Faster Homomorphic Trace-Type Function Evaluation

YU ISHIMAKI (Member, IEEE), AND HAYATO YAMANA (Member, IEEE)
Waseda University, Tokyo 169-8555, Japan
Corresponding author: Yu Ishimaki (yuishi@yama.info.waseda.ac.jp)

This work was supported in part by JST CREST, Japan, under Grant JPMJCR1503, and in part by the Japan–US Network Opportunity 2 by the Commissioned Research of National Institute of Information and Communications Technology, Japan.

ABSTRACT Homomorphic encryption enables computations over encrypted data without decryption, and can be used for outsourcing computations to some untrusted source. In homomorphic encryption based on the hardness of ring-learning with errors, offering promising security and functionality, a plaintext is represented by a polynomial. A plaintext is treated as a vector whose homomorphic evaluation enables component-wise addition and multiplication, as well as rotation across the components. We focus on a commonly used and time-consuming subroutine that enables homomorphically summing-up the components of the vector or homomorphically extracting the coefficients of the polynomial, and call it homomorphic trace-type function. We improve the efficiency of the homomorphic trace-type function evaluation. The homomorphic trace-type function evaluation is performed by repeating homomorphic rotation followed by addition (rotations-and-sums). To correctly add up a rotated ciphertext and an unrotated one, a special operation called key-switching should be performed on the rotated one. As key-switching is computationally expensive, the rotations-and-sums is inherently inefficient. We propose a more efficient trace-type function evaluation by using loop-unrolling, which is compatible with other optimization techniques such as hoisting, and can exploit multi-threading. We show that the rotations-and-sums is not the optimal solution in terms of runtime complexity and that a trade-off exists between time and space. Experimental results demonstrate that our proposed method works 1.32–2.12 times faster than the previous method.

INDEX TERMS Homomorphic encryption, secure outsourcing, ring-learning with errors.

I. INTRODUCTION
Homomorphic encryption (HE) allows computations over encrypted data without decryption. Since Gentry’s first construction of fully HE [1], numerous improvements in its theory and implementation have been made to increase efficiency as well as different types of computation (e.g., [2]–[5]).

These subsequent schemes are based on the hardness of learning with errors (LWE) problem [6] or its ring variant called ring LWE (RLWE) problem [7]. RLWE-based HE schemes provide amortization because a plaintext is represented by a polynomial. This allows one to pack multiple messages into the polynomial coefficients, thereby performing homomorphic operations such as component-wise addition, polynomial multiplication (convolution), and rotation (with the change of signs for a certain case). Moreover, the polynomial can be seen as a vector via batching [4], [8], where each of the elements is called plaintext slot. The batching enables homomorphic operations such as component-wise additions, multiplications, and rotations without changing signs across the slots.

Various useful methods for homomorphic function evaluation are expressed by a rotation followed by an addition, and they are performed recursively. For instance, total-sums operation [9] sums up all the slot values, and the result is stored back to all the slots. Total-sums and its partial variant (i.e., adding up the consecutive slots partially) are used in several applications (e.g., [10]–[12]). These operations are performed by $O(\log N)$ cyclic rotations and additions for $O(N)$ slots. Moreover, the trace function isolates coefficients by recursively performing the rotations and sums as in total-sums but using different rotation amounts [13]–[15]. The trace function is used during conversion between RLWE/LWE ciphertexts [15] and coefficient selection [14]. We refer to these types of operations (i.e., successive rotations and sums) as trace-type functions.
Because of the functionality above, the trace-type functions have been used in many applications. However, their processing is computationally expensive compared with homomorphic addition and multiplication due to the successive application of homomorphic rotations. Specifically, the trace-type function evaluation is 2.49–5.43x (resp. 85.83–124.26x) slower than the homomorphic multiplication in the B/FV scheme\(^1\) (resp. the CKKS scheme) according to the benchmark provided in [18, Table 1].\(^2\)

A homomorphic rotation is realized by an automorphism, which not only rotates plaintext slots but also changes the corresponding secret key. To homomorphically evaluate the addition of rotated ciphertext with respect to the changed key and original ciphertext, the rotated key should be reverted back to the original one via key-switching, which is a computationally expensive operation. Thus, trace-type functions become inefficient due to the \(O(\log N)\) sequential iterations, involving automorphisms followed by the expensive key-switchings.

**Contribution:** In this paper, we strive to improve the efficiency of the trace-type function evaluation. To this end, we propose a more efficient trace-type function evaluation using loop-unrolling to reduce the number of expensive operations by leveraging a property of automorphisms, special modulus technique for key-switching, and using a multi-core environment, thus reducing the computational cost compared with the sequential method [9], [15] at the expense of slightly increasing the required storage. The high-level idea of our approach is to successively unroll consecutive subloops in the trace-type function evaluation. Loop-unrolling enables parallelization and optimization that cannot be implemented in the sequential method. We parameterize the number of iterations after the unrolling and demonstrate that the proposed method with a certain unrolling parameter outperforms the sequential method both in the single- and the multi-threading modes.

The remainder of this paper is structured as follows. Section II presents notations and background for our development. Section III details the proposed method, and its performance evaluation is reported in Section IV. In Section V, we describe related work. We conclude the paper in Section VI.

II. PRELIMINARIES

A. NOTATION

For an integer \(q\), we denote by \(\mathbb{Z}_q := \mathbb{Z}/q\mathbb{Z}\) a quotient ring which is identified by \([-q/2, q/2) \cap \mathbb{Z}\). Let \(N\) be a power-of-two integer. We denote by \(R = \mathbb{Z}[X]/(X^N + 1)\) a polynomial ring, and \(R_q = R/qR\) its quotient ring. We also denote by \([a]_q\) the reduction of \(a\) modulo \(q\) into the interval for some integer \(a\). We use \([a]\) to denote range \([0, 1, \ldots, a - 1] \cap \mathbb{Z}\) for some positive integer \(a\). We use \(\mathbb{Z}_{2N} := \{2k + 1\}_{k \in [N]}\) to denote a set of positive integers smaller than \(2N\). For a set \(S\), we denote by \(\overset{\leftarrow}{S}\) sampling of an element from \(S\) uniformly at random. We denote \(X\) by sampling of an element from a distribution \(X\). Symbol \(\oplus\) denotes homomorphic addition. We denote \(\langle \cdot, \cdot \rangle\) by the inner product of two vectors consisting of elements in a ring. \(| \cdot |\) represents the length of an array or size of a set.

B. RESIDUE NUMBER SYSTEM (RNS)

Ciphertext modulus is typically larger than the CPU word size (e.g., 64 bits). To efficiently handle the ciphertext data, the modulus is represented by a product of small moduli relatively co-prime to each other. Let \(Q_i\) be a ciphertext modulus such that \(Q_i = \prod_{\ell=0}^{|\ell|} q_i\) is a product of \(\ell + 1\) small integers, each fitting into the word size, and each pair of the small moduli is relatively co-prime. Then, with the Chinese Remainder Theorem, the addition and multiplication of elements over \(\mathbb{Z}_{Q_i}\) can be performed on its residues each in \(\mathbb{Z}_{q_i}\) independently in parallel. Similarly, an element in \(R_{Q_i}\) can be represented via an array of elements, each in \(R_{q_i}\), for each \(i \in [\ell + 1]\). This representation is called RNS representation, where the addition, multiplication, and automorphism on the element in \(R_{Q_i}\) can be performed on the element in \(R_{q_i}\) in parallel for each \(i\). The parameter \(\ell\) is reduced throughout the homomorphic evaluation (typically after homomorphic multiplications), and is upper bounded by \(L\). We call a ciphertext w.r.t. modulus \(Q_i\) a level-\(\ell\) ciphertext.

Typically, two different representations are used to express elements in each \(R_{q_i}\). Coefficient representation is the most natural representation as the coefficient with respect to the power basis is aligned as a vector. This representation allows the direct application of nonlinear operations (e.g., scaling, lifting, and word decomposition) on \(R_{q_i}\). On the other hand, the multiplication of two elements in \(R_{q_i}\) suffers from quadratic complexity in \(N\). The evaluation representation enables multiplication in \(O(N)\) at the cost of \(O(N \log N)\) complexity of conversion between these two representations. To convert the coefficient representation into the evaluation representation, the number theoretic transform (NTT) is required, whereas the inverse NTT is required in the reverse direction. The (inverse) NTT causes a bottleneck in HE implementations with Full-RNS capability [19]–[22].

Throughout the paper, we use a superscript within parentheses to denote a ring element with respect to a large modulus in the RNS representation. Namely, for \(x \in R_{Q_i}\), we denote its RNS representation by \([x]_i\), where \(x^{(i)} = [x]_{q_i}\), and call each of \(x^{(i)}\) an RNS component.

C. AUTOMORPHISM AND KEY-SWITCHING

**Automorphism:** For some \(t \in \mathbb{Z}_{2N}^*\), an automorphism is defined as a map \(\psi_t : R \rightarrow R, a(X) \mapsto a(X^t) \mod \Phi_{2N}(X)\). After applying an automorphism on a ciphertext, the corresponding secret key changes. The homomorphic evaluation between ciphertexts under different secret keys is not supported. To perform further computations on ciphertexts under different secret keys, one needs to switch the corresponding key into another via key-switching. If we apply
The idea behind RNS-friendly key-switching is to decompose each of the polynomials. The effect of an automorphism is to rotate the plaintext slots/coeficient of a polynomial by a specific amount [23].

Key-Switching: For some $c_1 \in RQ_l$, let $\text{Decomp}$ be a function that outputs a set of small elements $(c'_0, c'_1, \ldots, c'_{d-1}) \in R^d$ for decomposition length (or the number of digits $d$) such that $c_1 = \sum_{i=0}^{d-1} c'_i (\text{mod } Q_i)$ for some $B_i \in \mathbb{Z}$. Let $\chi$ be a discrete Gaussian distribution. Key-switching consists of the following two algorithms:

- **SwitchKeyGen($s$, $s'$):** Sample $a_i \leftarrow RQ_l$, $e_i \leftarrow \chi$ and compute $\text{swk}_i = (e_i - a_is' + B_is, a_i) \in R^2_{Q_l}$ for each $i$.
  Output $\text{swk}_i \rightarrow s'$ = $[\text{swk}_i]_{\text{e}(d)} \in R^{d \times 2}_{Q_l}$.

- **KeySwitch($c$, $\text{swk}_i \rightarrow s'$):** For ciphertext $c = (c_0, c_1) \in R^2_{Q_l}$, compute $\hat{c} = \text{Decomp}(c_1)$. Output $(c'_0, c'_1) \in R^2_{Q_l}$ that is decryptable by $s'$, where $c'_0 = c_0 + \langle \hat{c}, \text{swk}_i \rightarrow s'[0] \rangle$ and $c'_1 = \langle \hat{c}, \text{swk}_i \rightarrow s'[1] \rangle$.

