Rhombus: Fast Homomorphic Matrix-Vector Multiplication for Secure Two-Party Inference

Jiaxing He[*] Kang Yang[†] Guofeng Tang[*] Zhangjie Huang[*]
Li Lin[*] Changzheng Wei[*] Ying Yan[*] Wei Wang[*]

Abstract

We present *Rhombus*, a new secure matrix-vector multiplication (MVM) protocol in the semi-honest two-party setting, which is able to be seamlessly integrated into existing privacy-preserving machine learning (PPML) frameworks and serve as the basis of secure computation in linear layers. *Rhombus* adopts RLWE-based homomorphic encryption (HE) with coefficient encoding, which allows messages to be chosen from not only a field $\mathbb{F}_p$ but also a ring $\mathbb{Z}_{2^\ell}$, where the latter supports faster computation in non-linear layers. To achieve better efficiency, we develop an input-output packing technique that reduces the communication cost incurred by HE with coefficient encoding by about $21\times$, and propose a split-point picking technique that reduces the number of rotations to that sublinear in the matrix dimension. Compared to the recent protocol *HELiKs* by Balla and Koushanfar (CCS'23), our implementation demonstrates that *Rhombus* improves the whole performance of an MVM protocol by a factor of $7.4\times \sim 8\times$, and improves the end-to-end performance of secure two-party inference of ResNet50 by a factor of $4.6\times \sim 18\times$.

1 Introduction

Machine learning (ML) has enabled numerous applications. Particularly, model owners can provide inference services for users without the capability of training models, i.e., so-called ML as a Service (MLaaS). Users can obtain value from ML services, while model owners are able to effectively monetize their services. However, MLaaS puts forward a challenging question regarding data privacy, i.e., a model owner may learn private data of a user, or the user may reveal the model. Privacy-preserving machine learning (PPML) addresses the challenging question by providing a solution of secure two-party inference (i.e., two parties can collaboratively perform secure inference without revealing data privacy).

Homomorphic encryption (HE), a powerful cryptographic primitive, enables the model owner to perform complex computations over the encrypted user data. Although many efforts [KKK+22, SFK+22, LKL+22] have been made to execute the HE-based computations efficiently, it's still impractical to perform all the inference computations on encrypted data because the model owner has to invoke bootstrapping operations, which are extremely computation-intensive. Instead, many protocols (e.g., [MZ17, LJLA17, JVC18, SGRP19, ASKG19, CGR+19, DEK19, RSC+19, MLS+20, RRK+20, KRC+, NC21, RRG+21, PSSY21, HJSK21, CZW+21, HLHD22, RBS+22, SDF+22, HLC+22, RBG+23, HLL+23, GJM+23, BK23]) have been developed to improve the efficiency of PPML under the hybrid framework combining the HE and secure multi-party computation (MPC) techniques. An ML model consists of a sequence of linear and non-linear layers. Specifically, the operations in linear layers, e.g., 3D convolution and fully connection (FC), can be often modeled as matrix-vector multiplications (MVMs), which can be realized from HE based on the

* Digital Technologies, Ant Group, {jiaxing.hjx,tangguofeng.gf,zhangjie.hzj,felix.ll, changzheng.wcz,fuying.yy,wei.wangwwei}@antgroup.com

† State Key Laboratory of Cryptology, yangk@sklc.org, corresponding author

ring learning with errors (RLWE) [LPR10], as that of in prior works [MZ17, JVC18, MLS+20, RRK+20, CZW+21, HLHD22], while the computations in non-linear layers are performed by MPC based on secret sharings. As a result, in this hybrid HE-MPC framework, each time when performing the MVM using HE, the encrypted results need to be converted to secret sharings for the subsequent computations in non-linear layers, and thus, the multiplicative depth is only one and bootstrapping is not needed. Figure 1 presents a HE-based two-party MVM paradigm, where the model owner (Bob) holds a matrix $\mathbf{W}$ while the user (Alice) holds a vector $\mathbf{v}$, and the protocol outputs a vector of additive secret sharings $(\mathbf{r}_1, \mathbf{r}_0)$ on $\mathbf{W} \cdot \mathbf{v}$ such that $\mathbf{r}_0 + \mathbf{r}_1 = \mathbf{W} \cdot \mathbf{v}$.

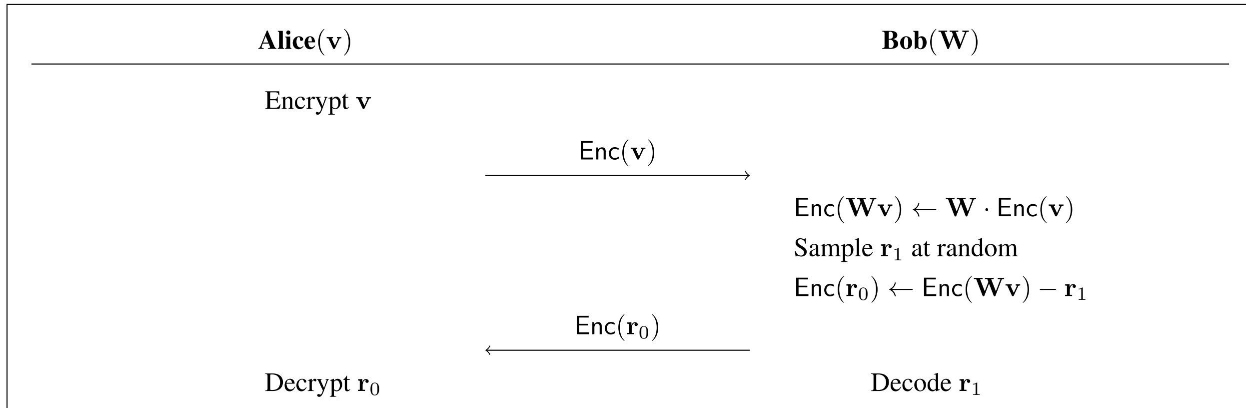

Figure 1: Secure two-party MVM paradigm based on HE.

There are two kinds of encoding approaches for RLWE-based HE to realize the MVM protocol: (1) NTT encoding that applies number theoretic transform (NTT) to the original data and then maps them to the coefficients of the plaintext polynomial, and (2) coefficient encoding that directly maps the original data to the coefficients of the plaintext polynomial. Most of prior works [JVC18, RRK+20, ZXW21, BK23, PZM+23] designed HE-based MVM protocols with NTT encoding. However, NTT encoding has three drawbacks. Firstly, the NTT encoding is more costly compared to coefficient encoding. Secondly, NTT encoding is limited to operating over a field $\mathbb{F}_p$. As shown in [RRK+20], the communication cost of the oblivious transfer (OT) based protocols in non-linear layers over $\mathbb{F}_p$ is approximately $1.5\times$ larger than that over a ring $\mathbb{Z}_{2^\ell}$ where $p \approx 2^\ell$. Thirdly, when converting homomorphic-computation results to additive secret sharings, larger HE parameters must be chosen to protect the privacy of secret values used in the homomorphic operations. The recent PPML protocol named *Cheetah* [HLHD22] employs coefficient encoding to overcome these drawbacks, at the cost of introducing a one-bit error to the computational result, which has no impact on the inference accuracy as the result would always be truncated. However, *Cheetah* encodes the MVM or 3D-convolution result in multiple ciphertexts (each encodes only a few elements of the results), which brings about a large communication overhead. In particular, the pointwise convolutions (with 1×1 filter size) incur a large communication overhead for *Cheetah*, accounting for 80% overhead of all convolutions for secure inference of ResNet50 [HZRS15], a popular deep neural network (DNN). This leads to a larger communication in linear layers of *Cheetah*, compared to the state-of-the-art PPML protocol [BK23] using HE with NTT encoding.

As analyzed above, we focus on coefficient encoding in this work. In the cleartexts, the MVM could be viewed as the inner products between each row of the matrix and the vector. The computation of each inner product is realized via placing the elements of matrix's row and vector into two separate polynomials, and then multiplying them. Based on this, one naive approach comes into play: Alice encodes her vector $\mathbf{v}$ into the coefficients of a polynomial in sequence, encrypts it, and then sends the ciphertext to Bob. After

receiving the ciphertext, Bob encodes each row of the matrix into the coefficients of a plaintext polynomial in some reversed order, and multiplies the encrypted vector by each encoded row. As a result, Bob generates r encrypted polynomials for a $r \times c$ matrix. To achieve $O(1)$ communication complexity, Bob packs the r ciphertexts into a single one by invoking the packing technique, known as PackLWEs [CDKS21], at the cost of $r - 1$ homomorphic rotations. In conclusion, the naive approach requires $O(r)$ plaintext-ciphertext multiplications and rotations.

1.1 Our Contribution

The naive approach, as described above, is inefficient when handling the matrices with a moderately large number r of rows (that is the case for most popular ML models), even if the number of columns c is small. In this paper, we propose two techniques to significantly reduce the computation cost, while keeping the communication complexity of $O(1)$. On one hand, we develop an *input-output packing* technique for MVM to reduce the number of rotations and plaintext-ciphertext multiplications to $O(rc/N)$, where $r, c\ (\leq N)$ are the numbers of rows and columns in a matrix, and N is the dimension of a polynomial ring used in RLWE-based HE. On the other hand, we present a *split-point picking* technique, which further reduces the number of rotations to $O(\sqrt{rc/N})$. This results in a lower computational cost for an end-to-end execution of the MVM protocol, compared to previous protocols.

Building upon the above techniques and RLWE-based HE with coefficient encoding, we design *Rhombus*, a new two-party matrix-vector multiplication (MVM) protocol with semi-honest security. When applying it to secure matrix multiplication, we further present two new encoding methods to reduce the communication overhead, at the cost of sightly larger computational cost. The two encoding methods enable *Rhombus* to better accommodate matrices of various dimensions. Our protocol *Rhombus* accepts the HE plaintexts from not only a field $\mathbb{F}_p$ but also a ring $\mathbb{Z}_{2^\ell}$. As a building block, *Rhombus* is able to be seamlessly integrated into existing two-party PPML frameworks to improve the performance of linear layers from simple models (e.g., logistic regression) to complicated models (e.g., Transformer-based models like GPT [BMR+20]). For performance comparison between *Rhombus* and the state-of-the-art protocols [HLHD22, BK23], we evaluate on the popular DNN model ResNet50 and use it as an example, as in the recent work [BK23].

We implemented *Rhombus* and integrated it to the open-sourced PPML library [HjLHD22]. Compared to the recent protocol *HELiKs* [BK23] (CCS 2023) using HE with NTT encoding, *Rhombus* improves the whole performance (resp., communication cost) of an MVM protocol by $7.4\times \sim 8\times$ (resp., $1.8\times$), and improves the end-to-end performance of secure ResNet50 inference by $4.6\times \sim 18\times$. Compared to the recent protocol *Cheetah* [HLHD22] (USENIX Security 2022) using HE with coefficient encoding, *Rhombus* reduces the communication cost of securely computing a matrix-vector multiplication (resp., pointwise convolutions) by $21\times$ (resp., $4.6\times$).

1.2 Technical Overview

We adopt the HE-based two-party MVM paradigm described in Figure 1, where Alice (resp., Bob) holds a vector $\mathbf{v}$ (resp., matrix $\mathbf{W}$) over an arithmetic domain (e.g., $\mathbb{F}_p$ or $\mathbb{Z}_{2^\ell}$), and $\mathbf{W} \cdot \mathbf{v}$ is converted to a vector of additive secret sharings $(\mathbf{r}_0, \mathbf{r}_1)$ such that $\mathbf{r}_0 + \mathbf{r}_1 = \mathbf{W} \cdot \mathbf{v}$. All secret values are encrypted with HE, and the computation is performed homomorphically. In this work, we first introduce two techniques, i.e., *input-output packing* and *split-point picking*, to improve homomorphic matrix-vector multiplication (MVM), which reduce the number of rotations to sublinear level to the dimension of the matrix, while keeping the communication complexity at $O(1)$. Then, we extend MVM to securely compute multiplication of two matrices by describing two new encoding methods to reduce the communication.

Matrix vector multiplication. Matrix vector multiplication can be regarded as either the inner product of each row of a matrix with a vector (called row-major), or as the linear combination of the matrix's columns

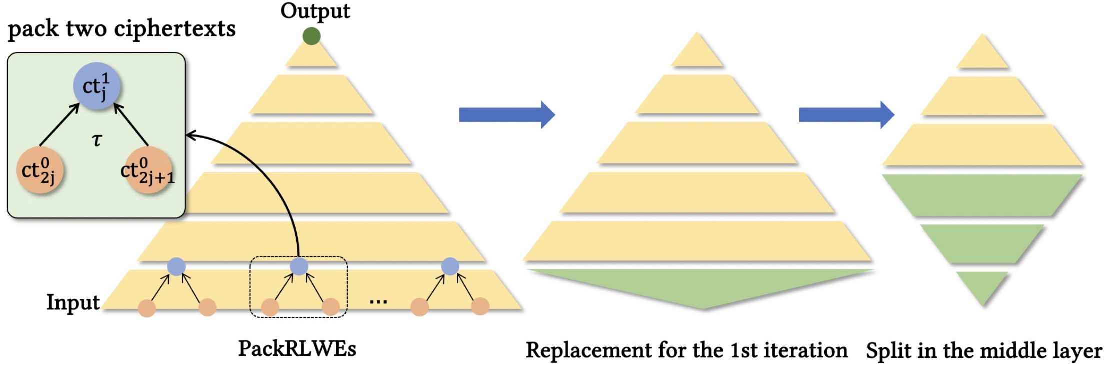

Figure 2: The high level idea of split-point picking.

with respect to the vector (called column-major). From these two perspectives, we respectively utilize the PackLWEs [CDKS21] and Expand [ACLS18, ALP+21] algorithms to design the HE-based MVM protocol, where the algorithms will be explained below. Suppose that a matrix $\mathbf{W}$ is of dimension $r \times c$, and r, c are powers of 2 for simplicity. In the row-major approach, Bob has to compute the inner product between each row of $\mathbf{W}$ and $\mathbf{v}$ homomorphically, which can be realized from the implicit convolution of the coefficients when multiplying two polynomials, as in [HLHD22]. Specifically, Bob encodes each row of $\mathbf{W}$ as a polynomial $\widehat{w}_i$ for each $i \in \{0, 1, \ldots, r-1\}$, and performs plaintext-ciphertext multiplication to obtain r ciphertexts, where the constant terms of the correponding plaintext polynomials are the results of inner product. Moreover, Bob can pack N/c rows into a length-N vector when $c < N$ (where N is the ring dimension in HE), and then computes N/c inner products from a single plaintext-ciphertext multiplication. To compress these ciphertexts, Bob first extracts the corresponding r LWE ciphertexts that encrypt r inner products from $m = rc/N$ RLWE ciphertexts ct_i for $i \in \{0, 1, \ldots m-1\}$, and then runs PackLWEs to merge the r LWE ciphertexts into a single RLWE ciphertext, as in [RCG+].

