_{i=1}^{k} x_i \cdot \tilde{q}_i \cdot q_i^* \pmod{q}$. This formula can be used

in more than one way to "reconstruct" the value $x \in \mathbb{Z}_q$ from the x_i's. In this work we use in particular the following two facts:

$$x = \Big(\sum_{i=1}^{k} \underbrace{[x_i \cdot \tilde{q}_i]_{q_i} \cdot q_i^*}_{\in \mathbb{Z}_q}\Big) + v \cdot q \text{ for some } v \in \mathbb{Z}, \tag{2}$$

$$\text{and } x = \Big(\sum_{i=1}^{k} \underbrace{x_i \cdot \tilde{q}_i \cdot q_i^*}_{\in [-\frac{q_i q}{4}, \frac{q_i q}{4})}\Big) + v' \cdot q \text{ for some } v' \in \mathbb{Z}. \tag{3}$$

2.2 CRT Basis Extension

Let $x \in \mathbb{Z}_q$ be given in CRT representation $(x_1, \ldots, x_k)$, and suppose we want to extend the CRT basis by computing $[x]_p \in \mathbb{Z}_p$ for some other modulus $p > 1$. Using Eq. 2, we would like to compute $[x]_p = \Big[\big(\sum_{i=1}^{k}[x_i \cdot \tilde{q}_i]_{q_i} \cdot q_i^*\big) + v \cdot q\Big]_p$. The main challenge here is to compute v (which is an integer in $\mathbb{Z}_k$). The formula for v is:

$$v = \left\lceil \big(\sum_{i=1}^{k}[x_i \cdot \tilde{q}_i]_{q_i} \cdot q_i^*\big)/q \right\rfloor = \left\lceil \sum_{i=1}^{k}[x_i \cdot \tilde{q}_i]_{q_i} \cdot \frac{q_i^*}{q} \right\rfloor = \left\lceil \sum_{i=1}^{k} \frac{[x_i \cdot \tilde{q}_i]_{q_i}}{q_i} \right\rfloor.$$

To get v, we compute for every $i \in \{1, \ldots, k\}$ the element $y_i := [x_i \cdot \tilde{q}_i]_{q_i}$ (using single-precision fixed-point arithmetic), and next the rational number $z_i := y_i/q_i$ (in floating-point). Then we sum up all the z_i's and round them to get v. Once we have the value of v, as well as all the y_i's, we can directly compute Eq. 2 modulo p to get $[x]_p = \Big[\big(\sum_{i=1}^{k} y_i \cdot [q_i^*]_p\big) + v \cdot [q]_p\Big]_p$.

In our setting p and the q_i's are parameters that can be pre-processed. In particular we pre-compute all the values $[q_i^*]_p$'s and $[q]_p$, so the last equation becomes just an inner-product of two $(k+1)$-vectors in $\mathbb{Z}_p$.

Complexity analysis. The computation of v requires k fixed-point single-precision multiplications to compute the y_i's, then k floating-point division operations to compute the z_i's, and then some additions and one rounding operation. In total it takes k fixed-point and $k+1$ floating-point operations. When p is a single-precision integer, the last inner product takes $k+1$ fixed-point multiplications, so the entire procedure takes $2k+1$ fixed-point and $k+1$ floating-point operations.

For larger p we may need to do $k+1$ multi-precision multiplications, but we may be able to use CRT representation again. When $p = \prod_{i=1}^{k'} p_i$ for single-precision co-prime p_j's, we can compute v only once and then compute the last inner product for each p_i (provided that we pre-computed $[q_i^*]_{p_j}$'s and $[q]_{p_j}$ for all i and j). The overall complexity in this case will be $kk' + k + m$ fixed-point and $k+1$ floating-point operations.

Correctness. The only source of errors in this procedure is the floating-point operations when computing v: Instead of the exact values $z_i = y_i/q_i$, we compute

their floating-point approximations z_i^* (with error ϵ_i), and so we obtain $\upsilon^* = \lceil \sum_i (z_i + \epsilon_i) \rfloor$ which may be different from $\upsilon = \lceil \sum_i z_i \rfloor$.

Since the z_i's are all in $[-\frac{1}{2}, \frac{1}{2})$, then using IEEE 754 double floats we have that the ϵ_i's are bounded in magnitude by 2^{-53}, and therefore the overall magnitude of the error term $\epsilon := \sum \epsilon_i$ is bounded, $|\epsilon| < k \cdot 2^{-53}$. Since we always have $k \leq 32$, then this gives us $|\epsilon| < 2^{-48}$.

When applying the procedure above, we must check that the resulting υ^* that it returns is outside the possible-error region $\mathbb{Z} + \frac{1}{2} \pm \epsilon$. If υ^* falls in the error region, then we have to re-run this procedure using higher precision (and hence smaller ϵ) until the result is outside the error region.

Alternatively, in our setting we can (a) argue that the the probability to fall in the error region is small, and (b) re-randomize the input x and re-run the procedure if the output does fall in that region. This yields a Las Vegas Algorithm that terminates in expected $1 + \epsilon'$ runs (for some ϵ' related to ϵ), see more details in Section 4.5.

Comparison to other approaches for computing υ. Two exact approaches for computing υ are presented in [19] and [13]. The first approach introduces an auxiliary modulus and performs the CRT computations both for p and the extra modulus, thus doubling the number of fixed-point operations and also increasing the implementation complexity [19]. The second approach computes successive fixed-point approximations until the computed value of υ is outside the error region (in one setting) or computes the exact value (in another setting with higher complexity) [13] . Both of these techniques incur higher computational costs than our method. But either of them can be used to handle the possible-error region condition, which would occur for our algorithm with a probability comparable to $|\epsilon|$, i.e., close to 2^{-48}.

2.3 Simple scaling in CRT representation

Let $x \in \mathbb{Z}_q$ be given in CRT representation $(x_1, \ldots, x_k)$, and let $t \in \mathbb{Z}$ be an integer modulus $t \geq 2$. We want to "scale down" x by a t/q factor, namely to compute the integer $y = \lceil t/q \cdot x \rfloor \in \mathbb{Z}_t$. We do it using Eq. 3, as follows:

$$\begin{aligned} y := \left\lceil \frac{t}{q} \cdot x \right\rfloor &= \left\lceil \Big(\sum_{i=1}^{k} x_i \cdot \tilde{q}_i \cdot q_i^* \cdot \frac{t}{q} \Big) + \upsilon' \cdot q \cdot \frac{t}{q} \right\rfloor \\ &= \left\lceil \Big(\sum_{i=1}^{k} x_i \cdot (\tilde{q}_i \cdot \frac{t}{q_i}) \Big) \right\rfloor + \upsilon' \cdot t \;=\; \left[\left\lceil \Big(\sum_{i=1}^{k} x_i \cdot (\tilde{q}_i \cdot \frac{t}{q_i}) \Big) \right\rfloor \right]_t \quad .(4) \end{aligned}$$

The last equation follows since the two sides are congruent modulo t and are both in the interval $[-t/2, t/2)$, hence they must be equal.

In our context, t and the q_i's are parameters that we can pre-process (while the x_i's are computed on-line). We pre-compute the rational numbers $t\tilde{q}_i/q_i \in [-t/2, t/2)$, separated into their integer and fractional parts:

$$t\tilde{q}_i/q_i \;=\; \omega_i + \theta_i, \quad \text{with } \omega_i \in \mathbb{Z}_t \text{ and } \theta_i \in [-\tfrac{1}{2}, \tfrac{1}{2}).$$

With the ω_i's and θ_i pre-computed, we take as input the x_i's, compute the two sums $w := [\sum_i x_i\omega_i]_t$ and $v := \lceil \sum_i x_i\theta_i \rfloor$, (using fixed-point arithmetic for w and floating-point arithmetic for v), then output $[w+v]_t$.

Complexity analysis. The procedure above takes k floating-point multiplications, some additions, and one rounding to compute v, and then an inner product mod t between two $(k+1)$-vectors, the single-precision vector $(x_1, \ldots, x_k, 1)$ and the mod-t vector $(\omega_1, \ldots, \omega_k, v)$. When the modulus t is a single-precision integer, the ω_i's are also single-precision integers, and hence the inner product takes k fixed-point multiplications. The total complexity is therefore $k+1$ floating-point operations and k single-precision modular multiplications.

