VerifyML: Obliviously Checking Model Fairness Resilient to Malicious Model Holder

Guowen Xu, Xingshuo Han, Gelei Deng, Tianwei Zhang, Shengmin Xu, Jianting Ning, Anjia Yang, and Hongwei Li, *Senior Member, IEEE*

***Abstract*— In this article, we present *VerifyML*, the first secure inference framework to check the fairness degree of a given Machine learning (ML) model. *VerifyML* is generic and is immune to any obstruction by the malicious model holder during the verification process. We rely on secure two-party computation (2 PC) technology to implement *VerifyML*, and carefully customize a series of optimization methods to boost its performance for both linear and nonlinear layer execution. Specifically, (1) *VerifyML* allows the vast majority of overhead to be performed offline, thus meeting the low latency requirements for online inference. (2) To speed up offline preparation, we first design novel homomorphic parallel computing techniques to accelerate the authenticated Beaver's triple (including matrix-vector and convolution triples) generation procedure. It achieves up to $1.7\times$ computation speedup and gains at least $10.7\times$ less communication overhead compared to state-of-the-art work. (3) We also present a new cryptographic protocol to evaluate the activation functions of non-linear layers, which is $4\times$–$42\times$ faster and has $> 48\times$ less communication than the existing 2 PC protocol against malicious parties. In fact, *VerifyML* even beats the state-of-the-art semi-honest ML secure inference system! We provide a formal theoretical analysis for *VerifyML* security and demonstrate its performance superiority on mainstream ML models including ResNet-18 and LeNet.**

***Index Terms*—Privacy protection, deep learning, cryptography.**

I. Introduction

MACHINE learning (ML) systems are increasingly being used to inform and influence people's decisions, with algorithmic outcomes that can have powerful implications for individuals and society. For instance, automated ML tools are often used to calculate personal loan default risks. This approach speeds up the decision-making process significantly. However, as with any decision-making algorithm, there is a tendency to provide accurate results for the majority, which can leave certain individuals and minority groups at a disadvantage [1], [44]. This problem is widely defined as the unfairness of the ML model, which often stems from inherent human bias in the training samples. A trained ML model can amplify this bias, causing discriminatory decisions about certain groups and individuals.

The unfairness of ML models is not limited to financial risk control but is present in every corner of society. One prime example is COMPAS [20], an automated software used in US courts to assess the likelihood of criminals reoffending. An investigation of the software reveals a bias against African-Americans, with COMPAS having a higher false positive rate for African-American offenders than white criminals due to incorrect risk estimation. Similar decision biases can be found in other real-world applications, including childcare systems [7], employment matching [36], AI chatbots, and ad-serving algorithms [17]. As mentioned earlier, these unfair decisions result from neglected biases and discrimination hidden in data and algorithms.

To alleviate the above problems, a series of recent works [4], [26], [34], [35], [37] have proposed for formalizing measures of fairness for classification models, as well as their variants, in aim to provide instructions for verifying the fairness of a given model. Several evaluation tools have also been released that facilitate automated checks for discriminatory decisions in a given model. For example, Aequitas [39] as a toolkit provides testing of models against several bias and fairness metrics corresponding to different population subgroups. It feeds back test reports to developers, researchers and governments to assist them in making conscious decisions to avoid tending to harm specific population groups. IBM also offers a toolkit AI Fairness 360 [3], which aims to bringing fairness research algorithms to the industrial setting, creating a benchmark where all fairness algorithms can be evaluated, and providing an environment for researchers to share their ideas.

Existing efforts in theory and tools have led the entire research community to work towards unbiased verification of the ML model fairness. However, existing verification mechanisms either require to white-box access the target model or require clients to send queries in plaintext to the model holder, which is impractical as it incurs a range of privacy concerns.

Manuscript received 7 October 2022; revised 17 May 2023; accepted 26 June 2023. Date of publication 29 June 2023; date of current version 11 July 2024. This work was supported by the Singapore Ministry of Education (MOE) AcRF Tier 2 under Grant MOE-T2EP20121-0006. The work of Anjia Yang was supported in part by the National Key Research and Development Program of China under Grant 2021ZD0112802, and in part by the National Natural Science Foundation of China under Grant 62072215. *(Corresponding author: Tianwei Zhang.)*

Guowen Xu, Xingshuo Han, Gelei Deng, and Tianwei Zhang are with the School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798 (e-mail: guowen.xu@foxmail.com; xingshuo001@e.ntu.edu.sg; gdeng003@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 350007, China (e-mail: smxu1989@gmail.com; jtning88@gmail.com).