We make two key improvements to the approach described as above. We first observe that the LWE extraction is unnecessary because the results of inner products are located at indices of multiples of c within each coefficient vector of the plaintext polynomials corresponding to ct_i for $i \in \{0, 1, \ldots, m-1\}$, where c is a power of two. We can modify the PackLWEs function to directly *extract-and-merge* the coefficients at powers-of-two positions of each plaintext polynomial into a single RLWE ciphertext, where we call the procedure as PackRLWEs (see Section 3.1 for details). In this way, only $O(rc/N)$ of homomorphic rotations are needed, compared to $O(r)$ in [RCG+]. Then, a more important observation is that the input ciphertexts to PackRLWEs are obtained by multiplying different plaintext polynomials $\{\widehat{y}_i\}$ (each packing N/c rows of matrix $\mathbf{W}$) with the same ciphertext $[\![\mathbf{v}]\!] = \mathsf{Enc}(\mathbf{v})$. In other words, the input ciphertexts of PackRLWEs are all in the form of $\widehat{y}_i \cdot [\![\mathbf{v}]\!]$, and we can try to separate the plaintexts from the ciphertexts. Then the homomorphic rotation (a.k.a., automorphism) in PackRLWEs can be performed on the common ciphertext $[\![\mathbf{v}]\!]$, and the rotation results can be reused.

To be specific, we delve into more details about PackRLWEs. Its computation pipeline is structured as a full binary tree (depicted in the left triangle of Figure 2), where each leaf node represents an input ciphertext, the internal nodes represent the intermediate ciphertexts during the computation, and the root node represents the output ciphertext. The ciphertext of each internal node is obtained via running the *pack two ciphertexts* procedure on the inputs of its two child ciphertexts and an automorphism τ, illustrated in the upper left corner of Figure 2 (see Algorithm 1 of Section 3.1 for details). Note that the ciphertexts at the same level (with the exception of those on the leaf nodes) of the binary tree are computed using the same automorphism τ. Given m ciphertexts as input, PackRLWEs processes through $\log m$ iterations, with the number of ciphertexts halving in each iteration, until only one ciphertext left (as the output). Below,

we set $ct_i^0 := ct_i$, where the superscript indicates the index of the iteration in which the ciphertext resides. Specifically, in the 1st iteration, the m ciphertexts ct_i's are fed into PackRLWEs, where every two adjacent ciphertexts will be packed to one ciphertext using the *pack two ciphertexts* procedure. At the 1st layer (i.e., the 1st iteration), this procedure executes as follows:

$$ct_j^1 = \left(ct_{2j}^0 + X^{c/2} \cdot ct_{2j+1}\right) + \tau\left(ct_{2j}^0 - X^{c/2} \cdot ct_{2j+1}^0\right)$$

for $j \in \{0, 1, \cdots, m/2 - 1\}$, and requires $N/2$ homomorphic automorphism calls, where τ satisfies $\tau(X^{c/2}) = -X^{c/2}$. By replacing ct_i^0 with $\widehat{y}_i \cdot [\![\mathbf{v}]\!]$, we derive the following equation:

$$ct_j^1 = \left(\widehat{y}_{2j} + X^{c/2} \cdot \widehat{y}_{2j+1}\right) \cdot [\![\mathbf{v}]\!] + \left(\tau(\widehat{y}_{2j}) + X^{c/2} \cdot \tau(\widehat{y}_{2j+1})\right) \cdot \tau([\![\mathbf{v}]\!]).$$

This equation implies that only one homomorphic automorphism is needed now, since $\tau([\![\mathbf{v}]\!])$ could be reused for the computation of all ct_j^1's. At this point, the computational flow has transformed into the shape like the middle part of Figure 2. Subsequently, we can move on to perform the replacement recursively to the 2nd, 3rd,... iterations, and will achieve the minimum number of homomorphic automorphism calls when we arrive the middle layer of the PackRLWEs tree (see Section 3.2 for details). At this time, we obtain a *rhombus-shaped computation flow* for homomorphic automorphism, as shown in the right part of Figure 2. In particular, the number of homomorphic automorphism calls becomes $1 + 2 + \cdots + 2^{\lfloor \log m/2 \rfloor - 1} = 2^{\lfloor \log m/2 \rfloor} - 1$ for the first $\lfloor \log m/2 \rfloor$ iterations, and $(2^{\lceil \log m/2 \rceil - 1} + \cdots + 2 + 1) = 2^{\lceil \log m/2 \rceil} - 1$ for the last $\lceil \log m/2 \rceil$ iterations. Totally, $2^{\lceil \log m/2 \rceil} + 2^{\lfloor \log m/2 \rfloor} - 2$, that is, $O(\sqrt{m}) = O(\sqrt{rc/N})$ of homomorphic automorphism calls are required now. During the process, we pick the optimal split-point to switch the algorithm between the replacement and PackRLWEs. We refer to the above technique as *split-point picking* (SPP).

Our column-major based approach leverages the Expand algorithm, following the combination of columns of the matrix with the vector. Specifically, Expand($[\![\mathbf{v}]\!]$) can output c ciphertexts $[\![\mathbf{v}[i]]\!]$ on the i-th element of $\mathbf{v}$ for $i \in \{0, 1, \ldots, c-1\}$. Hence, we encode each column of $\mathbf{W}$, multiply them with $[\![\mathbf{v}[i]]\!]$, and then accumulate the computational results. This process can also benefit from the above SPP approach, where we can similarly perform the replacement as in the row-major approach, and reduce the number of homomorphic automorphisms calls to $O(\sqrt{c})$. The column-major approach is less efficient than the row-major approach, as the latter can benefit from the input-packing. However, the column-major approach will be more efficient for tall matrices with large dimensions (i.e., $r > c \geq N$), due to the choice of split point during the SPP procedure. We postpone the explanation to Section 3.3.

Matrix Multiplication. Assume that Alice has a matrix $\mathbf{Y}_{m \times k}$ (to be encrypted), while Bob holds another matrix $\mathbf{X}_{n \times m}$. Many prior works such as [JVC18, HLHD22, BK23] realized the matrix multiplication from matrix vector multiplication directly. However, the direct approach incurs significant communication overhead when m is small. To solve the problem, we give two new matrix-encoding methods called V1 and V2, and both of them are more communication efficient than prior approaches. The intuition in V1 (resp., V2) is to pack as many columns (resp., rows) of matrix $\mathbf{Y}_{m \times k}$ as possible into one ciphertext. Then, we can follow the row-major approach of MVM as described above, with the only difference where we have essentially swapped the roles of the matrix and vector, i.e., the matrix is now encrypted, while the vector (each row of $\mathbf{X}_{n \times m}$) is in plaintext to perform homomorphic computation. We refer the reader to Section 4.1 for details.

1.3 Related Work

The HE-based MVM protocol could be traced back to Halevi and Shoup's work [HS14] (*HS* in short), which has been implemented in the open-sourced library *HELib* [HS20]. They encode a matrix diagonally such that

the output is aligned and the encrypted vector is rotated by all steps, and then multiply each encoded diagonal with the corresponding rotated vector and accumulate the multiplication results. *GAZELLE* [JVC18] first applied the method to secure two-party inference of DNN. Specifically, they proposed a hybrid method with linear complexity to the matrix dimension based on the diagonal encoding. Based on the approach of *GAZELLE*, *GALA* [ZXW21] deferred the final *rotate-and-sum* operation to be performed on cleartext. Very recently, *HELiKs* [BK23] further presented a MAC (multiply-accumulate) mode, such that the rotation operations are performed after plaintext-ciphertext multiplications, which reduces the noise growth. All the above HE-based protocols adopt NTT encoding.

For coefficient encoding, *Cheetah* [HLHD22] computes the inner product from the implicit negacyclic convolution computation on the coefficients when multiplying two polynomials. However, their encoding method results in the output of multiple ciphertexts. To compress the results, Ren et al. [RCG+] further extract specific positions of the RLWE ciphertexts to multiple LWE ciphertexts and then call the PackLWEs [CDKS21] algorithm to merge these LWE ciphertexts back into a single RLWE ciphertext. Unfortunately, the invocation of PackLWEs incurs significant computation overhead, and they accelerated it with FPGA hardware to achieve a performance that is comparable with *Cheetah*. We refer the reader to Section 5.1 for the efficiency comparison between *Rhombus* and previous MVM protocols using HE with NTT/coefficient encoding.

Matrix multiplication can be naturally generalized from matrix-vector multiplication, which is done in *GAZELLE*, *Cheetah* and *HELiKs*. Nevertheless, the generalization may incur a significant amount of redundant computation and communication for some dimensions of matrices. Specifically, for two matrices of respective dimension $n \times m$ and $m \times k$, it directly increases the communication and computation overhead by a factor of k with respect to the matrix-vector multiplication with parameters (n, m). Several recent works devote to solving this issue to a certain extent in the context of secure two-party inference of Transformers. In particular, *Iron* [HLC+22] generalizes the approach in *Cheetah*, adopts the coefficient encoding without being confined to column-wise partitioning of the second matrix, and achieves communication complexity of $O(\sqrt{nmk/N})$. Based on *Iron*, *BumbleBee* [LHG+23] further compresses the output ciphertexts with the proposed *interleaving* packing technique, but requires a large number of rotations. Very recently, *BOLT* [PZM+23] generalizes the *GAZELLE*'s encoding method, makes full use of each slot of the polynomial, and achieves the optimal communication complexity (i.e., $O(k(m+n)/N)$). However, it needs to run an additional protocol to convert the computation results from a field $\mathbb{F}_p$ to a ring $\mathbb{Z}_{2^\ell}$ for every execution of secure matrix multiplication, due to the use of NTT encoding. See Section 5.2 for efficiency comparison between *Rhombus* and prior matrix-multiplication protocols.

In addition, there are a few works [JKLS, HZ23, ZLY+24] that focus on matrix multiplication in the context of outsourced computation. In such context, both matrices need to be encrypted, and the resulting ciphertexts are involved in subsequent homomorphic computations (rather than being converted into secret sharings). Therefore, the encoding method must be compatible with subsequent computations. This brings about a large number of multiplications and rotations, resulting in a low efficiency when being applied in the two-party computation scenarios.

2 Preliminaries

2.1 Notation

We use $[n]$ to denote the set $\{0, 1, \cdots, n-1\}$. A vector (resp., matrix) is represented by a bold lower-case (resp., upper-case) letter such as $\mathbf{a}$ (resp., $\mathbf{W}$). The j-th element of vector $\mathbf{a}$ is denoted by $\mathbf{a}[j]$, and the i-th row vector and j-th column vector of a matrix $\mathbf{W}$ are denoted by $\mathbf{W}_{i:}$ and $\mathbf{W}_{:j}$, respectively. For two vectors $\mathbf{a}, \mathbf{b}$, we use $\langle \mathbf{a}, \mathbf{b} \rangle$ to denote the inner product of $\mathbf{a}$ and $\mathbf{b}$. We use lower-case letters with a "hat" symbol such as $\widehat{a}$ to represent a polynomial, and $\widehat{a}[j]$ to denote the j-th coefficient of $\widehat{a}$. For a polynomial $\widehat{a}$

(resp., a vector $\mathbf{a}$), we use $[\![\widehat{a}]\!]$ (resp., $[\![\mathbf{a}]\!]$) to denote its encryption. For a set (resp., distribution) $\mathcal{D}$, $a \leftarrow\!\!\$\, \mathcal{D}$ represents that a is sampled uniformly from $\mathcal{D}$ (resp., according to the distribution $\mathcal{D}$).

2.2 Cyclotomic Field

Let $\zeta = \exp(\pi i/N)$ for a power-of-two integer N, then $K = \mathbb{Q}[\zeta]$ is the $2N$-th cyclotomic field and $R = \mathbb{Z}[\zeta]$ is the ring of integers of K. We will identify K (resp., R) with $\mathbb{Q}[X]/(X^N+1)$ (resp., $\mathbb{Z}[X]/(X^N+1)$) with respect to the map $\zeta \mapsto X$. The residue ring of R modulo an integer q is denoted by $R_q = R/qR$. An element of K (resp., R, R_q) can be uniquely represented as a polynomial of degree less than N with coefficients in $\mathbb{Q}$ (resp., $\mathbb{Z}, \mathbb{Z}_q$), such as $\widehat{a}(X) = \widehat{a}[0] + \widehat{a}[1] \cdot X + \cdots + \widehat{a}[N-1] \cdot X^{N-1}$. To extend, we have the following field extension chain:

$$\mathbb{Q} = K_0 \leq K_1 \leq K_2 \leq \cdots \leq K_{\log N} = K$$

where K_ℓ denotes the $2^{\ell+1}$-th cyclotomic field. In this work, each element $\widehat{a} \in K_\ell$ is represented as a polynomial with only non-zero terms indexed by multiples of $N/2^\ell$, with $\widehat{a} = \sum_{i=0}^{2^\ell-1} \widehat{a}[i] \cdot X^{iN/2^\ell}$.

2.3 Galois Group and Automorphisms

We recall that $K \geq \mathbb{Q}$ is a Galois extension and its Galois group $\mathsf{Gal}(K/\mathbb{Q})$ consists of the automorphisms $\tau_d : \widehat{a}(X) \mapsto \widehat{a}(X^d)$ for $d \in \mathbb{Z}_{2N}^*$, the invertible residues modulo $2N$. The automorphism $\tau_d(\cdot)$ acts on the monomials for $d = 2^\ell + 1$ with $1 \leq \ell \leq \log N$. We note that $\tau_d(X^i) = X^i$ for $i = \frac{N}{2^\ell} \cdot e$ and $\tau_d(X^i) = -X^i$ for $i = \frac{N}{2^\ell} \cdot o$, where e (resp., o) represents an even (resp., odd) integer. In other words, the map $\widehat{a} \mapsto \widehat{a} + \tau_d(\widehat{a})$ doubles the coefficients $\widehat{a}[i]$ if $i = \frac{N}{2^\ell} \cdot e$, but zeroizes the coefficients $\widehat{a}[i]$ if $i = \frac{N}{2^\ell} \cdot o$ (More details can be found in [CDKS21]).

Given the automorphisms $\tau_{2^\ell+1}$ for $\ell \in \{1, \ldots, \log N\}$ as the bases, any automorphism $\tau \in \mathsf{Gal}(K/\mathbb{Q})$ can be expressed as a combination of them:

$$\tau = \tau_{2^1+1}^{b_1} \circ \tau_{2^2+1}^{b_2} \circ \cdots \circ \tau_{2^{\log N}+1}^{b_{\log N}}, b_\ell \in \{0, 1\}.$$

In general, for any $0 \leq s < t \leq \log N$, each automorphism $\tau \in \mathsf{Gal}(K_t/K_s)$ can be expressed as $\tau = \tau_{2^{s+1}+1}^{b_{s+1}} \circ \cdots \circ \tau_{2^t+1}^{b_t} |_{K_t}$. In particular, we define $\tau(\widehat{a}) := \tau_{2^{s+1}+1}^{b_{s+1}} \circ \cdots \circ \tau_{2^t+1}^{b_t}(\widehat{a})$ for $\widehat{a} \in K, \tau \in \mathsf{Gal}(K_t/K_s)$.