For a larger t we may need to do $O(k)$ multi-precision operations to compute the inner product. But in some cases we can also use CRT representation here: For $t = \prod_{j=1}^{k'} t_j$ (with the t_j's co-prime), we can represent each $\omega_i \in \mathbb{Z}_t$ in the CRT basis $\omega_{i,j} = [\omega_i]_{t_j}$. We can then compute the result y in the same CRT basis, $y_j = [y]_{t_j}$ by setting $w_j = [\sum_i x_i\omega_{i,j}]_{t_j}$ for all j, and then $y_j = [v+w_j]_{t_j}$. This will still take only $k+1$ floating-point operations, but kk' modular multiplications.

Correctness. The only source of errors in this routine is the computation of $v := \lceil \sum_i x_i\theta_i \rfloor$: Since we only keep the θ_i's with limited precision, we need to worry about the error exceeding the precision. Let $\tilde{\theta}_i$ be the floating-point values that we keep, while θ_i are the exact values ($\theta_i = t\tilde{q}_i/q - \omega_i$) and ϵ_i are the errors, $\epsilon_i = \tilde{\theta}_i - \theta_i$. Since $|\tilde{\theta}_i| \le \frac{1}{2}$, then for IEEE 754 double floats we have $|\epsilon_i| < 2^{-53}$. The value that our procedure computes for v is therefore $\tilde{v} := \lceil \sum_i x_i(\theta_i + \epsilon_i) \rfloor$, which may be different from $v := \lceil \sum_i x_i\theta_i \rfloor$.

We can easily control the magnitude of the error term $\sum x_i\epsilon_i$ by limiting the size of the q_i's: Since $|x_i| < q_i/2$ for all i, then $|\sum_i x_i\epsilon_i| < 2^{-54} \cdot \sum_i q_i$. In our implementation we always have $k < 32$, so for example as long as all our moduli satisfy $q_i \le 2^{47} < 2^{54}/4k$, we are ensured that $|\sum x_i\epsilon_i| < 1/4$.

When using the scaling procedure for decryption, we can keep $y = \lceil t/qx \rfloor$ close to an integer by controlling the ciphertext noise. For example, we can ensure that y (and therefore also v) is within $1/4$ of an integer, and thus if we also restrict the size of the q_i's as above, then we always get the correct result. Using the scaling procedure in other settings may require more care, see the next section for a discussion.

2.4 Complex scaling in CRT representation

The scaling procedure above was made simpler by the fact that we scale by a t/q factor, where the original integer is in $\mathbb{Z}_q$ and the result is computed modulo t. During homomorphic multiplication, however, we have a more complicated setting: Over there we have three parameters t, p, q, where $q = \prod_{i=1}^{k} q_i$ as before, we similarly have $p = \prod_{j=1}^{k'} p_j$, and we know that p is co-prime with q and $p \gg t$.

The input is $x \in \mathbb{Z} \cap [-qp/2t, qp/2t) \subset \mathbb{Z}_{qp}$, represented in the CRT basis $\{q_1, \ldots, q_k, p_1, \ldots, p_{k'}\}$. We need to scale it by a t/q factor and round, and we

want the result modulo q in the CRT basis $\{q_1, \ldots, q_k\}$. Namely, we want to compute $y := \left[\lceil t/q \cdot x \rfloor \right]_q$. This complex scaling is accomplished in two steps: [4]

1. First we essentially apply the CRT scaling procedure from Section 2.3 using $q' = qp$ and $t' = tp$, computing $y' := [\lceil tp/qp \cdot x \rfloor]_p$ (which we can think of as computing y' modulo tp and then discarding the mod-t CRT component). Note that since $x \in [-qp/2t, qp/2t)$ then $\lceil tp/qp \cdot x \rfloor \in [-p/2, p/2)$. Hence even though we computed y' modulo p, we know that $y' = \lceil t/q \cdot x \rfloor$ without modular reduction.
2. Having a representation of y' relative to CRT basis $\{p_1, \ldots, p_{k'}\}$, we extend this basis using the procedure from Section 2.2, adding $[y']_{q_i}$ for all the q_i's. Then we just discard the mod-p_j CRT components, thus getting a representation of $y = [y']_q$.

The second step is a straightforward application of the procedure from Section 2.2, but the first step needs some explanation. The input consists of the CRT components $x_i = [x]_{q_i}$ and $x'_j = [x]_{p_j}$, and we denote $Q := qp$, $Q_i^* := Q/q_i = q_i^* p$, ${Q'_j}^* := Q/p_j = qp_j^*$, and also $\tilde{Q}_i = [(Q_i^*)^{-1}]_{q_i}$ and $\tilde{Q'}_j = [({Q'_j}^*)^{-1}]_{p_j}$. Then by Eq. 3 we have

$$\frac{t}{q} \cdot x = \frac{t}{q}\Big(\sum_{i=1}^{k} x_i \tilde{Q}_i Q_i^* + \sum_{j=1}^{k'} x'_j \tilde{Q'}_j {Q'_j}^* + v'Q \Big) = \sum_{i=1}^{k} x_i \cdot \frac{t\tilde{Q}_i p}{q_i} + \sum_{j=1}^{k'} x'_j \cdot t\tilde{Q'}_j p_j^* + tv'p.$$

Rounding and reducing the above expression modulo one of the p_j's, all but one of the terms in the second sum drop out (as well as the term $tv'p$), and we get:

$$[\lceil t/q \cdot x \rfloor]_{p_j} = \left[\left\lceil \textstyle\sum_{i=1}^{k} x_i \cdot \frac{t\tilde{Q}_i p}{q_i} \right\rfloor + x'_j \cdot [t\tilde{Q'}_j p_j^*]_{p_j} \right]_{p_j}.$$

As in Section 2.3, we pre-compute all the values $\frac{t\tilde{Q}_i p}{q_i}$, breaking them into their integral and fractional parts, $\frac{t\tilde{Q}_i p}{q_i} = \omega'_i + \theta'_i$ with $\omega'_i \in \mathbb{Z}_p$ and $\theta'_i \in [-\frac{1}{2}, \frac{1}{2})$. We store all the θ'_i's as double floats, for every i, j we store the single-precision integer $\omega'_{i,j} = [\omega'_i]_{p_j}$, and for every j we also store $\lambda_j := [t\tilde{Q'}_j p_j^*]_{p_j}$. Then given the integer x, represented as $x \sim (x_1, \ldots, x_k, x'_i, \ldots, x'_{k'})$, we compute

$$v := \lceil \textstyle\sum_i \theta'_i x_i \rfloor, \text{ and for all } j \; w_j := \left[\lambda_j x'_j + \sum_i \omega'_{i,j} x_i \right]_{p_j} \text{ and } y'_j := \left[v + w_j \right]_{p_j}.$$

Then we have $y'_j = [\lceil t/q \cdot x \rfloor]_{p_j}$, and we return $y' \sim \{y'_1, \ldots, y'_{k'}\} \in \mathbb{Z}_p$.

Correctness. When computing the value $v = \lceil \sum_i \theta'_i x_i \rfloor$, we can bound the floating-point inaccuracy before rounding below $1/4$, just as in the simple scaling procedure from Section 2.3. However, when we use complex scaling during homomorphic multiplication, we do not have the guarantee that the exact value

[4] A somewhat different complex scaling procedure with similar complexity is presented in Appendix A.1, which can handle arbitrary $x \in \mathbb{Z}_{qp}$. However we did not implement that other procedure.

before rounding is close to an integer, and so we may encounter rounding errors where instead of rounding to the nearest integer, we will round to the second nearest. Contrary to the case of decryption, here such "rounding errors" are perfectly acceptable, as the rounding error is only added to the ciphertext noise.

Specifically, when analyzing the scaled ciphertext, we have a term of the form $\langle \mathbf{sk}^*, \epsilon \rangle$, where ϵ is the rounding error and we rely on it to be small (cf. Eq. 6 in Section A.2). Had we computed v with full accuracy, we could argue that the coefficients of ϵ are bounded in magnitude below $1/2$. But since we have an additional floating-point error of up to $1/4$, in our case some coefficients of ϵ could be as large as $3/4$. This adds less than one bit to the noise even in the worst case; in our tests the actual effect on the noise was too small to be noticed.