Key-switching introduces noise, which can be mitigated by using the **special modulus** [24]–[26] described below. The difference from the key-switching procedure above is that the evaluation key produced by $\text{SwitchKeyGen}$ is defined over $R^2 \times 2$ rather than over $R^2 \times 2$ for an extra modulus $P = \prod_{i=0}^{k-1} P_i$ as large as the key-switching noise, where $p_i$ values are relatively co-prime.

Key-Switching With Special Modulus:
- **SwitchKeyGen($s$, $s'$):** Sample $a_i \leftarrow R_PQ_l$, $e_i \leftarrow \chi$ and compute $\text{swk}_i = (e_i - a_is' + PB(Is, a_i) \in R^2_{Q_l}$ for each $i$.
  Output $\text{swk}_i \rightarrow s'$ = $[\text{swk}_i]_{\text{e}(d)} \in R^{d \times 2}_{Q_l}$.

- **KeySwitch($c$, $\text{swk}_i \rightarrow s'$):** For ciphertext $c = (c_0, c_1) \in R^2_{Q_l}$ w.r.t. the secret key $s$, compute $\hat{c} = \text{Decomp}(c_1)$. Output $(c'_0, c'_1) \in R^2_{Q_l}$ where $c'_0 = c_0 + \langle \hat{c}, \text{swk}_i \rightarrow s'[0] \rangle/P$ and $c'_1 = \langle \hat{c}, \text{swk}_i \rightarrow s'[1] \rangle/P$.

We defer the details of the $\text{KeySwitch}$ with the RNS representation to the next section.

D. RNS-FRIENDLY KEY-SWITCHING

The idea behind RNS-friendly key-switching is to decompose $c_1 \in RQ_l$, into small ring elements via simple modular reductions by partial products of $Q_\ell$, followed by constant multiplications. Such modular reductions are almost free with the RNS representation, as RNS components of $c_1$ w.r.t. the partial products are already available. Hence, they can be performed by simply splitting RNS components of $c_1$. The size of the partial products determines the decomposition granularity and the size of the special modulus. Since the decomposition granularity is controllable [26], we can choose the best decomposition granularity, depending on a target computation and the security level.

More specifically, the decomposition granularity is controlled by the maximal number of digits $d$ for a particular $L$. Let $\alpha = (L + 1)/d$ be the number of RNS moduli for each digit, where $(L + 1)$ is divisible by $d$. We use $\beta = [\ell + 1]/\alpha$ to denote the number of digits for key-switching on a ciphertext at level $\ell$. The difference between $d$ and $\beta$ is due to the fact that $\ell$ changes during the homomorphic evaluation. Let $\hat{Q}_j = \{q_{j, \ell+1} \mid \ell \in \mathbb{Z}\}$ be partial product associated with the $j$-th digit. Then, for $x \in RQ_l$, the decomposition function is defined by $\text{Decomp}(x) = \{x \cdot \hat{Q}_{j\ell}^{-1} \hat{Q}_{j, \ell+1} \mid \ell \in \mathbb{Z}\}$ with $B_j = [\hat{Q}_j/ Q_l]_\ell$.

We denote $C_\ell = \{q_0, q_1, \ldots, q_l\}$ and $B = \{p_0, \ldots, p_{\ell-1}\}$ by bases that respectively correspond to ciphertext modulus $Q_\ell$ and the special modulus $P$. We further denote $C_j' = \{q_{j, \ell}, \ldots, q_{j, \ell+1-1}\}$ by a base corresponding to partial product $Q_j$. We use $D_i = \{q_j \in C_j' \mid \ell \in \mathbb{Z}\}$ and $C_j' = B_j \backslash C_j'$. Note that when $\alpha \nmid \ell + 1$, the $(\beta - 1)$-th digit consists of $(\ell + 1)$ mod $\alpha$ elements, because $\alpha \cap ((\ell + 1)$, mod $\alpha$ elements are all zeros and do not need to be stored.

Recall that the inner-product is required during key-switching. Using the special modulus technique, an evaluation key for level-$\ell$ ciphertext is defined over $R_{PQ_l}$. However, each of the small digits is represented over $R_{P_j}$. To make these digits interact with the evaluation key, we need to extend the base of each digit from $C_j'$ to $D_j$ through (approximate) fast base conversion [27] (see Appendix C for more details). Note that the output of the fast base conversion must be in coefficient representation, whereas the inner-product requires its input to be in evaluation representation to perform efficient multiplications. Thus, we need several (inverse) NTT calls during key-switching.

Below, we describe the key-switching procedure by following the implementation of PALISADE v1.9.2 [28]. We assume that both input and output ciphertexts are in evaluation representation. For a ciphertext $c = (c_0, c_1) \in R^2_{Q_l}$ whose RNS representation is $\{c_0^{(j)} \mid c_1^{(j)} \}$ for $j \in [\ell + 1]$, the key-switching procedure comprises the following subroutines, in which the computationally expensive steps consisting of (inverse) NTTs are underlined.

1. RNS-decomposition and modulus-raise: This step inputs an element in $RQ_l$, and outputs an array of elements $R_{PQ_l}$ of length $\beta$. Both input and output are in evaluation representation.

1-1. RNS-decomposition:
   This step performs $\text{Decomp}(c_1)$ in RNS representation. First, it splits the RNS component of $c_1$ into $\beta$ digits to get $[c_1]_{Q_\ell}^{-1}$ for $j \in [\beta]$ via simple copy-and-paste. Each digit consists of $\alpha$ RNS components. Second, pre-computed constant $Q_{j\ell}^{-1}$ is multiplied to each of the digits for $j \in [\beta]$. Thus, we set $c_1^{(j)} \leftarrow [c_1^{(j\ell+1)} \cdot Q_{j\ell}^{-1}]_{\text{q}_{j\ell+1}}$ for $i \in [\alpha]$ and $[j] \in \beta$ when $j\alpha + i \leq \ell$. Then, on each $c_1^{(\ell)}$, we perform inverse NTTs to apply fast base conversion, and store original values in evaluation representation for subsequent use.

3To make the description generic, we do not treat the last digit as a special case when $\alpha \nmid \ell + 1$. 

VOLUME 9, 2021

53063
1.2. Modulus-raise:
For each $j \in \{\beta\}$, this step extends the base $C'_j$ of the
digit $\{c^{(i)}_{0,j}\}_{i \in [a]}$ into $D_B$ via fast base conversion. Since $C'_j \subset D_B$, it
suffices to extend from $C'_j$ to $C''_j$ followed by performing NTTs to get the results in evaluation
representation for the inner product in the next step. By concatenating the resulting RNS components
and the original digits already available in evaluation representation,
we have each digit $\hat{c}_{1,j}$ as an element in $R_{PQ}$. Denoting by $\text{ModUp}_{C'_j \rightarrow D_B}$
these procedures, we compute
$$\hat{c}_{1,j} = \{\hat{c}^{(i)}_{1,j}\}_{i \in [k+\ell+1]} \leftarrow \text{ModUp}_{C'_j \rightarrow D_B}(c^{(i)}_{1,j} = \{c^{(i)}_{1,j}\}_{i \in [a]}).$$ The details of $\text{ModUp}_{C'_j \rightarrow D_B}$ are provided in
Algorithm 7 in Appendix C.

2. Inner Product: This step performs the inner product
of the digits $(\hat{c}_{1,0}, \ldots, \hat{c}_{1,\beta-1})$ and evaluation key
$\text{swk} = (\text{swk}_{0,0}, \ldots, \text{swk}_{\beta-1})$ both in evaluation representation, where $\hat{c}_{1,j} = \{\hat{c}^{(i)}_{1,j}\}_{i \in [k+\ell+1]}$ and $\text{swk}_j = \{\text{swk}^{(i)}_{j}\}_{i \in [k+1]}$. We compute
$$\hat{c}' = \{\hat{c}'^{(i)}\} = \{\hat{c}^{(i)}_{1,0} \oplus \text{swk}_{j(i)}\}_{i \in [k+1]},$$
where
$$\hat{c}'^{(i)} = \sum_{j=0}^{\beta-1} \hat{c}^{(i)}_{1,j} \cdot \text{swk}^{(i)}_{j} \pmod{p_i} \text{ for } i \in [k]$$
and
$$\hat{c}'^{(k+i)} = \sum_{j=0}^{\beta-1} \hat{c}^{(k+i)}_{1,j} \cdot \text{swk}^{(k+i)}_{j} \pmod{q_i} \text{ for } i \in [\ell+1].$$

3. Modulus-down:
For $\hat{c}'$ (in evaluation representation), this step performs
division by $P$ followed by rounding, which shrinks the base $D_B$ into $C_e$ as well as reducing the key-switching
noise. These are performed by fast base conversion followed by subtraction and constant multiplication. More
specifically, we first parse RNS components of $\hat{c}'$ w.r.t $B$, and apply fast base conversion from $B$ to $C_e$ on it.
Then, we subtract the resulting elements from $\hat{c}'$ w.r.t $C_e$ followed by multiplication by $P^{-1}$. As $\hat{c}'$ is in evaluation
representation, we need inverse NTTs before performing fast base conversion. We denote the procedure
at this step by $\text{RNSModDown}$ whose details are available in Appendix C (Algorithm 8). Thus, we compute
$$\{c_0^{(i)}, c_{\ell}^{(i)}\}_{i \in [\ell+1]} \leftarrow \text{RNSModDown}(\hat{c}_0^{(i)}, \ldots, \hat{c}_{\ell}^{(i)}, \{c_1^{(i)}\}_{i \in [\ell+1]} = \text{RNSModDown}(\hat{c}_0^{(i)}, \ldots, \hat{c}_{\ell}^{(i)+1}),$$
and output $\{c_0^{(i)} \oplus \text{swk}_{\alpha\beta}\}_{i \in [\ell+1]}$. Then, we use fast base conversion from $B$ to $C_e$. Because $|B| = k = \alpha$ and $|C_e| = \ell + 1$, we need
$\alpha(1 + (\ell + 1)N) = \alpha(\ell + 2)N$ MMNs. Next, constant multiplications are performed on $(\ell + 1)$ elements, which involve
$(\ell + 1)N$ MMNs. Finally, NTTs are performed on $(\ell + 1)$ resulting elements having $(\ell + 1)Nlog N$ MMNs. Therefore,
$(\alpha + 2(\ell + 1))Nlog N + (\alpha + 2\alpha + \ell + 1)N$ MMNs are needed for one element. By doubling this, Step 3 requires
$2(\alpha + 2(\ell + 1))Nlog N + 2(\alpha + 2\alpha + \ell + 1)N$ MMNs. Step 4 (Addition and output) does not require any MMNs.