2.4 RLWE-based Homomorphic Encryption

Homomorphic encryption schemes based on RLWE, e.g., BGV [BGV14], BFV [Bra12, FV12] and CKKS [CKKS17], are defined over a ring R. The decisional RLWE assumption states that the distribution in the form of $(-\widehat{a} \cdot \widehat{s} + \widehat{e}, \widehat{a}) \in R_q^2$ is computationally indistinguishable from the uniform distribution in R_q^2, where $\widehat{a} \leftarrow\!\!\$\, R_q$ and $\widehat{e} \leftarrow\!\!\$\, \chi_\sigma$ (a discrete Gaussian distribution with standard deviation σ over R_q) and $\widehat{s} \in R$ is a random secret. In this work, we use BFV as the basic HE scheme. Let t, q be two positive integers with $t \ll q$, R_q^2, R_t denote the ciphertext space and plaintext space, and q, t are the ciphertext modulus and plaintext modulus, respectively. The scheme has a set of public parameters $\mathsf{HE.pp} = \{t, q, N, \sigma\}$ where N is the ring dimension. We will adopt the following algorithms to design an MVM protocol:

- **KeyGen.** We adopt the symmetric-key BFV-HE scheme [KPZ21], where the encryption and decryption depend on the same secret key $\widehat{sk}$, sampled uniformly from R_3 (i.e., each coefficient of polynomial $\widehat{sk}$ is sampled from $\{-1, 0, 1\}$).

- **Encode.** We use two coefficient encoding algorithms, denoted by Ecd_1 and Ecd_2. Specifically, Ecd_1 maps a vector $\mathbf{a} \in \mathbb{Z}_t^\ell$ to a polynomial $\widehat{a} \in R_t$ with $\widehat{a}[j] = \mathbf{a}[j]$ for $j \in [N]$ (padding zero if $\ell < N$). $\mathsf{Ecd}_2(\mathbf{a})$ is the reciprocal polynomial of $\mathsf{Ecd}_1(\mathbf{a})$, i.e., $\mathsf{Ecd}_2(\mathbf{a}) = \widehat{a}(X^{-1}) \in R_t$. We note that the constant term of $\mathsf{Ecd}_1(\mathbf{a}) \cdot \mathsf{Ecd}_2(\mathbf{b}) = \widehat{a}(X) \cdot \widehat{b}(X^{-1})$ is the inner product of $\mathbf{a}$ and $\mathbf{b}$ for two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_t^\ell$.
- **Decode.** Given a plaintext polynomial $\widehat{m} \in R_t$, this algorithm outputs the corresponding coefficient vector $\mathbf{m} \in \mathbb{Z}_t^\ell$.
- **Encrypt.** Given a plaintext polynomial $\widehat{m} \in R_t$, the encryption algorithm outputs a ciphertext $ct = (\widehat{c}_0, \widehat{c}_1)$, such that $\widehat{c}_0 = -\widehat{a} \cdot \widehat{sk} + \widehat{e} + \left\lfloor \frac{q}{t} \cdot \widehat{m} \right\rceil \in R_q$ and $\widehat{c}_1 = \widehat{a} \in R_q$ where $\widehat{a} \leftarrow\!\!\$\, R_q$ and $\widehat{e} \leftarrow\!\!\$\, \chi_\sigma$. We use Enc to denote the encryption algorithm, and also denote by $[\![\widehat{m}]\!]$ the ciphertext on $\widehat{m}$.
- **Decrypt.** Given a ciphertext $ct = (\widehat{c}_0, \widehat{c}_1) \in R_q^2$, the decryption algorithm outputs $\widehat{m} = \left\lfloor \frac{t}{q} (\widehat{c}_0 + \widehat{c}_1 \cdot \widehat{sk}) \right\rceil \in R_t$.
- **HomMulPt.** Given a polynomial $\widehat{a} \in R_t$ and a ciphertext $ct \in R_q^2$ on a plaintext polynomial $\widehat{m} \in R_t$, this algorithm $\mathsf{HomMulPt}$ outputs a ciphertext decrypted to $\widehat{m} \cdot \widehat{a} \in R_t$. This plaintext-ciphertext multiplication will amplify the noise from η to $\eta \cdot \eta_{mul}$ where η_{mul} is the multiplicative noise growth factor, which depends on the infinity norm of $\widehat{a}$. We refer the reader to [Bra12, FV12] for the details of $\mathsf{HomMulPt}$.
- **HomAut.** Given a ciphertext ct on a plaintext polynomial $\widehat{m}$ and an automorphism $\tau \in \mathsf{Gal}(K/\mathbb{Q})$, this algorithm homomorphically computes a new ciphertext $ct' \leftarrow \mathsf{HomAut}(ct, \tau)$, which is decrypted to $\tau(\widehat{m})$. The homomorphic automorphism (a.k.a., rotation) operation will introduce an additive noise η_{aut}, and amplifies the noise from η to $\eta + \eta_{aut}$. The details of HomAut can be found in [Bra12, FV12].
- **PackLWEs.** Given $2^\ell (\le N)$ ciphertexts $\{ct_i\}_{i \in [2^\ell]}$ where ct_i encrypts $\widehat{m}_i \in R_t$, this algorithm $\mathsf{PackLWEs}$ homomorphically merge the constant terms of them into a new ciphertext ct, decrypted to $\widehat{m} \in R_t$ with $\widehat{m}[(N/2^\ell) \cdot i] = 2^\ell \cdot \widehat{m}_i[0]$ for $i \in [2^\ell]$. The algorithm details can be found in [CDKS21].
- **Expand.** Given an integer $\ell \le \log N$ and a ciphertext ct on a plaintext polynomial $\widehat{m} = \sum_{i=0}^{2^\ell - 1} \widehat{m}[i] \cdot X^i \in R_t$, this algorithm homomorphically expands it to 2^ℓ ciphertexts $\{ct_i\}_{i \in [2^\ell]}$, where ct_i is decrypted to $2^\ell \cdot \widehat{m}[i] \in \mathbb{Z}_t$. We write it as $\{ct_i\}_{i \in [2^\ell]} \leftarrow \mathsf{Expand}(ct, \ell)$. We refer the reader to [ACLS18, ALP+21] for details.

Additionally, we will invoke an algorithm (denoted by HomAdd) for computing homomorphic additions (see [Bra12, FV12] for details). In general, for both $\mathsf{PackLWEs}$ and Expand, a single invocation requires $2^\ell - 1$ calls of HomAut, and the value 2^ℓ in either $\{2^\ell \cdot \widehat{m}_i[0]\}$ or $\{2^\ell \cdot \widehat{m}[i]\}$ could be removed via multiplying the input ciphertexts by its inverse $2^{-\ell}$.

2.5 Conversion between Additive SS and HE

We use additive secret sharing (SS) over a ring $\mathbb{Z}_t$. Concretely, for an element $x \in \mathbb{Z}_t$, an additive SS on x (denoted by $\langle x \rangle$) is generated by sampling two shares $\langle x \rangle_0$ and $\langle x \rangle_1$ from $\mathbb{Z}_t$, such that $\langle x \rangle_0 + \langle x \rangle_1 = x \mod t$. The conversion from additive secret sharings to HE ciphertexts (denoted by A2H) and reverse direction (denoted by H2A) are described as follows.

Conversion from SS to HE. Suppose that P_i for $i \in \{0, 1\}$ holds a vector of additive secret sharings $\langle \mathbf{x} \rangle_i$, where $\langle \mathbf{x} \rangle_0 + \langle \mathbf{x} \rangle_1 = \mathbf{x} \mod t$. To convert the secret values from SS to HE, for each $i \in \{0, 1\}$, P_i encodes its shares by $\widehat{x}_i \leftarrow \mathsf{Ecd}_1(\langle \mathbf{x} \rangle_i)$, and encrypts the resulting polynomial as $\mathsf{Enc}(\widehat{x}_i)$. Then, P_i sends $\mathsf{Enc}(\widehat{x}_i)$ to P_{1-i}, who performs the homomorphic addition $\mathsf{Enc}(\widehat{x}_i) + \mathsf{Ecd}_1(\langle \mathbf{x} \rangle_{1-i})$ to get the HE ciphertext $\mathsf{Enc}(\widehat{x})$ such that $\widehat{x} = \mathsf{Ecd}_1(\mathbf{x})$.

Conversion from HE to SS. Following prior works [HLHD22, CZ22], for the BFV scheme [Bra12, FV12] with coefficient encoding, the H2A conversion executes as follows:

1. Given a ciphertext $\mathsf{Enc}(\widehat{x})$ where $\widehat{x} \in R_t$ encodes a vector $\mathbf{x} \in \mathbb{Z}_t^N$, for some $i \in \{0,1\}$, P_i samples $\widehat{r} \leftarrow\!\!\$\, R_q$, and then sends $\mathsf{Enc}(\widehat{x}) - \widehat{r}$ to P_{1-i}. P_i computes $\langle \mathbf{x} \rangle_i := \lfloor (t/q) \cdot \mathbf{r} \rceil$ as its own shares, where $\mathbf{r} \in \mathbb{Z}_q^N$ is the coefficient vector of $\widehat{r}$.
2. P_{1-i} decrypts the ciphertext $\mathsf{Enc}(\widehat{x}) - \widehat{r}$ to a polynomial, and then transforms the polynomial to the coefficient vector as its own shares $\langle \mathbf{x} \rangle_{1-i}$.

Using the above H2A procedure, the noise flooding [Gen09] is eliminated, if a one-bit error is allowed (that is the case for PPML applications). For NTT encoding, noise flooding needs to be used, which leads to a noise 2^λ larger than that in normal encryption for statistical security parameter λ. This blows up the ciphertext modulus q of NTT encoding to at least λ bits larger than that of coefficient encoding.

2.6 Threat Model

Our protocol in the two-party setting is secure in the presence of any semi-honest adversary, who follows the protocol specification but tries to learn more than allowed from the protocol transcripts. Security against semi-honest adversaries has been considered in most of PPML protocols like [JVC18, HLHD22, BK23]. We always assume that the adversary is probabilistic polynomial time (PPT). For PPML applications, one party (i.e., client) holds an input data, while the other party (i.e., server) holds the model parameters. Through jointly running a secure two-party computation (2PC) protocol, the client will obtain an inference result on the input data, and the server obtains nothing. For security, a party cannot learn any information on private data of the other party beyond the inference results known only by the client.

3 Matrix Vector Multiplication

In this section, we present a new matrix vector multiplication (MVM) protocol, called *Rhombus*, shown in Figure 1. In particular, Alice (resp., Bob) holds a vector $\mathbf{v}$ (resp., a matrix $\mathbf{W}$), and the homomorphic-computation result will be converted to additive secret sharings. The conversion can be realized by invoking the H2A protocol shown in Section 2.5. Below, we focus on the part of homomorphic computation for MVM, and present the *input-output packing* (Section 3.1) and the *split-point picking* (Section 3.2) techniques to reduce the computation complexity based on the naive approach described in Section 1. Moreover, we also give the column-major based approach for MVM in Section 3.3, which is more suitable for tall matrices with large dimensions.

3.1 Input-Output Packing

The naive approach requires $O(r)$ of $\mathsf{HomMulPt}$ and HomAut calls. However, it fails to fully exploit the packing property of polynomials when $c < N$, thereby resulting in inefficient computation. For the matrix $\mathbf{W}$ with $c < N$, it is natural to encode N/c rows of $\mathbf{W}$ into a single plaintext polynomial, and then compute the N/c inner products by only one $\mathsf{HomMulPt}$. In this way, only rc/N $\mathsf{HomMulPt}$ calls are required, and we call it *input-packing*. However, the $\mathsf{PackLWEs}$ algorithm cannot be directly applied because the inner products are located at indices that are multiples of c in the coefficient vector of a polynomial, instead of only constant terms.

PackRLWEs. To solve the problem, we extend the $\mathsf{PackLWEs}$ algorithm to $\mathsf{PackRLWEs}$, and introduce an additional parameter h that indicates each ciphertext has 2^h coefficients to be merged. We present it in Algorithm 1 and give an example in Figure 3. In Proposition 1, we show the correctness of the $\mathsf{PackRLWEs}$ algorithm.

Algorithm 1: $\mathsf{PackRLWEs}(\{ct_j\}_{j\in[2^\ell]}, h)$

Input: Ciphertexts $\{ct_j\}_{j\in[2^\ell]}$ and an integer $h \in [\log N + 1]$ with $\ell + h \le \log N$.
Output: A single ciphertext ct.

1: **if** $\ell = 0$ **then**
2:     **return** $ct \leftarrow ct_0$
3: **else**
4:     $ct_e \leftarrow \mathsf{PackRLWEs}(\{ct_{2j}\}_{j\in[2^{\ell-1}]}, h)$
5:     $ct_o \leftarrow \mathsf{PackRLWEs}(\{ct_{2j+1}\}_{j\in[2^{\ell-1}]}, h)$
6:     $ct \leftarrow (ct_e + X^{N/2^{\ell+h}} \cdot ct_o) + \mathsf{HomAut}(ct_e - X^{N/2^{\ell+h}} \cdot ct_o, \tau_{2^{\ell+h}+1})$
7:     **return** ct
8: **end**

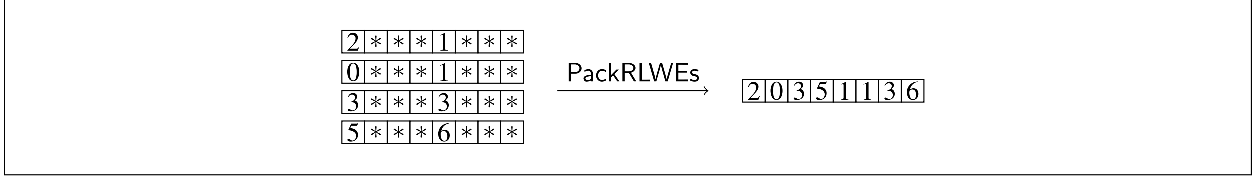

Figure 3: An example of PackRLWEs with $\ell = 2, h = 1, N = 8$

Proposition 1. *Given ciphertexts $\{ct_j\}_{j\in[2^\ell]}$ (decrypted to $\widehat{a}_j$ for each $j \in [2^\ell]$) and an integer $h \in [\log N + 1]$ with $\ell + h \le \log N$,* PackRLWEs *outputs a ciphertext ct which is decrypted to $\widehat{a}$, such that $\widehat{a}[(N/2^h)\cdot k + (N/2^{h+\ell})\cdot j] = 2^\ell \cdot \widehat{a}_j[(N/2^h)\cdot k]$ for each $j \in [2^\ell], k \in [2^h]$.*

The correctness of the above proposition directly comes from [CDKS21]. The number of HomAut calls required by PackRLWEs with input 2^ℓ ciphertexts is still $2^\ell - 1$, which is independent of h, compared to that PackLWEs requires HomAut calls of $2^{\ell+h} - 1$. Note that PackRLWEs will degenerate to PackLWEs when $h = 0$.

With the extended packing algorithm PackRLWEs, one can realize the input-output packing to reduce the number of HomMulPt and HomAut calls to $O(rc/N)$. In particular, we refer to the final step of PackRLWEs as *output-packing*, which packs rc/N ciphertexts into one ciphertext. A toy example for the input-output packing process is shown in Figure 4 with $N = 8$, $r = 4$ and $c = 4$.

3.2 Split-Point Picking

The *input-output packing* described in Section 3.1 reduces the number of HomMulPt and HomAut calls to $O(rc/N)$. Accordingly, the experimental results in Figure 8 imply that the HomAut operation will dominate the whole computation. In the following, we present the critical technique called split-point picking (SPP), which further reduces the number of HomAut calls to $O(\sqrt{rc/N})$.

For the SPP technique, a key observation is that the input ciphertexts of PackRLWEs are obtained by multiplying different plaintexts with the same ciphertext (the encrypted vector) in the *input-output packing* algorithm. By separating these plaintexts from the ciphertexts, it becomes possible to apply HomAut solely on the common ciphertext.

