SIMC 2.0: Improved Secure ML Inference Against Malicious Clients

Guowen Xu, *Member, IEEE*, Xingshuo Han, Tianwei Zhang, *Member, IEEE*, Shengmin Xu, Jianting Ning, *Member, IEEE*, Xinyi Huang, Hongwei Li, *Senior Member, IEEE*, and Robert H. Deng, *Fellow, IEEE*

***Abstract*—In this paper, we study the problem of secure ML inference against a malicious client and a semi-trusted server such that the client only learns the inference output while the server learns nothing. This problem is first formulated by Lehmkuhl et al. with a solution (MUSE, Usenix Security'21), whose performance is then substantially improved by Chandran et al.'s work (SIMC, USENIX Security'22). However, there still exists a nontrivial gap in these efforts towards practicality, giving the challenges of overhead reduction and secure inference acceleration in an all-round way. Based on this, we propose SIMC 2.0, which complies with the underlying structure of SIMC, but significantly optimizes both the linear and non-linear layers of the model. Specifically, (1) we design a new coding method for parallel homomorphic computation between matrices and vectors. (2) We reduce the size of the garbled circuit (GC) (used to calculate non-linear activation functions, e.g., ReLU) in SIMC by about two thirds. Compared with SIMC, our experiments show that SIMC 2.0 achieves a significant speedup by up to $17.4\times$ for linear layer computation, and at least $1.3\times$ reduction of both the computation and communication overhead in the implementation of non-linear layers under different data dimensions. Meanwhile, SIMC 2.0 demonstrates an encouraging runtime boost by $2.3 \sim 4.3\times$ over SIMC on different state-of-the-art ML models.**

***Index Terms*—Garbled circuit, homomorphic encryption, privacy protection, secure inference.**

Manuscript received 27 October 2022; revised 22 April 2023; accepted 18 June 2023. Date of publication 22 June 2023; date of current version 11 July 2024. This work was supported in part by Singapore Ministry of Education (MOE) AcRF Tier 2 under Grant MOE-T2EP20121-0006 and in part by the National Natural Science Foundation of China under Grant 62102090. *(Corresponding author: Tianwei Zhang.)*

Guowen Xu, Xingshuo Han, and Tianwei Zhang are with the School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798 (e-mail: guowen.xu@ntu.edu.sg; xingshuo001@e.ntu.edu.sg; tianwei.zhang@ntu.edu.sg).

Shengmin Xu and Jianting Ning are with the College of Computer and Cyber Security, Fujian Normal University, Fuzhou 350109, China (e-mail: smxu1989@gmail.com; jtning88@gmail.com).

Xinyi Huang is with Artificial Intelligence thrust, Information Hub, Hong Kong University of Science and Technology, Guangzhou 511458, China (e-mail: xinyi@ust.hk).

Hongwei Li is with the School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China (e-mail: hongweili@uestc.edu.cn).

Robert H. Deng is with the School of Information Systems, Singapore Management University, Singapore 188065 (e-mail: robertdeng@smu.edu.sg).

This article has supplementary downloadable material available at https://doi.org/10.1109/TDSC.2023.3288557, provided by the authors.

Digital Object Identifier 10.1109/TDSC.2023.3288557

I. Introduction

THE widespread application of machine learning (ML), especially the popularization of prediction services over pre-trained models, has increased the demand for *secure inference*. In this process, a server S_0 holds an ML model M whose weight W is considered private and sensitive, while a client S_1 has a private input t. The goal of secure inference is to make S_1 only get the model's output while S_0 knows nothing. Such a privacy-preserving paradigm has a variety of potential prospects, especially for privacy-critical applications, e.g., medical diagnosis, and financial data analysis. In theory, secure inference can be implemented by the secure two-party computing (2-PC) protocol in cryptographic primitives [1], [2]. It enables two parties to run an interactive protocol to securely calculate any function without revealing each party's private information. To instantiate it, many impressive works [3], [4], [5] have been proposed, built on various cryptographic technologies such as homomorphic encryption (HE) [6], secret sharing and Yao's garbled circuits (GC) [7]. Due to the inherent complexity of cryptographic primitives, existing efforts focus exclusively on improving efficiency and are based on a relatively weak threat assumption i.e., *semi-honest adversary model* [8], [9], [10]. In this model, S_0 and S_1 faithfully follow the specifications of secure inference and can only capture private information through passive observations.

A. Related Works

Recent works [11], [12], [13] show that this *semi-honest adversary model* can be insecure in real-world applications. In particular, Lehmkuhl et al. [11] argue that it may be reasonable that the ML model is held by a semi-trusted server, since the server is usually fixed and maintained by a reputable entity. However, it is impractical to assume that thousands of clients (which can be arbitrary entities) will faithfully comply with the protocol specification. To validate this hazard, they provide a systematic attack strategy, which enables a malicious client to violate the rules of the protocol and completely reconstruct the model's parameters during the inference interactions. Moreover, the number of inference queries required for such an attack is much smaller than the most advanced black-box model extraction attack [11].

While the above problem can be solved by resorting to the traditional 2-PC protocol against malicious adversaries [14], [15],

[16], it is inefficient in practice. To bridge this gap, Lehmkuhl et al. [11] pioneer the definition of *client-malicious adversary model*, where the server is still considered semi-honest, but the client can perform arbitrary malicious behaviors. To secure the inference process under such adversary model, they propose MUSE, a novel 2-PC protocol with considerably lower overhead compared to previous works. However, the computation and communication costs of MUSE are still not satisfactory, which are at least $15\times$ higher than the similar work DELPHI [17] under the semi-trusted threat model.

To alleviate the efficiency issue, Chandran et al. design SIMC [12], the state-of-the-art 2-PC protocol for secure inference under the client-malicious adversary model. Consistent with the underlying tone of MUSE, SIMC uses HE to execute the linear layers (including matrix-vector and convolution multiplication) of the ML model, and uses GC to implement the non-linear layers (mainly the ReLU function). Since almost 99% of the communication overhead in MUSE comes from non-linear layers, the core of SIMC is a novel protocol based on a customized GC to improve the performance of executing non-linear activation functions. As a result, compared to MUSE, SIMC gains $28 \sim 33\times$ reduction in the communication overhead for the implementation of popular activation functions (e.g., ReLU, ReLU6) and at least $4\times$ overall performance improvement.

However, we argue that SIMC is still far from practicality. This stems from two reasons. First, SIMC keeps the design of linear layer calculations in MUSE, which uses computationally heavy HE to perform dense matrix-vector and convolution multiplication. In reality, linear operations dominate the computation of modern neural networks: nearly 95% of the execution of the ML model is for intensive convolutional layers and fully connected (FC) layers [18]. This raises the requirement for efficient execution of matrix and convolution multiplication under ciphertext. Although the homomorphism of HE makes it suitable to achieve privacy-preserving linear operations, it is still computationally expensive for large-scale operations, especially when there is no appropriate parallel computing optimization (i.e., performing homomorphic linear computation in a Single Instruction Multiple Data (SIMD) manner) [19].

There is still much optimization space for the non-linear layers (such as ReLU) of SIMC, from the perspectives of both computation and communication. To be precise, given secret shares of a value t, SIMC designs a secure 2-PC protocol based on GC, which enables S_0 and S_1 to calculate the shares $s = \alpha t$ and authenticated shares of $v = f(t)$, i.e., shares of $f(t)$ and $\alpha f(t)$, where f denotes the activation function. Although SIMC has made great efforts to simplify the design of the GC and prevent malicious behaviors from the client, it requires a communication overhead of at least $2c\lambda + 4\kappa\lambda + 6\kappa^2$, where λ denotes the security parameter, κ is a field space, and c is the number of AND gates required to reconstruct shares of t and compute authenticated $f(t)$. Also, the main body of the activation function is still sealed in a relatively complex GC. As a consequence, SIMC inevitably results in substantial computation and communication costs in non-linear layers, since a modern model usually contains thousands of non-linear activation functions for calculation. We will perform experiments to demonstrate such overheads in Section VI.

B. Technical Challenges

This paper is dedicated to design a new 2-PC protocol to break through the performance bottleneck in SIMC, thereby promoting the practicality of secure inference against malicious clients. We follow the underlying structure of SIMC, i.e., using HE to execute the linear layers and GC to implement the non-linear layers, as such a hybrid method has shown advanced performance compared to other strategies [11], [17]. Therefore, our work naturally lies in solving two problems: (1) how to design an optimized mechanism to accelerate linear operations in HE, and (2) how to find a more simplified GC for the execution of non-linear activation functions.

There have been many impressive works [18], [19], [20], [21] exploring methods towards the above goals. To speed up HE's computation performance, existing efforts mainly focus on designing new coding methods to achieve parallelized element-wise homomorphic computation, i.e., performing homomorphic linear computation in an SIMD manner. For example, Jiang et al. [21] present a novel matrix encoding method for basic matrix operations, e.g., multiplication and transposition. Compared to previous approaches, this method reduces the computational complexity of multiplying two $d \times d$ matrices from $O(d^3)$ to $O(d)$. However, it is exclusively applicable to matrix operations between square or rectangular matrices, and is unfriendly to arbitrary matrix-vector multiplication and convolution operations in ML [22]. In addition, it considers parallel homomorphic calculations in a full ciphertext environment, contrary to our scenario where only the input of the client is ciphertext while the model parameters are clear. Sav et al. propose the Alternating Packing (AP) approach [19], which packs all elements of a vector into a ciphertext and then parallelizes homomorphic matrix-vector multiplication. However, similar to [21], AP focuses on SIMD operation between two packed ciphertexts.

Several works design solutions for the scenario of secure inference [18], [23], [24], mostly built on the *semi-honest adversary model*. Among them, one of the most remarkable works is GAZELLE [23]. It is customized for the HE-GC based secure inference and has been integrated into some advanced solutions such as DELPHI [17] and EzPC [25]. In fact, the core idea of GAZELLE is also used in the design of MUSE and SIMC, which inherit DELPHI's optimization strategy for HE-based linear operations. GAZELLE builds a new homomorphic linear algebra kernel, which provides fast algorithms for mapping neural network layers to optimized homomorphic matrix-vector multiplication and convolution routines. However, the computation complexity of GAZELLE is still non-negligible. For example, to calculate the product of a $n_o \times n_i$ plaintext matrix and a n_i dimensional ciphertext vector, GAZELLE requires at least $(\log_2(\frac{n}{n_o}) + \frac{n_i n_o}{n} - 1)$ rotation operations, where n is the number of slots in the ciphertext. This is very computationally expensive compared to other homomorphic operations such as addition and multiplication. To alleviate this problem, Zhang et al. presents GALA [18], an optimized solution over GAZELLE, to reduce the complexity of the rotation operations required by matrix vector calculations from $(\log_2(\frac{n}{n_o}) + \frac{n_i n_o}{n} - 1)$ to $(\frac{n_i n_o}{n} - 1)$, thus substantially improving the efficiency. However, GALA is specially customized for secure inference under *semi-honest adversary model*. Furthermore, we will demonstrate that the computational complexity of GALA is not optimal and can be further optimized.

To construct an efficient GC-based protocol, the following problems generally need to be solved: (1) avoiding using the GC to perform the multiplication between elements as much as possible, which usually requires at least $O(\kappa^2\lambda)$ communication complexity [26]; (2) ensuring the correctness of the client's input, so that the output of the GC is trusted and can be correctly propagated to the subsequent layers. SIMC presents a novel protocol that can meet these two requirements. Specifically, instead of using the GC directly to calculate $s = \alpha t$ and authenticated shares of $v = f(t)$, SIMC only resorts to the GC to obtain the garbled labels of the bits corresponding to each function, namely $s[i], v[i]$ and $\alpha v[i]$ for $1 \le i \le \kappa$. Such types of calculation are natural for the GC because it operates on a Boolean circuit. Moreover, it avoids performing a large number of multiplication operations in the GC to achieve binary-to-decimal conversion. Then, SIMC designs a lightweight input consistency verification method, which is used to force the client to feed the correct sharing of GC's input. Compared to MUSE, SIMC reduces the communication overhead of each ReLU function from $2d\lambda + 190\kappa\lambda + 232\kappa^2$ to $2c\lambda + 4\kappa\lambda + 6\kappa^2$, and accelerates the calculation several times.

We point out that it is possible to further simplify the protocol in SIMC. It comes from the insight that the ReLU function can be parsed as $f(t) = t \cdot sign(t)$, where the sign function $sign(t)$ equals 1 if $t \ge 0$ and 0 otherwise. Therefore, it is desirable if only the non-linear part of $f(t)$ (i.e., $sign(t)$) is encapsulated in the GC, i.e., for $1 \le i \le \kappa$, the output of the GC is $sign[i]$ and $s[i]$, instead of $s[i]$ and $v[i]$. Then, if we can find a lightweight alternative sub-protocol to privately compute the authenticated shares $v[i] = sign[i] \times s[i]$, it not only simplifies the size of the GC, but also further reduces the number of expensive multiplications between elements in the original GC. However, it is challenging to build such a protocol: The replacement sub-protocol should be lightweight compared to the original GC. In addition, it should be compatible with the input consistency verification method in SIMC, so as to realize the verifiability of the client input.

C. Our Contributions

In this work, we present SIMC 2.0, a new secure inference model that is resilient to malicious clients and achieves up to $5\times$ computation improvement over the previous state-of-the-art SIMC. SIMC 2.0 complies with the underlying structure of SIMC, but designs highly optimized methods to substantially reduce the overhead of linear and non-linear layers. In summary, the contributions of SIMC 2.0 are summarized as follows.