We remark, however, that even in the context of homomorphic multiplication we do need to watch out for rounding errors during the step of CRT basis extension above; this is done as described in Section 2.2 above and in Section 4.5.

Complexity analysis. The complexity of the first step above where we compute $y' = [\lceil t/q \cdot x \rfloor]_p$, is similar to the simple scaling procedure from Section 2.3. Namely we have k floating-point multiplications when computing v, and then for each modulus p_j we have $k+1$ single-precision modular multiplications to compute w_j. Hence the total complexity of this step is k floating-point multiplications and $k'(k+1)$ modular multiplications.

The complexity of the CRT basis extension, as described in Section 2.2, is k floating-point division operations and $k'(k+1)+k$ single-precision modular multiplications. Hence the total complexity of complex scaling is $2k$ floating-point operations and $2k'(k+1)+k$ modular multiplications.

3 Background: Scale-Invariant Homomorphic Encryption

For self-containment we briefly sketch in Appendix A.2 the first "scale-invariant" homomorphic encryption scheme, described by Brakerski in [5]. This section discusses the Fan-Vercauteren variant of the scheme and some optimizations due to Bajard et al. [3].

3.1 The Fan-Vercauteren Variant

In [8], Fan and Vercauteren ported Brakerski's scheme to the ring-LWE setting, working over polynomial rings rather than over the integers. Below we let $R = \mathbb{Z}[X]/\langle f(X) \rangle$ be a fixed ring, where $f \in \mathbb{Z}[X]$ is a monic irreducible polynomial of degree n (typically an m-th cyclotomic polynomial $\Phi_m(x)$ of degree $n = \phi(m)$). We use some convenient basis to represent R over $\mathbb{Z}$ (most often just the power basis, i.e., the coefficient representation of the polynomials). Also, let $R_t = R/tR$ denote the quotient ring for an integer modulus $t \in \mathbb{Z}$, represented in the same basis.

The plaintext space of this variant is R_t for some $t > 1$ (i.e., a polynomial of degree at most $n-1$ with coefficients in $\mathbb{Z}_t$), the secret key is a 2-vector $\mathbf{sk} = (1, s) \in R^2$ with $\|s\| \ll q/t$, ciphertexts are 2-vectors $\mathbf{ct} = (c_0, c_1) \in R_q^2$ for

another modulus $q \gg t$, and the decryption invariant is the same as in Brakerki's scheme, namely $[\lfloor \frac{t}{q}[\langle \mathbf{sk}, \mathbf{ct}\rangle]_q \rfloor]_t = [\lfloor \frac{t}{q}[c_0 + c_1 s]_q \rfloor]_t = m \cdot \frac{q}{t} + e$ for a small noise term $e \in R, \|e\| \ll q/t$.

For encryption, the public key includes a low-noise encryption of zero, $\mathbf{ct}^0 = (\mathbf{ct}_0^0, \mathbf{ct}_1^0)$, and to encrypt $m \in R_t$ they choose low-norm elements $u, e_1, e_2 \in R$ and set $Enc_{\mathbf{ct}^0}(m) := [u \cdot \mathbf{ct}^0 + (e_0, e_1) + (\Delta m, 0)]_q$, where $\Delta = \lfloor q/t \rfloor$. Homomorphic addition just adds the ciphertext vectors in R_q^2, and homomorphic multiplication is the same as in Brakerski's scheme, except (a) the special form of $\mathbf{sk}$ lets them optimize the relinearization "matrices" and use vectors instead, and (b) they use base-w decomposition (for a suitable word-size w) instead of base-2 decomposition. [5] In a little more detail:

(a) For the secret-key vector $\mathbf{sk} = (1, s)$, the tensor product $\mathbf{sk} \otimes \mathbf{sk}$ can be represented by the 3-vector $\mathbf{sk}^* = (1, s, s^2)$. Similarly, for the two ciphertexts $\mathbf{ct}^i = (c_0^i, c_1^i)$ $(i = 1, 2)$, it is sufficient to represent the tensor $\mathbf{ct}_1 \otimes \mathbf{ct}_2$ by the 3-vector $\mathbf{ct}^* = (c_0^*, c_1^*, c_2^*) = [c_0^1 c_0^2,\ (c_0^1 c_1^2 + c_1^1 c_0^2),\ c_1^1 c_1^2]_q$.

(b) For the relinearization gadget, all they need is to "encrypt" the single element s^2 using $\mathbf{sk}$. When using a base-w decomposition, they have vectors (rather than matrices) $W_i = (\beta_i, \alpha_i)$, with uniform α_i's and $\beta_i = [w^i s^2 - \alpha_i s + e_i]_q$ (for low-norm noise terms e_i).
After computing the three-vector $\mathbf{ct}^* = (c_0^*, c_1^*, c_2^*)$ as above during homomorphic multiplication, they decompose c_2^* into its base-w digits, $c_2^* = \sum_i w^i c_{2,i}^*$. Then computing $\mathbf{ct}^\times = \sum_i W_i \times \mathbf{ct}_i^*$ only requires that they set

$$\tilde{c}_0 := [\sum_{i=1}^{k} \beta_i c_{2,i}^*]_q,\ \tilde{c}_1 := [\sum_{i=1}^{k} \alpha_i c_{2,i}^*]_q, \quad \text{and then } \mathbf{ct}^\times := [(c_0^* + \tilde{c}_0, c_1^* + \tilde{c}_1)]_q.$$

3.2 CRT representation and optimized relinearization

Bajard et al. described in [3] several optimizations of the Fan-Vercauteren variant, centered around the use of CRT representation of the large integers involved. (They called it a *Residue Number System*, or RNS, but in this writeup we prefer the term CRT representation.) Specifically, the modulus q is chosen as a product of same-size, pairwise coprime, single-precision moduli, $q = \prod_{i=1}^{k} q_i$, and each element $x \in \mathbb{Z}_q$ is represented by the vector $(x_i = [x]_{q_i})_{i=1}^{k}$.

One significant optimization from [3] relates to the relinearization step in homomorphic multiplication. Recall that in that step we decompose the ciphertext $\mathbf{ct}^*$ into low-norm components $\mathbf{ct}_i^*$, such that reconstructing $\mathbf{ct}^*$ from the $\mathbf{ct}_i^*$'s is a linear operation, namely $\mathbf{ct}^* = \sum_i \tau_i \mathbf{ct}_i^*$ for some known coefficients τ_i. Instead of decomposing $\mathbf{ct}^*$ into bit or digits, Bajard et al. suggested to use its CRT components $\mathbf{ct}_i^* = [\mathbf{ct}^*]_{q_i}$ (which are also small), and rely on the reconstruction from Eq. 3 (which is linear).

[5] Fan and Vercauteren described in [8] a second relinearization procedure (which may be more efficient), using a technique of Gentry et al. from [10]. We ignore this alternative procedure here, since we did not implement it (yet).

4 Our Optimizations

4.1 The scheme that we implemented

The scheme that we implemented is the Fan-Vercauteren variant of Brakerski's scheme, with the CRT-based relinearization step of Bajard et al.; we refer to this variant as the "BFV scheme". We begin with a concrete stand-alone description of the functions that we implemented, then describe our simper/faster CRT-based implementation of these functions.

Parameters. Let $t, m, q \in \mathbb{Z}$ be parameters (where the single-precision t determines the plaintext space, and $m, |q|$ depend on t and the security parameter), such that $q = \prod_{i=1}^{k} q_i$ for same-size, pairwise coprime, single-precision moduli q_i.

Let $n = \phi(m)$, and let $R = \mathbb{Z}[X]/\Phi_m(X)$ be the m-th cyclotomic ring, and denote $R_q = R/qR$ and $R_t = R/tR$ to be quotient rings. In our implementation we represent elements in R, R_q, R_t in the power basis (i.e., polynomial coefficients), but note that other "small bases" are possible (such as the decoding basis from [15]), and for non-power-of-two cyclotomics they could sometimes result in better parameters. We let χ_e, χ_k be distributions over low-norm elements in R in the power basis, specifically we use discrete Gaussians for χ_e and the uniform distribution over $\{-1, 0, 1\}^n$ for χ_k.