We start with the *input-output packing* algorithm described as above, given the $r \times c$ matrix $\mathbf{W}$ and a ciphertext $[\![\widehat{v}]\!]$ as input, where $\mathbf{W}$ is encoded to $\{\widehat{y}_i\}_{i\in[m]}$ with $m = rc/N$ via the input packing. At this point, the ciphertexts input to PackRLWEs are in the form of $ct_i^0 = \widehat{y}_i \cdot [\![\widehat{v}]\!]$ for $i \in [m]$. Recall

$$\mathbf{W} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{bmatrix} \quad \mathbf{v} = \begin{bmatrix} 1 \\ 2 \\ 1 \\ 3 \end{bmatrix} \quad \mathbf{W}\mathbf{v} = \begin{bmatrix} 20 \\ 15 \\ 18 \\ 17 \end{bmatrix}$$

Encode matrix $\mathbf{W}$: $\widehat{w}_i \leftarrow \mathsf{Ecd}_2(\mathbf{W}_{i:}),\ i \in [4]$
Input packing:

$$\widehat{y}_0 = \widehat{w}_0 + X^4\widehat{w}_2 = 1 + 2X + X^2 + 4X^3 + 3X^4 - 4X^5 - 3X^6 - 2X^7$$
$$\widehat{y}_1 = \widehat{w}_1 + X^4\widehat{w}_3 = 2 + 3X + 2X^2 + X^3 + 4X^4 - X^5 - 4X^6 - 3X^7$$

$\Downarrow$ Multiply by $\widehat{v} = 1 + 2X + X^2 + 3X^3$

$$\widehat{y}_0 \cdot \widehat{v} = 20 + \cdots + 18X^4 + \cdots$$
$$\widehat{y}_1 \cdot \widehat{v} = 15 + \cdots + 17X^4 + \cdots$$

$\Downarrow$ Output packing

$$\widehat{a} = 20 + \cdots + 15X^2 + \cdots + 18X^4 + \cdots + 17X^6 + \cdots$$

Figure 4: Input-output packing for $N = 8, r = c = 4$.

that PackRLWEs involves $\log m$ iterations. For the i-th ($1 \leq i \leq \log m$) iteration, it reduces the number of ciphertexts from $m/2^{i-1}$ to $m/2^i$, using the automorphism $\tau_{2^i(N/c)+1}$. We denote the $m/2^{i-1}$ input ciphertexts in the i-th iteration as $\{ct_j^{i-1}\}_{j\in[m/2^{i-1}]}$. We first focus on the computation in the 1st iteration, which transforms the set of ciphertexts $\{ct_j^0\}_{j\in[m]}$ into $\{ct_j^1\}_{j\in[m/2]}$, with $m/2$ HomAut calls. In particular,

$$ct_j^1 = (ct_{2j}^0 + X^{c/2} \cdot ct_{2j+1}^0) + \mathsf{HomAut}(ct_{2j}^0 - X^{c/2} \cdot ct_{2j+1}^0, \tau_{2N/c+1})$$

for $j \in [m/2]$, according to step 6 in Algorithm 1. Based on the homomorphic property of $\tau_{2N/c+1}$, replacing ct_{2j}^0, ct_{2j+1}^0 with $\widehat{y}_{2j} \cdot [\![\widehat{v}]\!], \widehat{y}_{2j+1} \cdot [\![\widehat{v}]\!]$ respectively, we obtain

$$ct_j^1 = \left(\widehat{y}_{2j} + X^{c/2} \cdot \widehat{y}_{2j+1}\right) \cdot [\![\widehat{v}]\!] + \left(\tau_{2N/c+1}\left(\widehat{y}_{2j}\right) + X^{c/2} \cdot \tau_{2N/c+1}\left(\widehat{y}_{2j+1}\right)\right) \cdot \mathsf{HomAut}\left([\![\widehat{v}]\!], \tau_{2N/c+1}\right) \quad (1)$$

From the above equation (1), we find that all HomAut operations are performed on the same ciphertext $[\![\widehat{v}]\!]$. Therefore, HomAut now needs to be called only once in the 1st iteration. Note that the automorphisms on plaintexts are much faster than those on ciphertexts, hence, we ignore the automorphisms acting on plaintexts temporarily. Similarly, we can proceed to carry out the replacement for the 2nd iteration if $m \geq 4$. Specifically, the equation in this iteration is shown as follows:

$$ct_j^2 = \left(ct_{2j}^1 + X^{c/4} \cdot ct_{2j+1}^1\right) + \mathsf{HomAut}\left(ct_{2j}^1 - X^{c/4} \cdot ct_{2j+1}^1, \tau_{4N/c+1}\right)$$

for $j \in [m/4]$. Replacing ct_{2j}^1 with the ciphertext defined in the equation (1), we can derive the following equation:

$$ct_j^2 = \sum_{\tau \in \mathrm{Gal}(K_{h+2}/K_h)} \left(\sum_{k=0}^{3} \tau\left(\widehat{y}_{4j+k}\right) X^{kc/4}\right) \mathsf{HomAut}\left([\![\widehat{v}]\!], \tau\right) \quad (2)$$

where $h = \log(N/c)$, $\mathrm{Gal}(K_{h+2}/K_h) = \{1, \tau_{2^{h+1}+1}, \tau_{2^{h+2}+1}, \tau_{2^{h+2}+1} \circ \tau_{2^{h+1}+1}\}$. The equation (2) indicates that computing ct_j^2 in this manner requires only $\#\mathrm{Gal}(K_{h+2}/K_h) - 1 = 3$ HomAut calls. From the replacements (1) and (2), we can summarize that $\#\mathrm{Gal}(K_{h+L}/K_h) - 1 = 2^L - 1$ HomAut calls are required when computing ct_j^L for $j \in [m/2^L]$ in the replacement way. In the extreme case, $m-1$ HomAuts are needed if we compute $ct_0^{\log m}$ using the above approach, which is equivalent to invoking the PackRLWEs directly.

Based on the derivation above, we will achieve the optimal efficiency, if switching the computation algorithm in the middle layer of the PackRLWEs tree. In particular, we adopt the replacement mentioned as above in the first $\lfloor (\log m)/2 \rfloor$ iterations of the PackRLWEs tree with HomAut calls of $2^{\lfloor (\log m)/2 \rfloor} - 1$. Note that the number of ciphertexts is $m/2^{\lfloor (\log m)/2 \rfloor}$ at this time. Then we invoke PackRLWEs on the input of $m/2^{\lfloor (\log m)/2 \rfloor}$ ciphertexts, which requires $m/2^{\lfloor (\log m)/2 \rfloor} - 1$ HomAut calls. Totally, the number of HomAut calls is $2^{\lfloor (\log m)/2 \rfloor} + m/2^{\lfloor (\log m)/2 \rfloor} - 2$, which amounts to a complexity of $O(\sqrt{m}) = O(\sqrt{rc/N})$. We refer to this replacement approach as *split-point picking*. In the context of MVM, we usually choose the split point corresponding to the middle layer in the PackRLWEs tree. We summarize the process in Algorithm 2.

Proposition 2. *Given a matrix* $\mathbf{W}$ *with 2-power dimensions* $r, c \le N$*, an encrypted vector* $[\![\mathbf{v}]\!]$ *(encoding via* Ecd_1*) and the split-point* u*, Algorithm 2 outputs a ciphertext* ct' *which decrypts to* $\widehat{a}'$*, such that* $\widehat{a}'[(N/r) \cdot i] = (\mathbf{W} \cdot \mathbf{v})[i]$ *for* $i \in [r]$.

The proposition 2 shows the correctness of Algorithm 2, and we defer the proof to Appendix A. We note that only the coefficients with indices being multiples of N/r are useful for the output ciphertext. Therefore, for the output ciphertext in the form of $(\widehat{c}_0, \widehat{c}_1) \in R_q^2$, we can drop the irrelevant coefficients of $\widehat{c}_0$ to reduce the communication overhead.

This work follows the MVM protocol framework shown in Figure 1, and only improves the part of locally homomorphic computation by invoking Algorithm 2. Therefore, the improved MVM protocol with Algorithm 2 is natural to be secure in the presence of semi-honest adversaries under the RLWE assumption, by directly following the security proof of previous protocols, e.g., [HLHD22].

3.3 Column Major based MVM

Section 3.2 gives the MVM algorithm by regarding the MVM as inner products between the vector $\mathbf{v}$ and each row of the matrix $\mathbf{W}$. Alternatively, MVM can also be viewed as the linear combination of the columns of the matrix with the vector. In this way, we can leverage the Expand algorithm described in Section 2.4 to compute MVM as follows.

1. On input of a matrix $\mathbf{W}$ of dimension $r \times c$ and an encrypted vector $[\![\mathbf{v}]\!]$ with Ecd_1 for encoding, run the Expand algorithm to expand $[\![\mathbf{v}]\!]$ to c ciphertexts, each of them encrypts the value $\mathbf{v}[i]$ for $i \in [c]$.

2. For $i \in [c]$, encode the column vector $\mathbf{W}_{:i}$ of the matrix $\mathbf{W}$ by Ecd_1 into a plaintext polynomial $\widehat{w}_i$.

3. Execute the HomMulPt algorithm on these expanded ciphertexts and the encoded columns, and then accumulate these ciphertexts.

The correctness is straightforward based on the definition of Expand. The column-major approach requires c HomMulPt calls for computing the linear combination, and $c-1$ HomAut calls for running Expand. Similar to the row-major approach, the number of HomAut calls can be further reduced to $O(\sqrt{c})$ using the SPP optimization. We provide the details of the column-major MVM algorithm in Appendix B.

It is easy to see that the row-major approach outperforms the column-major approach when both the number of rows and columns of the matrix are smaller than N because the latter cannot benefit from the *input packing*. More precisely, we can pack multiple rows of the matrix to one plaintext polynomial with

Algorithm 2: $\mathsf{RhombusMVM_{RM}}$: Row major based matrix vector multiplication with SPP optimization.

Input: $r \times c$ Matrix with $r, c \leq N$ (powers of two), $ct \leftarrow [\![\mathbf{v}]\!] \in R_q^2$, and the split-point u.
Output: A ciphertext ct'.

1: **for** $i \in [r]$ **do**
2: $\quad \widehat{w}_i \leftarrow \mathsf{Ecd}_2(\mathbf{W}_{i:})$
3: **end**
4: $m \leftarrow r \cdot c/N, h \leftarrow \log(N/c)$
5: $ct \leftarrow m^{-1} \cdot ct$ // To remove the PackRLWEs factor
6: **for** $i \in [m]$ **do**
7: $\quad \widehat{y}_i \leftarrow \sum_{j=0}^{N/c-1} X^{j \cdot c} \cdot \widehat{w}_{i+j \cdot m}$ // Input packing
8: **end**
9: **for** $\tau \in \mathrm{Gal}(K_u/K_h)$ **do**
10: $\quad ct_\tau \leftarrow \mathsf{HomAut}(ct, \tau)$ // SPP: Replacement
11: **end**
12: **for** $i_1 \in [r/2^u]$ **do**
13: $\quad ct_{i_1} \leftarrow \mathsf{Enc}(0)$
14: $\quad$ **for** $\tau \in \mathrm{Gal}(K_u/K_h)$ **do**
15: $\quad\quad \widehat{z}_{i_1,\tau} \leftarrow \sum_{i_0=0}^{m2^u/r-1} X^{(N/2^u) \cdot i_0} \cdot \tau(\widehat{y}_{(r/2^u) \cdot i_0 + i_1})$
16: $\quad\quad ct_{i_1} \leftarrow \mathsf{HomAdd}(ct_{i_1}, \mathsf{HomMulPt}(ct_\tau, \widehat{z}_{i_1,\tau}))$
17: $\quad$ **end**
18: **end**
19: $ct' \leftarrow \mathsf{PackRLWEs}(\{ct_{i_1}\}_{i_1 \in [r/2^u]}, u)$ // SPP: PackRLWEs
20: **return** ct'

row-major approach, but for column-major approach we cannot pack multiple columns to one polynomial since each column will be multiplied by different element of the vector.

However, for matrix of large dimensions, i.e., the number of rows or columns is larger than N, the two approaches have their own merits. To analyze the case of large matrix, we suppose that $r = r'N, c = c'N$ for simplicity, where r', c' are positive integers. In this case, we have to partition the large matrix $\mathbf{W}$ into $r' \times c'$ submatrices, each of dimension $N \times N$, and partition the vector $\mathbf{v}$ into c' subvectors correspondingly, then perform the MVM on each pair of the submatrices and subvectors, respectively. We count the number of HomMulPt, HomAut calls and the complexity of matrix processing, corresponding to Step 15 in Algorithm 2, Step 10 in Algorithm 4, to compare the concrete efficiency of the two approaches. Before conducting the concrete comparison, we first observe the following facts:

- Both the two approaches require rc/N times of HomMulPt.
- With the row major approach, the PackRLWEs (Step 19 in Algorithm 2) can be called only once for the MVM computation with respect to the submatrices in the same row (note that the results of these MVMs will be accumulated), because it is additive homomorphic on packing two groups of ciphertexts: $\mathsf{PackRLWEs}(\{ct_i\}, u) + \mathsf{PackRLWEs}(\{ct'_i\}, u) = \mathsf{PackRLWEs}(\{ct_i + ct'_i\}, u)$.
- With the column major approach, the similar fact holds true, i.e., the HomAut (Step 13 in Algorithm 4) can also be merged, and thus be called only once, for the MVM computation with respect to the submatrices in the same row.

- For the matrix processing, larger split point u leads to more complex processing in both of the two algorithms.

Based on the analyses above, we can derive the concrete number of HomAut calls when employing the two approaches on large matrix (note that the *input packing* is not applicable), respectively. For row major approach with split point $u_r(0 \le u_r \le \log N)$, the vector $\mathbf{v}$ is encrypted to c' ciphertexts, each of them will go through $2^{u_r} - 1$ of HomAut calls in Step 10 in Algorithm 2 (note that $h = 0$ in this case), in addition, for the MVMs with respect to the submatrices in the same row, we need $N/2^{u_r} - 1$ of HomAut calls inside the PackRLWEs. Totally, $c'(2^{u_r} - 1) + r'(N/2^{u_r} - 1)$ of HomAut are needed. For the column major approach with split point $u_c(0 \le u_c \le \log N)$, each of the c' input ciphertexts will go through the $N/2^{u_c} - 1$ of HomAut calls at Step 5, besides, $2^{u_c} - 1$ of HomAut are needed for the MVMs with respect to the submatrices in the same row. Totally, $c'(N/2^{u_c} - 1) + r'(2^{u_c} - 1)$ of HomAut are needed when using column-major approach. As a result, the column major approach has the same complexity of HomAut calls as the row major approach if we set $u_c = \log N - u_r$, and the only difference lies on the matrix processing.

In summary, for the tall matrix with $r' > c'$ (i.e., $r > c$), the column major approach will outperform the row major approach. The reason is in this case, row major approach will need a larger $u_r(> (\log N)/2)$ to minimize the HomAut calls, leading to a more complex matrix processing than column-major one. However, for rectangular matrix with $r' < c'$ (i.e., $r < c$), row major will perform better than column major approach, due to a cheaper matrix processing in this case. We present the concrete efficiency comparison between these two approaches in Table 2 in Section 5.1.