Algorithm 1 Automorphism and Key-Switching (AKS)

Input:
- $t$: an automorphism index
- $c = (c_0, c_1) \in R_{Q_e}^2$: ciphertext \(\triangleright\) Eval. Rep.
- $\text{swk}_{\alpha\beta}\rightarrow x \in R_{PQ}^2$: evaluation key \(\triangleright\) Eval. Rep.

Output: ciphertext $c' \in R_{Q_e}^2$ \(\triangleright\) Eval. Rep.

1: $(c_0', c_1') \leftarrow c$
2: $(d_0, d_1) \leftarrow (\psi_t(c_0'), \psi_t(c_1'))$
3: $d \leftarrow \text{RNSDecompModUp}(d_1)$
4: $c_0' \leftarrow (d, \text{swk}_{\alpha\beta}\rightarrow x)[0]$
5: $c_1' \leftarrow (d, \text{swk}_{\alpha\beta}\rightarrow x)[1]$
6: $(c''_0, c''_1) \leftarrow (\text{RNSModDown}(c_0'), \text{RNSModDown}(c_1'))$
7: $(c_0', c_1') \leftarrow (d_0, 0) \oplus (c''_0, c''_1)$
8: return $(c_0', c_1')$

We outline the key-switching procedure performed immediately after the automorphism by index $t \in Z_{2N}^\ell$ in Algorithm 1. Whenever referring to Algorithm 1 (AKS), we omit the evaluation key $\text{swk}_{\alpha\beta}\rightarrow x$ if it is clear from the context. For the details of $\text{RNSDecompModUp}$ and $\text{RNSModDown}$, please see Algorithm 6 and Algorithm 8 in Appendix C, respectively.
E. HOMOMORPHIC TRACE-TYPE FUNCTION

A trace-type function can be expressed as recursive automorphisms-and-sums. We consider that the recursion proceeds $M \leq \log N$ times, as it is naturally derived from existing functionalities such as total-sums and trace. Both functionalities are often used with several choices of $M$, depending on applications. For instance, when $M = \log N$, the algorithm exactly corresponds to LWE-to-RLWE [15] and total-sums [9].

The choice of the generator depends on libraries. For instance, SEAL uses HEAAN and PALISADE uses $g = 5$.

\begin{algorithm}
\caption{Homomorphic Trace-Type Function Evaluation}
\textbf{Input:} ciphertext $c$, list of automorphism indices $k$
\textbf{Output:} ciphertext $c'$
\begin{algorithmic}[1]
\State $c' \leftarrow c$
\For {$i = 0$ to $M - 1$}
\State $c' \leftarrow c' \oplus \text{AKS}(k[i], c')$
\EndFor
\State \textbf{return} $c'$
\end{algorithmic}
\end{algorithm}

The framework of the trace-type function evaluation is described in Algorithm 2. The concrete functionality is determined by specifying the list of automorphism indices $k$. For instance, the trace is defined by setting $k = \{2^i + 1\}_{i \in [M]}$ [15], and goes down from the largest to smaller ring in a tower of rings. The total-sums procedure is defined by a list of automorphism indices $k = \{g^i\}_{i \in [M]}$ for some generator $g \in \mathbb{Z}_{2N}$, homomorphically permuting the slots by a power-of-two amount followed by addition at every iteration.

Despite the usefulness of the trace-type function, Algorithm 2 is inefficient because it requires $O(\log N)$ key-switching processes. In the next section, we provide a solution to reduce complexity.

III. PROPOSED LOOP-UNROLLING METHOD

The trace-type function is inherently sequential, with each loop consisting of $O(\log N)$ (inverse) NTT calls. To gain more efficiency, we strive to reduce the total number of (inverse) NTT calls by reducing the number of iterations via loop-unrolling. As $N$ typically ranges from $2^{10}$ to $2^{15}$ [29], unrolling all iterations is not efficient when $M$ is large (see section IV).

Instead, we successively unroll fewer iterations. Namely, let $M \leq \log N$ be the number of iterations required in Algorithm 2, and let $h(M)$ be the total number of iterations after successive unrollings. We unroll $\lceil M/h \rceil$ consecutive (and non-overlapping) iterations ($h - (M \mod h)$) times. If $(M \mod h) \neq 0$, we then unroll the remaining consecutive and non-overlapping $\lfloor M/h \rfloor + 1$ iterations ($M \mod h$) times.

By unrolling the iterations, consecutive automorphisms need to be applied on a ciphertext, which can be composed into a single one. This is because for some $t_1, t_2 \in \mathbb{Z}_2$, it holds that $\psi_{t_1} \circ \psi_{t_2} = \psi_{t_1 t_2}$ mod 2. For instance, we consider the total-sums. For $c \in \mathbb{R}^2$, when $M = 2$, Algorithm 2 computes $c' \leftarrow \text{AKS}(g, c) \oplus c'$ followed by $c'' \leftarrow \text{AKS}(g^2, c') \oplus c'$. By unrolling with $h = 1$, we obtain $c''' \leftarrow c \oplus \text{AKS}(g, c) \oplus \text{AKS}(g^2, c) \oplus \text{AKS}(g^3, c)$, where $\text{AKS}(g^3, c)$ term is derived from $\text{AKS}(g^2, \text{AKS}(g, c))$.

Thus, the number of automorphisms required per unrolled iteration is $O(2^{(\log_2 h)})$.

As $2^M = O(N)$, each unrolled iteration consists of $O(\sqrt{N})$ automorphisms and sums, and the number of required automorphism over all iterations becomes $O(h \sqrt{N})$. Due to the unrolling and composition, each unrolled loop requires additional automorphisms and the corresponding evaluation keys compared with Algorithm 2. As a result, this incurs higher storage complexity. Specifically, the increase of the complexity is substantial for smaller $h$ while slight for larger $h$, as shown in Subsection III-B. The proposed algorithm is outlined in Algorithm 3 as a generalization of the existing trace-type function evaluation.

\begin{algorithm}
\caption{Concept of Proposed Homomorphic Evaluation of Unrolled Trace-Type Function Evaluation}
\textbf{Input:}
\begin{itemize}
\item $c$: a ciphertext
\item $h$: # iterations after unrollings
\item $K = \{k_0, \ldots, k_{h-1}\}$: a set of arrays of automorphism indices for each iteration
\end{itemize}
\textbf{Output:} ciphertext $c'$
\begin{algorithmic}[1]
\State $c' \leftarrow c$
\For {$i = 0$ to $h - 1$}
\State $c' \leftarrow c' \oplus \text{AKS}(k_i[0], c') \oplus \ldots \oplus \text{AKS}(k_i[|k_i| - 1], c')$
\EndFor
\State \textbf{return} $c'$
\end{algorithmic}
\end{algorithm}

At a first glance, each of the iterations in Algorithm 3 requires different automorphisms on a ciphertext followed by key-switching, each consisting of the several sequential steps described in Subsection II-D. Indeed, Algorithm 3 can harness many optimizations to reduce the complexity of key-switchings, as described below.

A. OPTIMIZATION TECHNIQUES

The proposed algorithm harnesses many well-known optimization techniques that can be independently applied to each unrolled iteration. We here consider one unrolled iteration
consisting of \( n \) automorphisms and key-switching processes. In addition, we assume the automorphisms are applied to ciphertext \( c = (c_0, c_1) \in R_{Q_t}^\beta \) with indices \( t_0, t_1, \ldots, t_{n-1} \in \mathbb{Z}_{2N}^* \).

1) HOISTING, PRE-AUTOMORPHISMS, AND LAZY ModDown

**Hoisting and Pre-Automorphism:** Given that the automorphisms per unrolled iteration are applied to the same ciphertext, we can leverage the hoisting technique [22], [30]. Specifically, RNS-decomposition and modulus-raise described in SubSection II-D are performed once. The resulting decomposed element in \( R_{PQ_t}^\beta \) is reused for different automorphisms followed by key switching. Although the initial idea in [22] is to apply \( n \) automorphisms to the decomposed ring element, it incurs a non-negligible cost due to coping and performing the automorphisms on the decomposed ring element. Recently, Bossuat et al. [30] suggested avoiding automorphisms on the decomposed ring elements by using its inverse on the corresponding evaluation keys during the key-generation. This method avoids the automorphism on the array of large ring elements.

**Lazy Modulus-Down:** While the hoisting avoids \( n \) invocations of RNS-decomposition and modulus-raise, it still consists of \( n \) invocations of modulus-down. We further adopt lazy modulus-down as in [30] (where it is called the second layer of hoisting). With the hoisting and the automorphism on the evaluation keys, \( n \) distinct automorphisms are applied to \( c_0 \in R_{Q_t} \) and the result of the inner product over \( R_{PQ_t}^\beta \). Then, with the lazy modulus-down, the results of the inner product are summed before the modulus-down is performed. Specifically, our goal is to compute

\[
(\hat{c}_0, \hat{c}_1) = (c_0 + \sum_{e=0}^{n-1} \psi_{\ell_e}(c_0) + \left\lfloor \frac{\sum_{e} \psi_{\ell_e}((\text{swk}_e^\prime)[0], \hat{d}^\prime)}{P} \right\rfloor, \\
\quad \left\lfloor \frac{\sum_{e} \psi_{\ell_e}((\text{swk}_e^\prime)[1], \hat{d}^\prime)}{P} \right\rfloor),
\]

where \( \hat{d} \leftarrow \text{Decomp}(c_1) \in R_{PQ_t}^\beta \) and \( \text{swk}_e^\prime \leftarrow \psi_{\ell_e-1} \text{(swk}_e) \in R_{PQ_t}^{\beta \times 2} \) (performed during key generation). As a result, the number of (inverse) NTTs during the modulus-down is reduced down from \( O(n) \) in the naïve implementation to \( O(1) \).

2) PUTTING IT ALL TOGETHER

By incorporating the aforementioned techniques, the number of all operations in key-switchings before and after inner-product is reduced compared with the naïve approach to Algorithm 3. For convenience, we refer to RNS-decomposition, modulus-raise, and lazy modulus-down as hoisted operations. The hoisted operations are performed exactly \( h \) times. When \( h < M \), the number of (inverse) NTT calls are reduced compared with Algorithm 2 as expected. However, the number of inner-products is \( O(nh\sqrt{N}) \), which increases as \( h \) decreases, and becomes non-negligible for relatively large \( M \). Hence, we need to balance the complexities of hoisted operations and inner products.

As seen in the next subsection, the increase of the complexity of the hoisted operations and the increase of the complexity of the inner-products are asymmetric, thereby improving the computational efficiency by choosing a proper \( h \). This comes from the difference between the unit costs of the hoisted operations and inner-product (i.e., the cost for a single inner product is cheaper than those for hoisted operations).