Key generation. For the secret key, choose a low-norm secret key $s \leftarrow \chi_k$ and set $\mathbf{sk} := (1, s) \in R^2$. For the public encryption key, choose a uniform random $a \in R_q$ and $e \leftarrow \chi_e$, set $b := [-(as + e)]_q \in R_q$, and compute $\mathbf{pk} := (b, a)$.

Recall that we denote $q_i^* = \frac{q}{q_i}$ and $\tilde{q}_i = \left[q_i^{*-1}\right]_{q_i}$. For relinearization, choose a uniform $\alpha_i \in R_q$ and $e_i \leftarrow \chi_e$, and set $\beta_i = [\tilde{q}_i q_i^* s^2 - \alpha_i s + e_i]_q$ for each $i = 1, \ldots, k$. The public key consists of $\mathbf{pk}$ and all the vectors $W_i := (\beta_i, \alpha_i)$.

Encryption. To encrypt $m \in R_t$, choose $u \leftarrow \chi_k$ and $e'_0, e'_1 \leftarrow \chi_e$ and output the ciphertext $\mathbf{ct} := [u \cdot \mathbf{pk} + (e'_0, e'_1) + (\Delta m, 0)]_q$.

Decryption. For a ciphertext $\mathbf{ct} = (c_0, c_1)$, compute $x := [\langle \mathbf{sk}, \mathbf{ct} \rangle]_q = [c_0 + c_1 s]_q$ and output $m := [\lceil x \cdot t/q \rfloor]_t$.

Homomorphic Addition. On input $\mathbf{ct}^1$, $\mathbf{ct}^2$, output $[\mathbf{ct}^1 + \mathbf{ct}^2]_q$.

Homomorphic Multiplication. Given $\mathbf{ct}^i = (c_0^i, c_1^i)_{i=1,2}$, do the following:

1. **Tensoring:** Compute $c'_0 := c_0^1 c_0^2$, $c'_1 := c_0^1 c_1^2 + c_1^1 c_0^2$, $c'_2 := c_1^1 c_1^2 \in R$ without modular reduction, then set $c_i^* = [\lceil t/q \cdot c'_i \rfloor]_q$ for $i = 0, 1, 2$.
2. **Relinearization:** Decompose c_2^* into its CRT components $c_{2,i}^* = [c_2^*]_{q_i}$, set $\tilde{c}_0 := [\sum_{i=1}^{k} \beta_i c_{2,i}^*]_q$, $\tilde{c}_1 := [\sum_{i=1}^{k} \alpha_i c_{2,i}^*]_q$, output $\mathbf{ct}^\times := [(c_0^* + \tilde{c}_0, c_1^* + \tilde{c}_1)]_q$.

4.2 Pre-computed values

When setting the parameters, we pre-compute some tables to help speed things up later. Specifically:

- We pre-compute and store all the values that are needed for the simple CRT scaling procedure in Section 2.3: For each $i = 1, \ldots, k$, we compute the rational number $t\tilde{q}_i/q_i$, split into integral and fractional parts. Namely, $\omega_i := \left\lceil t \cdot \frac{\tilde{q}_i}{q_i} \right\rfloor \in \mathbb{Z}_t$ and $\theta_i := \frac{t \cdot \tilde{q}_i}{q_i} - \omega_i \in [-\frac{1}{2}, \frac{1}{2})$. We store ω_i as a single-precision integer and θ_i as a double float.
- We also choose a second set of single-precision coprime numbers $\{p_j\}_{j=1}^{k'}$ (coprime to all the q_i's), such that $p := \prod_j p_j$ is bigger than q by a large enough margin. Specifically we will need to ensure that for $c_0^1, c_1^1, c_0^2, c_1^2 \in R$ with coefficients in $[-q/2, q/2)$, the element $c^* := c_0^1 c_1^2 + c_1^1 c_0^2 \in R$ (without modular reduction) has coefficients in the range $[-qp/2t, qp/2t)$. For our setting of parameters, where all the q_i's and p_j's are 47-bit primes and t is up to 32 bits, it is sufficient to take $k' = k + 1$. For smaller CRT primes or larger values of t, a higher value of k' may be needed.
  Below we denote for all j, $p_j^* := p/p_j$ and $\tilde{p}_j := [(p_j^*)^{-1}]_{p_j}$. We also denote $Q := qp$, and for every i, j we have $Q_i^* := Q/q_i = q_i^* p$, $Q_j'^* := Q/p_j = q p_j^*$, and also $\tilde{Q}_i = [(Q_i^*)^{-1}]_{q_i}$ and $\tilde{Q'}_j = [(Q_j'^*)^{-1}]_{p_j}$.
- We pre-compute and store all the values that are needed in the procedure from Section 2.2 to extend the CRT basis $\{q_1, \ldots, q_k\}$ by each of the p_j's, as well the values that are needed to extend the CRT basis $\{p_1, \ldots, p_{k'}\}$ by each of the q_i's. Namely for all i, j we store the single-precision integers $\mu_{i,j} = [q_i^*]_{p_j}$ and $\nu_{i,j} = [p_j^*]_{q_i}$, as well as $\phi_j = [q]_{p_j}$ and $\psi_i = [p]_{q_i}$.
- We also pre-compute and store all the values that are needed for the complex CRT scaling procedure in Section 2.4. Namely, we pre-compute all the values $\frac{t\tilde{Q}_i p}{q_i}$, breaking them into their integral and fractional parts, $\frac{t\tilde{Q}_i p}{q_i} = \omega_i' + \theta_i'$ with $\omega_i' \in \mathbb{Z}_p$ and $\theta_i' \in [-\frac{1}{2}, \frac{1}{2})$. We store all the θ_i''s as double floats, for every i, j we store the single-precision integer $\omega_{i,j}' = [\omega_i']_{p_j}$, and for every j we also store $\lambda_j := [t\tilde{Q'}_j p_j^*]_{p_j}$.

4.3 Key-generation and encryption

The key-generation and encryption procedures are implemented in a straightforward manner. Small integers such as noise and key coefficients are drawn from χ_e or χ_k and stored as single-precision integers, while uniform elements in $a \leftarrow \mathbb{Z}_q$ are chosen directly in the CRT basis by drawing uniform values $a_i \in \mathbb{Z}_{q_i}$ for all i.

Operations in R_q are implemented directly in CRT representation, often requiring the computation of the number-theoretic-transform (NTT) modulo the separate q_i's. The only operations that require computations outside of R_q are decryption and homomorphic multiplications, as described next.

4.4 Decryption

Given the ciphertext $\mathbf{ct} = (c_0, c_1)$ and secret key $\mathbf{sk} = (1, s)$, we first compute the inner product in R_q, setting $x := [c_0 + c_1 s]_q$. We obtain the result in coefficient representation relative to the CRT basis $q_1, \ldots, q_k$. Namely for each coefficient

of x (call it $x_\ell \in \mathbb{Z}_q$) we have the CRT components $x_{\ell,i} = [x_\ell]_{q_i}$, $i = 1, \ldots, k, \ell = 0, \ldots, n-1$.

We then apply to each coefficient x_ℓ the simple scaling procedure from Section 2.3. This yields the scaled coefficients $m_\ell = [\lceil t/q \cdot x_\ell \rfloor]_t$, representing the element $m = [\lceil t/q \cdot x \rfloor]_t \in R_t$, as needed.

As we explained in Section 2.3, in the context of decryption we can ensure correctness by controlling the noise to guarantee that each $t/q \cdot x_\ell$ is within $1/4$ of an integer, and limit the size of the q_i's to 47 bits to ensure that the error is bounded below $1/4$.