- We design a new coding method for homomorphic linear computation in a SIMD manner. It is custom-built through the insight into the complementarity between cryptographic primitives in an HE-GC based framework, where the property of secret sharing is used to convert a large number of private rotation operations to be executed in the plaintext environment. Moreover, we design a block-combined diagonal encoding method to further reduce the number of rotation operations. As a result, compared to SIMC, we reduce the complexity of rotation operations required by matrix-vector computations from $(\log_2(\frac{n}{n_o}) + \frac{n_i n_o}{n} - 1)$ to $(l - 1)$, where l is a hyperparameter set by the server. We also present a new method for homomorphic convolution operation with SIMD support.
- We reduce the size of the GC in SIMC by about two thirds. As discussed above, instead of using the GC to calculate the entire ReLU function, we only encapsulate the non-linear part of ReLU into the GC. Then, we construct a lightweight alternative protocol that takes the output of the simplified GC as input to calculate the sharing of the desired result. We exploit the authenticated shares [27] as the basis for building the lightweight alternative protocol. As a result, compared to SIMC, SIMC 2.0 reduces the communication overhead of calculating each ReLU from $2c\lambda + 4\kappa\lambda + 6\kappa^2$ to $2e\lambda + 4\kappa\lambda + 6\kappa^2 + 2\kappa$, where $e < c$ denotes the number of AND gates required in the GC.
- We demonstrate that SIMC 2.0 has the same security properties as SIMC, i.e., it is secure against the malicious client model. We prove this with theoretical analysis following a similar logic of SIMC. We use several datasets (e.g., MNIST, CIFAR-10) to conduct extensive experiments on various ML models. Our experiments show that SIMC 2.0 achieves a significant speedup by up to $17.4\times$ for linear layer computation and at least $1.3\times$ communication reduction in non-linear layer computation under different data dimensions. Meanwhile, SIMC 2.0 demonstrates an encouraging runtime boost by $2.3 \sim 4.3\times$ over SIMC on different state-of-the-art ML models (e.g., AlexNet, VGG, ResNet).

Roadmap: The remainder of this paper is organized as follows. In Section II, we review some basic concepts and introduce the scenarios and threat models involved in this paper. In Sections III–V, we give the details of our SIMC 2.0. The performance evaluation is presented in Section VI, and Section VII concludes the paper.

II. Preliminaries

A. Threat Model

We consider a two-party ML inference scenario consisting of a server S_0 and a client S_1. S_0 holds an ML model M with the private and sensitive weight W. It is considered semi-honest: it obeys the deployment procedure of ML inference but may be curious to infer the client's data by observing the data flow in the running process. S_1 holds the private input t. It is malicious and can arbitrarily violate the protocol specification. The model architecture NN of M is assumed to be known to both the server and the client. Our goal is to design a secure inference framework, which enables S_1 to learn the inference result of t

without knowing any details about the model weight W, and S_0 knows nothing about the input t. A formal definition of the threat model is provided in Appendix A, available online.

B. Notations

We describe some notations used in this paper. Specifically, λ and σ denote the computational and statistical security parameters. For any $n > 0$, $[n]$ denotes the set $\{1, 2, \dots n\}$. In our SIMC 2.0, all arithmetic operations, such as addition and multiplication, are performed in the field $\mathbb{F}_p$, where p is a prime and $\kappa = \lceil \log p \rceil$. We assume that any element $x \in \mathbb{F}_p$ can be naturally mapped to the set $\{1, 2, \dots \kappa\}$, where we use $x[i]$ to represent the i-th bit of x, i.e., $x = \sum_{i \in [\kappa]} x[i] \cdot 2^{i-1}$. For a vector $\mathbf{x}$ and an element $\alpha \in \mathbb{F}_p$, $\alpha + \mathbf{x}$ and $\alpha\mathbf{x}$ denote the addition and multiplication of each component of $\mathbf{x}$ with α, respectively. Given a function $f : \mathbb{F}_p \to \mathbb{F}_p$, $f(\mathbf{x})$ represents the evaluation of f for each component of $\mathbf{x}$. Given two elements $x, y \in \mathbb{F}_p$, $x||y$ denotes the concatenation of x and y.

For simplicity, we assume that the targeted ML architecture NN consists of alternating linear and non-linear layers. Let the specifications of the linear layers and non-linear layers be $\mathbf{N}_1$, $\mathbf{N}_2, \dots, \mathbf{N}_m$ and $f_1, f_2, \dots, f_{m-1}$, respectively. Given an input vector $\mathbf{t}_0$, the model sequentially computes $\mathbf{u}_i = \mathbf{N}_i \cdot \mathbf{t}_{i-1}$ and $\mathbf{t}_i = f_i(\mathbf{u}_i)$, where $i \in [m-1]$. As a result, we have $\mathbf{u}_m = \mathbf{N}_m \cdot \mathbf{t}_{m-1} = \mathrm{NN}(\mathbf{t}_0)$.

C. Fully Homomorphic Encryption (FHE)

FHE [28] is a public key encryption system that supports the evaluation of any function parsed as a polynomial in ciphertext. Informally, assuming that the message space is $\mathbb{F}_p$, FHE contains the following four algorithms:

- $\texttt{KeyGen}(1^\lambda) \to (pk, sk)$: Given the security parameter λ, $\texttt{KeyGen}$ is a randomized algorithm that generates a public key pk and the corresponding secret key sk.
- $\texttt{Enc}(pk, t) \to c$: Given the public key pk and a message $t \in \mathbb{F}_p$ as input, $\texttt{Enc}$ outputs a ciphertext c.
- $\texttt{Dec}(sk, c) \to t$: Given the secret key sk and a ciphertext c as input, $\texttt{Dec}$ decrypts c and obtains the plaintext t.
- $\texttt{Eval}(pk, c_1, c_2, F) \to t$: Given the public key pk and two ciphertexts c_1, c_2 encrypting t_1, t_2, respectively, and a function F parsed as a polynomial, $\texttt{Eval}$ outputs a ciphertext c' encrypting $F(t_1, t_2)$.

We require FHE to satisfy correctness, semantic security and additive homomorphism [28]. In addition, we use ciphertext packing technology (CPT) [29] to accelerate the parallelism of homomorphic computation. In brief, CPT is capable of packing up to n plaintexts into one ciphertext that contains n plaintext slots, thus improving the parallelism of the computation. This makes homomorphic addition and multiplication for the ciphertext equivalent to performing the same operation on every plaintext slot at once. For example, given two ciphertexts c_1 and c_2 encrypting the plaintext vectors $\mathbf{t} = (t_0, t_1, \dots, t_n)$ and $\mathbf{t}' = (t'_0, t'_1, \dots, t'_n)$ respectively, $\texttt{Eval}(pk, c_1, c_2, F = (a + b))$ outputs a ciphertext c encrypting the plaintext vector $\mathbf{t}'' = (t_0 + t'_0, t_1 + t'_1, \dots, t_n + t'_n)$.

CPT also provides a *rotation* function $\mathbf{Rot}$ to facilitate rotation operations between plaintexts in different plaintext slots. To be precise, given a ciphertext c encrypting a plaintext vector $\mathbf{t} = (t_0, t_1, \dots, t_n)$, $\mathbf{Rot}(pk, c, j)$ transforms c into an encryption of $\mathbf{t}' = (t_j, t_{j+1}, \dots, t_0, \dots, t_{j-1})$, which enables the packed vector in the appropriate position to realize the correct element-wise addition and multiplication. Since the computation cost of the rotation operation in FHE is much more expensive than other operations such as homomorphic addition and multiplication, in our SIMC 2.0, we aim to design homomorphic parallel computation methods customized for matrix-vector multiplication and convolution, thereby minimizing the number of rotation operations incurred during execution.

D. Secret Sharing

We describe some terms used for secret sharing as follows.

- *Additive secret sharing [26]:* Given $x \in \mathbb{F}_p$, we say that a 2-of-2 additive secret sharing of x is a pair $(\langle x \rangle_0, \langle x \rangle_1) = (x - r, r) \in \mathbb{F}_p^2$ that satisfies $\langle x \rangle_0 + \langle x \rangle_1 = x$, where r is randomly selected from $\mathbb{F}_p$. Additive secret sharing is perfectly hidden, i.e., given $\langle x \rangle_0$ or $\langle x \rangle_1$, the value of x is perfectly hidden.
- *Authenticated shares:* Given a randomly selected $\alpha \in \mathbb{F}_p$ (known as the MAC key), the authenticated share of $x \in \mathbb{F}_p$ is denoted as $\{\langle x \rangle_b, \langle \alpha x \rangle_b\}_{b \in \{0,1\}}$, where each party S_b holds $(\langle x \rangle_b, \langle \alpha x \rangle_b)$. In the fully malicious protocol, α should be shared secretly with all parties. In our client-malicious model, consistent with previous works [11], [12], α is picked uniformly by the server S_0 and secretly shared with the client S_1. Authenticated shares provide $\lfloor \log p \rfloor$ bits of statistical security. Specifically, assuming that malicious S_1 has manipulated the share of x as $x + \beta$ by changing the shares $(\langle x \rangle_1, \langle \alpha x \rangle_1)$ to $(\langle x \rangle_1 + \beta, \langle \alpha x \rangle_1 + \beta')$, the probability of parties authenticating holding the share of $x + \beta$ (i.e., $\alpha x + \beta' = \alpha(x + \beta)$) is at most $2^{-\lfloor \log p \rfloor}$.
- *Authenticated Beaver's multiplicative triples:* Given a randomly selected triple $(A, B, C) \in \mathbb{F}_p^3$ satisfying $AB = C$, an authenticated Beaver's multiplicative triple denotes that S_b holds the following shares
$$\{(\langle A \rangle_b, \langle \alpha A \rangle_b), (\langle B \rangle_b, \langle \alpha B \rangle_b), (\langle C \rangle_b, \langle \alpha C \rangle_b)\}$$
for $b \in \{0, 1\}$. The process of generating triples is offline and is used to facilitate multiplication between authenticated shares. We give details of generating such triples in Fig. 8 in Appendix B, available online.

E. Oblivious Transfer

The 1-out-of-2 Oblivious Transfer (OT) [30] is denoted as OT_n, where the inputs of the sender (S_0) are two strings $s_0, s_1 \in \{0,1\}^n$, and the input of the receiver (S_1) is a choice bit $b \in \{0, 1\}$. At the end of the OT-execution, S_1 obtains s_b while S_0 receives nothing. Succinctly, the security properties of OT_n require that 1) the receiver learns nothing but s_b and 2) the sender knows nothing about the choice b. In this paper, we require that the instance of OT_n is secure against a semi-honest sender and a malicious receiver. We use OT_n^κ to represent κ

instances of OT_n. We exploit [30] to implement OT_n^{κ} with the communication complexity of $\kappa\lambda + 2n$ bits.

F. Garbled Circuits

A garbled scheme [31] for Boolean circuits consists of two algorithms, Garble and GCEval, defined as follows.

- $\texttt{Garble}(1^\lambda, C) \rightarrow (\texttt{GC}, \{\{\texttt{lab}_{i,j}^{in}\}_{i\in[n]}, \{\texttt{lab}_j^{out}\}\}_{j\in\{0,1\}})$: Given the security parameter λ and a Boolean circuit $C:\{0,1\}^n \rightarrow \{0,1\}$, the Garble function outputs a garbled circuit GC, an input set $\{\texttt{lab}_{i,j}^{in}\}_{i\in[n],j\in\{0,1\}}$ of labels and an output set $\{\texttt{lab}_j^{out}\}_{j\in\{0,1\}}$, where each label is of λ bits. In brief, $\{\texttt{lab}_{i,x[i]}^{in}\}_{i\in[n]}$ represents the garbled input for any $x \in \{0,1\}^n$ and the label $\texttt{lab}_{C(x)}^{out}$ represents the garbled output for $C(x)$.
- $\texttt{GCEval}(\texttt{GC}, \{\texttt{lab}_i\}_{i\in[n]}) \rightarrow \texttt{lab}'$: Given the garbled circuit GC and a set of labels $\{\texttt{lab}_i\}_{i\in[n]}$, GCEval outputs a label $\texttt{lab}'$.

The above garbled scheme (Garble, GCEval) is required to satisfy the following properties:

- *Correctness:* GCEval is faithfully evaluated on GC and outputs $C(x)$ if the garbled x is given. Formally, for any circuit C and $x \in \{0,1\}^n$, we have
$$\texttt{GCEval}(\texttt{GC}, \{\texttt{lab}_{i,x[i]}^{in}\}_{i\in[n]}) \rightarrow \texttt{lab}_{C(x)}^{out}$$
- *Security:* For any circuit C and input $x \in \{0,1\}^n$, there exists a polynomial probability-time simulator Sim that can simulate the GC and garbled input of x generated by Garble in real execution, i.e., $(\texttt{GC}, \{\texttt{lab}_{i,x[i]}^{in}\}_{i\in[n]}) \approx \texttt{Sim}(1^\lambda, C)$, where $\approx$ indicates computational indistinguishability of two distributions $(\texttt{GC}, \{\texttt{lab}_{i,x[i]}^{in}\}_{i\in[n]})$ and $\texttt{Sim}(1^\lambda, C)$.
- *Authenticity:* It is infeasible to guess the output label of $1 - C(x)$ given the garbled input of x and GC. Formally, for any circuit C and $x \in \{0,1\}^n$, we have $(\texttt{lab}_{1-C(x)}^{out} | \texttt{GC}, \{\texttt{lab}_{i,x[i]}^{in}\}_{i\in[n]}) \approx U_\lambda$, where U_λ represents the uniform distribution on the set $\{0,1\}^n$.

Note that the garbled scheme described above can be naturally extended to the case with multiple garbled outputs. We also utilize state-of-the-art optimization strategies, including point-and-permute, free-XOR and half-gates [32] to construct the garbled circuit. We provide a high-level description of performing a secure two-party computation with GC: assuming that a semi-honest server S_0 and a malicious client S_1 hold private inputs x and y, respectively. Both parties evaluate $C(x, y)$ on GC as follows. S_0 first garbles the circuit C to learn GC and garbled labels about the input and output of GC. Then, both parties invoke the OT functionality, where the sender S_0 inputs garbled labels corresponding to the input wires of S_1, while the receiver S_1 inputs y to obtain garbled inputs of y. S_0 additionally sends the garbled input of x, GC and a pair of ciphertexts for every output wire w of C to S_1. As a result, S_1 evaluates GC to learn the garbled output of $C(x, y)$ with the received garbled input about x and y. From the garbled output and the pair of ciphertexts given for each output wire, S_1 finally gets $C(x, y)$. S_1 sends $C(x, y)$ along with the hash of the garbled output to S_0 for verification. S_0 accepts it if the hash value corresponds to $C(x, y)$.