Anjia Yang is with the College of Cyber Security, Jinan University, Guangzhou 510632, China (e-mail: anjiayang@gmail.com).

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).

This article has supplementary downloadable material available at https://doi.org/10.1109/TDSC.2023.3290562, provided by the authors.

Digital Object Identifier 10.1109/TDSC.2023.3290562

Specifically, model holders are often reluctant to disclose model details because training a commercial model requires a lot of human cost, resources, and experience. Therefore, ML models, as precious intellectual property rights, need to be properly protected to ensure the company's competitiveness in the market. On the other hand, the queries that clients used to test model fairness naturally contain sensitive information, including loan records, disease history, and even criminal information. These highly private data should clearly be guaranteed confidentiality throughout the verification process. Hence, these requirements for privacy raises a challenging but meaningful question:

Can we design a verification framework that only returns the fairness of the model to the client and the parties cannot gain any private information?

We materialize the above question to a scenario where a client interacts with the model holder to verify the fairness of the model. Specifically, before using the target model's inference service, the client sends a set of queries for testing fairness to the model holder, which returns inference results to the client enabling it to locally evaluate how fair the model is. In such a scenario, the client is generally considered to be semi-honest since it needs to evaluate the model correctly for subsequent service. The model holder may be malicious, it may trick the client into believing that the model is of high fairness by arbitrarily violating the verification process. A natural solution to tackle such concerns is to leverage state-of-the-art generic 2 PC tools [6], [22], [24] that provide *malicious security*. It guarantees that if either entity behaves maliciously, they will be caught and the protocol aborted, protecting privacy. However, direct grafting of standard tools incurs enormous redundant overhead, including heavy reliance on zero-knowledge proofs [12], tedious computational authentication and interaction [16] (see Section III for more details).

To reduce the overhead, we propose *VerifyML*, a 2PC-based secure verification framework implemented on the *model holder-malicious* threat model. In this model, the client is considered semi-honest but the model holder is malicious and can arbitrarily violate the specification of the protocol. We adaptively customize a series of optimization methods for *VerifyML*, which show much better performance than the fully malicious baseline. Our key insight is to move the vast majority of operations to the client to bypass cumbersome data integrity verification and reduce the frequency of interactions between entities. Further, we design highly optimized methods to perform linear and nonlinear layer functions for ML, which brings at least $4-40\times$ speedup compared to state-of-the-art techniques. Overall, our contributions are as follows:

- We leverage the hybrid combination of HE-GC to design *VerifyML*. In *VerifyML*, the execution of ML's linear layer is implemented by homomorphic encryption (HE) while the non-linear layer is performed by the garbled circuit (GC). VerifyML allows more than 95% of operations to be completed in the offline phase, thus providing very low latency in the online inference phase. Actually, VerifyML's online phase even beats DELPHI [32], the state-of-the-art scheme for secure ML inference against only semi-honest adversaries.
- We design a series of optimization methods to reduce the overhead of the offline stage. Specifically, we design new homomorphic parallel computation methods, which are used to generate authenticated Beaver's triples, including matrix-vector and convolution triples, in a Single Instruction Multiple Data (SIMD) manner. Compared to existing techniques, we generate triples of matrix-vector multiplication without any homomorphic rotation operation, which is very computationally expensive compared to other homomorphic operations including addition and multiplication. Besides, we reduce the communication complexity of generating convolution triples (aka matrix multiplication triples) from cubic to quadratic with faster computing performance.
- We design computationally-friendly GC to perform activation functions of nonlinear layers (mainly ReLU). Our key idea is to minimize the number of expensive multiplication operations in the GC. Then, we use the GC as a one-time pad to simplify verifying the integrity of the input from the server. Compared to the state-of-the-art works, our non-linear layer protocol achieves at least an order of magnitude performance improvement.
- We provide formal theoretical analysis for *VerifyML* security and demonstrate its performance superiority on various datasets and mainstream ML models including ResNet-18 and LeNet. Compared to state-of-the-art work, our experiments show that *VerifyML* achieves up to $1.7\times$ computation speedup and gains at least $10.7\times$ less communication overhead for linear layer computation. For non-linear layers, *VerifyML* is also $4\times$–$42\times$ faster and has $> 48\times$ lesser communication than existing 2 PC protocol against malicious parties. Meanwhile, *VerifyML* demonstrates an encouraging online runtime boost by $32.6\times$ and $32.2\times$ over existing works on LeNet and ResNet-18, respectively, and at least an order of magnitude communication cost reduction.