Decryption complexity. The dominant factor in decryption is NTTs modulo the individual q_i's, that are used to compute the inner product $x := [c_0 + c_1 s]_q \in R_q$. Specifically we need $2k$ of them, k in the forward direction (one for each $[c_1]_{q_i}$) and k inverse NTTs (one for each $[c_1 s]_{q_i}$). These operations require $O(kn \log n)$ single-precision modular multiplications, where $n = \phi(m)$ is the degree of the polynomials and k is the number of moduli q_i. Once this computation is done, the simple CRT scaling procedure takes kn floating-point multiplications and kn integer multiplications modulo t. (The more sophisticated CRT scaling procedure in [3] takes $2kn$ modular multiplications after the NTT computations.)

4.5 Homomorphic Multiplication

The input to homomorphic multiplication is two ciphertexts $\mathbf{ct}^1 = (c_0^1, c_1^1), \mathbf{ct}^2 = (c_0^2, c_1^2)$, where each $c_b^a \in R_q$ is represented in the power basis with each coefficient represented in the CRT basis $\{q_i\}_{i=1}^k$. The procedure consists of three steps, where we first compute the "double-precision" elements $c'_0, c'_1, c'_2 \in R$, then scale them down to get $c_i^* := [\lceil t/q \cdot c'_i \rfloor]_q$, and finally apply relinearization.

Multiplication with double precision. We begin by extending the CRT basis using the procedure from Section 2.2. For each coefficient x in any of the c_b^a's, we are given the CRT representation $(x_1, \ldots, x_k)$ with $x_i = [x]_{q_i}$ and compute also the CRT components $(x'_1, \ldots, x'_{k'})$ with $x'_j = [x]_{p_j}$. This gives us a representation of the same integer x, in the larger ring $\mathbb{Z}_{qp}$, which in turn yields a representation of the c_b^a's in the larger ring R_{qp}.

Next we compute the three elements $c'_0 := [c_0^1 c_0^2]_{pq}$, $c'_1 := [c_0^1 c_1^2 + c_1^1 c_0^2]_{pq}$ and $c'_2 := [c_1^1 c_1^2]_{pq}$, where all the operations are in the ring R_{qp}. By our choice of parameters (with p sufficiently larger than q), we know that there is no modular reduction in these expressions, so in fact we obtain $c'_0, c'_1, c'_2 \in R$. These elements are represented in the power basis, with each coefficient $x \in \mathbb{Z}_{qp}$ represented by $(x_1, \ldots, x_k, x'_1, \ldots, x'_{k'})$ with $x_i = [x]_{q_i}$ and $x'_j = [x]_{p_j}$.

Scaling back down to R_q. By our choice of parameters, we know that all the coefficients of the c'_ℓ's are integers in the range $[-qp/2t, qp/2t)$, as needed for the complex CRT scaling procedure from Section 2.4. We therefore apply that procedure to each coefficient $x \in \mathbb{Z}_{qp}$, computing $x^* = [\lceil t/q \cdot x \rfloor]_q$. This gives as the power-basis representation of the elements $c_\ell^* = [\lceil t/q \cdot c'_\ell \rfloor]_q \in R_q$ for $\ell = 0, 1, 2$.

Relinearization. For relinearization, we use the same technique as Bajard et al. [3]. Namely, at this point we have the elements $c_0^*, c_1^*, c_2^* \in R_q$ in CRT representation, $c_{\ell,i}^* = [c_\ell^*]_{q_i}$ (for $\ell = 0, 1, 2$ and $i = 1, \ldots, k$). To relinearize we use the relinearization gadget vectors (β_i, α_i) that were computed during key generation. For each q_i, we first compute $\tilde{c}_{0,i} := \left[\sum_{j=1}^{k} [\beta_j]_{q_i} \cdot c_{2,j}^*\right]_{q_i}$ and $\tilde{c}_{1,i} := \left[\sum_{j=1}^{k} [\alpha_j]_{q_i} \cdot c_{2,j}^*\right]_{q_i}$, and then $c_{0,i}^\times := [c_{0,i}^* + \tilde{c}_{0,i}]_{q_i}$ and $c_{1,i}^\times := [c_{1,i}^* + \tilde{c}_{1,i}]_{q_i}$.

This gives the relinearized ciphertext $\mathbf{ct}^\times = (c_0^\times, c_1^\times) \in R_q^2$, which is the output of the homomorphic multiplication procedure.

Correctness. Correctness of the CRT basis-extension and complex scaling procedures was discussed in Sections 2.2 and 2.4, respectively. As was explained in Section 2.4, for homomorphic multiplication we need not worry about rounding errors in the simple-scaling subroutine of the complex-scaling procedure. But we do need to worry about them in both the CRT-extension from the q_i's to the p_j's, and the CRT-extension from the p_j's back to the q_j's inside the complex-scaling procedure.

As explained in Section 2.2, the imprecision due to floating-point operations in the CRT-extension procedure can be bounded in our case by $\epsilon \leq 2^{-48}$. We apply this procedure to ciphertexts, which are pseudo-random, and thus the fractional part of the exact value before rounding is uniformly distributed in $[-\frac{1}{2}, \frac{1}{2})$ (at least heuristically). This means that for each coefficient to which we apply the CRT-extension, we have roughly an ϵ probability to be rounding to the second-nearest integer instead of the the nearest one. And as opposed to the scaling operation, incorrect rounding here is a real error, leading to the result being invalid. With $\epsilon \leq 2^{-48}$ and each ciphertext having less than 2^{17} coefficients, the probability of encountering an error when applying the CRT-extension procedure to a ciphertext is bounded below 2^{-31}. The implementation must watch out for the computed values (before rounding) falling in the plausible-error regions $\mathbb{Z} + \frac{1}{2} \pm \epsilon$, and take corrective measures when it happens.

One possibility for correcting these errors is simply to repeat the calculation with higher precision, thus shrinking the plausible-error regions. (Specifically, we would need to compute the ratios y_q/q_i with higher precision.) Alternatively, if we detect a possible error while processing some ciphertext $\mathbf{ct} = (c_0, c_1)$, we can try adding to it the zero encryption $\mathbf{pk}$ from the public key, getting another ciphertext $\mathbf{ct}' = [\mathbf{ct} + \mathbf{pk}]_q$ that encrypts the same value, then try again with the new ciphertext.[6] Since $\mathbf{pk}$ is pseudo-random, the result rerandomizes all the coefficients. This yields a Las-Vegas type procedure, where the probability of success in each trial is at least $1 - 2^{-31}$, so the expected number of trials is very close to one.

Multiplication complexity. As for decryption, here too the dominant factor is the NTTs that we must compute when performing multiplication operations in R_q and R_{qp}. Specifically we need to transform the four elements $c_b^a \in R_{qp}$ after the CRT extension in order to compute the three $c'_\ell \in R_{qp}$, then transform

[6] When applying CRT-extension to the extended ciphertext $\mathbf{ct}^*$, we can use the tensor $\mathbf{pk} \otimes \mathbf{pk}$ to rerandomize it if needed.

back the c'_ℓ's before scaling them back to R_q to get the c^*_ℓ's. For relinearization we need to transform all the elements $c^*_{2,i} \in R_q$ before multiplying them by the α_i's and β_i's, and also transform c^*_0, c^*_1 before we can add them. Each transform in R_q takes k single-precision NTTs, and each transform in R_{qp} takes $k + k'$ NTTs, so the total number of single-precision NTTs is $k^2 + 9k + 7k'$. Each transform takes $O(n \log n)$ multiplications, so the NTTs take $O(k^2 n \log n)$ modular multiplications overall. In our experiments, these NTTs account for 60-75% of the homomorphic multiplication running time.

In addition to these NTTs, we spend $4(k + k')n$ modular multiplications computing the c'_ℓ's in the transformed domain and $2k^2n$ modular multiplications computing the products $c^*_{2,i}\beta_i$ and $c^*_{2,i}\beta_i$ in the transformed domain. We also spend $4n(kk' + k + k')$ modular multiplications and $(k + 1)n$ floating-point operations in the CRT-extension procedure in Section 4.5, and additional $3n(2k'(k + 1) + k)$ modular multiplications and $2kn$ floating-point operations in the complex scaling in Section 4.5. Hence other than the NTTs, we have a total of $(3k + 1)n$ floating point operations and $(2k^2 + 10kk' + 11k + 14k')n$ modular multiplications. (The more sophisticated homomorphic multiplication procedure in [3] has essentially the same complexity, i.e., same coefficient for the quadratic term of k and similar coefficients for the linear terms of k and k'.)