4 Matrix Multiplication

In this section, we focus on secure matrix multiplication, where Alice holds a matrix $\mathbf{Y}_{m\times k}$ (to be encrypted), and Bob holds another matrix $\mathbf{X}_{n\times m}$. Both parties will execute a protocol similar to Figure 1 to securely compute additive secret sharings of $\mathbf{X} \cdot \mathbf{Y}$. Although matrix multiplication could be naturally generalized from MVM, the cost of matrix multiplication with parameters (n, m, k) becomes k times that of MVM with matrix dimension (n, m) in this case. Especially when $n, m \ll N$, it results in significant communication waste. Below, we present two new matrix encoding methods that fully utilize all coefficients of the polynomial to achieve optimal asymptotic communication complexity. We consider the row-major approach in Section 4.1, and then show the corresponding column-major method in Section 4.2.

4.1 Matrix Encoding for Row-Major Approach

Recall that in the MVM algorithm shown in Section 3, the matrix is encoded in a packed manner, while the vector is encoded alone. When it comes to matrix multiplication, we can reverse the encoding manner of the matrix and vector to give a communication efficient method. That is, for the matrix $\mathbf{Y}$ to be encrypted, we pack multiple columns into one plaintext polynomial, similar to the encoding of matrices in MVM. Once the columns of matrix $\mathbf{Y}$ are packed, each row of matrix $\mathbf{X}$ must be encoded separately, and then has to be multiplied with the encrypted $\mathbf{Y}$. Figure 5 gives an example for the case that $n = m, mk = N$ where n, m, k are powers of two. Furthermore, we can likewise apply the SPP approach. We describe it in Algorithm 3.

For general dimensions n, m, k of matrices, we have to partition them into several blocks satisfying the above requirements, and perform the computation on each pair of blocks of $\mathbf{X}$ and $\mathbf{Y}$ using Algorithm 3. In the following, we give two partition methods V1, V2, which enable a trade-off between the matrix encoding and homomorphic automorphism.

Matrix Partition V1. The first method is to partition the matrix $\mathbf{Y}$ vertically. Specifically, we pad m to a power of 2, denoted by $\overline{m}$. Then, every $N/\overline{m}$ adjacent columns of $\mathbf{Y}$ will be packed into a single polynomial, and thus $\mathbf{Y}$ will be encrypted to $\lceil k\overline{m}/N \rceil$ ciphertexts. For the matrix $\mathbf{X}$, we divide it into $\lceil n/\overline{m} \rceil$ blocks

$$\mathbf{X} = \begin{bmatrix} 1 & 2 & 5 & 1 \\ 2 & 3 & 1 & 1 \\ 3 & 1 & 2 & 1 \\ 4 & 2 & 1 & 2 \end{bmatrix} \quad \mathbf{Y} = \begin{bmatrix} 3 & 1 \\ 1 & 2 \\ 2 & 2 \\ 1 & 3 \end{bmatrix} \quad \mathbf{Z} = \mathbf{X}\mathbf{Y} = \begin{bmatrix} 11 & 18 \\ 12 & 13 \\ 15 & 12 \\ 18 & 16 \end{bmatrix}$$

Encode matrix $\mathbf{Y}$: $\{\, \widehat{y} \leftarrow \mathsf{Ecd}_1(\mathbf{Y}_{:0}) + X^4 \mathsf{Ecd}_1(\mathbf{Y}_{:1}) \,\}$

Encode matrix $\mathbf{X}$: $\{\, \widehat{x}_i \leftarrow \mathsf{Ecd}_2(\mathbf{X}_{i:}),\ i \in [4] \,\}$

$\Downarrow$ Multiply $\widehat{x}_i$ by $\widehat{y}$

$$\widehat{x}_i \cdot \widehat{y} = \langle \mathbf{X}_{i:}, \mathbf{Y}_{:0} \rangle + \cdots + \langle \mathbf{X}_{i:}, \mathbf{Y}_{:1} \rangle X^4 + \cdots, i \in \{0, 1, 2, 3\}$$

$\Downarrow$ output packing

$$\widehat{z} = 11 + 12X + 15X^2 + 18X^3 + 18X^4 + 13X^5 + 12X^6 + 16X^7$$

Figure 5: Example for row-major based matrix multiplication with $n = m = 4, k = 2, N = 8$.

horizontally, each of them is of dimension $\overline{m} \times \overline{m}$ (padding zero for the last block if necessary). In this way, each block of $\mathbf{X}$ will be multiplied by every ciphertext on $\mathbf{Y}$, as shown in the left part of Figure 6.

Matrix Partition V2. Instead of padding m, we can also pad k to a power of 2, denoted by $\overline{k}$. Then, we divide the matrix $\mathbf{Y}$ into $\lceil m\overline{k}/N \rceil$ blocks horizontally, each of dimension $N/\overline{k} \times \overline{k}$. Accordingly, $\mathbf{X}$ is partitioned into $\lceil n\overline{k}/N \rceil \times \lceil m\overline{k}/N \rceil$ blocks, where each block is of dimension $N/\overline{k} \times N/\overline{k}$, as shown in the right part of Figure 6.

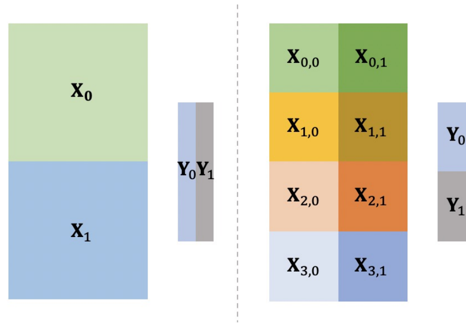

Figure 6: Two methods of partitioning matrices $\mathbf{X}$, $\mathbf{Y}$

Below, we analyze the following performance characteristics for the two partition methods with SPP optimization:

V1. The number of HomMulPt calls is $n \cdot \lceil \overline{m}k/N \rceil$. The total number of HomAut calls is $\lceil \overline{m}k/N \rceil (s_0 - 1) + \lceil \overline{m}k/N \rceil \lceil n/\overline{m} \rceil (s_1 - 1)$ where $s_0 s_1 = \overline{m}$ implicitly gives the split point. The total number of ciphertexts sent between the two parties is $\lceil \overline{m}k/N \rceil + \lceil n/\overline{m} \rceil \lceil \overline{m}k/N \rceil$.

V2. The number of HomMulPt calls is $n \cdot \lceil m\overline{k}/N \rceil$. For HomAut, we note that the HomAut operations for the blocks of $\mathbf{X}$ in the same row could be merged and performed only once. As a result, the number of HomAut operations is $\lceil m\overline{k}/N \rceil (s_0 - 1) + \lceil n\overline{k}/N \rceil (s_1 - 1)$, where $s_0 s_1 = N/\overline{k}$ indicates the split point. The number of ciphertexts sent is $\lceil m\overline{k}/N \rceil + \lceil n\overline{k}/N \rceil$.

Algorithm 3: $\mathsf{RhombusMatMul}_{\mathsf{RM}}$: Row major based matrix multiplication with SPP optimization.

Input: Matrix $\mathbf{X}_{n\times m}$, encrypted matrix $ct \leftarrow [\![\mathbf{Y}_{m\times k}]\!]$ with $n = m, mk = N$, and split-point u.
Output: $ct' \leftarrow [\![\mathbf{X}\cdot\mathbf{Y}]\!]$

1: $ct \leftarrow m^{-1}\cdot ct$ // Remove the PackRLWEs factor.
2: $h \leftarrow \log k$
3: **for** $i \in [n]$ **do**
4: $\quad \widehat{x}_i \leftarrow \mathsf{Ecd}_2(\mathbf{X}_{i:})$ // Encode each row of $\mathbf{X}$ separately.
5: **end**
6: **for** $\tau \in \mathrm{Gal}(K_u/K_h)$ **do**
7: $\quad ct_\tau \leftarrow \mathsf{HomAut}(ct, \tau)$ // SPP: Replacement
8: **end**
9: **for** $i_1 \in [nk/2^u]$ **do**
10: $\quad ct_{i_1} \leftarrow \mathsf{Enc}(0)$
11: $\quad$ **for** $\tau \in \mathrm{Gal}(K_u/K_h)$ **do**
12: $\quad\quad \widehat{z}_{i_1,\tau} \leftarrow \sum_{i_0=0}^{2^{u-h}-1} X^{(N/2^u)\cdot i_0}\cdot\tau(\widehat{x}_{i_0\cdot(nk/2^u)+i_1})$;
13: $\quad\quad ct_{i_1} \leftarrow \mathsf{HomAdd}(ct_{i_1}, \mathsf{HomMulPt}(ct_\tau, \widehat{z}_{i_1,\tau}))$
14: $\quad$ **end**
15: **end**
16: $ct' \leftarrow \mathsf{PackRLWEs}(\{ct_{i_1}\}_{i_1\in[nk/2^u]}, u)$ // SPP: PackRLWEs
17: **return** ct'

Table 3 gives the comparison with prior works for matrix multiplication, where Encode/NTT represents the number of plaintexts of the encoded matrix $\mathbf{X}$. We note that the matrix $\mathbf{X}$ will be encoded to multiple plaintexts, and then these plaintexts will be transformed to the NTT form and are multiplied by the encrypted blocks of $\mathbf{Y}$. The encoding and transformation (to NTT) could be performed only once. The number of HomAut calls in Table 3 are the minimized by choosing the optimal split-point.

4.2 Matrix Encoding for Column-Major Method

Similar to the column-major MVM described in Section 3.3, matrix multiplication can also be realized with the Expand algorithm. Specifically, we note that $\mathbf{X}\cdot\mathbf{Y} = \sum_{i\in[m]}\mathbf{X}_{:i}\cdot\mathbf{Y}_{i:}$. Taking the product of $\mathbf{X}_{:0}$ and $\mathbf{Y}_{0:}$ as an example, this tensor product can be computed as shown in Figure 7 (where $\mathbf{X}, \mathbf{Y}$ are the same as that in Figure 5).

Encode $\mathbf{Y}_{0:}$: $\quad \widehat{y}_0 = 3 + 1X^4$
Encode $\mathbf{X}_{:0}$: $\quad \widehat{x}_0 = 1 + 2X + 3X^2 + 4X^3$

$\Downarrow$ Multiply $\widehat{x}_0$ and $\widehat{y}_0$

$$\widehat{x}_0\cdot\widehat{y}_0 = 3 + 6X + 9X^2 + 12X^3 + 1X^4 + 2X^5 + 3X^6 + 4X^7$$

Figure 7: Tensor of $\mathbf{X}_{:0}$ and $\mathbf{Y}_{0:}$ from HE with $N = 8$.

Note that $\mathbf{Y}$ is encrypted in a packed manner. Hence, our goal is to extract the $\widehat{y}_0, \widehat{y}_1, \widehat{y}_2, \widehat{y}_3$, in the same encoding manner as in Figure 7. Exactly, the extraction can be realized from the Expand algorithm by

setting the parameter $\ell = 2$.

For the general matrix parameters n, m, k, we can follow the same matrix partition methods described in Section 4.1. In this way, the column-major matrix multiplication has the same complexity as the row-major approach in terms of HomAut, HomMulPt and Encode/NTT operations. The only difference lies in the processing of plaintext matrices (similar to MVM described in Section 3.3). In particular, for the dimension (n, m) of the matrix $\mathbf{X}$, the column major approach has the cheaper matrix processing than the row major approach when $n > m$.

5 Performance Evaluation

In this section, we evaluate the performance of *Rhombus*, and compare it with prior arts. This includes the efficiency comparison of MVM, matrix multiplication, pointwise convolution and the end-to-end inference of ResNet50. We adopt the SEAL library [SEA21] of version 3.7 with HEXL acceleration to implement HE-based MVM and matrix multiplication. For the pointwise convolution and ResNet50 inference, our implementation builds upon the open-sourced PPML library OpenCheetah [HjLHD22], and simulates the network environment with EMP-toolkit [WMK16].

Experimental Setup. All our experiments were conducted using Intel(R) Xeon(R) Platinum 8358P CPU at 2.60GHz with 32GB of memory. We run our benchmarks in two network settings: LAN (3Gbps of bandwidth and 0.3ms of round-trip time) and WAN (100Mbps of bandwidth and 40ms of round-trip time). All experiments are run with 4 threads. We use the tc tool on Linux to control the network traffic.

HE parameters. Our protocols select the dimension $N = 8192$ of polynomial ring R, ciphertext modulus $q \approx 2^{100}$, plaintext modulus $t = 2^{37}$ and standard deviation $\sigma = 3.2$ of a discrete Gaussian distribution, which achieves 128-bit security level. For two recent protocols [HLHD22, BK23] to be compared, *Cheetah* chooses the HE parameters as $\mathsf{HE.pp}[\textit{Cheetah}] = \{N = 4096, q \approx 2^{105}, t = 2^{37}, \sigma = 3.2\}$, and *HELiKs* sets the HE parameters as $\mathsf{HE.pp}[\textit{HELiKs}] = \{N = 8192, q \approx 2^{120}, p \approx 2^{37}, \sigma = 3.2\}$, which enables the noise flooding to provide 32-bit statistical security according to the smudging lemma [AJL+12], where p is the plaintext modulus. In Figure 8, we provide the overhead (including the runtime and noise growth) of the basic homomorphic operations, which helps to understand the improvements we have made compared to previous approaches.

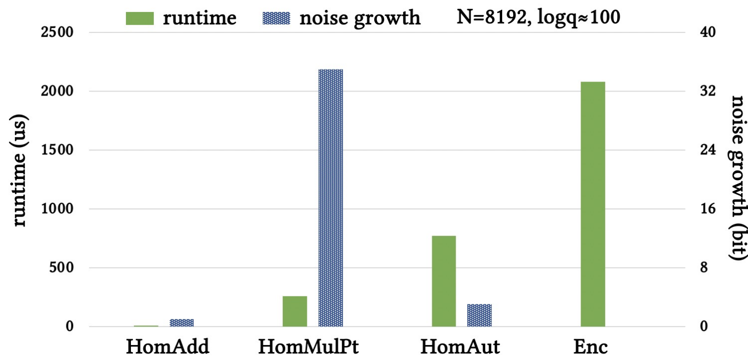

Figure 8: Running time (in μs) and noise growth (in bit) of homomorphic operations for parameters $N = 8192$ and $\log q \approx 100$.

5.1 Matrix Vector Multiplication

In Table 1, we give a theoretical comparison between our MVM protocol *Rhombus* and previous protocols, where HomAut, HomMulPt and Encrypt are three dominated operations. We note that for the NTT encoding based method, such as *HELiKs* [BK23], the multiplicative noise $\widehat{\eta}_{mul}$ introduced by HomMulPt will be greater than (or equal to) η_{mul} of coefficient encoding. This is because the noise growth factor depends only on the value of the coefficients of the plaintext polynomial and not on the size of the plaintext modulus, and in most cases, the NTT encoding will transform a vector even with small elements to a polynomial whose coefficients are spread throughout the entire plaintext space.