II. Preliminaries

A. Threat Model

We consider a secure ML inference scenario, where a model holder P_0 and a client P_1 interact with each other to evaluate the fairness of the target model. In such a *model holder-malicious* threat model, P_0 holds the model $\mathbf{M}$ while the client owns the private test set used to verify the fairness of the model. The client is generally considered to be semi-honest, that is, it follows the protocol's specifications in the interaction process for evaluating the fairness of the model unbiased. However, it is possible to infer model parameters by passively analyzing data streams captured during interactions. The model holder is malicious. It may arbitrarily violate the specification of the protocol to trick clients into believing that they hold a high-fairness model. The network architecture is assumed to be known to both P_0 and P_1. *VerifyML* aims to construct such a secure inference framework that enables P_1 to correctly evaluate the fairness of model without knowing any details of the model parameters, meanwhile, P_0 knows nothing about the client's input. We provide a

formal definition of the threat model in Appendix A, available online.

B. Notations

We use λ and σ to denote the computational security parameter and the statistical security parameter, respectively. $[k]$ represents the set $\{1, 2, \cdots k\}$ for $k > 0$. In our *VerifyML*, all the arithmetic operations are calculated in the field $\mathbb{F}_p$, where p is a a prime and we define $\kappa = \lceil \log p \rceil$. This means that there is a natural mapping for elements in $\mathbb{F}_p$ to $\{0,1\}^\kappa$. For example, $a[i]$ indicates the i-th bit of a on this mapping, i.e, $a = \sum_{i\in[\kappa]} a[i] \cdot 2^{i-1}$. Given two vectors $\mathbf{a}$ and $\mathbf{b}$, and an element $\alpha \in \mathbb{F}_p$, $\mathbf{a} + \mathbf{b}$ indicates the element-wise addition, $\alpha + \mathbf{a}$ and $\alpha\mathbf{a}$ mean that each component of $\mathbf{a}$ performs addition and multiplication with α, respectively. $\mathbf{a} * \mathbf{b}$ represents the inner production between vectors $\mathbf{a}$ and $\mathbf{b}$. Similarly, given any function $f : \mathbb{F}_p \to \mathbb{F}_p$, $f(\mathbf{a})$ denotes evaluation of f on each component on $\mathbf{a}$. $a||b$ represents the concatenation of a and b. U_n is used to represent the uniform distribution on the set $\{0,1\}^n$ for any $n > 0$.

For ease of exposition, we consider an ML model, usually a neural network model $\mathbf{M}$, consisting of alternating linear and nonlinear layers. We assume that the specification of the linear layer is $\mathbf{L}_1, \cdots \mathbf{L}_m$ and the non-linear layer is $f_1, \ldots, f_{m-1}$. Given an initial input (i.e., query) $\mathbf{x}_0$, the model holder will sequentially execute $\mathbf{v}_i = \mathbf{L}_i\mathbf{x}_{i-1}$ and $\mathbf{x}_i = f_i(\mathbf{v}_i)$. Finally, $\mathbf{M}$ outputs the inference result $\mathbf{v}_m = \mathbf{L}_m\mathbf{x}_{m-1} = \mathbf{M}(\mathbf{x}_0)$.

C. ML Fairness Measurement

Let $\mathcal{X}$ be the set of possible inputs and $\mathcal{Y}$ be the set of all possible labels. In addition, let $\mathcal{O}$ be a finite set related to fairness (e.g., ethnic group). We assume that $\mathcal{X} \times \mathcal{Y} \times \mathcal{O}$ is drawn from a probability space Ω with an unknown distribution $\mathcal{D}$, and use $\mathbf{M}(\mathbf{x})$ to denote the model inference result given an input $\mathbf{x}$. Based on these, we review the term of the *empirical fairness gap* (EFG) [41], which is widely used to measure the fairness of ML models against a specific group. To formalize the formulation of EFG, we first describe the definition of *conditional risk* as follows:

$$F_o(\mathbf{M}) = \mathop{\mathbb{E}}_{(\mathbf{x},y,o')\sim\mathcal{D}}[\mathbb{I}\{\mathbf{M}(\mathbf{x}) \neq y\}|o' = o] \tag{1}$$