5 Implementation Details and Performance Results

5.1 Implementation Details

Software Implementation. The BFV scheme based on the decryption and homomorphic multiplication algorithms described in this paper was implemented in PALISADE[7], a modular C++11 lattice cryptography library that supports several SHE and proxy re-encryption schemes based on cyclotomic rings [17]. The results presented in this work were obtained for a power-of-two cyclotomic ring $\mathbb{Z}[x]/\langle x^n + 1\rangle$, which supports efficient polynomial multiplication using Fermat Theoretic Transform [2]. For efficient modular multiplication implementation in NTT, scaling, and CRT basis extension, we used the Number Theory Library (NTL)[8] function MulModPrecon, which is described in Lines 5-7 of Algorithm 2 in [12]. All single-precision integer computations were done in unsigned 64-bit integers. Floating-point computations were done in IEEE 754 double-precision floating-point format.

Our implementation of the BFV scheme is publicly accessible (included in PALISADE starting with version 1.1).

Parameter Selection. To select the ciphertext modulus q, we used the correctness constraints presented in Section 3.5 of [14]. The polynomial multiplication expansion factor $\delta_R = \sup\{\|ab\|_\infty / \|a\|_\infty \|a\|_\infty : a, b \in R\}$ was set to $\sqrt{n}$ by applying the Central Limit Theorem since all dominant polynomial multiplication terms result from the multiplication of polynomials with zero-centered random

[7] https://git.njit.edu/palisade/PALISADE
[8] http://www.shoup.net/ntl/

coefficients. We empirically confirmed the correctness of this average-case analysis by running a large number of experiments for various multiplicative depths.

To choose the ring dimension n, we ran the LWE security estimator[9] (commit 58662bc) [1] to find the lowest security levels for the uSVP, decoding, and dual attacks following the standard homomorphic encryption security recommendations [7]. We selected the least value of the number of security bits λ for all 3 attacks on classical and quantum computers based on the estimates for the BKZ sieve reduction cost model.

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 $\lambda > 128$.

Loop parallelization. Multi-threading in our implementation is achieved via OpenMP[10]. The loop parallelization in the scaling and CRT basis 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).

Experimental setup. We ran the experiments in PALISADE version 1.1, which includes NTL version 10.5.0 and GMP version 6.1.2. The evaluation environment for the single-threaded experiments was a commodity desktop computer system with an Intel Core i7-3770 CPU with 4 cores rated at 3.40GHz and 16GB of memory, running Linux CentOS 7. The compiler was g++ (GCC) 5.3.1. The evaluation environment for the multi-threaded experiments was a server system with 2 sockets of 16-core Intel Xeon E5-2698 v3 at 2.30GHz CPU (which is a Haswell processor) and 250GB of RAM. The compiler was g++ (GCC) 4.8.5.

5.2 Results

Single-threaded mode. Table 1 presents the timing results for the range of multiplicative depths L from 1 to 50 for the single-threaded mode of operation. It also demonstrates the contributions of CRT basis extension, scaling, and NTT to the total homomorphic multiplication time (excluding the relinearization).

Table 1 suggests that the relative contribution of CRT basis extension and scaling operations to the homomorphic multiplication runtime (without relinearization) grows from 34% at $L = 1$ up to 46% for $L = 50$. The remaining execution time is dominated by NTT operations. Our complexity and profiling analysis indicated that this increase in relative execution time is due to the $O(k^2n)$ modular multiplications needed for CRT basis extension and scaling operations, which start contributing more than the $O(kn \log n)$ modular multiplications in the NTT operations for polynomial multiplications as k increases.

Our profiling analysis of CRT basis extension and scaling showed that the contribution of floating-point operations to each of these functions was under 5%,

[9] https://bitbucket.org/malb/lwe-estimator
[10] http://www.openmp.org/

which corresponded to at most 2.5% of the total homomorphic multiplication time (for most settings, this contribution did not exceed 1%). This result justifies the practical use of our much simpler algorithms, as compared to [3], considering that both approaches have approximately the same computational complexity.

Table 1 also shows that the contribution of the relinearization procedure to the total homomorphic multiplication time grows from 10% ($L = 1$) to 45% ($L = 50$) due to the quadratic dependence of the number of NTTs in the relinearization procedure on the number of coprime moduli k.

The profiling of the decryption operation showed that only 6% ($L = 50$) to 10% ($L = 10$) was spent on CRT scaling while at least 75% was consumed by NTT operations and up to 10% by component-wise vector products. This supports our analysis, asserting that the decryption operation is dominated by NTT, and the effect of the scaling operation is insignificant.

Since the performance is dominated by NTTs (for which there is no difference between our implementation and the one in [3]), the runtimes of decryption and homomorphic multiplication are very similar between the two implementations.

Table 1: Timing results for decryption, homomorphic multiplication, and relinearization in the single-threaded mode; $t = 2$, $\log_2 q_i \approx 47$, $\lambda > 128$

L	n	$\log_2 q$	k	Dec. [ms]	Mul. [ms]	Relin. [ms]	Multiplication [%]		
							CRT ext.	Scaling	NTT
1	2^{12}	94	2	0.76	15.9	1.76	27	7	60
5	2^{13}	141	3	2.30	46.3	7.42	26	8	62
10	2^{14}	235	5	7.79	158	39.8	24	9	62
20	2^{14}	376	8	13.0	258	91.6	25	12	59
30	2^{15}	564	12	42.3	858	476	26	14	56
50	2^{16}	940	20	149	3,339	2,705	28	18	51

Multi-threaded mode. Table 2 illustrates the runtimes for $L = 20$ on a 32-core server system when the number of threads is varied from 1 to 32. The highest runtime improvement factors for decryption and homomorphic multiplication (with relinearization) are 4.5 and 6.0, respectively.

The decryption runtime is dominated by NTT, and the NTTs are parallelized at the level of CRT moduli (parameter k, which is 8 in this case). Table 2 shows that the maximum improvement is indeed achieved at 8 threads. Any further increase in the number of threads increases the overhead related to multi-threading without providing any improvement in speed. The theoretical maximum improvement factor of 8 is not reached most likely due to the distribution of the load between the cores of two sockets in the server. A more careful fine-tuning of OpenMP thread affinity settings would be needed to achieve a higher improvement factor, which is beyond the scope of this work.

The runtime of homomorphic multiplication (without relinearization) shows a more significant improvement with increase in the number of threads: it continues improving until 32 threads and reaches the speedup of 7.4 compared to the

Table 2: Timing results with multiple threads for decryption, multiplication, and relinearization, for the case of $L = 20, n = 2^{14}, k = 8$ from Table 1

# of threads	Dec. [ms]	Mul. [ms]	Relin. [ms]	Mul. + Relin. [ms]
1	13.59	266.8	107.4	374.2
2	7.53	155.0	57.0	212.0
3	5.95	112.1	46.4	158.5
4	4.45	90.7	34.0	124.7
5	4.61	79.3	35.2	114.5
6	4.68	72.0	35.9	107.9
7	4.61	67.5	36.0	103.5
8	3.04	58.5	24.5	83.0
9	3.19	49.8	25.1	74.9
16	3.46	46.6	25.7	72.3
17	3.42	40.6	25.8	66.4
32	3.47	35.9	26.9	62.7

single-threaded execution time. This effect is due to the CRT basis extension and scaling operations, which are parallelized at the level of polynomial coefficients (parameter $n = 2^{14}$). However, as the contribution of NTT operations is high (nearly 60% for the single-threaded mode, as illustrated in Table 1), the benefits of parallelization due to CRT basis extension and scaling are limited (their relative contribution becomes smaller as the number of threads increases).

The relinearization procedure is NTT-bound and, therefore, shows approximately the same relative improvement as the decryption procedure, i.e., a factor of 4.4, which reaches its maximum value at 8 threads.

In summary, our analysis suggests that the proposed CRT basis extension and scaling operations parallelize well (w.r.t. ring dimension n) but the overall parallelization improvements of homomorphic multiplication and decryption largely depend on the parallelization of NTT operations. In our implementation, no intra-NTT parallelization was applied and thus the overall benefits of parallelization were limited.