We use Algorithm 2 and choose the optimal split-point to minimize the number of rotations (i.e., HomAut) called by *Rhombus*. We focus on analyzing the comparison between *Rhombus* and the previous approaches listed in Table 1. For the approaches with NTT encoding, *HELiKs* has the lower complexity than *GAZELLE* and *HS*, and has a smaller noise growth compared to *GALA* by swapping the order of HomMulPt and Rotation, such that *HELiKs* can choose a smaller ciphertext modulus q than *GALA*. For the coefficient encoded approaches, we note that the $O(r)$ number of HomAut calls in *CHAM* is more costly than the $O(\sqrt{rc/N})$ number of Encrypt operations in *Cheetah* in most cases according to Figure 8, and *CHAM* adopted the FPGA hardware to accelerate their approach to achieve a comparable performance as *Cheetah*. Therefore, in terms of concrete performance (shown below), we only conduct the comparsion with the two state-of-the-art protocols, i.e., *HELiKs* with NTT encoding (over a field $\mathbb{F}_p$) and *Cheetah* with coefficient encoding (over a ring $\mathbb{Z}_t$). Compared to *Cheetah*, *Rhombus* (Algorithm 2) requires additional $O(\sqrt{rc/N})$ rotations. However, *Cheetah* requires additional $O(\sqrt{rc/N})$ calls of Encrypt (and decryption), which is more expensive than rotation, as shown in Figure 8. For communication, *Cheetah* incurs a significant overhead, especially for matrices with large dimensions. While *HELiKs* requires the number of rotations linear in the matrix dimension, *Rhombus* needs only *sublinear* number of rotations. Furthermore, *Rhombus* can drop the unused coefficients of ciphertext polynomials to further reduce the communication cost, which is *not* supported by *HELiKs* with NTT encoding.

Table 1: **Comparison of HE-based MVM protocols between Rhombus and known protocols in the two-party setting**, where $r, c \leq N$ are the number of rows and columns in a matrix respectively, c is also the dimension of a vector, and N is the dimension of polynomial ring R in HE. The computational costs of the basic operations satisfy: Encrypt $\succ$ Rotation $\succ$ HomMulPt, as shown in Figure 8. η is the initial noise in the input ciphertext, η_{mul} (resp., $\widehat{\eta}_{mul}$) is the noise introduced by HomMulPt with coefficient (resp., NTT) encoding, which satisfies $\widehat{\eta}_{mul} \geq \eta_{mul}$. η_{aut} is the noise introduced by HomAut.

Protocol	Rotation (i.e., HomAut)	HomMulPt	Encrypt& Comm.	Encoding	Noise growth
HS [HS14]	$O(\max\{\sqrt{r}, \sqrt{c}\})$	$O(\max\{r, c\})$	$O(1)$	NTT	$(\eta + \eta_{aut}) \cdot \widehat{\eta}_{mul} + \eta_{aut}$
GAZELLE [JVC18]	$O(\frac{rc}{N} + \log(\frac{N}{c}))$	$O(\frac{rc}{N})$	$O(1)$	NTT	$(\eta + \eta_{aut}) \cdot \widehat{\eta}_{mul} + \eta_{aut}$
GALA [ZXW21]	$O(\frac{rc}{N})$	$O(\frac{rc}{N})$	$O(1)$	NTT	$(\eta + \eta_{aut}) \cdot \widehat{\eta}_{mul}$
HELiKs [BK23]	$O(\frac{rc}{N})$	$O(\frac{rc}{N})$	$O(1)$	NTT	$\eta \cdot \widehat{\eta}_{mul} + \eta_{aut}$
CHAM [RCG+]	$O(r)$	$O(\frac{rc}{N})$	$O(1)$	Coefficient	$\eta \cdot \eta_{mul} + \eta_{aut}$
Cheetah [HLHD22]	0	$O(\frac{rc}{N})$	$O(\sqrt{\frac{rc}{N}})$	Coefficient	$\eta \cdot \eta_{mul}$
Rhombus	$O(\sqrt{\frac{rc}{N}})$	$O(\frac{rc}{N})$	$O(1)$	Coefficient	$(\eta + \eta_{aut}) \cdot \eta_{mul} + \eta_{aut}$

For concrete performance (running time and concrete communication cost), we compare *Rhombus* with the state-of-the-art protocols *HELiKs* and *Cheetah*, which is reported in Figure 9. In the WAN setting, we observe that *Rhombus* achieves a speedup of up to $8\times$ over *HELiKs* and $3.4\times$ over *Cheetah*. In the LAN setting, we find that *Rhombus* improves the performance of *Cheetah* by $1.8\times$ and that of *HELiKs* by $7.4\times$. In terms of communication cost, *Rhombus* reduces the communication by a factor of $21\times$ compared to

Cheetah, and achieves the improvement of $1.8\times$ over *HELiKs*.

In Table 2, we present the comparison between row major approach and column major approach on large matrices of different dimensions. As analyzed in 3.3, we choose the optimal split point u to minimize the number of HomAut calls, such that the only differences between the two approaches lies in the matrix processing. In particular, when $r < c$, the row major approach has a cheaper matrix processing than the column major approach, and thus performs better than the column major approach, conversely, the column major approach outperforms the row major approach when $r > c$. The time difference between the two approaches come from the different complexity of the matrix processing. And the two approaches perform comparably when $r = c$.

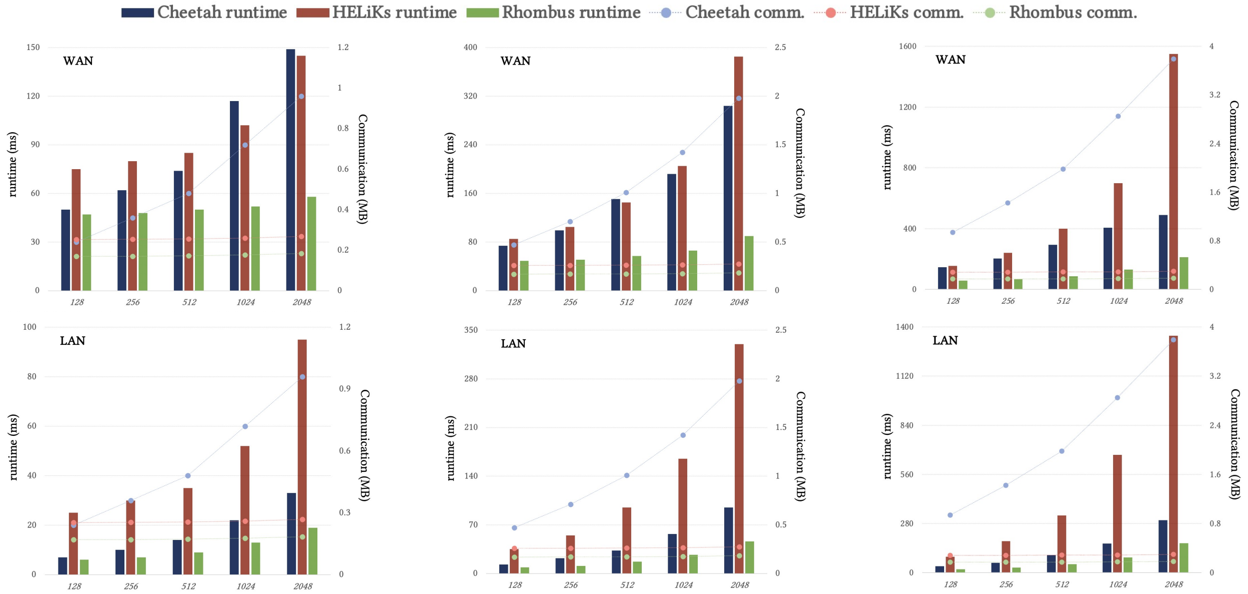

Figure 9: Comparison of the end-to-end performance of secure two-party MVM protocols for matrices with different dimensions in the LAN and WAN settings. Comparison of communication efficiency is also involved in this figure. For six sub-figures, the number of columns in a matrix are 256 (two sub-figures on the left), 1024 (two sub-figures in the middle) and 4096 (two sub-figures on the right) respectively, and these numbers are also the dimensions of vectors. The number of rows in a matrix is shown in the x-coordinate of these sub-figures, i.e, 128, 256, 512, 1024 and 2048.

Table 2: **Comparison between row major and column major approaches for large matrices of different dimensions**, the performance data includes only the time (measured in seconds) for the computation of MVM, excluding the time for network communication because the two approaches have the same communication overhead.

Approach	Dimension ($r \times c$)					
	$2^{13} \times 2^{14}$	$2^{14} \times 2^{13}$	$2^{14} \times 2^{14}$	$2^{13} \times 2^{15}$	$2^{15} \times 2^{13}$	$2^{15} \times 2^{15}$
row major, 3.2	1.32	1.68	2.66	2.56	3.86	10.8
column major, 3.3	1.68	1.33	2.67	3.88	2.55	10.8

Table 3: **Complexity comparison of HE-based two-party matrix-multiplication protocols among Rhombus, Iron, BumbleBee and BOLT.** Two matrices have dimensions $n \times m$ and $m \times k$, respectively. N denotes the polynomial-ring dimension in HE. *Rhombus-V1* (resp., *Rhombus-V2*) adopts the matrix partition method V1 (resp., V2) shown in Section 4.1. η is the initial noise of the input ciphertexts.

	HomAut	HomMulPt	Encode/NTT	Communication	Encoding	Noise growth
Iron [HLC+22]	0	$O(\frac{nmk}{N})$	$O(\frac{nmk}{N})$	$O(\sqrt{\frac{nmk}{N}})$	Coefficient	$\eta \cdot \eta_{mul}$
BumbleBee [LHG+23]	$O(\frac{kn \log N}{\sqrt{N}})$	$O(\frac{nmk}{N})$	$O(\frac{nm}{\sqrt{N}})$	$O(\frac{km}{N} + \frac{kn}{\sqrt{N}})$ †	Coefficient	$\eta \cdot \eta_{mul} + \eta_{aut}$
BOLT [PZM+23]	$O(\frac{mk\sqrt{n}}{N})$	$O(\frac{nmk}{N})$	$O(\frac{nmk}{N})$	$O(\frac{k(n+m)}{N})$	NTT	$(\eta + \eta_{aut}) \cdot \widehat{\eta}_{mul} + \eta_{aut}$
Rhombus-V1	$O(\frac{mk\sqrt{n}}{N})$	$O(\frac{nmk}{N})$	$O(n)$	$O(\frac{k(n+m)}{N})$	Coefficient	$(\eta + \eta_{aut}) \cdot \eta_{mul} + \eta_{aut}$
Rhombus-V2	$O(\sqrt{\frac{nmk}{N}})$	$O(\frac{nmk}{N})$	$O(\frac{nmk}{N})$	$O(\frac{k(n+m)}{N})$	Coefficient	$(\eta + \eta_{aut}) \cdot \eta_{mul} + \eta_{aut}$

† The complexity is derived from the partition window $n_w = 1, m_w = \sqrt{N}, k = \sqrt{N}$ given in *BumbleBee*.

5.2 Matrix Multiplication

Firstly, we show a theoretical comparison between the secure matrix-multiplication protocol *Rhombus* (described in Section 4) and the most efficient protocols in the two-party setting, which is shown in Table 3. Compared to *Iron*, our protocol *Rhombus* takes about $O(\sqrt{N/k})$ times less communication, where $k \ll N$ for the parameters used in real-world applications. As a trade-off, *Rhombus* requires additional rotations of either $O(\frac{mk\sqrt{n}}{N})$ or $O(\sqrt{\frac{nmk}{N}})$, which is *sublinear* in the dimension (either n or m) of a matrix. *BumbleBee* [LHG+23] proposed the IntrLeave algorithm to realize the same functionality as our PackRLWEs, but required $O(\min\{\ell \cdot 2^\ell, N\})$ rotation calls when packing 2^ℓ ciphertexts, with each having $N/2^\ell$ coefficients to be merged, where PackRLWEs only requires $O(2^\ell)$ rotations and $\ell \leq \log N$. While *BumbleBee* requires the almost linear complexity for the number of rotations, *Rhombus* adopts the SPP optimization, which further reduces the number of rotations to sublinear complexity. Therefore, *Rhombus* has a significantly lower computation cost than *BumbleBee*. Protocol *BOLT* [PZM+23] adopts NTT encoding, and requires a noise growth larger than *Rhombus*, where $\widehat{\eta}_{mul} \geq \eta_{mul}$. For the parameters N, n, m, k used in real-world applications, *BOLT* requires significantly more either encoding operations than *Rhombus-V1* or rotation operations than *Rhombus-V2*. Therefore, *Rhombus* has the lower computation cost than *BOLT*. Furthermore, *BOLT* with NTT encoding not only incurs more complex computation for encoding matrices, but also requires additional overhead when converting the computational results to additive secret sharings over a ring (used by *BOLT* to compute non-linear layers).

For the end-to-end performance of two-party matrix multiplication, we compare *Rhombus* with the recent protocols *Iron* [HLC+22] and *BumbleBee* [LHG+23], reported in Table 4. The HE parameters for *Iron* and *BumbleBee* are set as $\mathsf{HE.pp}[\textit{Iron}] = \{N = 4096, q \approx 2^{92}, t = 2^{37}, \sigma = 3.2\}$ and $\mathsf{HE.pp}[\textit{BumbleBee}] = \{N = 8192, q \approx 2^{112}, t = 2^{37}, \sigma = 3.2\}$, respectively. Note that the code of *BOLT* is not publicly available for now, and thus we only conduct a theoretical comparison with *BOLT*. For different matrix dimensions, we choose a suitable matrix partition method (V1 or V2) described in Section 4.1 to achieve a better performance. Compared to *Iron*, *Rhombus* achieves a speedup by $2.7\times \sim 3.9\times$ (resp., $1.3\times \sim 2.3\times$) in the WAN (resp., LAN) setting, and reduces the communication cost by a factor of $4.6\times \sim 8\times$. Compared to *BumbleBee*, *Rhombus* demonstrates a performance improvement by a factor of $1.9\times \sim 3.9\times$ (resp. $2.5\times \sim 4.5\times$) in the WAN (resp., LAN) setting, and achieves a communication reduction by around $1.1\times$.

The comparison with BumbleBee in this work was conducted using this version: https://eprint.iacr.org/archive/2023/1678/1698733772.pdf

Table 4: Comparison of the end-to-end performance and communication efficiency between Rhombus with Iron for secure matrix multiplication. The dimensions of matrices are chosen as same as that in the BERT-base model [DCLT19] with input of 128 tokens. We refer the reader to [HLC+22, PZM+23] for more details.

Dimension (n, m, k)	Protocol	Running Time (s)		Comm. (MB)
		LAN	WAN	
$(128, 128, 64)$	*Iron* [HLC+22]	0.08	0.35	1.75
	BumbleBee [LHG+23]	0.16	0.23	0.24
	Rhombus-V1	0.06	0.12	0.22
$(768, 768, 128)$	*Iron* [HLC+22]	1.27	2.96	15.6
	BumbleBee [LHG+23]	2.39	2.45	2.93
	Rhombus-V2	0.63	0.75	2.7
$(3072, 768, 128)$	*Iron* [HLC+22]	4.86	7.67	31.3
	BumbleBee [LHG+23]	9.4	10	7.35
	Rhombus-V2	2.07	2.55	6.75
$(768, 3072, 128)$	*Iron* [HLC+22]	5.26	8.52	31.3
	BumbleBee [LHG+23]	7.2	7.4	7.3
	Rhombus-V2	2.88	3.12	6.75

5.3 Pointwise Convolutions

3D convolution is a common linear operation used in DNNs. In most cases, the convolutional kernels/filters are quite small, such as 3×3 and 1×1. In particular, pointwise convolution (with filter size 1×1) is commonly used to combine feature channels, reduce dimensionality and decrease the model's parameters. For example, in SqueezeNet [IHM+16] and DenseNet [HLvdMW18], half of the convolutions are pointwise; in ResNet50, ResNet101 and ResNet152 [HZRS15], there are about two-thirds of the convolutions are pointwise. Although *Cheetah* has proposed a generic method for convolution computation, it performs poorly for pointwise convolutions. Specifically, for the inference of ResNet50, the overhead for pointwise convolutions already accounts for 80% of all convolution layers.