Given a set of samples $(\mathbf{x}, y, o')$ satisfying distribution $\mathcal{D}$, $F_o(\mathbf{M})$ is the expectation of the number of misclassified entries in the test set that belong to group o, where $\mathbb{I}\{\Phi\}$ represents the indicator function with a predicate Φ. Given an independent sample set $\Psi = \{(\mathbf{x}^{(1)}, y^{(1)}, o^{(1)}), \ldots, (\mathbf{x}^{(t)}, y^{(t)}, o^{(t)})\} \sim \mathcal{D}^t$, the *empirical conditional risk* is defined as follows:

$$\tilde{F}_o(\mathbf{M}, \Psi) = \frac{1}{t_o}\sum_{i=1}^{t}\left[\mathbb{I}\{\mathbf{M}(\mathbf{x}^{(i)}) \neq y^{(i)}\}|o^{(i)} = o\right] \tag{2}$$

where t_o indicates the number of samples in Ψ from group o. Then, we describe the term *fairness gap (FG)*, which is used to measure the maximum margin of any two groups, specifically,

$$FG = \max_{o_o,o_1\in\mathcal{O}} |F_{o_o}(\mathbf{M}) - F_{o_1}(\mathbf{M})| \tag{3}$$

Likewise, the *empirical fairness gap (EFG)* is defined as

$$EFG = \max_{o_o,o_1\in\mathcal{O}} |\tilde{F}_{o_o}(\mathbf{M}, \Psi) - \tilde{F}_{o_1}(\mathbf{M}, \Psi)| \tag{4}$$

Lastly, we say a ML model $\mathbf{M}$ is ϵ-**fair** on $(\mathcal{O}, \mathcal{D})$, if its fairness gap is smaller than ϵ with confidence $1 - \delta$. Formally, a ϵ-**fair** $\mathbf{M}$ is defined as satisfying the following conditions:

$$Pr\left[\max_{o_o,o_1\in\mathcal{O}} |F_{o_o}(\mathbf{M}) - F_{o_1}(\mathbf{M})| > \epsilon\right] \leq \delta \tag{5}$$

In practice, we usually replace *FG* in (5) with *EFG* to facilitate the measurement of fairness. Note that once the client gets enough predictions in the target model, it can locally evaluate the fairness of the model according to (5).

D. Fully Homomorphic Encryption

Let the plaintext space be $\mathbb{F}_p$, informally, a Fully homomorphic encryption (FHE) under the public key encryption system usually contains the following algorithms:

- $\texttt{KeyGen}(1^\lambda) \to (pk, sk)$. Taking the security parameter λ as input, `KeyGen` is a random algorithm used to output the public key pk and the corresponding secret key sk required for homomorphic encryption.
- $\texttt{Enc}(pk, x) \to c$. Given pk and a plaintext $x \in \mathbb{F}_p$, the algorithm `Enc` outputs a ciphertext c encrypting x.
- $\texttt{Dec}(sk, c) \to x$. Taking sk and a ciphertext c as input, `Dec` decrypts c and outputs the corresponding plaintext x.
- $\texttt{Eval}(pk, c_1, c_2, F) \to c'$. Given pk, two ciphertexts c_1 and c_2, and a function F, the algorithm `Eval` outputs a ciphertext c' encrypting $F(c_1, c_2)$.

We require FHE to satisfy correctness, semantic security, and functional privacy.[1] In *VerifyML*, we use the SEAL library [40] to implement the fully homomorphic encryption. In addition, we utilize ciphertext packing technology (CPT) [42] to encrypt multiple plaintexts into a single ciphertext, thus enabling homomorphic computation in a SIMD manner. Specifically, given two plaintext vectors $\mathbf{x} = (x_1, \ldots, x_n)$ and $\mathbf{x}' = (x'_1, \ldots, x'_n)$, we can pack $\mathbf{x}$ and $\mathbf{x}'$ into ciphertexts c and c' each of them containing n plaintext slots. Homomorphic operations between c and c' including addition and multiplication are equivalent to performing the same element-wise operations on the corresponding plaintext slots. FHE also provides algorithm `Rotation` to handle operations between data located in different plaintext slots. Informally, given a plaintext vector $\mathbf{x} = (x_0, \ldots, x_n)$ is encrypted into a single ciphertext c, $\texttt{Rotation}(pk, c, j)$ transforms c into another ciphertext c' whose encrypted plaintext vector is $x' = (x_{j+1}, x_{j+2}, \ldots, x_1, \ldots, x_j)$. In this way, data on different plaintext slots can be moved to the same position to achieve element-wise operations under ciphertext. In FHE, *rotation* operations are computational expensive compared to homomorphic addition and multiplication operations. Therefore, the optimization criterion for homomorphic SIMD operations is to minimize the number of *rotation* operations.