III. Linear Layer Optimization

We start by describing the secure execution of linear layers in SIMC. To be precise, SIMC designs two protocols (*InitLin* and *Lin*) to securely perform linear layer operations, as shown in Fig. 9 and 10 in Appendix C, available online. We observe that the most computationally intensive operations in *InitLin* and *Lin* are homomorphically computing $\mathbf{N} \cdot \mathbf{t}$ and $\alpha\mathbf{N} \cdot \mathbf{t}$ (including matrix-vector multiplication in FC layers and convolution operations in convolution layers), where $\mathbf{N}$ is a plaintext weight matrix held by the server, and $\mathbf{t}$ (will be encrypted as $[\mathbf{t}]_\mathbf{c}$) is the input held by the client.

We focus on the optimization of matrix-vector parallel multiplication. More specifically, we consider an FC layer with n_i inputs and n_o outputs, i.e., $\mathbf{N} \in \mathbb{F}_p^{n_o \times n_i}$. The client's input is a ciphertext vector $\mathbf{t} \in \mathbb{F}_p^{n_i}$. n is the number of slots in a ciphertext. We first describe a naive approach to parallelize homomorphic multiplication of $\mathbf{N} \cdot \mathbf{t}$, followed by a state-of-the-art method proposed by GAZELLE [23]. Finally, we introduce our proposed scheme, which substantially reduces the computation cost of matrix-vector multiplication.

A. Naive Method

The naive method of matrix-vector multiplication is shown in Fig. 1, where $\mathbf{N}$ is the $n_o \times n_i$ dimensional plaintext matrix held by the server and $[\mathbf{t}]_\mathbf{c}$ is the encryption of the client's input vector. The server encodes each row of the matrix into a separate plaintext vector (Step (a) in Fig. 1), where each encoded vector is of length n (including zero-padding if necessary). We denote these encoded plaintext vectors by $\mathbf{N_0}$, $\mathbf{N_1}$, ..., $\mathbf{N_{n_o-1}}$. For example, there are four vectors in Fig. 1, namely $\mathbf{N_0}$, $\mathbf{N_1}$, $\mathbf{N_2}$, and $\mathbf{N_3}$.

The purpose of the server is to homomorphically compute the dot product of the plaintext $\mathbf{N}$ and the ciphertext $[\mathbf{t}]_\mathbf{c}$. Let ScMult be the scalar multiplication of a plaintext and a ciphertext in HE. The server first uses ScMult to compute the element-wise multiplication of $\mathbf{N_i}$ and $[\mathbf{t}]_\mathbf{c}$. As a result, the server gets $[\mathbf{v_i}]_\mathbf{c} = [\mathbf{N_i} \odot \mathbf{t}]_\mathbf{c}$ (Step (b) in Fig. 1). We observe that the sum of all the elements in $[\mathbf{v_i}]_\mathbf{c}$ is the i-th element of the desired dot product of $\mathbf{N}$ and $[\mathbf{t}]_\mathbf{c}$. Since there is no direct way to obtain this sum in HE, we rely on the rotation operation to do it. To be precise, as shown in Step (c) in Fig. 1, $[\mathbf{v_i}]_\mathbf{c}$ is first rotated by permuting $\frac{n_i}{2}$ positions so that the first $\frac{n_i}{2}$ entries of the rotated $[\mathbf{v_i}]_\mathbf{c}$ are the second $\frac{n_i}{2}$ entries of the original $[\mathbf{v_i}]_\mathbf{c}$. The server then performs element-wise addition on the original $[\mathbf{v_i}]_\mathbf{c}$ and rotated one homomorphically, which derives a ciphertext whose sum of the first $\frac{n_i}{2}$ entries is actually the desired result. The server iteratively performs the above rotation operation for $\log_2 n_i$ times. Each time, it operates on the ciphertext derived from the previous iteration. The server finally receives the ciphertext whose first entry is the i-th element of $\mathbf{Nt}$. By executing this procedure for each $[\mathbf{v_i}]_\mathbf{c}$, i.e., $[\mathbf{v_0}]_\mathbf{c}$, $[\mathbf{v_1}]_\mathbf{c}$, $[\mathbf{v_2}]_\mathbf{c}$, and $[\mathbf{v_3}]_\mathbf{c}$ in Fig. 1, the server obtains n_o ciphertexts. Consistently, the first entries of these ciphertexts correspond to the elements in $\mathbf{Nt}$.

Now we analyze the complexity of the above matrix-vector multiplication. We consider the process beginning with the server receiving the ciphertext (i.e., $[\mathbf{t}]_\mathbf{c}$) from the client until

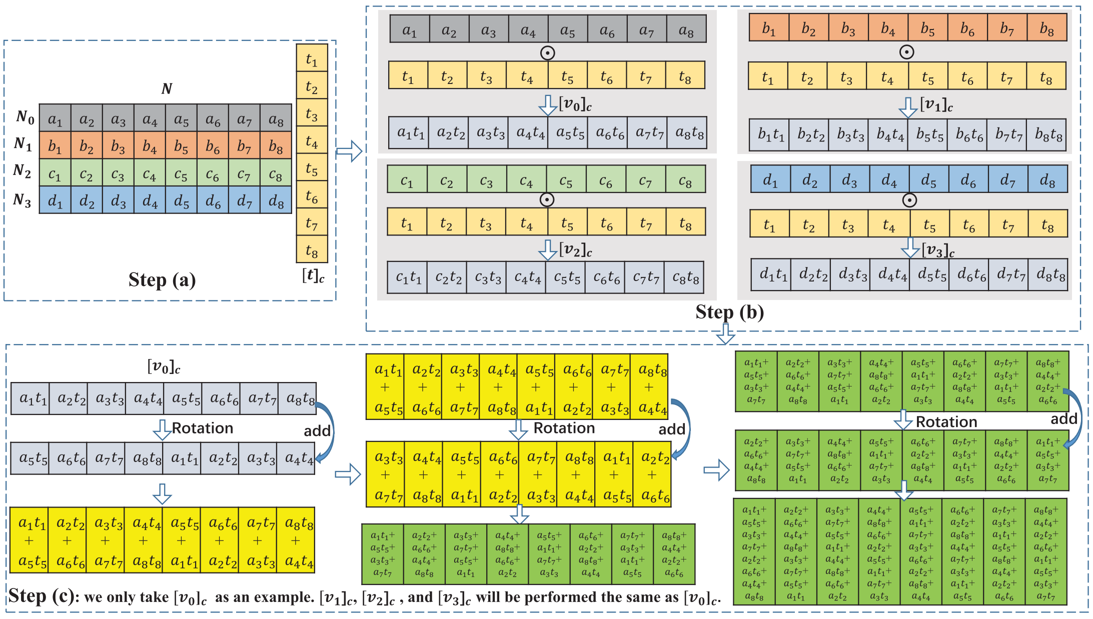
N
N_0
N_1
N_2
N_3
Step (a)
$[t]_c$
$[v_0]_c$
$[v_1]_c$
$[v_2]_c$
$[v_3]_c$
Step (b)
Rotation
add
Step (c): we only take $[v_0]_c$ as an example. $[v_1]_c$, $[v_2]_c$, and $[v_3]_c$ will be performed the same as $[v_0]_c$.

Fig. 1. Naive matrix-vector multiplication.

it obtains the ciphertext (i.e., n_o ciphertexts) to be shared.[1] There are totally n_o scalar multiplication operations (i.e., ScMult), $n_o \log_2 n_i$ rotation operations, and $n_o \log_2 n_i$ addition operations. It produces n_o output ciphertexts, each containing an element in $\mathbf{Nt}$. Such naive use of the ciphertext space inevitably incurs inefficiencies for linear computations.

B. Hybrid Method (GAZELLE)

In order to fully utilize the n slots and substantially reduce computational complexity, Juvekar et al. [23] propose GAZELLE, which is also used in SIMC. GAZELLE is actually a variant of diagonal encoding [29], which exploits the fact that n_o is usually much smaller than n_i in the FC layer. Based on this, GAZELLE shows that the most expensive rotation operation is a function of n_o rather than n_i, thus speeding up the calculation of the FC layer. The basic idea of GAZELLE is shown in Fig. 2. The server encodes the matrix $\mathbf{N}$ diagonally into a vector of n_o plaintexts. For example, as shown in Step (a) in Fig. 2, the first plaintext vector $\mathbf{N_0}$ consists of gray elements in matrix $\mathbf{N}$, i.e., $(a_1, b_2, c_3, d_4, a_5, b_6, c_7, d_8)$. The second plaintext vector $\mathbf{N_1}$ consists of orange elements $(a_2, b_3, c_4, d_5, a_6, b_7, c_8, d_1)$. The composition of $\mathbf{N_2}$ and $\mathbf{N_3}$ is analogous. Note that the meaning of $\mathbf{N_0}$ to $\mathbf{N_3}$ in the hybrid approach is different from the previous naive approach in Section III-A.

For $i = 1$ to $n_o - 1$, the server rotates $[\mathbf{t}]_\mathbf{c}$ by i positions, as shown in Step (b) in Fig. 2. Afterwards, ScMult is used to perform the element-wise multiplication of $\mathbf{N_i}$ and the corresponding ciphertext. For example, as shown in Step (c) in Fig. 2, $\mathbf{N_0}$ is multiplied by the ciphertext $[\mathbf{t}]_\mathbf{c}$, while $\mathbf{N_1}$ is multiplied by $[\mathbf{t}']_\mathbf{c}$, and so on. As a result, the server obtains n_o multiplied ciphertexts, $\{[\mathbf{v_i}]_\mathbf{c}\}$. The server receives four ciphertexts $\{[\mathbf{v_0}]_\mathbf{c}, [\mathbf{v_1}]_\mathbf{c}, [\mathbf{v_2}]_\mathbf{c}, [\mathbf{v_3}]_\mathbf{c}\}$, whose elements are all part of the desired dot product. Then, the server sums all ciphertexts $[\mathbf{v_i}]_\mathbf{c}$ (Step (d) in Fig. 2) in an element-wise way to form a new ciphertext. At this point, the server performs rotation operations similar to the naive approach, i.e., iteratively performing $\log_2 \frac{n_i}{n_o}$ rotations followed by addition to obtain a final single ciphertext. The first n_o entries in this ciphertext correspond to the n_o ciphertext elements in $\mathbf{Nt}$.

To further reduce computation cost, GAZELLE proposes combining multiple copies of $\mathbf{t}$ into a single ciphertext (called $[\mathbf{t_{pack}}]_\mathbf{c}$), since the number n of slots in a single ciphertext is usually larger than the dimension n_i of the input vector. As a result, $[\mathbf{t_{pack}}]_\mathbf{c}$ has $\frac{n}{n_i}$ copies of $\mathbf{t}$ so that the server can perform ScMult operations on $[\mathbf{t_{pack}}]_\mathbf{c}$ with $\frac{n}{n_i}$ encoded vectors at once. This causes the server to get $\frac{n_i n_o}{n}$ ciphertexts instead of n_o ciphertexts, resulting in a single ciphertext that contains $\frac{n}{n_o}$ rather than $\frac{n_i}{n_o}$ blocks. Finally, the server iteratively performs $\log_2 \frac{n}{n_o}$ rotations followed by addition to obtain a final single ciphertext, where the first n_o entries in this ciphertext correspond to the n_o ciphertext elements of $\mathbf{Nt}$.

In terms of complexity, GAZELLE requires $\frac{n_i n_o}{n}$ scalar multiplications, $\log_2 \frac{n}{n_o} + \frac{n_i n_o}{n} - 1$ rotations, and $\log_2 \frac{n}{n_o} + \frac{n_i n_o}{n} - 1$ additions. It outputs only one ciphertext, which greatly improves the efficiency and utilization of slots over the naive approach.

C. Our Method

In the above hybrid method, the rotation operation comes from two parts: rotations required for the client's encrypted input (Step (b) in Fig. 2) and the subsequent rotations followed

[1]In the neural network inference process with HE-GC as the underlying structure, the ciphertext generated in the linear phase will be securely shared between the client and the server, and used as the input of the subsequent GC-based nonlinear function. The reader can refer to [11], [12], [13] for more details.

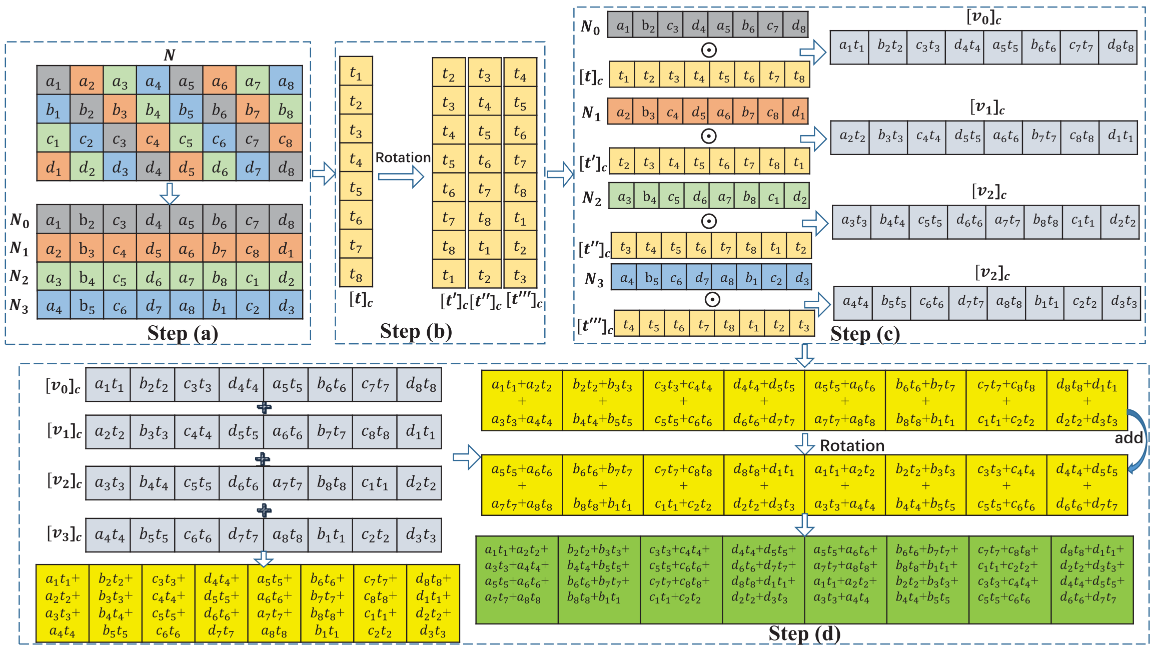

Fig. 2. Hybrid matrix-vector multiplication.