As an application of our matrix-multiplication protocol *Rhombus*, we implemented the pointwise convolution and evaluate its performance. In particular, pointwise convolution of a (c_i, h, w) input image with $(c_o, f_h = 1, f_w = 1)$ filter of stride s could be transformed to the matrix multiplication with parameters $(hw/s^2, c_i, c_o)$ [BK23] without any additional costs. We report in Figure 10 the performance comparison under the WAN setting among *Rhombus*, *Cheetah* [HLHD22] and *HELiKs* [BK23] for pointwise convolutions in ResNet50. From this figure, we observe that *HELiKs* (resp., *Cheetah*) takes 338 MB (resp., 842 MB) of total communication for all pointwise convolutions, and *Rhombus* needs the total communication of 182 MB, which achieves the improvement by $1.9\times$ (resp. $4.6\times$), compared to *HELiKs* (resp., *Cheetah*). For running time of computing all pointwise convolutions, while *HELiKs* (resp., *Cheetah*) takes 56 seconds (resp., 101 seconds), *Rhombus* runs in about 30 seconds, which gives a speedup of $1.9\times$ (resp., $3.4\times$). We also measured the performance of pointwise convolutions in ResNet50 under the LAN setting for these protocols, which is reported in Figure 13 in Appendix. In particular, while *HELiKs* (resp., *Cheetah*) takes 35 seconds (resp. 30 seconds) for all pointwise convolutions, *Rhombus* needs 15 seconds, which achieves the improvement by $2.3\times$ (resp., $2\times$).

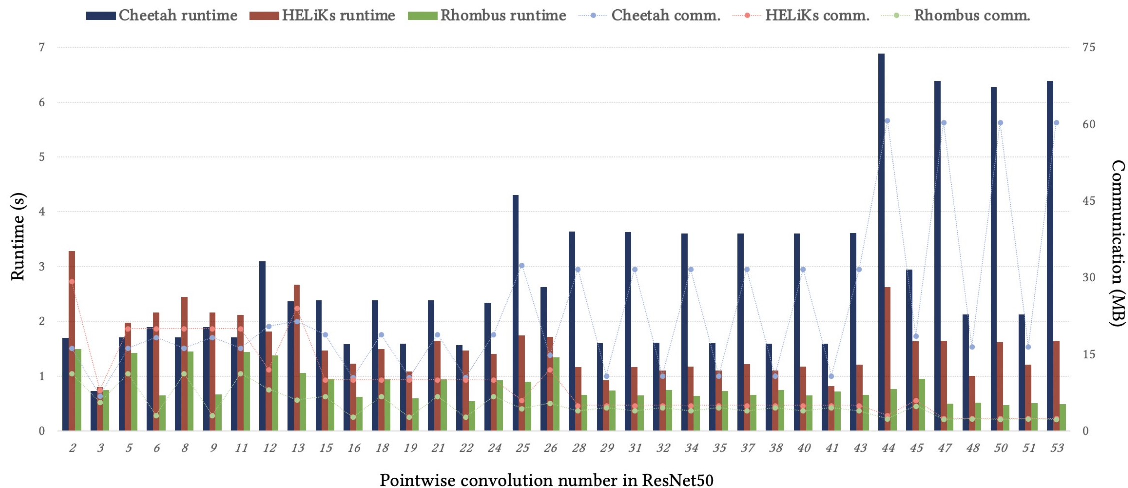

Figure 10: **Comparison of running time and communication among Rhombus, Cheetah and HELiKs for pointwise convolutions in ResNet50 under the WAN setting.** There are 53 convolutions in ResNet50 indexed by $1 \sim 53$, where the x-coordinate in this figure represents the index of pointwise convolutions among them.

5.4 End-to-End PPML Inference

We integrate *Rhombus* into the OpenCheetah library to demonstrate the efficiency improvement of end-to-end PPML inference. We use the popular DNN model ResNet50 as an example just like as in *HELiKs* [BK23]. In Figure 11, we illustrate the comprehensive comparison of running time between *Rhombus* and the state-of-the-art protocols *Cheetah* [HLHD22] and *HELiKs* [BK23] for the end-to-end inference of ResNet50. In this figure, we divide the convolutions into two parts: pointwise convolutions (pw-conv) and non-pointwise convolutions (npw-conv). Figure 12 reports the comparison of communication cost among these protocols for the end-to-end inference of ResNet50. The experiments are run under two frameworks: (1) one is *CryptFlow2* [RRK+20] (CF2 in short) where all sub-protocols builds over a field $\mathbb{F}_p$ with $p \approx 2^{37}$; (2) the other is *Cheetah* [HLHD22], in which all sub-protocols work over a ring $\mathbb{Z}_{2^{37}}$. Below, we give the performance analysis of ResNet50 inference.

Performance of HELiKs. The recent work [BK23] integrated *HELiKs* into the CF2 framework [RRK+20] by replacing the original matrix-multiplication and 3D-convolution protocols with their new protocols, which is called *HELiKs+CF2*. Additionally, *HELiKs* performed the encoding of convolutional filters and NTT transformation in the preprocessing phase to improve the performance of the online phase. In our benchmark, we focus on the end-to-end running time and total communication without dividing the protocol into the preprocessing and online phases. We note that both *Cheetah* and *Rhombus* can also benefit from executing a similar preprocessing. *HELiKs+CF2* works a field $\mathbb{F}_p$, which leads to be less efficient than the best-known protocol over a ring $\mathbb{Z}_t$ for secure computation of non-linear layers. Besides, *HELiKs+CF2* uses the IKNP OT extension [IKNP03] in the implementation, which incurs a large communication. The experimental result shows that *HELiKs+CF2* executes one inference in 42 minutes (resp., 195 seconds) in the WAN (resp., LAN) setting and the communication cost of 28.8 GB.

Performance of Cheetah. *Cheetah* [HLHD22] builds upon a ring $\mathbb{Z}_{2^{37}}$. For non-linear layers, *Cheetah* employs Ferret OT extension [YWL+20] with sublinear communication and uses it to realize the approximate truncation, which significantly reduces the communication cost of non-linear operators. For linear layers, *Cheetah* adopted the coefficient encoding to compute 3D convolution and matrix multiplication. However,

Cheetah does not fully utilize each coefficient of the polynomial, resulting in significant waste in both communication and computation. The experimental result shows that *Cheetah* completes one inference within 213 seconds in the WAN setting and 58 seconds in the LAN setting. Figure 11 shows the proportion of running time of each module in *Cheetah*, among which 50% of running time takes at the pointwise convolutions. The total communication of *Cheetah* is 1.77 GB, while 46% of communication is used for computing pointwise convolutions.

Performance of Rhombus+Cheetah. We integrate *Rhombus* into the *Cheetah* framework by replacing the original pointwise convolution and MVM protocols with our protocols. Using our protocol *Rhombus*, the end-to-end running time for ResNet50 inference is improved to 141 seconds (resp., 42 seconds) in the WAN (resp., LAN) setting, and the total communication is reduced to 1.13 GB. In terms of the end-to-end performance, we achieve the improvement of $18\times$ (resp., $1.5\times$) in the WAN setting and the improvement of $4.6\times$ (resp., $1.4\times$) in the LAN setting, compared to *HELiKs* (resp., *Cheetah*). In terms of communication efficiency, *Rhombus* achieves the improvements by $26\times$ over *HELiKs* and $1.6\times$ over *Cheetah*, as shown in Figure 12. Due to the performance advantage of computing pointwise convolutions as shown in Section 5.3, our solution *Rhombus* will also achieve better efficiency for secure two-party inference of DNN models SqueezeNet, DenseNet, ResNet101 and ResNet152, compared to the state-of-the-art approaches *HELiKs* and *Cheetah*, where these DNNs have at least a half of convolutions are pointwise convolutions.

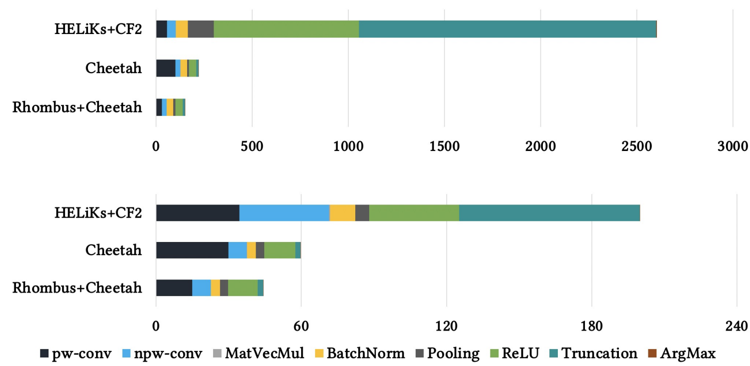

Figure 11: Comparison of running time (in second) among Rhombus, HELiKs and Cheetah for the end-to-end inference of ResNet50. The above (resp., below) image is the running time in the WAN (resp., LAN) setting.

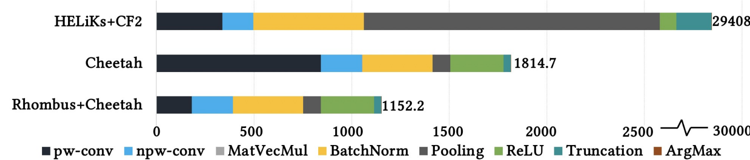

Figure 12: Comparison of the total communication (in MB) among secure two-party protocols for ResNet50 inference.

Acknowledgments

Work of Kang Yang is supported by the National Key Research and Development Program of China (Grant No. 2022YFB2702000), and by the National Natural Science Foundation of China (Grant Nos. 62102037, 61932019).

References

[ACLS18] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. IEEE Computer Society, 2018.

[AJL+12] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, volume 7237 of *Lecture Notes in Computer Science*, pages 483–501. Springer, 2012.

[ALP+21] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication-computation trade-offs in PIR. USENIX Association, 2021.

[ASKG19] Nitin Agrawal, Ali Shahin Shamsabadi, Matt J. Kusner, and Adrià Gascón. QUOTIENT: Two-party secure neural network training and prediction. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1231–1247, London, UK, November 11–15, 2019. ACM Press.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory*, 2014.

[BK23] Shashank Balla and Farinaz Koushanfar. Heliks: HE linear algebra kernels for secure inference. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 2306–2320. ACM, 2023.

[BMR+20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 868–886, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

[CDKS21] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient homomorphic conversion between (ring) LWE ciphertexts. Springer, 2021.

[CGR+19] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. EzPC: Programmable and efficient secure two-party computation for machine learning. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 496–511, 2019.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. Springer, 2017.

[CZ22] Geoffroy Couteau and Maryam Zarezadeh. Non-interactive secure computation of inner-product from LPN and LWE. Springer, 2022.

[CZW+21] Chaochao Chen, Jun Zhou, Li Wang, Xibin Wu, Wenjing Fang, Jin Tan, Lei Wang, Alex X. Liu, Hao Wang, and Cheng Hong. When homomorphic encryption marries secret sharing: Secure large-scale sparse logistic regression and applications in risk control. ACM, 2021.

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[DEK19] Anders Dalskov, Daniel Escudero, and Marcel Keller. Secure evaluation of quantized neural networks. Cryptology ePrint Archive, Report 2019/131, 2019. https://eprint.iacr.org/2019/131.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, page 144, 2012.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 169–178, Bethesda, MD, USA, May 31 – June 2, 2009. ACM Press.

[GJM+23] Kanav Gupta, Neha Jawalkar, Ananta Mukherjee, Nishanth Chandran, Divya Gupta, Ashish Panwar, and Rahul Sharma. SIGMA: secure GPT inference with function secret sharing. *IACR Cryptol. ePrint Arch.*, page 1269, 2023.

[HjLHD22] Zhicong Huang, Wen jie Lu, Cheng Hong, and Jiansheng Ding. Opencheetah: Lean and fast secure Two-Party deep neural network inference. Open-sourced libary is available at https://github.com/Alibaba-Gemini-Lab/OpenCheetah, 2022.

[HJSK21] Siam Umar Hussain, Mojan Javaheripi, Mohammad Samragh, and Farinaz Koushanfar. COINN: Crypto/ML codesign for oblivious inference via neural networks. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 3266–3281, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

[HLC+22] Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, Guowen Xu, and Tianwei Zhang. Iron: Private inference on transformers. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

[HLHD22] Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding. Cheetah: Lean and fast secure two-party deep neural network inference. USENIX Association, 2022.

[HLL+23] Xiaoyang Hou, Jian Liu, Jingyu Li, Yuhan Li, Wen jie Lu, Cheng Hong, and Kui Ren. CipherGPT: Secure two-party gpt inference. Cryptology ePrint Archive, Paper 2023/1147, 2023. https://eprint.iacr.org/2023/1147.

[HLvdMW18] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks, 2018.

[HS14] Shai Halevi and Victor Shoup. Algorithms in helib. Springer, 2014.

[HS20] Shai Halevi and Victor Shoup. Design and implementation of helib: a homomorphic encryption library. *IACR Cryptol. ePrint Arch.*, page 1481, 2020.

[HZ23] Hai Huang and Haoran Zong. Secure matrix multiplication based on fully homomorphic encryption. *J. Supercomput.*, 79(5):5064–5085, 2023.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[IHM+16] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size, 2016.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2003.

[JKLS] Xiaoqian Jiang, Miran Kim, Kristin E. Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018.*

[JVC18] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha P. Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. USENIX Association, 2018.

[KKK+22] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. BTS: an accelerator for bootstrappable fully homomorphic encryption. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 711–725. ACM, 2022.

[KPZ21] Andrey Kim, Yuriy Polyakov, and Vincent Zucca. Revisiting homomorphic encryption schemes for finite fields. Springer, 2021.

[KRC+] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow: Secure tensorflow inference. In *2020 IEEE Symposium on Security and Privacy, SP 2020.*

[LHG+23] Wen-jie Lu, Zhicong Huang, Zhen Gu, Jingyu Li, Jian Liu, Kui Ren, Cheng Hong, Tao Wei, and Wenguang Chen. Bumblebee: Secure two-party inference framework for large transformers. *IACR Cryptol. ePrint Arch.*, page 1678, 2023.

[LJLA17] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. ACM, 2017.

[LKL+22] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, and Jong-Seon No. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *IEEE Access*, 10:30039–30054, 2022.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. Springer, 2010.

[MLS+20] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. USENIX Association, 2020.

[MZ17] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 19–38. IEEE Computer Society, 2017.

[NC21] Lucien K. L. Ng and Sherman S. M. Chow. GForce: GPU-friendly oblivious and rapid neural network inference. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 2147–2164. USENIX Association, August 11–13, 2021.

[PSSY21] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: improved mixed-protocol secure two-party computation. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2165–2182. USENIX Association, 2021.