**Algorithm 4** Homomorphic Evaluation of Unrolled Trace-Type Function Evaluation (HoistKS)

**Input:**
- \( c = (c_0, c_1) \): ciphertext
- \( k = (k_0, k_1, \ldots, k_{|k|-1}) \): array of automorphism indices
- \( \text{ek} = (\text{swk}_0, \ldots, \text{swk}_{\beta-1}, \hat{d}_0) \in (R_{PQ_t}^\beta)^{|k|} \)

**Output:** ciphertext \( c' \)

1: \((c'_0, c'_1) \leftarrow c \)
2: \( \hat{d} \leftarrow \text{RNSDecompModUp}(c'_1) \) \( \triangleright \) \( \hat{d} \in R_{PQ_t}^\beta \)
3: \( v \leftarrow [] \) \( \triangleright \) Empty array of \( R_{Q_t} \) of length \(|k|\)
4: \( \tilde{v}_0, \tilde{v}_1 \leftarrow [] \) \( \triangleright \) Empty arrays of \( R_{PQ_t} \) of length \(|k|\)
5: for \( i = 0 \rightarrow |k| - 1 \) do
6: \( v[i] \leftarrow \psi_{k_i}(c'_0) \)
7: \( \tilde{v}_0[i] \leftarrow \psi_{k_i}(\langle \hat{d}, \text{ek}[i][0]\rangle) \)
8: \( \tilde{v}_1[i] \leftarrow \psi_{k_i}(\langle \hat{d}, \text{ek}[i][1]\rangle) \)
9: end for
10: \( d_0 \leftarrow v[0] \)
11: \( \tilde{c}_0 \leftarrow \tilde{v}_0[0] \)
12: \( \tilde{c}_1 \leftarrow \tilde{v}_1[0] \)
13: for \( i = 1 \rightarrow |k| - 1 \) do
14: \( d_0 \leftarrow d_0 + v[i] \)
15: \( \tilde{c}_0 \leftarrow \tilde{c}_0 + \tilde{v}_0[i] \)
16: \( \tilde{c}_1 \leftarrow \tilde{c}_1 + \tilde{v}_1[i] \)
17: end for
18: \((c'_0, c'_1) \leftarrow (\text{RNSModDown}(\tilde{c}_0), \text{RNSModDown}(\tilde{c}_1)) \)
19: \((c'_0, c'_1) \leftarrow (d_0, 0) \oplus (c'_0, c'_1) \) \( \triangleright \) Eval. Rep.
20: return \((c'_0, c'_1)\)

Algorithm 4 shows the procedure for each unrolled iteration.\(^7\) We hide the details of the procedures consisting of (inverse) NTTs, and defer them to Appendix C. Hence, all polynomials are in evaluation representation at the level of Algorithm 4. As an application of Algorithm 4, we also provide the unrolled version of the homomorphic trace-type function evaluation in Algorithm 5.

### B. APPROXIMATE COST ANALYSIS

We derive an approximately optimal \( h \) for a given HE parameter in terms of modular-multiplication complexity as commonly considered in literature (e.g., [26], [30]).

\(^6\)More specifically, \( \text{swk}_e^\prime \) can be generated via \( \text{SwitchKeyGen}(s, s(X^{(\ell_e)}+1)) \).

\(^7\)It should be noted that Algorithm 4 is similar to the linear transformation provided in [30, Algorithm 6]. The difference is that the trace-type function does not contain any multiplicative factor.
Algorithm 5 Optimized Homomorphic Evaluation of Unrolled Trace-Type Function Evaluation

Input:
- \(c\): ciphertext
- \(h\): # iterations after unrolling
- \(K = (k_0, \ldots, k_{h-1})\): array of vectors of automorphism indices per iteration
- \(E = (ek_0', \ldots, ek_{h-1}')\): array of vectors of evaluation keys for each of the automorphisms, where \(ek_i' = \{\text{swk}\}_{x=h(i-1)}^{h(i-1)+1} \in (R_{PQ})_k\) for each \(i \in [h]\)

Output: ciphertext \(c'\)

1: \(c' \leftarrow c\)
2: for \(i = 0 \odot h - 1\) do
3: \(c' \leftarrow c' \oplus \text{HoistKS}(c', k_i, ek_i')\)
4: end for
5: return \(c'\)

More specifically, we consider the cost as the number of MMs over either \(Z_{pq}\) or \(Z_{pq}\). Namely, we have the following cost for each unrolled iteration given in Algorithm 4. Note that some of the procedures are exactly the same as the non-hoisted key-switching process already presented in Subsection II-D.

1) RNS-Decomposition and Modulus-Raise (Line 2): This step is identical to the one in the key-switching in Subsection II-D. The complexity is \((\beta + 1)(\ell + 1)N \log N + \alpha \beta (\ell + 2)N + (\ell + 1)N\) MMs. 

2) Inner-product-and-automorphism (Lines 5–9): This step iterates \(n := |k|\) times. Each iteration consists of automorphisms on three ring elements, one in \(R_{PQ}\), and the other two (i.e., result of inner product) in \(R_{PQ}\). The complexity of the inner product is exactly the same as the key-switching described in Subsection II-D, which is \(2\beta (\ell + 1 + \alpha)N\) MMs. Thus, the total complexity is given by \(2n\cdot \beta (\ell + 1 + \alpha)N\) MMs. Note that the number of per iteration automorphisms are \(2(\ell + k + 1) + (\ell + 1)\), as opposed to \(2(\ell + 1)\) in the rotations-and-sums.

3) Lazy Modulus-down (Lines 10–18):
   a) Addition: Lines 10–17: This step aggregates arrays of ring elements, and does not contain any MMs.
   b) Modulus-down (Line 18): This step performs modulus-down on two ring elements in \(R_{PQ}\), which is exactly the same as in Subsection II-D. The complexity is given by \(2(\alpha + 2(\ell + 1))N \log N + 2(\alpha \ell + 2\alpha + \ell + 1)N\)

4) Addition and output (Line 19): This step is the same as the key-switching described in Subsection II-D, which does not require any MMs.

For convenience, we denote the total cost for all operations before (resp. after) inner-product-and-automorphism per unrolled iteration by \(C_{H1}\) (resp. \(C_{H2}\)). These two cost as

Table 1. Cost in MMs for hoisted operations as well as the cost for a single inner product \(C_{IP}\) are summarized in Table 1 to analyze the cost as a function of \(h\). \(^9\)

We now define our cost function. The loop-unrolling controls the number of the hoisted operations and the number of inner-product-and-automorphism. Specifically, the hoisted operations are performed exactly \(h\) times. For each unrolled iteration, we require either \(2(\lfloor M/h \rfloor - 1) \) or \(2(\lfloor M/h \rfloor + 1 - 1)\) inner-product-and-automorphism, repeating \(h - r \) or \(r \) times over \(h\) iterations respectively. Thus, we have

\[
C(h) = h \cdot (C_{H1} + C_{H2}) + (h - r) \cdot (2(\lfloor M/h \rfloor - 1) \cdot C_{IP} + r \cdot (2(\lfloor M/h \rfloor + 1 - 1) \cdot C_{IP})
\]

(1)

where \(r = M \mod h\). Now, let \(g(h) = 2(\lfloor M/h \rfloor \cdot (h + (M \mod h))\) is monotonic decreasing.

Proof: See Appendix A. \(\square\)

From Lemma 1, \(C(h = M)\) gives minimal complexity if \(C_{H} \leq C_{IP}\). To see whether our loop unrolling works better, we need to know if \(C_{H} > C_{IP}\) holds. Since \(C_{H} > C_{H1}\), it is sufficient to check if \(C_{H1} > C_{IP}\). To simplify the comparison, from Table 1, we compare \(N \log N\) term of \(C_{H1}\) with \(C_{IP}\). That is, we consider

\[
C_{H1} = (\beta + 1)(\ell + 1)N \log N + \text{other terms} = (\beta \ell + \beta + \ell + 1)N \log N + \text{other terms}
\]

\[
C_{IP} = 2\beta (\ell + 1 + \alpha)N = 2(\beta \ell + \beta + \alpha \beta)N \approx 2(\beta \ell + \beta + \ell + 1)N,
\]

where the last approximation is done by \(\alpha \beta \approx \ell + 1\). As \(\log N > 2\) holds in practice, \(C_{H1} > C_{IP}\) and thus

\(^9\)Techniques used in \([30]\) drop the costs for the hoisted operations by \((\ell + 1)N\) and \((\ell + 1)N \log N\) MMs on \(C_{H1}\) and \(C_{H2}\). Although we did not adopt such optimizations, it is expected that our loop-unrolling still works better than the rotations-and-sums. However, speed-up factor will get slightly lower than what we show in the experimental results.
Thus, it is clear that \( \in\{e\} \) where \((\mod\ H\) parameters, such as \(\alpha, \beta\) expected to run better, regardless of the concrete choices of

Below, we show theoretical analysis to understand the effect of \(h\). From Lemma 1, we derive the following corollary.

**Corollary 1.1:** For positive integers \(M(> 1)\) and \(h \in \{1, \ldots, M\}\), \(g(h) = 2^{[M/h]} \cdot (h + (M \mod h)) = 2M\) holds for \(h \in \{M – [M/2], \ldots, M\}\).

**Proof:** We consider the case when \(h = M – e\) with \(e \in \{1, \ldots, [M/2]\}\). When \(M\) is odd, \([M/h] = 1\) and \(M \mod h = e\) holds, and thus \(g(h = M – e) = 2 \cdot (M – e + e) = 2M\). When \(M\) is even, the proof is the same for \(h = M – e\) with \(e \in \{1, \ldots, [M/2]\}\). For \(e = [M/2] = M/2\), we have \(h = M – M/2 = M/2\), \(M \mod h = 0\), and \([M/h] = 2\). Thus, it is clear that \(g(h) = 2^2 \cdot (M/2) = 2M\).

**Lemma 2:** \(C(M – e) < C(M – e + 1)\) holds for some \(e \in \{1, 2, \ldots, M – 1\}\) over all \(M \in \{2, 3, \ldots, \log N\}\) if

\[
\frac{CH}{CIP} > g(M – e) – g(M – e + 1) + 1,
\]

where \(g(M – e) – g(M – e + 1) + 1 \geq 1\).

**Proof:** We have

\[
(M – e + 1) \cdot (CH – CIP) + g(M – e + 1) \cdot CIP > (M – e) \cdot (CH – CIP) + g(M – e) \cdot CIP,
\]

which simplifies as follows

\[
\frac{CH}{CIP} > g(M – e) – g(M – e + 1) + 1 \geq 1,
\]

where the last inequality holds from Corollary 1.1.

Given \(CH\) and \(CIP\), we can find maximal \(e\) satisfying the inequality in Lemma 2. The bound \(g(M – e) – g(M – e + 1) + 1\) becomes higher as \(M\) and \(e\) increase. The \(HE\) parameter only has an effect on the ratio \(CH/CIP\), which tends to set the cost-efficient \(h\) smaller as the ratio goes higher. Nevertheless, we also provide that there exists an upper bound of \(h\) from the following theorem.