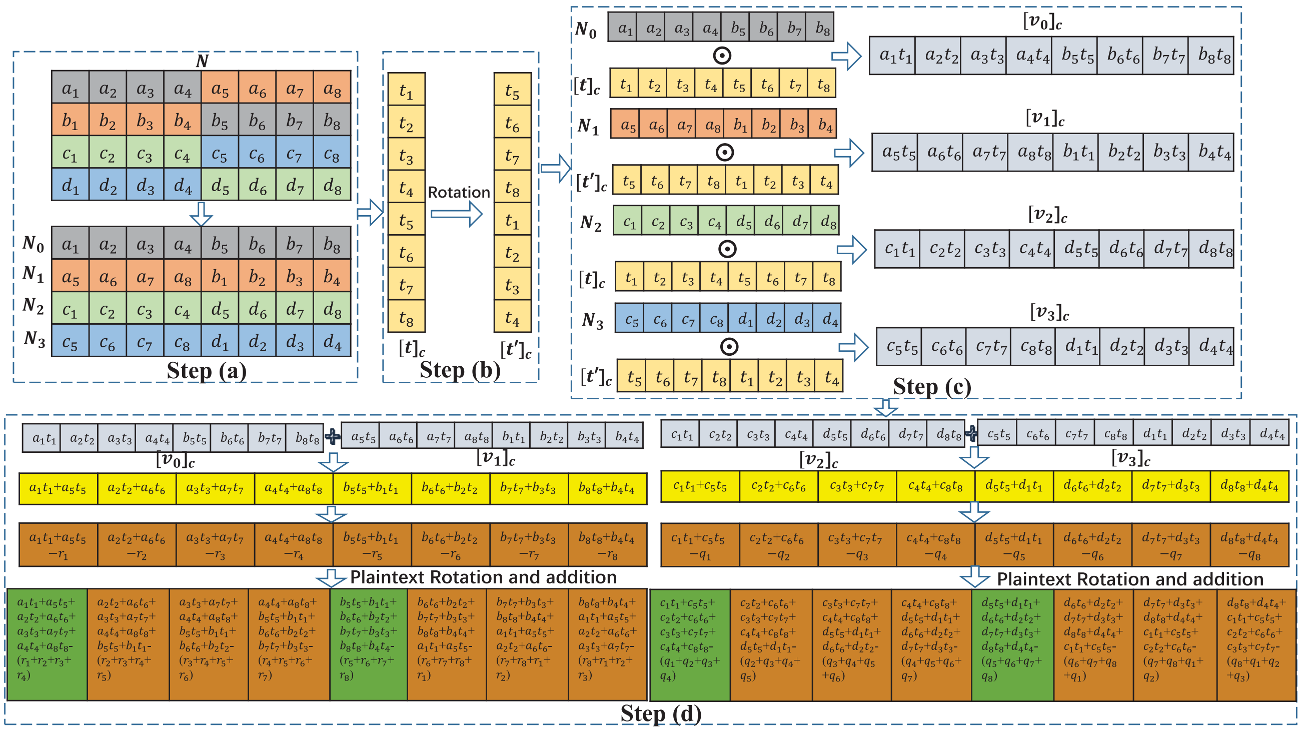
Step (a)
Step (b)
Rotation
Step (c)
Step (d)
Plaintext Rotation and addition
Plaintext Rotation and addition

Fig. 3. Our matrix-vector multiplication algorithm.

by addition involved in obtaining the final matrix-vector result (Step (d) in Fig. 2). Therefore, our approach is motivated by two observations in order to substantially reduce the number of rotation operations incurred by these two aspects. First, we divide the server's original $n_o \times n_i$-dimensional weight matrix $\mathbf{N}$ into multiple sub-matrices, and calculate each sub-matrix separately (explained later based on Fig. 3). This can effectively reduce the number of rotations for the client's ciphertext $[\mathbf{t}]_\mathbf{c}$ from $n_o - 1$ to $l - 1$, where l is a hyperparameter set by the server.

Second, for the rotations followed by addition (Step (d) in Fig. 2), we claim that they are not necessary. This is determined by the characteristics of the HE-GC based secure inference framework. To be precise, the ciphertext output from the FC layer will be secretly shared with the client and server, and used as the input of the next non-linear layer function. As the shares are in plaintext, we can completely convert the rotation followed by addition to be performed under plaintext. This will significantly reduce the computational complexity. For example, given a 16×256 -dimensional matrix and a vector of length 256, our experiments indicate at least a $3.5\times$ speedup compared to the hybrid method (see more details in Section VI).

Fig. 3 illustrates our matrix-vector calculation procedure. Specifically, we first split the matrix $\mathbf{N}$ into multiple sub-matrices. As shown in Step (a) in Fig. 3, for simplicity, we take the first $l = 2$ rows (i.e., $\mathbf{N}_0$ and $\mathbf{N}_1$) of $\mathbf{N}$ as a sub-matrix, thus $\mathbf{N}_2$ and $\mathbf{N}_3$ form another sub-matrix. Then, we exploit the diagonal method to arrange the $1 \times (n_i/l)$ sized sub-matrices of $\mathbf{N}$ as a sub-matrix, which results in two new sub-matrices. For subsequent parallel ciphertext computation, we need to sequentially perform $l-1$ rotations of the client's input $[\mathbf{t}]_\mathbf{c}$, starting from moving the $(n_i/l+1)$-th entry to the first entry. In Step (b) in Fig. 3, $[\mathbf{t}]_\mathbf{c}$ is rotated into $[\mathbf{t}']_\mathbf{c}$, where the first entry of $[\mathbf{t}']_\mathbf{c}$ is the $(8/2+1) = 5$-th entry of $[\mathbf{t}]_\mathbf{c}$.

ScMult is further used to perform the element-wise multiplication of $\mathbf{N_i}$ and the corresponding ciphertext. For example, as shown in Step (c) in Fig. 3, $\mathbf{N_0}$ is multiplied by the ciphertext $[\mathbf{t}]_\mathbf{c}$, while $\mathbf{N_1}$ is multiplied by $[\mathbf{t}']_\mathbf{c}$. $\mathbf{N_2}$ and $\mathbf{N_3}$ are operated similarly. As a result, the server obtains n_o multiplied ciphertexts $\{[\mathbf{v_i}]_\mathbf{c}\}$, which can be divided into l independent ciphertext pairs. We can observe that the server gets two independent ciphertext pairs, i.e., $\{[\mathbf{v_0}]_\mathbf{c}, [\mathbf{v_1}]_\mathbf{c}\}$ and $\{[\mathbf{v_2}]_\mathbf{c}, [\mathbf{v_3}]_\mathbf{c}\}$. For each individual set of ciphertexts, the server adds all ciphertexts in that set in an element-wise way, thus forming l new ciphertexts (yellow boxes in Step (d) in Fig. 3).

Until now, a natural approach is to perform rotation followed by addition on each ciphertext, as in the hybrid approach. This will take a total of n_i/l rotations and derive n_o/l ciphertexts. However, we argue that such ciphertext operations are not necessary and can be converted to plaintext to be performed. Our key insight is that the ciphertext result of the FC layer will be secretly shared with the client and server. Therefore, we require the server to generate random vectors (i.e., $\{\mathbf{r_i}\}_{i\in[8]}$ and $\{\mathbf{q_i}\}_{i\in[8]}$ in Step (d) of Fig. 3), and then subtract the random vectors from the corresponding ciphertexts. The server returns the subtracted ciphertexts to the client, which decrypts them and executes n_i/l plaintext rotation followed by addition operations on them to obtain its share. Similarly, the server gets its share by executing n_i/l plaintext rotation followed by addition operations on its random vectors. Compared to GAZELLE, our method only needs to perform $l-1$ rotations on the client's input, and does not require any rotations subsequently. This significantly improves the performance of matrix-vector computations.

We further reduce the computational cost by packing multiple copies of $\mathbf{t}$ in a single ciphertext (called $[\mathbf{t_{pack}}]_\mathbf{c}$). As a result, $[\mathbf{t_{pack}}]_\mathbf{c}$ has $\frac{n}{n_i}$ copies of $\mathbf{t}$ so that the server can perform ScMult operations on $[\mathbf{t_{pack}}]_\mathbf{c}$ with $\frac{n}{n_i}$ encoded vectors at once. The computational cost of our method is $\frac{n_i n_o}{n}$ homomorphic multiplication operations, $l-1$ rotation operations, and $\frac{n_i n_o}{n} - 1$ homomorphic addition operations. It outputs $n_i \cdot n_o/(l \cdot n)$ ciphertexts.

TABLE I
COMPUTATION COMPLEXITY OF EACH METHOD

Method	**#Rotation**	**#ScMul**	**#Add**
Naive	$n_o \log_2 n_i$	n_o	$n_o \log_2 n_i$
Hybrid	$\log_2 \frac{n}{n_o} + \frac{n_i n_o}{n} - 1$	$\frac{n_i n_o}{n}$	$\log_2 \frac{n}{n_o} + \frac{n_i n_o}{n} - 1$
Our method	$l-1$	$\frac{n_i n_o}{n}$	$\frac{n_i n_o}{n} - 1$

Remark 3.1: Table I shows the comparison of our method with existing approaches in computation complexity. It is obvious that our method has better complexity, especially for rotation operations. Note that GALA [18] also designs an improved version of GAZELLE [23] for matrix-vector multiplication, which has the computational complexity of $(\frac{n_i n_o}{n} - 1)$ for rotation operations. However, GALA is specially customized for secure inference under *semi-honest adversary model*. Moreover, our method still outperforms GALA in computation complexity.

Remark 3.2: We also describe the optimization of convolution operations for linear layers. In brief, we assume that the server has c_o plaintext kernels of size $k_w \times k_h \times c_i$, and the ciphertext input sent by the client to the server is c_i kernels of size $u_w \times u_h$. The server is required to perform homomorphic convolution operations between the ciphertext input and its own plaintext kernel to obtain the ciphertext output. To improve the parallel processing capabilities, GAZELLE proposes to pack the input data of c_n channels into one ciphertext and requires a total of $\frac{c_i(k_w k_h - 1)}{c_n}$ rotation operations to achieve convolution. We present an improved solution over GAZELLE to execute convolution operations. We design strategies to significantly reduce the number of rotation operations involved in computing the convolution between each of the $\frac{c_0 c_i}{c_n^2}$ blocks and the corresponding input channels (see Appendix D, available online). As a result, our method reduces the computation complexity with respect to rotations by a factor of $\frac{c_i}{c_n}$ compared to GAZELLE. Readers can refer to Appendix D for more details, available online.

IV. Nonlinear Layer Optimization

In this section, we describe our proposed optimization method for non-linear layers. We focus on the secure computation of the activation function ReLU, one of the most popular functions in non-linear layers of modern DNNs.

A. Overview

As shown in Section III, the output of each linear layer will be securely shared with the server and client (Fig. 9 and 10 in Appendix C, available online), and used as the input of the next non-linear layer to obtain authenticated shares of the output of the non-linear layer. Specifically, assume that the output of a linear layer is $\mathbf{u} = \langle\mathbf{u}\rangle_0 + \langle\mathbf{u}\rangle_1$, where $\langle\mathbf{u}\rangle_0$ and $\langle\mathbf{u}\rangle_1$ are the shares held by the server and the client, respectively. Then the functionality of the next non-linear layer is shown in Fig. 4. At the high level, the main difference between our method and SIMC comes from parsing ReLU as $f(\mathbf{u}) = \mathbf{u} \cdot sign(\mathbf{u})$ and

Function $f : \mathbb{F}_p \rightarrow \mathbb{F}_p$.
Input: S_0 holds $\langle \mathbf{u} \rangle_0 \in \mathbb{F}_p$ and a MAC key α uniformly chosen from $\mathbb{F}_p$. S_1 holds $\langle \mathbf{u} \rangle_1 \in \mathbb{F}_p$.
Output: S_b obtains $\{(\langle \alpha \mathbf{u} \rangle_b, \langle f(\mathbf{u}) \rangle_b, \langle \alpha f(\mathbf{u}) \rangle_b)\}$ for $b \in \{0, 1\}$.

Fig. 4. Functionality of the nonlinear layer

encapsulating only the non-linear part ($sign(\mathbf{u})$) into the GC. The sign function $sign(t)$ equals 1 if $t \geq 0$ and 0 otherwise. In this way, we can reduce the original GC size by about two-thirds, thereby further reducing the computation and communication costs incurred by running non-linear layers.

B. Our Protocol for the Non-Linear Layer

Similar to SIMC, our method can be divided into four phases: Garbled Circuit phase, Authentication phase 1, Local Computation phase, and Authentication phase 2. Assume that the server's (S_0) input is $(\langle \mathbf{u} \rangle_0, \alpha)$ and the client's (S_1) input is $(\langle \mathbf{u} \rangle_1)$. We provide a high-level view of the protocol. Fig. 5 gives a detailed technical description.

Garbled Circuit Phase: SIMC uses GC to calculate every bit of $\mathbf{u}$ and $ReLU(\mathbf{u})$, instead of directly calculating these values. This is efficient since it avoids incurring a large number of multiplication operations in the GC. Our method follows a similar logic, but instead of computing $ReLU(\mathbf{u})$, we compute every bit of $sign(\mathbf{u})$. In this phase, we can reduce the original GC size by about two-thirds. To achieve this, S_0 first constructs a garbled circuit for $booln^f$, where the input of the circuit is the share of $\mathbf{u}$ and its output is $(\mathbf{u}, sign(\mathbf{u}))$. S_1 evaluates this garbled circuit on $\langle \mathbf{u} \rangle_0$ and $\langle \mathbf{u} \rangle_1$ once the correct input labels are obtained using the OT protocol. At the end, S_1 learns the set of output labels for the bits of $\mathbf{u}$ and $sign(\mathbf{u})$. Note that the operation $\mathbf{Trun}$ is used to assist in implementing secure multiplication operations outside the GC, and it can be seen as a kind of random noise that can be eliminated if the client performs the GC evaluation correctly. As a result, it does not affect the accuracy of inference (see correctness analysis below).