[PZM+23] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. BOLT: privacy-preserving, accurate and efficient inference for transformers. *IACR Cryptol. ePrint Arch.*, page 1893, 2023.

[RBG+23] Deevashwer Rathee, Anwesh Bhattacharya, Divya Gupta, Rahul Sharma, and Dawn Song. Secure floating-point training. In *32nd USENIX Security Symposium (USENIX Security 2023)*, pages 6329–6346. USENIX Association, 2023.

[RBS+22] Deevashwer Rathee, Anwesh Bhattacharya, Rahul Sharma, Divya Gupta, Nishanth Chandran, and Aseem Rastogi. SecFloat: Accurate floating-point meets secure 2-party computation. In *2022 IEEE Symposium on Security and Privacy*, pages 576–595, San Francisco, CA, USA, May 22–26, 2022. IEEE Computer Society Press.

[RCG+] Xuanle Ren, Zhaohui Chen, Zhen Gu, Yanheng Lu, Ruiguang Zhong, Wen-Jie Lu, Jiansong Zhang, Yichi Zhang, Hanghang Wu, Xiaofu Zheng, Heng Liu, Tingqiang Chu, Cheng Hong, Changzheng Wei, Dimin Niu, and Yuan Xie. CHAM: A customized homomorphic encryption accelerator for fast matrix-vector product. In *60th ACM/IEEE Design Automation Conference, DAC 2023*.

[RRG+21] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. Sirnn: A math library for secure RNN inference. IEEE, 2021.

[RRK+20] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow2: Practical 2-party secure inference. ACM, 2020.

[RSC+19] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin E. Lauter, and Farinaz Koushanfar. XONN: XNOR-based oblivious deep neural network inference. In Nadia Heninger and Patrick Traynor, editors, *USENIX Security 2019*, pages 1501–1518, Santa Clara, CA, USA, August 14–16, 2019. USENIX Association.

[SDF+22] Liyan Shen, Ye Dong, Binxing Fang, Jinqiao Shi, Xuebin Wang, Shengli Pan, and Ruisheng Shi. Abnn2: Secure two-party arbitrary-bitwidth quantized neural network predictions. Association for Computing Machinery, 2022.

[SEA21] Microsoft SEAL (release 3.7). https://github.com/Microsoft/SEAL, September 2021. Microsoft Research, Redmond, WA.

[SFK+22] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sánchez. Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 173–187. ACM, 2022.

[SGRP19] Phillipp Schoppmann, Adrià Gascón, Mariana Raykova, and Benny Pinkas. Make some ROOM for the zeros: Data sparsity in secure distributed machine learning. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1335–1350, London, UK, November 11–15, 2019. ACM Press.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[YWL+20] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1607–1626. ACM, 2020.

[ZLY+24] Jiawen Zhang, Jian Liu, Xinpeng Yang, Yinghao Wang, Kejia Chen, Xiaoyang Hou, Kui Ren, and Xiaohu Yang. Secure transformer inference made non-interactive. *IACR Cryptol. ePrint Arch.*, page 136, 2024.

[ZXW21] Qiao Zhang, Chunsheng Xin, and Hongyi Wu. GALA: greedy computation for linear algebra in privacy-preserved neural networks. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

A Proof of Proposition 2

We first recall the properties of *field trace*, and then give the proof of proposition 2 based on it. Assuming that $0 \leq s \leq t \leq \log N$, the *field trace* is the map $\mathrm{Tr}_{K_t/K_s} : K_t \to K_s, \widehat{a} \mapsto \sum_{\tau \in \mathrm{Gal}(K_t/K_s)} \tau(\widehat{a})$, which is an additive homomorphism from K_t to K_s. The trace $\mathrm{Tr}_{K_t/K_s}(\widehat{a})$ could be decomposed as $\mathrm{Tr}_{K_{s+1}/K_s} \circ \cdots \circ \mathrm{Tr}_{K_t/K_{t-1}}(\widehat{a})$, which can zeroize the coefficients $\widehat{a}[i]$ if $\frac{N}{2^t} \mid i$ and $\frac{N}{2^s} \nmid i$. It has the following desired properties for any $\widehat{a}, \widehat{b} \in K_t, \widehat{c} \in K_s$:

1. $\mathrm{Tr}_{K_t/K_s}(\widehat{a} + \widehat{b}) = \mathrm{Tr}_{K_t/K_s}(\widehat{a}) + \mathrm{Tr}_{K_t/K_s}(\widehat{b})$

2. $\mathrm{Tr}_{K_t/K_s}(\widehat{c} \cdot \widehat{a}) = \widehat{c} \cdot \mathrm{Tr}_{K_t/K_s}(\widehat{a})$

3. $\mathrm{Tr}_{K_t/K_s}(\widehat{c}) = 2^{t-s} \cdot \widehat{c}$

We next give the formal proofs of proposition 2 based on the field trace described above. At first, we note that the $\mathsf{PackRLWEs}(\{ct_i\}_{i\in[2^\ell]}, h)$ and $\mathsf{Expand}(ct, \ell)$ could be expressed as follows:

$$\mathsf{PackRLWEs}(\{ct_i\}_{i\in[2^\ell]}, h) \rightarrow \sum_{i=0}^{2^\ell-1} X^{(N/2^{\ell+h})\cdot i} \cdot \mathrm{Tr}_{K_{\ell+h}/K_h}(ct_i)$$

$$\mathsf{Expand}(ct, \ell) \rightarrow \{\mathrm{Tr}_{K_{\log N}/K_{\log N-\ell}}(X^{-i} \cdot ct)\}_{i\in[2^\ell]}$$

Proof of proposition 2. We claim that Algorithm 2 outputs a ciphertext which could be expressed as

$$ct' \leftarrow \mathsf{PackRLWEs}(\{\widehat{y}_i \cdot ct\}_{i\in[m]}, h\})$$

Let $\ell = \log m$, then it could be further expressed as

$$ct' \leftarrow \sum_{i=0}^{2^\ell-1} X^{(N/2^{\ell+h})\cdot i} \cdot \mathrm{Tr}_{K_{\ell+h}/K_h}(\widehat{y}_i \cdot ct)$$

Assume K_u is the middle field between K_h and $K_{\ell+h}$, then

$$ct' = \sum_{i=0}^{2^\ell-1} X^{(N/2^{\ell+h})\cdot i} \cdot \mathrm{Tr}_{K_{\ell+h}/K_u}(\mathrm{Tr}_{K_u/K_h}(\widehat{y}_i \cdot ct))$$

due to $\mathrm{Tr}_{K_{\ell+h}/K_h} = \mathrm{Tr}_{K_u/K_h} \circ \mathrm{Tr}_{K_{\ell+h}/K_u} = \mathrm{Tr}_{K_{\ell+h}/K_u} \circ \mathrm{Tr}_{K_u/K_h}$ (Note that we are abusing notation here, as strictly speaking, Tr_{K_u/K_h} is only defined for the input from K_u, however, we can easily extend it to the large filed K, as decribed in Section 2.3). Let $i = i_0 \cdot 2^{\ell+h-u} + i_1$, then the monomial $X^{N/2^{\ell+h}\cdot i}$ can be rewritten as $X^{(N/2^u)\cdot i_0} \cdot X^{(N/2^{\ell+h})\cdot i_1}$, and the first part can be moved into $\mathrm{Tr}_{K_{\ell+h}/K_u}$ according to property 2. Specifically, we have

$$ct' = \sum_{i_1=0}^{2^{\ell+h-u}-1} X^{(N/2^{\ell+h})\cdot i_1} \cdot \mathrm{Tr}_{K_{\ell+h}/K_u}(ct_{i_1}) \tag{3}$$

where $ct_{i_1} = \sum_{i_0=0}^{2^{u-h}-1} X^{(N/2^u)\cdot i_0} \cdot \mathrm{Tr}_{K_u/K_h}(\widehat{y}_{i_0\cdot 2^{\ell+h-u}+i_1} \cdot ct)$. And the ct_{i_1} could be further written as

$$ct_{i_1} = \sum_{\tau\in\mathrm{Gal}(K_u/K_h)} \left(\sum_{i_0=0}^{2^{u-h}-1} X^{(N/2^u)\cdot i_0} \cdot \tau(\widehat{y}_{i_0\cdot 2^{\ell+h-u}+i_1}) \right) \cdot \tau(ct)$$

The expression in parentheses for some τ is just the $\widehat{z}_{i_1,\tau}$ in step 15 of Algorithm 2 (Note that $2^{\ell+h-u} = r/2^u$).

Finally, it's easy to find that the equation 3 happens to be the $\mathsf{PackRLWEs}(\{ct_{i_1}\}_{i_1\in[r/2^u]}, u)$ which corresponds to the step 19. This completes the proof. □

Algorithm 4: $\mathsf{RhombusMVM_{CM}}$: Column-major based approach with SPP optimization.

Input: $r \times c$ matrix $\mathbf{W}$ with $r, c \le N$ (powers of two), a ciphertext $ct \leftarrow [\![\mathbf{v}]\!]$, and the split-point u.

Output: A ciphertext $ct' \leftarrow [\![\mathbf{W}\mathbf{v}]\!]$.

1: **for** $i \in [c]$ **do**
2: $\quad \widehat{w}_i \leftarrow \mathsf{Ecd}_1(\mathbf{W}_{:i})$
3: **end**
4: $ct \leftarrow c^{-1} \cdot ct$ // To remove the Expand factor
5: $\{ct_{i_1}\}_{i_1 \in [N/2^u]} \leftarrow \mathsf{Expand}(ct, \log N - u)$
6: $ct' \leftarrow \mathsf{Enc}(0)$
7: **for** $\tau \in \mathrm{Gal}(K_u / K_{\log N - \log c})$ **do**
8: $\quad ct_{acc} \leftarrow \mathsf{Enc}(0)$
9: $\quad$ **for** $i_1 \in [N/2^u]$ **do**
10: $\quad\quad \widehat{y}_{i_1,\tau} \leftarrow \sum_{i_0=0}^{2^u c/N-1} \tau^{-1}(\widehat{w}_{i_0(N/2^u)+i_1}) \cdot X^{-i_0(N/2^u)}$
11: $\quad\quad ct_{acc} \leftarrow \mathsf{HomAdd}(ct_{acc}, \mathsf{HomMulPt}(ct_{i_1}, \widehat{y}_{i_1,\tau}))$
12: $\quad$ **end**
13: $\quad ct' \leftarrow \mathsf{HomAdd}(ct', \mathsf{HomAut}(ct_{acc}, \tau))$
14: **end**
15: **return** ct'

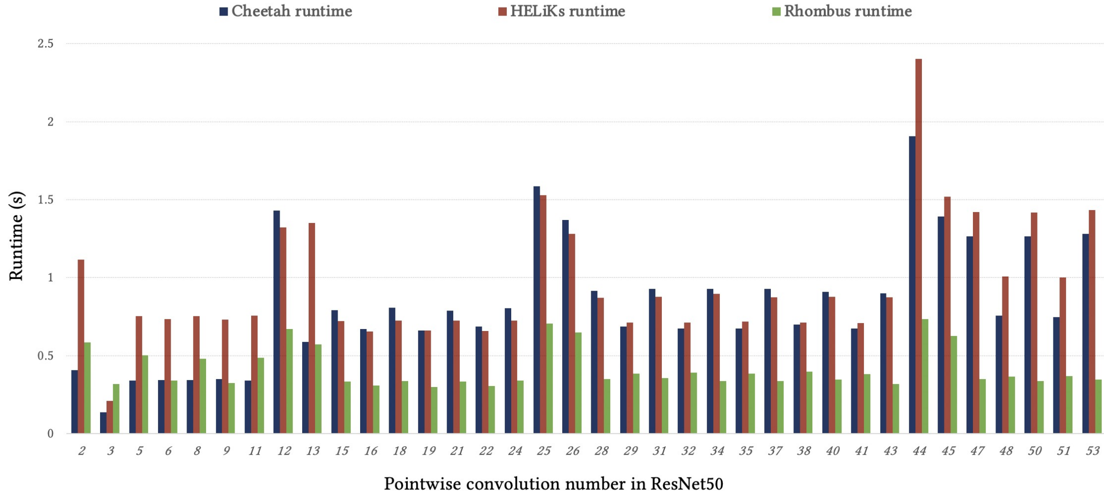

Figure 13: **Performance comparison among Rhombus, Cheetah and HELiKs for pointwise convolutions in ResNet50 under the LAN setting.** There are 53 convolutions in ResNet50 indexed by $1 \sim 53$, where the x-coordinate in this figure represents the index of pointwise convolutions among them.

B Column major based MVM

The column major based MVM with SPP optimization is described in Algorithm 4. The following proposition 3 shows its correctness.

Proposition 3. *Given a matrix* $\mathbf{W}$ *with dimensions* $r, c \le N$ *and an encrypted vector* $[\![\mathbf{v}]\!]$ *(encoded by* Ecd_1*), Algorithm 4 outputs a ciphertext* ct'*, which decrypts to a polynomial* $\widehat{a}'$ *with* $\widehat{a}'[i] = (\mathbf{W} \cdot \mathbf{v})[i]$ *for* $i \in [r]$.

Proof. Similar to the proof of proposition 2, we prove that Algorithm 4 could be expressed as

$$ct' \leftarrow \sum_{i=0}^{c-1} \widehat{w}_i \cdot \mathsf{Expand}(ct, \ell)[i]$$

where $\ell = \log c$. It could be also written as

$$ct' \leftarrow \sum_{i=0}^{c-1} \widehat{w}_i \cdot \mathrm{Tr}_{K_{\log N}/K_{\log N-\ell}}(X^{-i} \cdot ct).$$

Choosing u as the split point with $\log N - \ell \le u \le \log N$, the equation described as above could be further written as

$$ct' = \sum_{i=0}^{c-1} \widehat{w}_i \cdot \mathrm{Tr}_{K_u/K_{\log N-\ell}} \left(\mathrm{Tr}_{K_{\log N}/K_u}(X^{-i} \cdot ct)\right).$$

Next, we decompose $i = i_0 \cdot (N/2^u) + i_1$, and then we have

$$ct' = \sum_{i_0=0}^{c2^u/N-1} \sum_{i_1=0}^{N/2^u-1} \widehat{w}_{i_0(N/2^u)+i_1} \cdot \mathrm{Tr}_{K_u/K_{\log N-\ell}}(X^{-i_0(N/2^u)} ct_{i_1})$$

where $ct_{i_1} = \mathrm{Tr}_{K_{\log N}/K_u}(X^{-i_1} \cdot ct)$. The $ct_{i_1}, i_1 \in [N/2^u]$ could be computed by $\mathsf{Expand}(ct, \log N - u)$. Expanding the trace $\mathrm{Tr}_{K_u/K_{\log N-\ell}}$ and rearranging the expression, we have the following result:

$$ct' = \sum_{\tau \in \mathrm{Gal}(K_u/K_{\log N-\ell})} \tau \left(\sum_{i_1=0}^{N/2^u-1} \widehat{y}_{i_1,\tau} \cdot ct_{i_1} \right)$$

where $\widehat{y}_{i_1,\tau} = \sum_{i_0=0}^{c2^u/N-1} \tau^{-1}(\widehat{w}_{i_0(N/2^u)+i_1}) \cdot X^{-i_0 \cdot (N/2^u)}$ corresponds to the step 10 in Algorithm 4. This completes the proof. □