6 Conclusion and Future Work

In this work we described simpler alternatives to the CRT basis-extension and scaling procedures of Bajard et al. [3], and implemented them in the PALISADE library [16]. There is still plenty of room for improving this implementation; a few directions that we plan to experiment with include the following:

– *Better relinearization.* Fan and Vercauteren described in [8] an alternative relinearization procedure, using techniques similar to the BGV-relinearization in [10]. Implementing this method could save a significant number of NTTs.

– *Larger moduli.* We also plan to use higher-precision floating-point numbers to support larger CRT moduli. Switching from 47-bit moduli to 60-bit moduli would result in fewer moduli and hence fewer operations.

We hope that using the above approaches, we can obtain an implementation of the BFV scheme with performance close to the BGV implementation from [11].

References

1. Albrecht, M., Scott, S., Player, R.: On the concrete hardness of learning with errors. Journal of Mathematical Cryptology 9(3), 169203 (10 2015)
2. Aysu, A., Patterson, C., Schaumont, P.: Low-cost and area-efficient fpga implementations of lattice-based cryptography. In: 2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST). pp. 81–86 (June 2013)
3. Bajard, J.C., Eynard, J., Hasan, M.A., Zucca, V.: A full rns variant of fv like somewhat homomorphic encryption schemes. In: Avanzi, R., Heys, H. (eds.) SAC 2016. pp. 423–442 (2017)
4. Bos, J.W., Lauter, K., Loftus, J., Naehrig, M.: Improved security for a ring-based fully homomorphic encryption scheme. In: IMACC 2013. pp. 45–64 (2013)
5. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical gapsvp. In: CRYPTO 2012 - Volume 7417. pp. 868–886 (2012)
6. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: ITCS '12. pp. 309–325 (2012)
7. Chase, M., Chen, H., Ding, J., Goldwasser, S., Gorbunov, S., Hoffstein, J., Lauter, K., Lokam, S., Moody, D., Morrison, T., Sahai, A., Vaikuntanathan, V.: Security of homomorphic encryption. Tech. rep., HomomorphicEncryption.org, Redmond WA (July 2017)
8. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144 (2012)
9. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: STOC '09. pp. 169–178 (2009)
10. Gentry, C., Halevi, S., Smart, N.: Homomorphic evaluation of the AES circuit. In: "CRYPTO 2012". LNCS, vol. 7417, pp. 850–867 (2012)
11. Halevi, S., Shoup, V.: Design and implementation of a homomorphic-encryption library. `https://shaih.github.io/pubs/he-library.pdf` (2013)
12. Harvey, D.: Faster arithmetic for number-theoretic transforms. Journal of Symbolic Computation 60, 113 – 119 (2014)
13. Kawamura, S., Koike, M., Sano, F., Shimbo, A.: Cox-rower architecture for fast parallel montgomery multiplication. In: EUROCRYPT 2000. pp. 523–538 (2000)
14. Lepoint, T., Naehrig, M.: A comparison of the homomorphic encryption schemes fv and yashe. In: AFRICACRYPT 2014. pp. 318–335 (2014)
15. Lyubashevsky, V., Peikert, C., Regev, O.: A toolkit for ring-lwe cryptography. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. pp. 35–54 (2013)
16. Polyakov, Y., Rohloff, K., Ryan, G.W.: PALISADE lattice cryptography library. `https://git.njit.edu/palisade/PALISADE` (Accessed January 2018)
17. Polyakov, Y., Rohloff, K., Sahu, G., Vaikuntanathan, V.: Fast proxy re-encryption for publish/subscribe systems. ACM Trans. Priv. Secur. 20(4), 14:1–14:31 (2017)
18. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. J. ACM 56(6) (2009)
19. Shenoy, A.P., Kumaresan, R.: Fast base extension using a redundant modulus in rns. IEEE Transactions on Computers 38(2), 292–297 (Feb 1989)

A Appendices

A.1 Alternative variant of complex scaling in CRT representation

This section presents an alternative (slightly more efficient) variant of complex scaling in CRT representation. This variant has a reduced size requirement for p (by a factor of t) and very similar computational complexity.

The input is $x \in \mathbb{Z}_{qp}$, represented in the CRT basis $\{q_1, \ldots, q_k, p_1, \ldots, p_{k'}\}$. We need to scale it by t/q and round, and we want the result modulo q in the CRT basis $\{q_1, \ldots, q_k\}$. Namely, we want to compute $\left[\lceil t/q \cdot x \rfloor\right]_{q_i}$ for all i. We combine techniques from the two procedures above, computing the ratio υ as in Section 2.2, then computing Eq. 3 modulo each of the q_i's similarly to Section 2.3. Let us denote: $Q := qp$, $Q_i^* := Q/q_i = q_i^* p$, ${Q'_j}^* := Q/p_j = q p_j^*$, and also $\tilde{Q}_i = [(Q_i^*)^{-1}]_{q_i}$ and $\tilde{Q}'_j = [({Q'_j}^*)^{-1}]_{p_j}$. Then by Eq. 3 we have

$$\begin{aligned}\frac{t}{q} \cdot x &= \frac{t}{q}\Big(\sum_{i=1}^{k} [x_i \tilde{Q}_i]_{q_i} Q_i^* + \sum_{j=1}^{k'} [x'_j \tilde{Q'}_j]_{p_j} {Q'_j}^* + \upsilon Q\Big) \\ &= \sum_{i=1}^{k} [x_i \tilde{Q}_i]_{q_i} \cdot tp/q_i + \sum_{j=1}^{k'} [x'_j \tilde{Q'}_j]_{p_j} \cdot tp_j^* + \upsilon tp, \end{aligned} \tag{5}$$

and the ratio υ is computed as

$$\upsilon = \left\lceil \frac{\sum_{i=1}^{k} [x_i \tilde{Q}_i]_{q_i} Q_i^* + \sum_{j=1}^{k'} [x'_j \tilde{Q'}_j]_{p_j} {Q'_j}^*}{Q} \right\rfloor = \left\lceil \sum_{i=1}^{k} \frac{[x_i \tilde{Q}_i]_{q_i}}{q_i} + \sum_{j=1}^{k'} \frac{[x'_j \tilde{Q'}_j]_{p_j}}{p_j} \right\rfloor .$$

We thus compute $y_i := [x_i \tilde{Q}_i]_{q_i}$ and $z_i = y_i/q_i$ for all i, and $y'_j := [x'_j \tilde{Q'}_j]_{p_j}$ and $z'_j = y'_j/p_j$ for all j, and set $\upsilon := \left\lceil \sum_i z_i + \sum_j z'_j \right\rfloor$.

As in Section 2.3, we pre-compute all the values $\frac{tp/q_i}{q_i}$, breaking them into their integral and fractional parts, $\frac{tp}{q_i} = \omega'_i + \theta'_i$ with $\omega'_i \in \mathbb{Z}_{tp}$ and $\theta'_i \in [-\frac{1}{2}, \frac{1}{2})$. We store all the θ'_i's as double floats, and for every i, i' we store the single-precision integer $\omega'_{i,i'} = [\omega'_{i'}]_{q_i}$. In addition, and for every i, j we store $\zeta_{i,j} = [tp_j^*]_{q_i}$, and for every i we store $\lambda_j := [tp]_{q_i}$.

On inputs $(x_1, \ldots, x_k, x'_1, \ldots, x'_{k'})$ we compute υ and all the y_i's and y'_j's as above, then compute Eq. 5 modulo each of the q_i's, by setting

$$v' := \lceil \textstyle\sum_i \theta'_i y_i \rfloor \text{ , and for all } i \; w'_i := \left[\sum_{i'} y_{i'} \omega'_{i,i'} + \sum_j y'_j \zeta_{i,j} + \upsilon \lambda_j \right]_{q_i}.$$

The complex scaling procedure returns $[\lceil t/q \cdot x \rfloor]_{q_i} = [v' + w'_i]_{q_i}$ for all i.