**Theorem 3:** If \(CH > CIP\), \(C(M – e) < C(M)\) holds for \(e \in \{1, 2, \ldots, [M/2]\}\).

**Proof:** From Corollary 1.1, we know that \(g(M – e) – g(M – e + 1) = 0\) for \(e \in \{1, \ldots, [M/2]\}\), making the bound in Lemma 2 equal to one. Hence, \(C(M – [M/2]) < C(M – ([M/2] – 1)) < \ldots < C(M)\) holds as long as \(CH > CIP\).

**Theorem 3** implies that the optimal \(h\) providing the minimal cost is upper bounded by \(h \leq M – [M/2]\), as long as \(CH > CIP\). In other words, the cost-efficient \(h\) is located on the half of \(M\) or below the half.

Moreover, the speed-up factor over \(C(M)\) can be given by

\[
\frac{C(M)}{C(h)} = \frac{M \cdot (CH + CIP)}{h \cdot CH + (g(h) – h) \cdot CIP} = \frac{CH + CIP}{h} \cdot \frac{1}{CH + \frac{g(h) – h}{h}},
\]

where \((g(h) – h)/h \geq 1\), whose equality holds when \(h = M\).

Thus, for \(h < M\), the speed-up factor is less than \(M/h\) but approaches it as the ratio grows. As we see above, the ratio impacts both the cost-efficient \(h\) and the speed-up factor. We describe how the ratio changes along with the effect of \(k\). (\(= \alpha\)).

**Effect of \(k\) (\(= \alpha\)) and \(\ell\):** The choice of \(k\) affects the complexity and the effect of the loop-unrollings. The effect of \(k\) depends on \(\ell\) as they determine \(\beta\). Figure 1 shows the change of the ratio by varying \(\ell\) and \(k\). For each \(k\), the graph generally shows the decrease/increase of the ratio according to the growth of \(\ell\). The decrease of the ratio is caused by the increase of \(\beta\), whereas the increase of the ratio is caused when \(\beta\) does not change with the increase of \(\ell\). For example, these extreme cases are observable when \(k = 1\) and \(k = 10\), respectively. These behaviors are explained by considering \(CH\) and \(CIP\) as functions of \(\ell\), and by comparing \(CH(\ell)/CIP(\ell)\) and \(CH(\ell + 1)/CIP(\ell + 1)\), where we defer the details to Appendix B. The loop-unrolling is expected to work more effectively in terms of performance when \(\ell\) and \(k\) are large.

![Figure 1. CH to CIP ratio with different k and \ell for log N = 15 and L = 9.](image)

![Figure 2. Cost comparison for log N = 15 and M = 2, 7, 15.](image)

We show the shape of the cost function in Figure 2 according to \(h\) for \(\log N = 15, \ell = 9\) and \(M \in \{2, 7, 15\}\). The choice of \(M\) is to capture the wide range of the trace-type function evaluation. For each \(M\), the right-most point \((h = M)\) corresponds to the rotations-and-sums in Algorithm 2. We can observe from the figure that the
minimum cost is provided when choosing around the center of possible $h$’s for each $M$, following the theorem above.

Breakdown Analysis: We show how each of the terms, $(C_{H1}, C_{IP}$, and $C_{H2})$, changes as $h$ and $(\ell, k)$ change for $M = 15$ in Figure 3 with stacked-bars, where $h = 1, 2$ are omitted for visualizing purpose. We can see that dominant terms depend on $\beta$ for relatively large $h$, whereas the $C_{IP}$ term becomes dominant for relatively small $h$. For larger (resp. smaller) $\beta$, $C_{H1}$ (resp. $C_{H2}$) tends to become dominant. Assuming all parameters except $h$ are fixed, the dominant cost gets shifted from the hoisted part to the inner-product part as $h$ decreases while reducing overall complexities. The cost for $h = M$ is clearly bounded by the hoisted operations. Hence, with loop-unrolling, a linear decrease of the hoisted costs and non-linear increase of the inner-product cost in $h$ enables the overall speedup of the homomorphic trace-type function evaluation.

Storage Cost: Although the loop-unrolling decreases the cost compared with the case $h = M$, it also increases the number of evaluation keys required, thereby increasing storage cost. The number of evaluation keys is exactly the same as the number of inner-products (i.e., $g(h) - h$). We show how the number differs depending on $M$ and $h$ for a fixed log $N$ in Figure 4. The figure precisely shows the shape of $g(h) - h$, which is monotonically decreasing in $h$. Thus, the rotations-and-sums approach [9], [15] $(h = M)$ can be considered as a storage-efficient solution. Compared with the Figure 2, where the increase of $h$ from some particular point typically increases the cost, we know that there exists a trade-off between storage and time.

Note for Parallelism: We note that the proposed method has more opportunities to adopt parallelization than Algorithm 2 does per iteration. Specifically, we can parallelize the line 5 of Algorithm 4, where such parallelization
never appears in Algorithm 2. Other available parallelism, such as modulus raise (line 2 in Algorithm 4) and modulus down (line 18 in Algorithm 4), are equally available in each of the methods with the difference of the total number of invocations.

As the analysis is approximate in the sense that costs for automorphism, addition, and implementation factors such as memory access and cache-miss are ignored, the optimal h can differ for a given HE parameter and an actual implementation. In the next section, we provide experimental results.

IV. PERFORMANCE EVALUATION
A. EXPERIMENTAL SETUP
We compared the performance of the proposed method with that of two existing methods, namely, the recursive rotations-and-sums procedure [9], [15] and the baby-step giant-step (BS/GS) algorithm to evaluate all the necessary automorphism indices [22]. These methods can be respectively captured in our parameter setting, namely h = M and h = 2. Since the loop-unrolling in the proposed method enables parallelization of the entire procedure at the level of inner-product-and-automorphism (i.e., line 5 in Algorithm 4), we also compare all methods in both single- and multi-threaded modes.

We conducted all experiments with PALISADE library (v1.9.2) [28], as it is the only library that provides both parallelization and the generic key-switching with the special modulus [26]. All experiments were run on a machine running ubuntu 16.04 equipped with Intel Xeon Gold 5122 CPU 3.60 GHz and 32-GB RAM. All codes were written in C++, compiled with g++ 7.2.0 using ‘-O3’ option. We used the CKKS scheme with a scaling factor of 2^{40} in APPROX-RESCALE mode by varying the multiplicative depth to see the scalability in the number of RNS moduli ℓ + 1. Note that the choice of the scaling parameter depends on applications. We aim to show the trend for variable choices of ℓ. All parameter sets satisfy 128-bit security level according to the standardization white paper [29]. We modified the implementation of the automorphism on polynomials in evaluation representation such that it is realized via a simple permutation. This is to reduce the automorphism costs. Instead of performing automorphisms on three (resp. two) ring elements in our method (resp. in the rotations-and-sums), we pre-computed a permutation vector once on the fly, and then applied a simple permutation on each of RNS components.

We assume that multi-threading is used differently depending on the choice of h. For h = M, we use it at the level of RNS moduli except for the fast base conversion used in modulus-raise and modulus-down which are parallelized at the level of ring-dimension, as provided in PALISADE. We used four threads for all experiments with multi-threading. For h < M, we used the multi-threading in nearly the same way as h = M, except that we used it at the level of the inner-product-and-automorphism in an unrolled iteration.

Next, we confirm the actual speed-up in an actual computing environment with both single- and multi-threaded modes. All results are averaged over 10 trials. We show the runtime comparison for log N = 15 with M ∈ {2, ⌊log N/2⌋, log N}, L = 9, ℓ ∈ {1, 4, 9}, and k ∈ {1, 5, 10} to see the scalability of our method in ring dimension, multiplicative depth, and number of RNS moduli for the special modulus.

B. RESULTS
Tables 2a–2d show the runtime comparison for M ∈ {2, 15}, where we omitted M = 7 (shown in a subsequent plot) for the ease of readability. All tables show that our method works faster than h = M for all parameter settings. Tables 2a and 2b (resp. 2c and 2d) show the result for M = 15 (resp. M = 2) with single- and multi-threaded modes, respectively. They show that loop-unrolling runs faster than the rotations-and-sums. More specifically, it gives speed-ups by a factor of 1.32–1.48 and 1.47–2.12 with single- and multi-threaded modes, respectively. Below, we describe the result with single-threading followed by that with multi-threading in detail.

With single-threading for M = 15, it can be seen that the most runtime-efficient h is less than 15 − ⌊15/2⌋ = 8, following the theoretical analysis. The difference of the concrete value of the runtime-efficient h depends on (k, ℓ) pair. More specifically, when (k, ℓ) changes such that the ratio goes up, h tends to become smaller. For instance, we can see that, for ℓ = 4, 9, the runtime-efficient h decreases as k grows. For M = 2, the runtime-efficient h is simply equal to one as the loop-unrolling effectively works.

We can also observe that the magnitude of the runtime is affected by k, regardless of the concrete choice of h. For a fixed ℓ, increasing k generally decreases the overall runtime as long as β decreases. This can be observed from the tables when ℓ = 9 with all k, and ℓ = 4 with k ranging from 1 to 5.
TABLE 2. Runtime comparison (in millisecond) for log $N = 15$, $M \in (2, 15)$ with different $(k, \ell)$ pairs in both single- and multi-threaded modes. “N/A” represents the case where a process was killed by the operating system due to large memory consumption. For each table, $h = 2$ and $h = M$ respectively correspond to [22] and rotations-and-sums [9, 15].