Authentication Phase 1: In the previous phase, S_1 obtains the garbled output labels for each bit of $\mathbf{u}$ and $sign(\mathbf{u})$, denoted as $\mathbf{u}[i]$ and $sign(\mathbf{u})[i]$, respectively. The goal of this phase is to compute the shares of $\alpha \mathbf{u}[i]$, $sign(\mathbf{u})[i]$, and $\alpha sign(\mathbf{u})[i]$. We take $\alpha \mathbf{u}[i]$ as an example to briefly describe the procedure. Specifically, we observe that the shares of $\alpha \mathbf{u}[i]$ are either shares 0 or α, depending on whether $\mathbf{u}[i]$ is 0 or 1. Also, the output of the GC contains two output labels corresponding to each $\mathbf{u}[i]$ (each one for $\mathbf{u}[i] = 0$ and 1). Therefore, we denote these labels as $\mathtt{lab}_{i,0}^{out}$ and $\mathtt{lab}_{i,1}^{out}$ for $\mathbf{u}[i] = 0$ and $\mathbf{u}[i] = 1$, respectively. To calculate the shares of $\alpha \mathbf{u}[i]$, S_0 chooses a random number $\tau_i \in \mathbb{F}_p$ and converts it to $\mathtt{lab}_{i,0}^{out}$. Similarly, $\tau_i + \alpha$ is converted to $\mathtt{lab}_{i,1}^{out}$. S_0 sends the two ciphertexts to S_1, and sets its share of $\alpha \mathbf{u}[i]$ as $-\tau_i$. Since S_1 has obtained $\mathtt{lab}_{i,\mathbf{u}[i]}^{out}$ in the previous phase, it can obviously decrypt one of the two ciphertexts to obtain its share for $\alpha \mathbf{u}[i]$. Computing the share of $\alpha sign(\mathbf{u})[i]$ and $sign(\mathbf{u})[i]$ follows a similar method using the output labels for $sign(\mathbf{u})$.

Local Computation Phase: Given the shares of $\alpha \mathbf{u}[i]$, $sign(\mathbf{u})[i]$, and $\alpha sign(\mathbf{u})[i]$, this phase locally calculates the shares of $\alpha \mathbf{u}$, $sign(\mathbf{u})$, and $\alpha sign(\mathbf{u})$. Taking $\alpha \mathbf{u}$ as an example, as shown in Fig. 5, each party locally multiplies $\alpha \mathbf{u}[i]$ and 2^{i-1} and then sums these results to get the share on $\alpha \mathbf{u}$. The shares of $sign(\mathbf{u})$ and $\alpha sign(\mathbf{u})$ are calculated similarly.

Authentication Phase 2: This phase calculates the shares of $f(\mathbf{u}) = \mathbf{u} sign(\mathbf{u})$, and $\alpha f(\mathbf{u})$. Since each party holds the authenticated shares of $\mathbf{u}$ and $sign(\mathbf{u})$, it is easy to compute the authenticated shares of $f(\mathbf{u})$ for each party (see Fig. 5).

Remark 4.1: As described above, we reduce the size of the GC in SIMC by about two thirds. Therefore, instead of using the GC to calculate the entire ReLU function, we only encapsulate the non-linear part of ReLU into the GC. Then, we construct a lightweight alternative protocol that takes the output of the simplified GC as input to calculate the shares of the desired result. Therefore, compared to SIMC, SIMC 2.0 reduces the communication overhead of calculating each ReLU from $2c\lambda + 4\kappa\lambda + 6\kappa^2$ to $2e\lambda + 4\kappa\lambda + 6\kappa^2 + 2\kappa$, where $e < c$ denotes the number of AND gates required in the GC. Clearly, $c - e > 1$ since we only need to compute $sign(\mathbf{u})$ instead of $f(\mathbf{u}) = \mathbf{u} \cdot sign(\mathbf{u})$ in SIMC 2.0. Therefore, compared with SIMC, we save communication cost with $(2c\lambda + 4\kappa\lambda + 6\kappa^2) - (2e\lambda + 4\kappa\lambda + 6\kappa^2 + 2\kappa) = 2\lambda(c - e) - 2\kappa$. Based on the setting of $\lambda \geq 2\kappa$, it is obviously $(c - e)\lambda \geq \kappa$. Also, we experimentally demonstrate that this simplified GC improves the running efficiency by one third compared to the original one.

Remark 4.2: Our method can be easily extended to other non-linear functions, such as `Maxpool`. We follow the same idea to compute all the shares of `Maxpool`'s output, where we construct a Boolean circuit for `Maxpool` that feeds multiple inputs $(\mathbf{u}_1, \mathbf{u}_2, \ldots \mathbf{u}_\kappa)$, and outputs the reconstructed $(\mathbf{u}_1, \mathbf{u}_2, \ldots \mathbf{u}_\kappa)$ and $f(\mathbf{u}_1, \mathbf{u}_2, \ldots \mathbf{u}_\kappa)$.

Correctness: We analyze the correctness of our protocol in Fig. 5 as follows. Based on the correctness of the $\mathrm{OT}_\lambda^\kappa$, the client S_1 holds $\{\tilde{\mathtt{lab}}_i^{in} = \mathtt{lab}_{i,\langle \mathbf{u} \rangle_1[i]}\}_{i \in \{\kappa+1, \ldots 2\kappa\}}$. Since for $i \in [\kappa]$, $\{\tilde{\mathtt{lab}}_i^{in} = \mathtt{lab}_{i,\langle \mathbf{u} \rangle_0[i]}\}_{i \in [\kappa]}$, we can get $\tilde{\mathtt{lab}}_i^{out} = \mathtt{lab}_{i,\mathbf{u}[i]}^{out}$ and $\tilde{\mathtt{lab}}_{i+\kappa}^{out} = \mathtt{lab}_{i+\kappa,sign(\mathbf{u})[i]}^{out}$, with the correctness of (`Garble`, `GCEval`) for the circuit $booln^f$. Therefore, for $i \in [k]$, we have $\tilde{\xi}_i || \tilde{\zeta}_i = \xi_{i,\mathbf{u}[i]} || \zeta_{i,\mathbf{u}[i]}$ and $\tilde{\xi}_{i+\kappa} || \tilde{\zeta}_{i+\kappa} = \xi_{i+\kappa,sign(\mathbf{u})[i]} || \zeta_{i+\kappa,sign(\mathbf{u})[i]}$. We also have $c_i = ct_{i,\xi_{i,\mathbf{u}[i]}} \oplus \mathbf{Trun}_\kappa(\zeta_{i,\mathbf{u}[i]}) = \tau_{i,\mathbf{u}[i]}$ and $(d_i || e_i) = \hat{ct}_{i,\xi_{i+\kappa,sign(\mathbf{u})[i]}} \oplus \mathbf{Trun}_{2\kappa}(\zeta_{i+\kappa,sign(\mathbf{u})[i]}) = \rho_{i,sign(\mathbf{u})[i]} || \sigma_{i,sign(\mathbf{u})[i]}$. On the basis of these, we have

- $g_1 = \sum_{i \in [\kappa]} (c_i - \tau_{i,0}) 2^{i-1} = \sum_{i \in [\kappa]} \alpha(\mathbf{u}[i]) 2^{i-1} = \alpha \mathbf{u}$.
- $g_2 = \sum_{i \in [\kappa]} (d_i - \rho_{i,0}) 2^{i-1} = \sum_{i \in [\kappa]} (sign(\mathbf{u})[i]) 2^{i-1} = sign(\mathbf{u})$.
- $g_3 = \sum_{i \in [\kappa]} (e_i - \sigma_{i,0}) 2^{i-1} = \sum_{i \in [\kappa]} \alpha(sign(\mathbf{u})[i]) 2^{i-1} = \alpha sign(\mathbf{u})$.

Since each party holds the authenticated shares of $\mathbf{u}$ and $sign(\mathbf{u})$, we can easily calculate the shares of $f(\mathbf{u}) = \mathbf{u} sign(\mathbf{u})$, and $\alpha f(\mathbf{u})$. This concludes the proof of correctness.

Security: Our protocol for non-linear layers has the same security properties as SIMC, i.e., it is secure against the malicious client model. We provide the following theorem and prove it

Preamble: To compute the ReLU function $f : \mathbb{F}_p \rightarrow \mathbb{F}_p$, we consider a Boolean circuit $booln^f$ that takes the share of $\mathbf{u}$ as input and outputs $(\mathbf{u}, f(\mathbf{u}))$. In addition, we define a truncation function $\mathbf{Trun}_h : \{0,1\}^\lambda \rightarrow \{0,1\}^h$, which outputs the last h bits of the input. We require that λ satisfies $\lambda \geq 2\kappa$, which stems from the function $\mathbf{Trun}$ requiring (parts of) output labels of garbled circuit (i.e., λ-bit strings) to one-time pad values of length κ bits or 2κ bits. The purpose of this setting is for the smooth execution of $\mathbf{Trun}$, which is exactly the same as the original SIMC [12] setting.
Input:S_0 holds $\langle\mathbf{u}\rangle_0 \in \mathbb{F}_p$ and a MAC key α uniformly chosen from $\mathbb{F}_p$. S_1 holds $\langle\mathbf{u}\rangle_1 \in \mathbb{F}_p$.
Output: S_b obtains $\{(\langle\alpha\mathbf{u}\rangle_b, \langle f(\mathbf{u})\rangle_b, \langle\alpha f(\mathbf{u})\rangle_b)\}$ for $b \in \{0,1\}$.
Protocol:

1. Garbled Circuit Phase:
   - S_0 computes $\texttt{Garble}(1^\lambda, booln^f) \rightarrow (\texttt{GC}, \{\{\texttt{lab}_{i,j}^{in}\}_{i\in[2\kappa]}, \{\texttt{lab}_{i,j}^{out}\}_{i\in[2\kappa]}\}_{j\in\{0,1\}})$. Given the security parameter λ and a Boolean circuit $booln^f$, $\texttt{Garble}$ outputs a garbled circuit $\texttt{GC}$, an input set $\{\texttt{lab}_{i,j}^{in}\}_{i\in[2\kappa],j\in\{0,1\}}$ of labels, and an output set $\{\texttt{lab}_{i,j}^{out}\}_{i\in[2\kappa],j\in\{0,1\}}$, where each label is of λ-bits.
   - S_0 (as the sender) and S_1 (as the receiver) invoke the OT_λ^κ (see Section 2.5), where S_0's inputs are $\{\texttt{lab}_{i,0}^{in}, \texttt{lab}_{i,1}^{in}\}_{i\in\{\kappa+1,\cdots,2\kappa\}}$ while S_1's input is $\langle\mathbf{u}\rangle_1$. As a result, S_1 obtains $\{\tilde{\texttt{lab}}_i^{in}\}_{i\in\{\kappa+1,\cdots,2\kappa\}}$. In addition, S_0 sends the garbled circuit $\texttt{GC}$ and its garbled inputs $\{\tilde{\texttt{lab}}_i^{in} = \texttt{lab}_{i,\langle\mathbf{u}\rangle_0[i]}\}_{i\in[\kappa]}$ to S_1.
   - Given the $\texttt{GC}$ and the garbled inputs $\{\tilde{\texttt{lab}}_i^{in}\}_{i\in[2\kappa]}$, S_1 computes $\texttt{GCEval}(\texttt{GC}, \{\tilde{\texttt{lab}}_i^{in}\}_{i\in[2\kappa]}) \rightarrow \{\tilde{\texttt{lab}}_i^{out}\}_{i\in[2\kappa]}$.
2. Authentication Phase 1:
   - For every $i \in [\kappa]$, S_0 randomly selects $\rho_{i,0}$, $\sigma_{i,0}$ and $\tau_{i,0} \in \mathbb{F}_p$ and sets $(\rho_{i,1}, \sigma_{i,1}, \tau_{i,1}) = (1+\rho_{i,0}, \alpha+\sigma_{i,0}, \alpha+\tau_{i,0})$.
   - For every $i \in [2\kappa]$ and $j \in \{0,1\}$, S_0 parses $\{\texttt{lab}_{i,j}^{out}\}$ as $\xi_{i,j}||\zeta_{i,j}$ where $\xi_{i,j} \in \{0,1\}$ and $\zeta_{i,j} \in \{0,1\}^{\lambda-1}$.
   - For every $i \in [\kappa]$ and $j \in \{0,1\}$, S_0 sends $ct_{i,\xi_{i,j}}$ and $\hat{ct}_{i,\xi_{i+\kappa,j}}$ to S_1, where $ct_{i,\xi_{i,j}} = \tau_{i,j} \oplus \mathbf{Trun}_\kappa(\zeta_{i,j})$ and $\hat{ct}_{i,\xi_{i+\kappa,j}} = (\rho_{i,j}||\sigma_{i,j}) \oplus \mathbf{Trun}_{2\kappa}(\zeta_{i+\kappa,j})$.
   - For every $i \in [2\kappa]$, S_1 parses $\tilde{\texttt{lab}}_i^{out}$ as $\tilde{\xi}_i||\tilde{\zeta}_i$ where $\tilde{\xi}_i \in \{0,1\}$ and $\tilde{\zeta}_i \in \{0,1\}^{\lambda-1}$.
   - For every $i \in [\kappa]$, S_1 computes $c_i = ct_{i,\tilde{\xi}_i} \oplus \mathbf{Trun}_\kappa(\tilde{\zeta}_i)$ and $(d_i||e_i) = \hat{ct}_{i,\tilde{\xi}_{i+\kappa}} \oplus \mathbf{Trun}_{2\kappa}(\tilde{\zeta}_{i+\kappa})$.
3. Local Computation Phase:
   - S_0 outputs $\langle g_1\rangle_0 = (-\sum_{i\in[\kappa]} \tau_{i,0}2^{i-1})$, $\langle g_2\rangle_0 = (-\sum_{i\in[\kappa]} \rho_{i,0}2^{i-1})$ and $\langle g_3\rangle_0 = (-\sum_{i\in[\kappa]} \sigma_{i,0}2^{i-1})$.
   - S_1 outputs $\langle g_1\rangle_1 = (\sum_{i\in[\kappa]} c_i2^{i-1})$, $\langle g_2\rangle_1 = (\sum_{i\in[\kappa]} d_i2^{i-1})$ and $\langle g_3\rangle_1 = (\sum_{i\in[\kappa]} e_i2^{i-1})$.
4. Authentication Phase 2:
   - $S_b, b \in \{0,1\}$ randomly select a triplet of fresh authentication shares $\{(\langle A\rangle_b, \langle\alpha A\rangle_b), (\langle B\rangle_b, \langle\alpha B\rangle_b), (\langle C\rangle_b, \langle\alpha C\rangle_b)\}$ (see **Fig.** 8 for selection details), where triple $(A, B, C) \in \mathbb{F}_p^3$ satisfying $AB = C$.
   - S_0 interacts with S_1 to reveal $\Gamma = \mathbf{u} - A$ and $\Lambda = g_2 - B$.
   - $S_b, , b \in \{0,1\}$ computes the $\langle z_2\rangle_b = \langle\mathbf{u} \cdot sign(\mathbf{u})\rangle_b$ through $\langle z_2\rangle_b$=$\langle C\rangle_b + \Gamma \cdot \langle B\rangle_b + \Lambda \cdot \langle A\rangle_b + \Gamma \cdot \Lambda$. In addition, $\langle z_3\rangle_b = \langle\alpha\mathbf{u} \cdot sign(\mathbf{u})\rangle_b$ can be obtained in a similar way.