Correctness. This procedure has two possible sources of errors, namely the rounding operation when computing the ratio υ, and the rounding operation when computing v'. The error when computing υ is handled just as in the CRT-extension procedure from Section 2.2: The inaccuracy there is bounded for our

parameters by $\epsilon \leq 2^{-48}$, and we must check that the value before rounding does not fall in the error region $\mathbb{Z} + \frac{1}{2} \pm \epsilon$ and take some corrective measures if it does.

Regarding the error when setting the value $v' := \lceil \sum_i \theta'_i y_i \rfloor$, we can bound the floating-point inaccuracy before rounding below $1/4$, just as in the simple scaling procedure from Section 2.3. However, when we use complex scaling during homomorphic multiplication, we do not have the guarantee that the exact value before rounding is close to an integer. We thus may encounter rounding errors where instead of rounding to the nearest integer, we will round to the second nearest.

For our purposes, when we use complex scaling in the context of homomorphic multiplication, such "rounding errors" are perfectly acceptable, as the rounding error is only added to the ciphertext noise. Specifically, when analyzing the scaled ciphertext, we have a term of the form $\langle \mathbf{sk}^*, \epsilon \rangle$, where ϵ is the rounding error and we rely on it to be small (cf. Eq. 6 in Section A.2). Had we computed v with full accuracy, we could argue that the coefficients of ϵ are bounded in magnitude below $1/2$. But since we have an additional floating-point error of upto $1/4$, in our case some coefficients of ϵ could be as large as $3/4$. This adds less than one bit to the noise even in the worst case, in our tests the actual effect on the noise was too small to be noticed.

Complexity analysis. Computing v and the y_i's and y'_j's takes $k + k'$ modular multiplications and $k + k' + 1$ floating point operations. Then computing v' takes $k + 1$ more floating-point operations, and computing each w'_i takes $k + k' + 1$ modular multiplications. In total, complex CRT scaling therefore takes $2k + k' + 2$ floating point operations and $kk' + k^2 + 2k + k'$ single-precision modular multiplications.

A.2 Brakerski's Scheme

The starting point for Brakerski's scheme is Regev's encryption scheme [18], with plaintext space $\mathbb{Z}_t$ for some modulus $t > 1$, where secret keys and ciphertexts are dimension-n vectors over $\mathbb{Z}_q^n$ for some other modulus $q \gg t$. (Throughout this section we assume for simplicity of notations that q is divisible by t. It is well known that this condition in superfluous, however, and replacing q/t by $\lceil q/t \rfloor$ everywhere works just as well.)

The decryption invariant of this scheme is that a ciphertext $\mathbf{ct}$, encrypting a message $m \in \mathbb{Z}_t$ relative to secret key $\mathbf{sk}$, satisfies

$$[\langle \mathbf{sk}, \mathbf{ct} \rangle]_q = m \cdot q/t + e, \text{ for a small noise term } |e| \ll q/t,$$

where $\langle \cdot, \cdot \rangle$ denotes inner product. Decryption is therefore implemented by setting $m := \left[\left\lceil \frac{t}{q} \cdot [\langle \mathbf{sk}, \mathbf{ct} \rangle]_q \right\rfloor \right]_t$. [11] Homomorphic addition of two ciphertext vectors $\mathbf{ct}_1, \mathbf{ct}_2$ consists of just adding the two vectors over $\mathbb{Z}_q$, and has the effect

[11] We ignore the encryption procedure in this section, since it is mostly irrelevant for the current work. For suitable choices, Regev proved that this encryption scheme is CPA-secure under the LWE assumption.

of adding the plaintexts and also adding the two noise terms. Homomorphic multiplication is more involved, consisting of the following parts:

Key generation. In Brakerski's scheme, the secret key $\mathbf{sk}$ must also be small, namely $\|\mathbf{sk}\| \ll q/t$. Moreover, the public key includes a "relinearization gadget", consisting of $\log q$ matrices $W_i \in \mathbb{Z}_q^{n \times n^2}$. Denoting the tensor product of $\mathbf{sk}$ with itself (over $\mathbb{Z}$) by $\mathbf{sk}^* = \mathbf{sk} \otimes \mathbf{sk} \in \mathbb{Z}^{n^2}$, the relinearization matrices satisfy

$$[\mathbf{sk} \times W_i]_q = 2^i \mathbf{sk}^* + \mathbf{e}_i^*, \text{ for a small noise term } \|\mathbf{e}^*\| \ll q/t.$$

Homomorphic multiplication. Let $\mathbf{ct}_1, \mathbf{ct}_2$ be two ciphertexts, satisfying the decryption invariant $[\langle \mathbf{sk}, \mathbf{ct}_i \rangle]_q = m_i \cdot q/t + e_i$. Homomorphic multiplication consists of:

1. **Tensoring.** Taking the tensor product $\mathbf{ct}_1 \otimes \mathbf{ct}_2$ *without modular reduction*, then scaling down by t/q, hence getting $\mathbf{ct}^* := \left[\lceil t/q \cdot \mathbf{ct}_1 \otimes \mathbf{ct}_2 \rfloor \right]_q$.
2. **Relinearization.** Decomposing $\mathbf{ct}^*$ into bits $\mathbf{ct}_i^* \in \{0,1\}^{n^2}$ (where $\mathbf{ct}^* = \sum_i 2^i \mathbf{ct}_i^*$), then setting $\mathbf{ct}^\times := [\sum_i W_i \times \mathbf{ct}_i^*]_q$.

To see that $\mathbf{ct}^\times$ is indeed an encryption of the product $m_1 m_2$ relative to $\mathbf{sk}$, denote the rational vector before rounding by $\mathbf{ct}' = t/q \cdot \mathbf{ct}_1 \otimes \mathbf{ct}_2$, and the rounding error by ϵ (so $\mathbf{ct}^* = \epsilon + \mathbf{ct}' + q \cdot \text{something}$), and we have

$$\begin{aligned}
\langle \mathbf{sk}^*, \mathbf{ct}' \rangle &= \left\langle \mathbf{sk} \otimes \mathbf{sk}, \tfrac{t}{q} \mathbf{ct}_1 \otimes \mathbf{ct}_2 \right\rangle = t/q \cdot (\langle \mathbf{sk}, \mathbf{ct}_1 \rangle \cdot \langle \mathbf{sk}, \mathbf{ct}_2 \rangle \\
&= t/q \cdot (m_1 \cdot q/t + e_1 + k_1 q)(m_2 \cdot q/t + e_2 + k_2 q) \\
&= m_1 m_2 \cdot q/t + \underbrace{e_1 m_2 + m_1 e_2 + e_1 e_2 t/q + t(k_1 + k_2)}_{e' \ll q/t} + q \cdot \text{something}.
\end{aligned}$$

Including the rounding error, and since $\mathbf{sk}$ is small (and hence so is $\mathbf{sk}^*$), we get

$$\langle \mathbf{sk}^*, \mathbf{ct}^* \rangle = \langle \mathbf{sk}^*, \epsilon + \mathbf{ct}' + \mathbf{k}^* q \rangle = m_1 m_2 \cdot q/t + \underbrace{e' + \langle \mathbf{sk}^*, \epsilon \rangle}_{e'' \ll q/t} + q \cdot \text{something}, \tag{6}$$

so $\mathbf{ct}^*$ encrypts $m_1 m_2$ relative to $\mathbf{sk}^*$. After relinearization, we have

$$\begin{aligned}
&\left\langle \mathbf{sk}, \mathbf{ct}^\times \right\rangle = \mathbf{sk} \times \sum_i W_i \times \mathbf{ct}_i^* = \sum_i \left\langle (2^i \mathbf{sk}^* + \mathbf{e}_i^*), \mathbf{ct}_i^* \right\rangle \\
&= \left\langle \mathbf{sk}^*, \sum_i 2^i \mathbf{ct}_i^* \right\rangle + \sum_i \langle \mathbf{e}_i^*, \mathbf{ct}_i^* \rangle = m_1 m_2 \cdot q/t + \underbrace{e'' + \sum_i \langle \mathbf{e}_i^*, \mathbf{ct}_i^* \rangle}_{\tilde{e}} \pmod q.
\end{aligned}$$

Since the $\mathbf{ct}_i^*$'s are small then so is the noise term $\tilde{e}$, as needed.