(a) $M = 15$ (single-threading)

| $k$ | 1 | 5 | 10 |
|-----|---|---|----|
| $h \vee(\beta)$ | (2) | (5) | (10) |
| 1 | N/A | N/A | N/A |
| 2 | N/A | N/A | N/A |
| 3 | 491.7 | 1933.7 | 6484.8 |
| 4 | 325.4 | 1241.1 | 4113.4 |
| 5 | 270.1 | 995.4 | 3257.6 |
| 6 | 270.3 | 975.6 | 3159.8 |
| 7 | 270.0 | 945.3 | 3068.9 |
| 8 | 278.8 | 957.5 | 3123.3 |
| 9 | 295.8 | 1009.3 | 3285.2 |
| 10 | 313.7 | 1059.5 | 3446.3 |
| 11 | 332.0 | 1108.2 | 3626.6 |
| 12 | 349.5 | 1159.7 | 3742.2 |
| 13 | 367.6 | 1209.0 | 3902.5 |
| 14 | 384.8 | 1255.4 | 4024.5 |
| 15 | 387.4 | 1266.3 | 4062.5 |

(b) $M = 15$ (multi-threading)

| $k$ | 1 | 5 | 10 |
|-----|---|---|----|
| $h \vee(\beta)$ | (2) | (5) | (10) |
| 1 | N/A | N/A | N/A |
| 2 | N/A | N/A | N/A |
| 3 | 181.1 | 666.4 | 2069.9 |
| 4 | 140.1 | 458.4 | 1430.3 |
| 5 | 130.3 | 410.7 | 1253.8 |
| 6 | 140.5 | 430.2 | 1306.0 |
| 7 | 149.2 | 446.8 | 1365.6 |
| 8 | 162.1 | 484.9 | 1481.9 |
| 9 | 181.6 | 546.8 | 1652.4 |
| 10 | 200.0 | 603.0 | 1836.4 |
| 11 | 220.7 | 661.6 | 2008.0 |
| 12 | 239.2 | 716.6 | 2192.8 |
| 13 | 255.5 | 781.5 | 2380.0 |
| 14 | 270.0 | 840.2 | 2542.6 |
| 15 | 277.0 | 825.5 | 2573.9 |

(c) $M = 2$ (single-threading)

| $k$ | 1 | 5 | 10 |
|-----|---|---|----|
| $h \vee(\beta)$ | (2) | (5) | (10) |
| 1 | 35.8 | 124.4 | 400.8 |
| 2 | 52.5 | 167.2 | 536.8 |

(d) $M = 2$ (multi-threading)

| $k$ | 1 | 5 | 10 |
|-----|---|---|----|
| $h \vee(\beta)$ | (2) | (5) | (10) |
| 1 | 20.1 | 61.3 | 186.1 |
| 2 | 37.3 | 109.7 | 343.7 |

When the increase of $(\ell, k)$ does not change $\beta$, the runtime simply increases. This is shown for $\ell = 4$ from $k = 5$ to 10. An exception happens when $\ell = 1$ where $k = 1$ incurring $\beta = 2$ provides faster runtime than larger $k$ and thus reaches the minimum.

With the multi-threading, loop-unrolling has more opportunities to parallelize the entire procedure. Regardless of the concrete values of $M$, the multi-threading gives speed-ups over single-threaded mode. For $M = 15$, as opposed to the single-threaded mode, the optimal $h$ becomes lower, which can be observed at $h = 5$. This simply means that the parallelization on the inner-product-and-automorphism can take advantage of multi-threading for smaller $h$. For $M = 2$, the runtime-efficient $h$ is equal to one, which is the same as the one in the single-threaded mode. Since the trend of the other performance numbers is the same as the single-threaded mode, we do not go into further details.

As opposed to the theoretical analysis, we see that $h = M$ can give lower runtime than $h = M - 1$ in the experiment. This comes from the difference between our algorithm...
(Algorithm 5) and the rotations-and-sums (Algorithm 2), which cannot be captured by the approximate cost analysis. For instance, the rotations-and-sums requires automorphisms on two elements in $R_{Q_0}$, resulting in $2(\ell + 1)$ automorphisms on RNS components. On the other hand, our algorithm needs to perform automorphism on two elements in $R_{Q_0}$ and one element in $R_{Q_1}$. Hence, it requires $2(\ell + 1 + k) + (\ell + 1) = (3\ell + 2k + 3)$ automorphisms on RNS components. As a result, our algorithm has $(\ell + 2k + 1)$ extra automorphisms on RNS components per each inner-product. Such extra overhead on our method accumulates as $M$ becomes larger. In our experiments, we see this when $M = 15$.

![Figure 5. Runtime comparison for $\log N = 15$ and $\ell = 9$ with $M \in \{2, 7, 15\}$ in the single-threaded mode.](image)

Figure 5 shows plots to see how actual implementations look close to the theoretical analysis. For $M = 2$, the shapes are the same as in the theoretical estimate.

For $M = 7, 15$, we see that each of the optimal $h$’s tends to become higher than the theoretical estimate. One reason would be that the cost for hoisted operations is a bit underestimated. In the implementation, the approximate base conversion indeed leverages lazy mod reduce (e.g., see [31]), which is a technique of bypassing the modular reduction until the summation is done during the computation of inner-product. This contributes to lower runtime on the hoisted operations than expected. Hence, the ratio becomes lower, yielding higher runtime-efficient $h$.

**Speed-up Factor vs. Memory Usage:** We show the trade-off between runtime and storage in Figure 6 by changing $k$ for fixed $\ell$ and $M$ in the single-threaded mode, where we plot the speedup factor and the memory blowup over $h = 15$ with solid and dotted lines, respectively. Moreover, we add a horizontal blue line on the y-axis being one to visualize valid choice of $h$’s outperforming $h = M$ in terms of runtime. In Figure 6a, such $h$ can be chosen from $\{5, 6, \ldots, 12\}$, where the maximal speedup is given at $h = 8$. Since the storage cost increases as $h$ decreases, the valid $h$ should be chosen from $\{8, 9, \ldots, 12\}$ in practice. Note that the linear decrease with a slope being one from $h = 8$ to 14 for the storage cost is because of the property of $g(h) = h$ and Corollary 1.1. We can also observe that all storage blowups are higher than the speed-up factors. In Figures 6b and 6c, we see some choices of $h$ provide speed-up factors that outperform the storage blowup, showing a better trade-off than the above. This is because of choosing higher $(k, \ell)$ pairs and that the storage blowup factor for a fixed $h$ is independent of $(k, \ell)$, which enables the loop-unrolling to work more effectively. Therefore, we can conclude that our loop-unrolling can give a time-space trade-off, and can even provide a speed-up factor larger than the storage blowup factor for some choice of HE parameter in practice.

Overall, apart from the existence of implementation factors, our experimental results follow the trend of the approximate cost function. Thus, it is worth applying the loop-unrolling to the trace-type function evaluation. Although we did not observe a parameter set when our method completely failed over $h = M$, there can be a case where the loop-unrolling cannot run due to large memory usage, which may not happen for the rotations-and-sums. This situation can happen with larger HE parameter settings having higher ring dimension (e.g., $\log N \geq 16$) and large multiplicative depth (large $L$), depending on storage availability.

**C. DISCUSSION**

As we have shown in the tables, the best $h$ providing the minimum runtime depends on application-dependent parameters, such as $M$, $\log N$, $k$, and $\ell$. Based on the fact that a parameter of homomorphic encryption is pre-determined depending on applications, and the search space of $h$ is bounded by $M \leq \log N$ at around 15, it is easy to determine the best $h$ (e.g., by examining all possible $h \leq M$) depending on objectives. For instance, since the choice of $h$ incurs a trade-off between the time and storage, it is recommended to find $h$ satisfying a given storage requirement.

Moreover, it is possible to consider $k$ as another parameter for each $\ell$, rather than considering it as a static parameter over all $\ell$’s. We can set the number of special moduli $k_\ell$ for each level. As can be seen from the tables, $k = 1$ provides the minimum runtime for $\ell = 1$, over all $h$. However, for $\ell = 9$, $k = 10$ provides the minimum. Hence, it is reasonable to create evaluation keys for each level. However, such optimization requires additional memory costs, and we leave such adoption as a part of our future work.

**V. RELATED WORK**

The trace-type function evaluation is studied differently for total-sums and trace, and it can be seen as a special type of linear transformation. The total-sums algorithm was first presented in [9]. It works via a “repeated-doubling” approach, which recursively applies rotations-and-sums. Its design is generic for a vector of any length (including non-power-of-two lengths) to be processed. In contrast, we focus on the vectors of power-of-two length as they are naturally derived from the power-of-two cyclotomic ring. Summing up all the slots is not always required. Instead, partial summation is used (e.g., [11]) by performing rotations-and-sums fewer than $\log N$ times.

While the total-sums operation works over the plaintext slots, the trace function works over polynomial coefficients. The main effect of the trace function is isolation of the
coefficient of a polynomial. When only a specific coefficient should be extracted, the problem is same as LWE-to-RLWE conversion [15], [32]. The initial solution [32] applies as many key-switching processes as the ring dimension on an LWE ciphertext. Although this approach provides high parallelism, it suffers from computational inefficiency and requires gigabyte or terabyte of storage due to the required $O(N)$ evaluation keys. Recently, these limitations have been removed in a LWE-to-RLWE conversions by recursive automorphism-and-sums [15], resembling total-sums but using different indices for the automorphisms. As a result, both storage and computational complexities have been reduced by an exponential factor. Our development further improves the efficiency of the homomorphic function evaluation expressed by rotations-and-sums (or equivalently automorphisms-and-sums) with a trade-off between storage and computational costs.

The trace-type function can be considered as a linear transformation. The functionality of the linear transformation is achieved by homomorphically summing the rotated ciphertexts multiplied by some constants. In trace-type function evaluation, the multiplicative factor is always one, and there can be other potential solutions from the literature of the linear transformation. Halevi and Shoup [22] presented a set of optimization techniques to reduce the cost of homomorphic linear transformations. We leverage techniques, such as the hoisting and baby-step giant-step (BS/GS).\textsuperscript{11} Note that the naïve adaption of the BS/GS algorithm suffers from inefficiency, specifically for large $M$. Our algorithm represents a recursive application of BS/GS, with improved efficiency over previous work. In [22], the authors introduced three variants of key-switching strategies: minimal key-switching, BS/GS, and a full strategy. These strategies differ in the number of key-switchings needed to perform a single automorphism on a ciphertext. Our implementation follows the full-strategy in the sense that a single homomorphic automorphism is realized by exactly one automorphism and key-switching, whereas the other strategies require at most two or more key-switchings to realize a single homomorphic automorphism. As we are interested in the improvement of the computational efficiency, adapting the other key-switching strategies is beyond the scope of this paper.

Recently, Bossuat et al. [30] proposed a set of optimization techniques for hoisted automorphisms. Their techniques are generic and were adopted in our method. However, their problem setting for the generic matrix–vector multiplication is slightly different from ours. Specifically, their cost model regarding the key-switching for a generic linear transformation is asymmetric in BS and GS. The asymmetry of the cost model is caused by the existence of multiplication after applying an automorphism on a ciphertext, which requires the lazy modulus-down for a baby-step in each iteration of a giant-step. In our case, no multiplication is require. Hence, the lazy modulus-down inside the baby-step can be immediately performed independently from the giant-step. Moreover, they do not provide the timing results in the multi-threading mode, which is important in our loop-unrolling technique.

VI. CONCLUSION
In this paper, we developed a faster method to evaluate homomorphic trace-type function by using loop-unrolling along with a set of optimizations to amortize costly key-switchings. We approximately demonstrated that the loop-unrolling provides efficiency, and empirically showed that our method achieves 1.32–2.12x faster runtime than the previous rotations-and-sums method. Our method helps construct homomorphic encryption-based privacy-preserving applications to which we adopt our method in our future work. It would also be interesting to develop a way to automatically determine the best unrolling parameter $h$ for a given computing environment.