Fig. 5. Our protocol for the non-linear layer $\pi_{\texttt{Non-lin}}^f$

in Appendix E, available online following a similar logic of SIMC.

Theorem 1: Let ($\texttt{Garble}$, $\texttt{GCEval}$) be a garbling scheme with the properties defined in Section II-F. Authenticated shares have the properties defined in Section II-D. Then our protocol for non-linear layers is secure against any malicious adversary $\mathcal{A}$ corrupting the client S_1.

Proof: Please refer to Appendix E, available online. □

V. Secure Inference

In this section, we describe the details of our secure inference protocol (called π_{inf}). For simplicity, suppose that a neural network (NN) consists of alternating linear and nonlinear layers. Let the specifications of the linear layer be $\mathbf{N}_1, \mathbf{N}_2, \ldots, \mathbf{N}_m$, and the nonlinear layer be $f_1, f_2, \ldots, f_{m-1}$. Given an input vector $\mathbf{t}_0$, one needs to sequentially compute $\mathbf{u}_i = \mathbf{N}_i \cdot \mathbf{t}_{i-1}, \mathbf{t}_i = f_i(\mathbf{u}_i)$, where $i \in [m-1]$. As a result, we have $\mathbf{u}_m = \mathbf{N}_m \cdot \mathbf{t}_{m-1} = \text{NN}(\mathbf{t}_0)$. In secure inference, the server S_0' input is the weights of all linear layer, i.e., $\mathbf{N}_1, \ldots, \mathbf{N}_m$ while the input of S_1 is $\mathbf{t}$. The goal of π_{inf} is to learn $\text{NN}(\mathbf{t}_0)$ for the client S_1. We provide an overview of π_{inf} below and give details of the protocol in Fig. 6.

Our protocol can be roughly divided into two phases: the evaluation phase and the consistency check phase. We perform the computation of alternating linear and nonlinear layers with appropriate parameters in the evaluation phase. After that, the server performs a consistency check phase to verify the consistency of the calculations so far. The output will be released to the client if the check is successful. Specifically,

- *Linear Layer Evaluation:* To evaluate the first linear layer, S_0 and S_1 execute the function *InitLin* to learn $(\langle\mathbf{u}_1\rangle_b, \langle\mathbf{r}_1\rangle_b)$ for $b \in \{0,1\}$, where S_0's inputs for *InitLin* are $(\mathbf{N}_1, \alpha)$ while S_1's input is $\mathbf{t}_0$. This process is to compute the authenticated shares of $\mathbf{u}_1$, i.e., shares of $\mathbf{u}_1$ and $\mathbf{r}_1$. We use $\mathbf{r}_1$ to represent the authentication of $\mathbf{u}_1$. To evaluate the i-th linear layer ($i > 1$), S_0 and S_1 execute the function *Lin* to learn $(\langle\mathbf{u}_i\rangle_b, \langle\mathbf{r}_i\rangle_b, \langle\mathbf{z}_i\rangle_b)$ for $b \in \{0,1\}$, where S_0's inputs for *Lin* are $(\langle\mathbf{t}_{i-1}\rangle_0, \langle\mathbf{d}_{i-1}\rangle_0, \mathbf{N}_i, \alpha)$ while S_1's inputs are $(\langle\mathbf{t}_{i-1}\rangle_1, \langle\mathbf{d}_{i-1}\rangle_1)$. We use $\mathbf{d}_{i-1}$ to denote the authentication on $\mathbf{t}_{i-1}$. Therefore, *Lin* outputs shares of $\mathbf{u}_i = \mathbf{N}_i\mathbf{t}_{i-1}, \mathbf{r}_i = \mathbf{N}_i\mathbf{d}_{i-1}$ and an additional tag $\mathbf{z}_i = (\alpha^3\mathbf{t}_i - \alpha^2\mathbf{d}_{i-1})$.
- *Non-Linear Layer Evaluation:* To evaluate the i-th ($i \in [m-1]$) non-linear layer, S_0 and S_1 execute the function $\pi_{\texttt{Non-lin}}^{f_i}$ to learn $(\langle\mathbf{k}_i\rangle_b, \langle\mathbf{t}_i\rangle_b, \langle\mathbf{d}_i\rangle_b)$ for $b \in \{0,1\}$, where

Preamble: Consider a neural network (NN) consisting of m linear layers and $m-1$ non-linear layers. Let the specifications of the linear layer be $\mathbf{N}_1$, $\mathbf{N}_2$, $\cdots$, $\mathbf{N}_m$, and the nonlinear layer be f_1, f_2, $\cdots$, f_{m-1}.
Input: S_0 holds $\{\mathbf{N}_j \in \mathbb{F}_p^{n_j \times n_{j-1}}\}_{j\in[m]}$, *i.e.*, weights for the m linear layers. S_1 holds $\mathbf{t}_0 \in \mathbb{F}_p^{n_0}$ as the input of NN.
Output: S_1 obtains $\mathrm{NN}(\mathbf{t}_0)$.
Protocol:

1. S_0 uniformly selects a random MAC key α from $\mathbb{F}_p$ to be used throughout the execution of the protocol.
2. **First Linear Layer**: S_0 and S_1 execute the function **InitLin** to learn $(\langle \mathbf{u}_1\rangle_b, \langle \mathbf{r}_1\rangle_b)$ for $b \in \{0,1\}$, where S_0's inputs for **InitLin** are $(\mathbf{N}_1, \alpha)$ while S_1's input is $\mathbf{t}_0$.
3. For each $j \in [m-1]$,
   Non-Linear Layer f_j: S_0 and S_1 execute the function $\pi^{f_j}_{\mathtt{Non-lin}}$ to learn $(\langle \mathbf{k}_j\rangle_b, \langle \mathbf{t}_j\rangle_b, \langle \mathbf{d}_j\rangle_b)$ for $b\in\{0,1\}$, where S_0's inputs for $\pi^{f_j}_{\mathtt{Non-lin}}$ are $(\langle \mathbf{u}_j\rangle_0, \alpha)$ while S_1's input is $\langle \mathbf{u}_j\rangle_1$.
   Linear layer $j+1$: S_0 and S_1 execute the function **Lin** to learn $(\langle \mathbf{u}_{j+1}\rangle_b, \langle \mathbf{r}_{j+1}\rangle_b, \langle \mathbf{z}_{j+1}\rangle_b)$ for $b\in\{0,1\}$, where S_0's inputs for **Lin** are $(\langle \mathbf{t}_j\rangle_0, \langle \mathbf{d}_j\rangle_0, \mathbf{N}_{j+1}, \alpha)$ while S_1's inputs are $(\langle \mathbf{t}_j\rangle_1, \langle \mathbf{d}_j\rangle_1)$.
4. **Consistency Check**:
   - For $j\in[m-1]$, S_0 selects $\mathbf{s}_j \in_R \mathbb{F}_p^{n_j}$ and $\mathbf{s}'_j \in_R \mathbb{F}_p^{n_{j+1}}$. S_0 sends $(\mathbf{s}_j, \mathbf{s}'_j)$ to S_1.
   - S_1 computes $\langle \mathbf{q}\rangle_1 = \sum_{j\in[m-1]} \left((\langle \mathbf{r}_j\rangle_1 - \langle \mathbf{k}_j\rangle_1)\cdot \mathbf{s}_j + \langle \mathbf{z}_{j+1}\rangle_1 \cdot \mathbf{s}'_j \right)$ and sends it to S_0.
   - S_0 computes $\langle \mathbf{q}\rangle_0 = \sum_{j\in[m-1]} \left((\langle \mathbf{r}_j\rangle_0 - \langle \mathbf{k}_j\rangle_0)\cdot \mathbf{s}_j + \langle \mathbf{z}_{j+1}\rangle_0 \cdot \mathbf{s}'_j \right)$.
   - S_0 aborts if $\langle \mathbf{q}\rangle_0 + \langle \mathbf{q}\rangle_1 \mod p \neq 0$. Otherwise, sends $\langle \mathbf{u}_m\rangle_0$ to S_1.
5. **Output Phase**: S_1 outputs $\langle \mathbf{u}_m\rangle_0 + \langle \mathbf{u}_m\rangle_1 \mod p$ if S_0 did not abort in the previous step.

Fig. 6. Secure Inference Protocol π_{inf}

S_0's inputs for $\pi^{f_i}_{\mathtt{Non-lin}}$ are $(\langle \mathbf{u}_i\rangle_0, \alpha)$ while S_1's input is $\langle \mathbf{u}_i\rangle_1$, where we use $\mathbf{k}_i$ to represent another set of shares of authentication on $\mathbf{u}_i$.

- *Consistency Check Phase:* The server performs the following process to check the correctness of the calculation.
  – For each $i \in \{2,\ldots,m\}$, verify that the authentication share entered into the function *Lin* is valid by verifying that $\mathbf{z}_i = 0^{n_i - 1}$.
  – For each $i\in[m-1]$, check that the input of the function $\pi^{f}_{\mathtt{Non-lin}}$ is the same as the output of *Lin* under the i-th linear layer by verifying that $\mathbf{r}_i - \mathbf{k}_i = 0^{n_i}$.

Finally, all of the above checks can be combined into a single check by S_0 to pick up random scalars. If the check passes, the final prediction can be reconstructed by S_1, otherwise S_0 aborts and returns the final share to S_1.

Correctness: We briefly describe the correctness of our secure inference protocol π_{inf}. In more detail, we first have $\mathbf{u}_1 = \mathbf{N}_1 \cdot \mathbf{t}_0$ and $\mathbf{r}_1 = \alpha \mathbf{u}_1$ by the correctness of *InitLin*. Then, for each $i\in\{2,\ldots,m\}$, we have $\mathbf{u}_1 = \mathbf{N}_i \cdot \mathbf{t}_{i-1}$, $\mathbf{r}_i = \mathbf{N}_i \cdot \mathbf{d}_{i-1}$, and $\mathbf{z}_i = \alpha^3 \mathbf{t}_{i-1} - \alpha^2 \mathbf{d}_{i-1}$ by the correctness of *Lin*. Furthermore, for each $i\in[m-1]$, it holds that $\mathbf{k}_i = \alpha \mathbf{u}_i$, $\mathbf{t}_i = f_i(\mathbf{u}_i)$, and $\mathbf{d}_i = \alpha f_i(\mathbf{u}_i)$ by the correctness of $\pi^{f}_{\mathtt{Non-lin}}$. On the other hand, we can observe that $q=0$ since for each $i\in[m-1]$, $\mathbf{z}_{i+1} = 0$ and $\mathbf{r}_i = \mathbf{k}_i$. Finally, we have $\mathbf{u}_m = NN(\mathbf{t}_0)$.

Security: Our secure inference protocol has the same security properties as SIMC, i.e., it is secure against the malicious client model. We provide the following theorem.

Theorem 2: Our secure inference protocol π_{inf} is secure against a semi-honest server S_0 and any malicious adversary $\mathcal{A}$ that corrupts the client S_1.

Proof: Please refer to Appendix F, available online. □

VI. Performance Evaluation

In this section, we conduct experiments to demonstrate the performance of SIMC 2.0. Since SIMC 2.0 is derived from SIMC [12][2], the most advanced solution for secure inference under the client-malicious model, we take it as the baseline to compare computation and communication overhead. In more detail, we first describe the performance comparison of these two approaches for linear layer computation (including matrix-vector computation and convolution), and then discuss their overheads in secure computation of non-linear layers. Finally, we demonstrate the end-to-end performance advantage of SIMC 2.0 compared to SIMC for various mainstream DNN models (e.g., AlexNet, VGG-16, ResNet-18, ResNet-50, ResNet-100, and ResNet-152).

[2]Code is available at https://aka.ms/simc.

A. Implementation Details

Consistent with SIMC, SIMC 2.0 uses the homomorphic encryption library provided by SEAL [33] (the maximum number n of slots allowed for a single ciphertext is set to 4096) to realize the calculation of linear layers, and uses the components of the garbled circuit in the EMP toolkit [34] (with the OT protocol that resists active adversaries) to realize the execution of nonlinear layers. As a result, SIMC 2.0 provides 128 bits of computational security and 40 bits of statistical security. Like SIMC, our system is implemented on the 44-bit prime field. Other parameters, such as the configuration of the zero-knowledge proof protocol [24] and the random numbers used to verify the consistency of the calculation, are completely inherited from the SIMC settings (refer to [12] for more details). All running times refer to the average of 10 calculations. Our experiments are carried out in both the LAN and the WAN settings. LAN is implemented with two workstations in our lab. The client workstation has AMD EPYC 7282 1.4 GHz CPUs with 32 threads on 16 cores and 32 GB RAM. The server workstation has Intel(R) Xeon(R) E5-2697 v3 2.6 GHz CPUs with 28 threads on 14 cores and 64 GB RAM. The WAN setting is based on a connection between a local PC and an Amazon AWS server with an average bandwidth of 963 Mbps and running time of around 14 ms.