APPENDIX A
PROOF OF LEMMA
Lemma 1 Proof: We aim to show $g(n) \geq g(n + 1)$ for any $n \in \{1, \ldots, M - 1\}$. For positive integers $x, y$, we denote
Since \( [M/n] \geq [M/(n+1)] \), we consider the following two cases:

Case 1) \( [M/n] = [M/(n+1)] \):

Let \( k = [M/n] = [M/(n+1)] \). Then, we have

\[
\begin{aligned}
r(M, n) &= M - nk, \\
r(M, n+1) &= M - (n+1)k = r(M, n) - k.
\end{aligned}
\]

So,

\[
g(n) - g(n+1) = 2^k(n+r(M, n)) - 2^k(n+1+r(M, n+1)) = 2^k(n+r(M, n)) - 2^k(n+1+r(M, n) - k) = 2^k(k - 1).
\]

As \( n \in \{1, \ldots, M-1\}, k \geq 1 \). Hence, \( g(n) - g(n+1) \geq 0 \), as needed.

Case 2) \( [M/n] > [M/(n+1)] \): We let \( k = [M/n] \geq 2 \). Then, we have \( [M/(n+1)] = k - s \) for some \( s \in \{1, \ldots, k-1\} \). Using \( r(M, n) = M - nk \), we have

\[
r(M, n+1) = M - (n+1)(k-s) = M - nk - k + (h+1) = r(M, n) - k + s(n+1).
\]

Then,

\[
g(n) - g(n+1) = 2^k(n+r(M, n)) - 2^{k-s}((n+1) + r(M, n+1)) = 2^k(n+r(M, n)) - 2^{k-s}((n+1) + r(M, n) - k + s(n+1)) = 2^{k-s}(2^{s-1}(n+r(M, n)) + (k-1) - s(n+1)).
\]

As \( k - s \geq 1 \), it suffices to check if \( f(s) := (2^s-1)(n+r(M, n)) + k-1 - s(n+1) \geq 0 \). We prove this by induction on \( s \).

For \( s = 1 \) (base case), we can clearly see \( f(1) = r(M, n) + k - 2 \geq 0 \), since \( k \geq 2 \) and \( r(M, n) \geq 0 \). For the inductive step, suppose that for all \( j \) with \( 1 \leq j < k \), \( f(j) = (2^j - 1)(n+r(M, n)) + k - 1 - j(n+1) \geq 0 \). Then, we need to prove \( f(j+1) = (2^{j+1}-1)(n+r(M, n)) + k - 1 - (j+1)(n+1) \geq 0 \). Using

\[
\begin{aligned}
(2^{j+1}-1)(n+r(M, n)) &= 2^j + 2^j - 1)(n+r(M, n)) \\
k - 1 - (j+1)(n+1) &= k - 1 - j(n+1) - (n+1),
\end{aligned}
\]

\( r(M, n) \geq 0 \), and \( j \geq 1 \), it is easy to show that \( f(j+1) = f(j) + 2(n+r(M, n)) - (n+1) \geq 0 \) holds by the induction hypothesis.

\[\square\]

### APPENDIX B

**Details of the Behavior of the \( C_H \) to \( C_{IP} \) Ratio**

**A. The Ratio According to the Increase of \( \ell \)**

For a fixed \( \alpha \geq 1 \), we denote by \( \beta_\ell = [(\ell +1)/\alpha] \). We treat \( C_H = C_{H1} + C_{H2} \) and \( C_{IP} \) as functions of \( \ell \). Namely, let

\[
C_H(\ell) = (\beta_\ell \ell + \ell \ell + 5\ell + 2\alpha + 3)N \log N + (2\alpha \beta_\ell + 2\alpha \beta_\ell + 2\alpha \ell + 4\alpha + 3\ell + 3)N
\]

be a function representing the cost for all hoisted operations per unrolled-iteration, and

\[
C_{IP}(\ell) = 2\beta_\ell (\ell + 1 + \alpha)N
\]

be a function representing the cost for a single inner-product. We aim to show

\[
\frac{C_H(\ell + 1)}{C_{IP}(\ell + 1)} > \frac{C_H(\ell)}{C_{IP}(\ell)} \quad \text{when } \beta_{\ell+1} = \beta_\ell
\]

and

\[
\frac{C_H(\ell)}{C_{IP}(\ell)} > \frac{C_H(\ell + 1)}{C_{IP}(\ell + 1)} \quad \text{when } \beta_{\ell+1} > \beta_\ell.
\]

For \( \ell \) and \( \ell + 1 \), we have either 1) \( \beta_{\ell+1} = \beta_\ell \) or 2) \( \beta_{\ell+1} = \beta_\ell + 1 \). Moreover, \( C_H(\ell + 1) \) and \( C_{IP}(\ell + 1) \) can be expressed as \( C_H(\ell + 1) = C_H(\ell) + a \) and \( C_{IP}(\ell + 1) = C_{IP}(\ell) + b \) for some \( a, b \), respectively. Thus, we need to show inequalities by specifying \( a \) and \( b \) depending on the conditions on \( \beta_\ell \) and \( \beta_{\ell+1} \). Below, we scale everything by \( 1/N \) since \( N \) appears as a common multiplicative factor.

**Case 1) \( \beta_\ell = \beta_{\ell+1} \):** Since \( C_{IP}(\ell), C_H(\ell) > 0 \), we need to show

\[
C_{IP}(\ell) \cdot a - C_H(\ell) \cdot b > 0,
\]

where

- \( a = (\beta_\ell + 5) \log N + (2\alpha + \alpha \beta_\ell + 3) \)
- \( b = 2\beta_\ell \).

We show that the inequality holds for the \( \log N \) terms and the other terms separately.

**log \( N \) terms:**

\[
The \log N \text{ terms of } C_{IP}(\ell) \cdot a - C_H(\ell) \cdot b \text{ scaled by } 1/\log N \text{ is given by}
\]

\[
2\beta_\ell (\ell + 1 + \alpha) \cdot (\beta_\ell + 5) - (\beta_\ell + \beta_\ell + 5\ell + 2\alpha + 3) \cdot 2\beta_\ell = 2\alpha \beta_\ell (\beta_\ell + 3) > 0.
\]

**Other non-\( \log \) \( N \) terms:**

The other non-\( \log N \) terms of \( C_{IP}(\ell) \cdot a - C_H(\ell) \cdot b \) is given by

\[
2\beta_\ell (\ell + 1 + \alpha) \cdot (2\alpha + \alpha \beta_\ell + 3) - (2\alpha \beta_\ell + 2\alpha \ell + 4\alpha + 3\ell + 3) \cdot 2\beta_\ell = 2\alpha \beta_\ell (\alpha - 1) + 2\beta_\ell (2\alpha + 1) > 0.
\]

Thus, \( C_{IP}(\ell) \cdot a - C_H(\ell) \cdot b > 0 \) holds.

**Case 2) \( \beta_{\ell+1} = \beta_\ell + 1 \):**

Similarly to case 1, we need to show

\[
C_H(\ell) \cdot b - C_{IP}(\ell) \cdot a > 0,
\]

\[\square\]
where

- \( a = (\ell + \beta \ell + 7) \log N + (5\alpha + \alpha \beta + \alpha \ell + 3) \)
- \( b = 2(\beta \ell + \ell + 2 + \alpha) \).

As before, we show the inequality holds for the log \( N \) terms and the other non-log \( N \) terms separately.

\( \log N \) terms:

The \( \log N \) terms of \( C_H(\ell) \cdot b - C_{IP}(\ell) \cdot a \) scaled by \( 1/\log N \) is given by

\[
(\beta \ell \ell + \beta \ell + 5\ell + 2\alpha + 5) \cdot 2(\beta \ell + \ell + 2 + \alpha) \\
- 2\beta \ell (\ell + 1 + \alpha) \cdot (\ell + \beta \ell + 7) \\
= 4\alpha^2 + 14\alpha \ell + 18\alpha + 10\ell^2 + 30\ell + 20 - 2\alpha \beta^2 - 8\alpha \beta \ell \\
= 2(2\alpha^2 + 7\alpha \ell + 9\alpha + 5\ell^2 + 15\ell + 10 - \alpha \beta \ell (\ell + 4)).
\]

By letting \( \alpha \beta \ell = \ell + 1 \), the negative terms become

\[-\alpha \beta \ell (\ell + 4) = -(\ell + 1)(\beta \ell + 4) = -(\beta \ell + \beta \ell + 4 + 4) \].

When \( \beta \ell \geq \ell \), we only have two cases: \( \alpha = 1 \) with any \( \ell \), or \( \ell = 0 \) with any \( \alpha \). In either case, we have \( \beta \ell = \ell + 1 \). Replacing \( \beta \ell \) with \( \ell + 1 \) yields the negative terms as

\[-(\ell^2 + 2\ell + 1 + \ell + 4) = -(\ell^2 + 6\ell + 5) \]

which are smaller than that of the positive terms. When \( \beta \ell < \ell \), the magnitude of the negative terms are clearly smaller than the that of the positive terms.

Other Non-log \( N \) Terms:

\( \log N \) terms of \( C_H(\ell) \cdot b - C_{IP}(\ell) \cdot a \) is given by

\[
(\alpha \beta \ell + 2\alpha \beta \ell + 2\alpha \ell + 4\alpha + 3\ell + 3) \cdot 2(\beta \ell + \ell + 2 + \alpha) \\
- 2\beta \ell (\ell + 1 + \alpha) \cdot (5\alpha + \alpha \beta + \alpha \ell + 3) \\
= 4\alpha^2 \ell + 2\alpha \beta^2 + 4\alpha \ell^2 + 8\alpha^2 + 22\alpha \ell^2 + 22\alpha \ell + 18\ell + 12 \\
- 2\alpha^2 \beta^2 - 6\alpha \beta \ell \\
= 2(2\alpha^2 \ell + \alpha \beta^2 + 2\alpha \ell^2 + 4\alpha^2 + 11\alpha + 3\ell^2 + 11\alpha \ell + 9\ell + 6 \\
- \alpha \beta \ell (\alpha \beta + 3\alpha)).
\]

Having \( \alpha \beta \ell = \ell + 1 \), the negative terms become

\[-\alpha \beta \ell (\alpha \beta + 3\alpha) = -(\ell + 1)(\ell + 1 + 3\alpha) \\
= -(\ell^2 + 2\ell + 1 + 3\alpha \ell + 3\alpha),
\]

whose magnitude is smaller than that of the positive terms. Thus, \( C_H(\ell) \cdot b - C_{IP}(\ell) \cdot a \geq 0 \) holds.

### B. THE RATIO WHEN \( \ell = 0 \)

When \( \ell = 0 \), we have \( \beta \ell = 1 \) for any \( \alpha \), and the ratio becomes

\[
\frac{C_H(0)}{C_{IP}(0)} = \frac{(2\alpha + 6) \log N + (6\alpha + 3)}{2(\alpha + 1)} \\
= \frac{2(\alpha + 3) \log N + 3(2\alpha + 1)}{2(\alpha + 1)} \\
= \frac{(\alpha + 3) \log N + (3\alpha + 3/2)}{(\alpha + 1)} \\
= \frac{(1 + \frac{3}{\alpha}) \log N + (3 + \frac{3}{\alpha \ell})}{(1 + \frac{1}{\alpha})},
\]

which converges to \( \log N + 3 \) as \( \alpha \) approaches \( \infty \). When \( \alpha \) is minimum (i.e., \( \alpha = 1 \)), the ratio becomes \( 2 \log N + 9/4 \), which is larger than \( \log N + 3 \) for \( \log N > 3/4 \). Hence, the ratio decreases as \( \alpha \) increases, and it reaches maximum when \( \alpha \) is minimum.

### APPENDIX C

#### DETAILS OF SUBROUTINES

##### A. APPROXIMATE BASE CONVERSION

We denote \( C' = \{ q_0', \ldots, q_{\ell-1}' \} \) and \( B' = \{ p'_0, \ldots, p'_{\ell-1} \} \) by bases, such that \( Q' = \prod_{i=0}^{\ell-1} q'_i \) and \( P' = \prod_{j=0}^{\ell-1} p'_j \). For \( x \in \mathbb{Z} Q' \), we denote its RNS representation by \( x_j = \lfloor x/Q_i \rfloor \) for \( i \in [\ell] \). Approximate base conversion \( \text{Conv}_{C' \rightarrow B'} \) produces a set of RNS components w.r.t. \( B' \), satisfying \( x_j = \lfloor x/Q_i \rfloor \) for some integer \( e \) with \( |e| \leq \ell/2 \) via

\[
\text{Conv}_{C' \rightarrow B'}(x) = \left( \sum_{i=0}^{\ell-1} \lfloor x/Q_i \rfloor p'_i \mod \left( Q'/q_i \right) \right)_{p \in B'}.
\]
We now describe the complexity of the base conversion in terms of the number of MM multiplications. For each $i \in [\ell_1]$, the $y_i := \left[ x_i \cdot q_i / Q \right]_q \text{ term is computed once and is then reused to compute } \left( \sum_{j=1}^{\ell_1} (Q/q_i) \mod p_j \right) \text{ for each } j \in [\ell_2].$ Thus, the total complexity is $\ell_1 + \ell_2 \ell_2 = \ell_1 (1 + \ell_2)$ MM multiplications. The base conversion on an element in $R_{Q^r}$ can be done by applying it coefficient-wise. Thus, the complexity is given by $\mathcal{N}_\ell (1 + \ell_2)$ MM multiplications. Note that a polynomial must be in coefficient representation.

**REFERENCES**

[1] C. Gentry, “Fully homomorphic encryption using ideal lattices,” in *Proc. 41st Annu. ACM Symp. Symp. Theory Comput. (STOC)*, 2009, pp. 169–178.

[2] Z. Brakerski and V. Vaikuntanathan, “Lattice-based FHE as secure as PKE,” in *Proc. 5th Conf. Innov. Comput. Sci.*, Jan. 2014, pp. 1–12.

[3] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, “(Leveled) fully homomorphic encryption without bootstrapping.” *ACM Trans. Comput. Theory*, vol. 6, no. 3, pp. 1–36, Jul. 2014.

[4] J. H. Cheon, A. Kim, M. Kim, and Y. Song, “Homomorphic encryption for arithmetic of approximate numbers,” in *Advances in Cryptology—ASIACRYPT* (Lecture Notes in Computer Science), vol. 10624. Cham, Switzerland: Springer, 2017, pp. 409–437.

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

[6] O. Regev, “On lattices, learning with errors, random linear codes, and cryptography,” in *Proc. 35th Annu. ACM Symp. Theory Comput. (STOC)*, 2005, pp. 84–93.

[7] V. Lyubashevsky, C. Peikert, and O. Regev, “On ideal lattices and learning with errors over rings,” in *Advances in Cryptology* (Lecture Notes in Computer Science), vol. 6110. H. Gilbert, Ed. Berlin, Germany: Springer, 2010, pp. 1–23.

[8] N. P. Smart and F. Vercauteren, “Fully homomorphic SIMD operations,” *Des., Codes Cryptogr.*, vol. 71, no. 1, pp. 57–81, Apr. 2014.

[9] S. Halevi and V. Shoup, “Algorithms in HElib,” in *Advances in Cryptology—CRYPTO* (Lecture Notes in Computer Science), vol. 8616. Berlin, Germany: Springer, 2014, pp. 554–571.

[10] C. Gentry, S. Halevi, C. Jutla, and M. Raykova, “Private database access with HE-over-ORAM architecture,” in *Applied Cryptography and Network Security* (Lecture Notes in Computer Science), vol. 9092. Cham, Switzerland: Springer, 2015, pp. 172–191.

[11] M. Kim, Y. Song, S. Wang, Y. Xia, and X. Jiang, “Secure logistic regression based on homomorphic encryption: Design and evaluation,” *JMIR Med. Informat.*, vol. 6, no. 2, p. e19, Apr. 2018.

[12] A. A. Badawi, L. Hoang, C. F. Man, K. Laine, and K. M. Aung, “PriviFT: Private and fast text classification with homomorphic encryption,” *IEEE Access*, vol. 8, pp. 226544–226556, 2020.

[13] J. Alperin-Sheriff and C. Peikert, “Practical bootstrapping in quasilinear time,” in *Advances in Cryptology—CRYPTO* (Lecture Notes in Computer Science), vol. 8042. Berlin, Germany: Springer, 2013, pp. 1–20.

[14] H. Chen and K. Han, “Homomorphic lower digits removal and improved FHE bootstrapping,” in *Advances in Cryptology—EUROCRYPT* (Lecture Notes in Computer Science), vol. 10820. Cham, Switzerland: Springer, 2018, pp. 315–337.

[15] H. Chen, W. Dai, M. Kim, and Y. Song, “Efficient homomorphic encryption based on ring LWE ciphertexts,” Cryptol. ePrint Arch., Tech. Rep. 2020/015, 2020.

[16] Z. Brakerski, “Fully homomorphic encryption without modulus switching from classical GapSVP,” in *Advances in Cryptology-CRYPTO* (Lecture Notes in Computer Science), vol. 7417, R. Safavi-Naini and R. Canetti, Eds. Berlin, Germany: Springer, 2012, pp. 868–886.

[17] J. Fan and F. Vercauteren, “Somewhat practical fully homomorphic encryption,” Cryptol. ePrint Arch., Tech. Rep. 2012/144, 2012.

[18] F. Boemer, R. Cammarota, D. Demmler, T. Schneider, and H. Yalame, “MP2ML: A mixed-protocol machine learning framework for private inference,” Cryptol. ePrint Arch., Tech. Rep. 2020/721, 2020.

[19] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, “A full RNS variant of FV like somewhat homomorphic encryption schemes,” in *Selected Areas in Cryptography—SAC* (Lecture Notes in Computer Science), vol. 10532. Cham, Switzerland: Springer, 2017, pp. 423–442.

[20] S. Halevi, Y. Polyakov, and V. Shoup, “An improved RNS variant of the BGV homomorphic encryption scheme,” in *Topics in Cryptology—CT-RSA* (Lecture Notes in Computer Science), vol. 11405. Cham, Switzerland: Springer, 2019, pp. 83–105.

[21] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, “Bootstrapping for approximate homomorphic encryption,” in *Advances in Cryptology—EUROCRYPT* (Lecture Notes in Computer Science), vol. 10620. Cham, Switzerland: Springer, 2018, pp. 360–384.

[22] S. Halevi and V. Shoup, “Faster homomorphic linear transformations in HElib,” in *Advances in Cryptology—CRYPTO* (Lecture Notes in Computer Science), vol. 10991. Santa Barbara, CA, USA: Springer, 2018, pp. 93–120.

[23] C. Gentry, S. Halevi, and N. P. Smart, “Fully homomorphic encryption with polylog overhead,” in *Advances in Cryptology—EUROCRYPT* (Lecture Notes in Computer Science), vol. 7237. Berlin, Germany: Springer, 2012, pp. 465–482.

[24] C. Gentry, S. Halevi, and N. P. Smart, “Homomorphic evaluation of the AES circuit,” in *Advances in Cryptology—CRYPTO* (Lecture Notes in Computer Science), vol. 7417. Berlin, Germany: Springer, 2012, pp. 850–867.

[25] M. Kim, Y. Song, B. Li, and D. Micciancio, “Semi-parallel logistic regression for GWAS on encrypted data,” *Advances in Cryptology—ASIACRYPT* (Lecture Notes in Computer Science), vol. 12006. Cham, Switzerland: Springer, 2019, pp. 347–368.

[26] K. Han and D. Ki, “Better bootstrapping for approximate homomorphic encryption,” in *Topics in Cryptology—CT-RSA* (Lecture Notes in Computer Science), vol. 12006. Cham, Switzerland: Springer, 2020, pp. 364–390.

[27] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, “Bootstrapping for approximate homomorphic encryption,” in *Topics in Cryptology—CT-RSA* (Lecture Notes in Computer Science), vol. 12006. Cham, Switzerland: Springer, 2020, pp. 364–390.

[28] K. Han and D. Ki, “Better bootstrapping for approximate homomorphic encryption,” in *Topics in Cryptology—CT-RSA* (Lecture Notes in Computer Science), vol. 12006. Cham, Switzerland: Springer, 2020, pp. 364–390.

[29] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, “A full RNS variant of approximate homomorphic encryption,” in *Selected Areas in Cryptography—SAC* (Lecture Notes in Computer Science), vol. 11349. Cham, Switzerland: Springer, 2019, pp. 347–368.

[30] (Apr. 2020). *PALISADE Lattice Cryptography Library (Release 1.9.2)*. [Online]. Available: https://palisade-crypto.org/
[30] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux, “Secure and efficient bootstrapping for approximate homomorphic encryption,” Cryptol. ePrint Arch., Tech. Rep. 2020/1203, 2020.

[31] A. Q. A. Al Badawi, Y. Polyakov, K. M. M. Aung, B. Veeravalli, and K. Rohloff, “Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme,” IEEE Trans. Emerg. Topics Comput., early access, Mar. 4, 2019, doi: 10.1109/TETC.2019.2902799.

[32] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, “Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE,” in Advances in Cryptology—ASIACRYPT (Lecture Notes in Computer Science), vol. 10624. Hong Kong: Springer, 2017, pp. 377–408.

YU ISHIMAKI (Member, IEEE) received the B.S. and M.S. degrees from the Department of Computer Science and Communications Engineering, Waseda University, in 2015 and 2017, respectively, where he is currently pursuing the Ph.D. degree under the supervision of Prof. H. Yamana.

HAYATO YAMANA (Member, IEEE) has been a Professor with the Faculty of Science and Engineering, Waseda University, since 2005. He is currently a member of ACM and a Senior Member of IEICE. He served on the Board of Governors of the IEEE Computer Society from 2018 to 2020.

***