B. Performance of Linear Layers

We compare the cost of SIMC 2.0 and SIMC in the linear layer. As mentioned in Section III-C, given a $n_o \times n_i$-dimensional

TABLE II
COST OF MATRIX-VECTOR MULTIPLICATION

Dimension	Metric	# operations		Running time (ms)			
		SIMC	SIMC 2.0	SIMC LAN	SIMC WAN	SIMC 2.0 (Speedup) LAN	SIMC 2.0 (Speedup) WAN
1×4096	Rotation ScMult Add	12 1 12	**0** **1** **0**	14.4	39.9	**1.6 (8.8×)**	**28.5 (1.4×)**
2×2048	Rotation ScMult Add	11 1 11	**0** **1** **0**	13.4	38.7	**3.0 (4.4×)**	**27.6 (1.4×)**
4×1024	Rotation ScMult Add	10 1 10	**0** **1** **0**	13.1	38.8	**2.98 (4.4×)**	**27.7 (1.4×)**
8×512	Rotation ScMult Add	9 1 9	**0** **1** **0**	11.3	36.6	**3.8 2.97 (×)**	**28.1 (1.3×)**
16×256	Rotation ScMult Add	8 1 8	**0** **1** **0**	10.8	36.5	**3.01 (3.5×)**	**28 (1.3×)**
32×128	Rotation ScMult Add	7 1 7	**0** **1** **0**	10.1	35.7	**3.1 (3.3×)**	**29.1 (1.2×)**

TABLE III
COST OF CONVOLUTION COMPUTATION

Input	Kernel	Metric	#operations		Running time (s)			
			SIMC	Ours	SIMC LAN	SIMC WAN	SIMC 2.0 (Speedup) LAN	SIMC 2.0 (Speedup) WAN
16×16 @128	1×1 @128	Rotation ScMult Add	960 1024 1016	**120** **1024** **1016**	1.3	1.4	**0.23 (5.7×)**	**0.35 (4.0×)**
16×16 @2048	1×1 @512	Rotation ScMult Add	61140 65536 65504	**480** **65536** **65504**	79.9	81.0	**4.60 (17.4×)**	**5.13 (15.8×)**
16×16 @128	3×3 @128	Rotation ScMult Add	1024 9216 9208	**184** **9216** **9208**	2.6	2.8	**1.04 (2.5×)**	**1.22 (2.3×)**
16×16 @2048	5×5 @64	Rotation ScMult Add	10752 204800 204796	**3132** **204800** **204796**	41.3	42.0	**24.3 (1.7×)**	**26.3 (1.6×)**

matrix and a vector of length n_i, the matrix-vector multiplication in SIMC only produces one ciphertext, while our method derives $n_i \cdot n_o/(l \cdot n)$ ciphertexts. The smaller l, the greater the computational advantage our scheme obtains compared to SIMC. For simplicity of comparison, we set $l = n_i \cdot n_o/(n)$ to ensure that SIMC outputs only one ciphertext for matrix-vector multiplication, which makes SIMC and SIMC 2.0 have exactly the same communication overhead at the linear layer, so we can focus on the comparison of computation cost.

1) Matrix-Vector Multiplication: Table II provides the computation complexity of SIMC and SIMC 2.0 for matrix-vector multiplication with different matrix dimensions. We observe that SIMC 2.0 greatly reduces the number of most expensive rotation operations (zero time) in HE while SIMC requires up to 12 operations (which can be calculated using the formulas in Table I). In addition, SIMC 2.0 is very friendly to other homomorphic operations, including addition and multiplication. For example, the execution process involves only one multiplication compared to SIMC. The running time is also provided in Table II, which quantifies the entire time cost for one inference session, including client processing, server processing, and network latency. We observe that in the LAN setting, SIMC 2.0 can achieve speedup of up to $8.8\times$ for matrix-vector multiplication of different dimensions. This is mainly due to our optimization strategy for this operation. Specifically, we first divide the server's original $n_o \times n_i$-dimensional weight matrix into multiple submatrices and calculate each submatrix separately. It can effectively reduce the number of rotations required for the client's ciphertext from $n_o - 1$ to $l - 1$. Then we convert the subsequent rotation operations by summing in the ciphertext into the plaintext (see Section III-C), thereby removing all the homomorphic rotation operations required at this stage.

We observe that the speedup of SIMC 2.0 is relatively smaller under the WAN setting, which is mainly caused by the communication latency between the local client and the cloud server. This dominates the total running time compared to the computation cost of the lightweight HE. Particularly, the running time is round 14 milliseconds in our setting, while the optimized HE in SIMC 2.0 only takes about 1 to 3 milliseconds.

2) Convolution Computation: The complexity and running time of the convolution operation for different input sizes and kernel sizes are provided in Table III, where the encrypted input is a data sample of size $u_w \times u_h$ with c_i channels (denoted as $u_w \times u_h@c_i$) and the server holds kernels with the size of $k_w \times k_h@c_o$ and c_i channels per kernel. We observe that our method substantially reduces the number of rotation operations required to compute the convolution compared to SIMC. For example, SIMC 2.0 reduces the number of rotation operations by up to $127\times$ compared to SIMC, given the input size of $16 \times 16@2048$ and the kernel size of $1 \times 1@512$. This benefits from our designed kernel grouping method, which reduces the most expensive rotation operation by a factor of $\frac{c_i}{c_n}$ (refer to Appendix D for more details, available online). The results indicate that a large speedup can be obtained if the input has more channels and a small kernel size. This is very beneficial for modern models that commonly have such characteristics. For the running time, it is obvious that our method significantly accelerates the inference execution. In more detail, in the LAN setting, SIMC 2.0 achieves speedups of $5.7\times$, $17.4\times$, $2.5\times$, and $1.7\times$ compared to SIMC. In the WAN setting, SIMC 2.0 also shows similar performance superiority.

C. Performance of Non-Linear Layers

We compare the computation and communication overheads of SIMC and our SIMC 2.0 in the implementation of non-linear layers. We mainly show the cost required by the two schemes to securely execute different numbers of ReLU functions.

1) Computation Overhead: Fig. 7(a) and (b) show the running time of SIMC and SIMC 2.0 for different numbers of ReLU functions in the LAN and WAN settings, respectively. We observe that SIMC 2.0 reduces the running time by about one-third compared to SIMC. Our improvement mainly comes from the optimization mechanism designed for the computation of non-linear layers. As mentioned earlier, compared to SIMC, we only encapsulate the non-linear part of ReLU into the GC. The goal of this strategy is to substantially reduce the number of AND gates required in the GC (AND operations are usually expensive in the GC). For example, given GC's security parameter $\lambda = 128$ bits and 44-bit prime field, SIMC needs 249 AND gates to calculate ReLU, while we only need 161 AND gates. This saves 88 AND gate operations! Moreover, the alternative

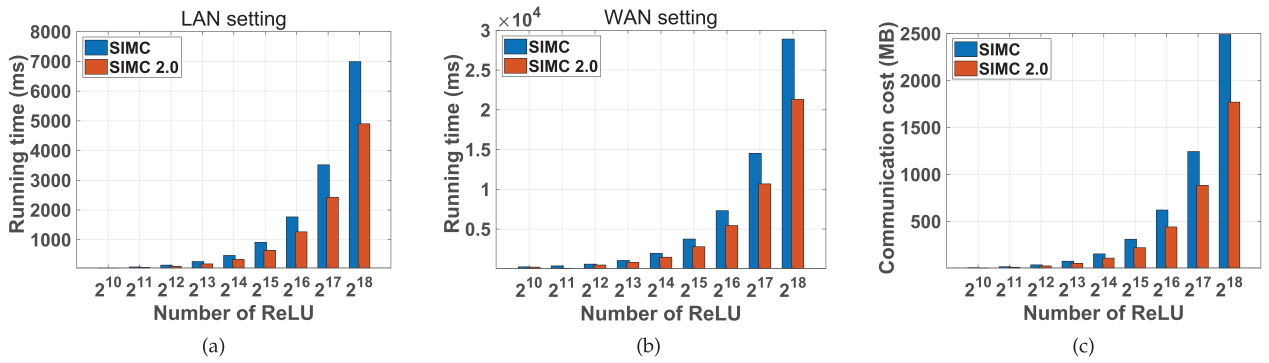

Fig. 7. Comparison of the nonlinear layer overhead. (a) Running time under the LAN setting. (b) Running time under the WAN setting. (c) Communication overhead with different numbers of ReLU functions.

protocol we designed is also resource-friendly, requiring only the server and client to exchange two elements and perform simple mathematical operations locally.

2) Communication Overhead: Fig. 7(c) provides the communication overhead incurred by SIMC and our method in computing different numbers of ReLU functions. We observe that SIMC 2.0 reduces the communication overhead of SIMC by approximately one-third. This also benefits from our optimized GC to perform non-linear operations. In more detail, given GC's security parameter $\lambda = 128$ bits and 44-bit prime field, SIMC requires to communicate 11.92 KB data to perform one ReLU. In contrast, since our optimized GC saves about 88 AND gates, which needs 2.69 KB communication data for each ReLU, our SIMC 2.0 only requires 9.21 KB data transfer for each ReLU. Note that such communication advantages are non-trivial since mainstream DNN models usually contain tens of thousands of ReLU functions.

TABLE IV
COST OF END-TO-END MODEL INFERENCE

Model	Metric	#operations		Running time (s)				Comm. (GB)	
		SIMC	Ours	SIMC		SIMC 2.0 (Speedup)		SIMC	Ours
				LAN	WAN	LAN	WAN		
MLP	Rotation	70	**55**	0.14	0.21	**0.11 (1.3×)**	**0.18 (1.2×)**	<0.01	<0.01
	ScMult	56	**56**						
	Add	70	**55**						
AlexNet	Rotation	72549	**4394**	81.0	85.8	**35.2 (2.3×)**	**47,7 (1.8×)**	0.42	0.29
	ScMult	931977	**931977**						
	Add	931643	**931630**						
VGG-16	Rotation	315030	**10689**	72.4	97.6	**34.5 (2.1×)**	**54.2 (1.8×)**	5.81	4.13
	ScMult	3677802	**3677802**						
	Add	3676668	**3676654**						
ResNet-18	Rotation	234802	**9542**	77.1	118.3	**35.0 (2.2×)**	**65.6 (1.8×)**	4.27	3.04
	ScMult	2737585	**2737585**						
	Add	2736632	**2736624**						
ResNet-50	Rotation	2023606	**50666**	434.8	683.9	**111.3 (3.9×)**	**296.4 (2.3×)**	29.50	21.00
	ScMult	5168181	**5168181**						
	Add	5162524	**5162518**						
ResNet-101	Rotation	3947190	**113770**	802.3	1195.2	**186.5 (4.3×)**	**478.1 (2.5×)**	46.13	31.85
	ScMult	9903157	**9903157**						
	Add	9890972	**9890964**						
ResNet-152	Rotation	5533878	**169450**	1209.2	1780.1	**274.8 (4.4×)**	**684.6 (2.6×)**	64.25	46.28
	ScMult	13802549	**13802549**						
	Add	13784604	**13784596**						

D. Performance of End-to-End Secure Inference

Finally we compare the computation and communication costs of SIMC and SIMC 2.0 for different complete DNN models. We employ 7 ML models in our experiments: (1) a multi-layer perceptron with the network structure of 784-128-128-10, which is often used for benchmarking private-preserving model deployment (e.g., SecureML [5], MiniONN [35], GAZELLE). This model is trained with the MNIST dataset. (2) Some mainstream CNN models: AlexNet [36], VGG-16 [37], ResNet-18 [38], ResNet-50 [38], ResNet-101 [38], and ResNet-152 [38]. These models are trained over the CIFAR-10 dataset.

Table IV provides the computation complexity of SIMC and our method for different models. It is clear that SIMC 2.0 reduces the number of rotation operations in SIMC by $16.5\times$, $29.4\times$, $24.6\times$, $39.9\times$, $34.6\times$ and $50.5\times$ for AlexNet, VGG-16, ResNet-18, ResNet-50, ResNet-10, and ResNet-152, respectively. The fundamental reason for this improvement is that our designed optimization technique is for HE-based linear operations. We observe that SIMC 2.0 does not significantly reduce the number of rotation operations in the MLP model. It stems from the small ratio between the number of slots and the dimension of the output in the MLP, which limits the performance gain. Table IV also shows the running time and the corresponding speedup of our scheme. Since we substantially reduce the number of expensive rotation operations in the linear layer, and design an optimized GC for the nonlinear layer, our method obtains speedups of $2.3\times$, $2.1\times$, $2.2\times$, $3.9\times$, $4.3\times$ and $4.4\times$ for AlexNet, VGG-16, ResNet-18, ResNet-50, ResNet-101, and ResNet-152, respectively, under the LAN Setting. A similar performance boost is obtained under the WAN setting. Due to network latency in the WAN, the running time of both SIMC and our method is inevitably increased. This can reduce the performance advantage of SIMC 2.0, but our solution is still better than SIMC.

It has been demonstrated that for the HE-GC based secure inference model, the communication overhead caused by the non-linear layer dominates the communication in SIMC (refer to Seciton 5 in SIMC [12]). In our experiments, the communication cost of linear layers in SIMC 2.0 remains consistent with SIMC, while the overhead of nonlinear layers is reduced by two-thirds. Therefore, we observe that the size of communication data required by SIMC 2.0 is still smaller than that of SIMC, remaining approximately between $2/3 \sim 3/4$ and the original size, depending on the model size. For example, for the ResNet-152 model, SIMC requires to transfer 64.25 GB data to perform an

TABLE V
COMPARISON OF MATRIX-VECTOR MULTIPLICATION

Dimension	Running time (ms)				Comm.(MB)	
	Cheetah		SIMC 2.0		Cheetah	SIMC 2.0
	LAN	WAN	LAN	WAN		
1024×2048	179	201	413	428	1.84	0.51
512×1024	48	67	103	121	0.62	0.16

TABLE VI
COMPARISON OF CONVOLUTION COMPUTATION

Input	Kernel	Running time (s)				Comm.(MB)	
		Cheetah		SIMC 2.0		Cheetah	SIMC 2.0
		LAN	WAN	LAN	WAN		
224×224 @3	3×3 @64	2.27	2.59	3.14	3.59	51.63	71.06
56×56 @64	1×1 @256	0.91	1.23	1.82	2.47	15.51	27.07
56×56 @256	1×1 @64	0.81	1.14	2.13	2.77	17.13	43.02

inference process while our SIMC 2.0 saves about 18 GB in the communication overhead, which is quite impressive.

E. Comparison With Other State-of-the-Art Work

We find a recent work by Cheetah [20], which provides a state-of-the-art homomorphic encryption-based solution for linear layer operations in neural networks [39]. In order to distinguish our work from that of Cheetah, here we first demonstrate the difference in technical design of the two methods, and experimentally compare the performance of the two methods in terms of computation and communication overhead.

Theoretical Difference: Cheetah encodes the plaintext vector onto a polynomial, where each coefficient of the polynomial represents a plaintext element. In contrast, in the original homomorphic encryption, the combination of all coefficients of a polynomial represents only one plaintext element. For instance, let us consider the matrix-vector inner product operation. Cheetah uses the principle of polynomial multiplication, where some coefficients of the new polynomials obtained after multiplication are the inner products between the rows of the matrix and the vectors. Cheetah then extracts the corresponding coefficients of the polynomial using the conversion technique between LWE ciphertext and RLWE ciphertext, which yields the desired result. Similarly, Cheetah performs convolution operations following the same principles described above but with specific optimizations. Consequently, Cheetah only involves multiplication operations and eliminates all rotation operations when performing linear operations on machine learning models.

In contrast, our method does not alter any primitives in homomorphic encryption. Instead, we focus on exploiting the intrinsic connection between secret sharing and homomorphic encryption to design optimization methods that minimize the number of rotations required in linear computation. For example, when performing the inner product operation between a matrix and a vector, we leverage the property of secret sharing to transform the rotation operation in a large number of ciphertexts from the ciphertext to the plaintext environment. Specifically, the ciphertext output from the fully connected (FC) layer is secretly shared with the client and server, and used as the input of the next nonlinear layer function [40]. Since the shares are in plaintext, we can entirely convert the rotation followed by addition to be performed under plaintext. This transformation significantly reduces the computational complexity.

Quantitative Comparison: We also conducted experimental comparisons between the computational and communication overheads of the two schemes in performing linear operations (shown in Table V and Table VI). Our observations show that Cheetah outperforms our SIMC 2.0 in terms of computational overhead. The primary reason is that Cheetah entirely eliminates the rotation operation in the linear homomorphic operation, whereas our method still requires some rotation operations. Additionally, the communication overhead required by Cheetah in the convolution process is lower than that of our method, primarily due to the lightweight plaintext encoding method designed by Cheetah, reducing the size of the ciphertext sent by the server to the user. However, the communication overhead of SIMC 2.0 in the matrix-vector multiplication operation is significantly lower than that of Cheetah. This is because SIMC uses Single Instruction Multiple Data (SIMD) technology to maximize the use of plaintext slots, while the same operation in Cheetah has to sacrifice many redundant coefficient items, thus requiring the server to send more ciphertext to the user.

We would like to clarify that although Cheetah is generally more efficient than SIMC 2.0 in linear operations, applying Cheetah to our scenario requires substantial modification of SIMC 2.0. This is primarily because Cheetah operates on rings, whereas our optimization methods for nonlinear operations operate on the field of prime numbers. Therefore, if we want to use Cheetah to replace the linear part of SIMC 2.0, non-trivial modifications to the protocol of the nonlinear part in SIMC 2.0 are necessary to accommodate ring distributions. While this is an interesting problem, we currently do not have a good solution and leave it for future work.

VII. CONCLUSION

In this paper, we proposed SIMC 2.0, which significantly improves the performance of SIMC for secure ML inference in a threat model with a malicious client and honest-but-curious server. We designed a new coding method to minimize the number of costly rotation operations during homomorphic parallel matrix and vector computations in the linear layers. We also greatly reduced the size of the GC in SIMC to substantially speed up operations in the non-linear layers. In the future, we will focus on designing more efficient optimization strategies to further reduce the computation overhead of SIMC 2.0, to make secure ML inference more suitable for a wider range of practical applications.

REFERENCES

[1] J. Katz et al., "Optimizing authenticated garbling for faster secure two-party computation," in *Proc. Annu. Int. Cryptol. Conf.*, Springer, 2018, pp. 365–391.
[2] X. Wang, A. J. Malozemoff, and J. Katz, "Faster secure two-party computation in the single-execution setting," in *Proc. Annu. Int. Conf. Theory Appl. Cryptogr. Techn.*, Springer, 2017, pp. 399–424.

[3] D. Rathee et al., "CrypTFlow2: Practical 2-party secure inference," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2020, pp. 325–342.
[4] A. Patra et al., "ABY2. 0: Improved mixed-protocol secure two-party computation," in *Proc. USENIX Secur. Symp.*, 2021, pp. 2165–2182.
[5] P. Mohassel and Y. Zhang, "SecureML: A system for scalable privacy-preserving machine learning," in *Proc. IEEE Symp. Secur. Privacy*, 2017, pp. 19–38.
[6] G. Xu, G. Li, S. Guo, T. Zhang, and H. Li, "Secure decentralized image classification with multiparty homomorphic encryption," *IEEE Trans. Circuits Syst. Video Technol.*, to be published, doi: 10.1109/TCSVT.2023.3234278.
[7] M. Bellare, V. T. Hoang, and P. Rogaway, "Foundations of garbled circuits," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 784–796.
[8] R. LaVigne et al., "Topology-hiding computation beyond semi-honest adversaries," in *Proc. Theory Cryptogr. Conf.*, Springer, 2018, pp. 3–35.
[9] S. U. Hussain et al., "COINN: Crypto/ML codesign for oblivious inference via neural networks," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2021, pp. 3266–3281.
[10] Q. Lou and L. Jiang, "HEMET: A homomorphic-encryption-friendly privacy-preserving mobile neural network architecture," in *Proc. Int. Conf. Mach. Learn.*, PMLR, 2021, pp. 7102–7110.
[11] R. Lehmkuhl et al., "Muse: Secure inference resilient to malicious clients," in *Proc. USENIX Secur. Symp.*, 2021, pp. 2201–2218.
[12] N. Chandran et al., "SIMC: ML inference secure against malicious clients at semi-honest cost," in *Proc. USENIX Secur. Symp.*, 2022, pp. 1361–1378.
[13] A. Patra and A. Suresh, "BLAZE: Blazing fast privacy-preserving machine learning," in *Proc. Netw. Distrib. Syst. Secur.*, 2020, pp. 1–18.
[14] I. Damgård, D. Escudero, T. Frederiksen, M. Keller, P. Scholl, and N. Volgushev, "New primitives for actively-secure MPC over rings with applications to private machine learning," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 1102–1120.
[15] D. Escudero et al., "Improved primitives for MPC over mixed arithmetic-binary circuits," in *Proc. Annu. Int. Cryptol. Conf.*, Springer, 2020, pp. 823–852.
[16] C. Hazay and A. Yanai, "Constant-round maliciously secure two-party computation in the ram model," *J. Cryptol.*, vol. 32, no. 4, pp. 1144–1199, 2019.
[17] P. Mishra et al., "Delphi: A cryptographic inference service for neural networks," in *Proc. USENIX Secur. Symp.*, 2020, pp. 2505–2522.
[18] Q. Zhang, C. Xin, and H. Wu, "GALA: Greedy computation for linear algebra in privacy-preserved neural networks," in *Proc. Netw. Distrib. Syst. Secur.*, 2021, pp. 1–16.
[19] S. Sav et al., "POSEIDON: Privacy-preserving federated neural network learning," in *Proc. Netw. Distrib. Syst. Secur.*, 2021, pp. 1–18.
[20] Z. Huang, W.-j.C. LuHong, and J. Ding, "Cheetah: Lean and fast secure two-party deep neural network inference," in *Proc. USENIX Secur. Symp.*, 2022, pp. 809–826.
[21] X. Jiang et al., "Secure outsourced matrix computation and application to neural networks," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 1209–1222.
[22] Z. Huang, C. Hong, W.-J. Lu, C. Weng, and H. Qu, "More efficient secure matrix multiplication for unbalanced recommender systems," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 1, pp. 551–562, Jan./Feb. 2023.
[23] C. Juvekar et al., "{GAZELLE}: A low latency framework for secure neural network inference," in *Proc. USENIX Secur. Symp.*, 2018, pp. 1651–1669.
[24] H. Chen et al., "Maliciously secure matrix multiplication with applications to private deep learning," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, Springer, 2020, pp. 31–59.
[25] N. Chandran et al., "EzPC: Programmable and efficient secure two-party computation for machine learning," in *Proc. IEEE Eur. Symp. Secur. Privacy*, 2019, pp. 496–511.
[26] D. Demmler, T. Schneider, and M. Zohner, "ABY—A framework for efficient mixed-protocol secure two-party computation," in *Proc. Netw. Distrib. Syst. Secur.*, 2015, pp. 1–15.
[27] X. Wang, S. Ranellucci, and J. Katz, "Authenticated garbling and efficient maliciously secure two-party computation," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 21–37.
[28] C. Gentry, "A fully homomorphic encryption scheme," Stanford, CA, USA: Stanford Univ., 2009.
[29] S. Halevi and V. Shoup, "Algorithms in HElib," in *Proc. Annu. Cryptol. Conf.*, Springer, 2014, pp. 554–571.
[30] M. Keller, E. Orsini, and P. Scholl, "Actively secure OT extension with optimal overhead," in *Proc. Annu. Cryptol. Conf.*, Springer, 2015, pp. 724–741.
[31] Z. Ghodsi, N. K. Jha, B. Reagen, and S. Garg, "Circa: Stochastic ReLUs for private deep learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2021, pp. 2241–2252.
[32] S. Zahur, M. Rosulek, and D. Evans, "Two halves make a whole," in *Proc. Annu. Int. Conf. Theory Appl. Cryptogr. Techn.*, Springer, 2015, pp. 220–250.
[33] "SEAL Microsoft (release 3.3)," Microsoft Research, Redmond, WA, Jun. 2019. [Online]. Available: https://github.com/Microsoft/SEAL
[34] X. Wang, A. J. Malozemoff, and J. Katz, "EMP-Toolkit: Efficient multiparty computation toolkit," 2016. [Online]. Available: https://github.com/emp-toolkit
[35] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious neural network predictions via minionn transformations," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 619–631.
[36] W. Yu et al., "Visualizing and comparing alexnet and VGG using deconvolutional layers," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 1–7.
[37] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. Int. Conf. Learn. Representations*, 2015, pp. 1–14.
[38] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[39] Y. Lindell, "How to simulate it–a tutorial on the simulation proof technique," in *Tutorials on the Foundations of Cryptography*, Berlin, Germany: Springer, 2017, pp. 277–346.
[40] M. Keller, V. Pastro, and D. Rotaru, "Overdrive: Making SPDZ great again," in *Proc. Annu. Int. Conf. Theory Appl. Cryptogr. Techn.*, Springer, 2018, pp. 158–189.

Guowen Xu (Member, IEEE) received the PhD degree from the University of Electronic Science and Technology of China, in 2020. He is currently a research fellow with Nanyang Technological University, Singapore. He has published papers in reputable conferences/journals, including ACM CCS, NeurIPS, ASIACCS, ACSAC, ESORICS, *IEEE Transactions on Information Forensics and Security*, and *IEEE Transactions on Dependable and Secure Computing*. His research interests include applied cryptography and privacy-preserving Deep Learning.

Xingshuo Han is currently working toward the PhD degree with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. He has published papers in reputable conferences/journals, including ACM MM, *IEEE Transactions on Intelligent Transportation Systems* and *IEEE Transactions on Dependable and Secure Computing*. His research interests include safety and privacy of deep learning, safety and security of autonomous vehicles, and intelligent transportation systems.

Tianwei Zhang (Member, IEEE) received the bachelor's degree from Peking University, in 2011, and the PhD degree from Princeton University, in 2017. He is an assistant professor with the School of Computer Science and Engineering, Nanyang Technological University. His research focuses on computer system security. He is particularly interested in security threats and defenses in machine learning systems, autonomous systems, computer architecture and distributed systems.

Shengmin Xu is currently an associate professor with the Fujian Provincial Key Laboratory of Network Security and Cryptology, College of Computer and Cyber Security, Fujian Normal University, Fuzhou, China. Previously, he was a senior research engineer with the School of Computing and Information Systems, Singapore Management University. His research interests include cryptography and information security.

Jianting Ning (Member, IEEE) received the Ph.D. degree from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, in 2016. He is currently a professor with the Fujian Provincial Key Laboratory of Network Security and Cryptology, College of Mathematics and Computer Science, Fujian Normal University, China. Previously, he was a research scientist with the School of Information Systems, Singapore Management University and a research fellow with the Department of Computer Science, National University of Singapore. His research interests include applied cryptography and information security. He has published papers in major conferences/journals such as ACM CCS, ESORICS, ACSAC, *IEEE Transactions on Intelligent Transportation Systems*, *IEEE Transactions on Dependable and Secure Computing*, etc

Xinyi Huang is currently an associate professor with the Thrust of Artificial Intelligence, Information Hub, Hong Kong University of Science and Technology (Guangzhou), China. His research interests include cryptography and information security. He is in the Editorial Board of *International Journal of Information Security* and SCIENCE CHINA Information Sciences. He has served as the program/general chair or program committee member in more than 120 international conferences.

Hongwei Li (Senior Member, IEEE) is currently the head and a professor with the Department of Information Security, School of Computer Science and Engineering, University of Electronic Science and Technology of China. His research interests include network security and applied cryptography. He is the distinguished lecturer of IEEE Vehicular Technology Society.

Robert H. Deng (Fellow, IEEE) is AXA chair professor of Cybersecurity, Singapore Management University. His research interests are in the areas of data security and privacy, cloud security and IoT security. He served on many editorial boards and conference committees, including the editorial boards of *IEEE Security & Privacy Magazine*, *IEEE Transactions on Dependable and Secure Computing*, *IEEE Transactions on Information Forensics and Security*, and Steering Committee chair of the ACM Asia Conference on Computer and Communications Security.