[1] Functional privacy ensures that given a ciphertext c, which is an encrypted share of $F(x_1, x_2)$ obtained by homomorphically evaluating L, c is indistinguishable from ciphertext c' encrypting a share of $F'(x_1, x_2)$ for any F'.

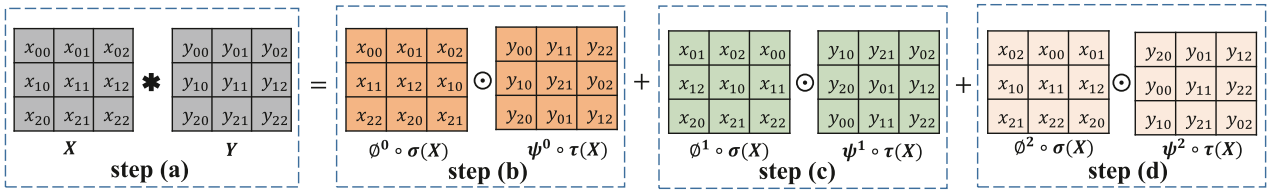


Fig. 1. Parallel matrix multiplication.

E. Parallel Matrix Homomorphic Multiplication

We review the parallel homomorphic multiplication method between arbitrary matrices proposed by Jiang et al. [19], which will be used to accelerate the generation of authenticated triples for convolution in *VerifyML*. We take the homomorphic multiplication of two $d \times d$ dimensional matrices as an example. Specifically, given a $d \times d$ dimensional matrix $\mathbf{X} = (x_{i,j})_{0 \le i,j < d}$, we first define four useful permutations, σ, τ, ϕ, and φ, over the field $\mathbb{F}_p^{d \times d}$. Let $\sigma(\mathbf{X})_{i,j} = \mathbf{X}_{i,i+j}$, $\tau(\mathbf{X})_{i,j} = \mathbf{X}_{i+j,j}$, $\phi(\mathbf{X})_{i,j} = \mathbf{X}_{i,j+1}$ and $\varphi(\mathbf{X})_{i,j} = \mathbf{X}_{i+1,j}$. Then for two square matrices $\mathbf{X}$ and $\mathbf{Y}$ of order d, we can calculate the matrix multiplication between the two by the following formula:

$$\mathbf{X} * \mathbf{Y} = \sum_{k=0}^{d-1} (\phi^k \circ \sigma(\mathbf{X})) \odot (\varphi^k \circ \tau(\mathbf{Y})) \tag{6}$$

where $\odot$ denotes the element-wise multiplication. We provide a toy example of the multiplication of two 3×3 matrices in Fig. 1 for ease of understanding.

We can convert a $d \times d$-dimensional matrix to a vector of length d^2 by encoding map $\mathbb{F}_p^{d^2} \to \mathbb{F}_p^{d \times d}$: $\mathbf{x} = (x_0, \ldots, x_{d^2-1}) \mapsto \mathbf{X} = (x_{d \cdot i+j})_{0 \le i,j < d}$. A ciphertext is said to encrypt a matrix $\mathbf{X}$ if it encrypts the corresponding plaintext vector $\mathbf{x}$. Therefore, given two square matrices $\mathbf{X}$ and $\mathbf{X}$, the multiplication of the two under the ciphertext is calculated as follows:

$$\mathbf{c_X} \circledast \mathbf{c_Y} = \sum_{k=0}^{d-1} (\phi^k(\mathtt{Enc}_{pk}(\sigma(\mathbf{X})))) \boxtimes (\varphi^k(\mathtt{Enc}_{pk}(\tau(\mathbf{Y})))) \tag{7}$$

In the following sections, we will use $\mathbf{c_X} \circledast \mathbf{c_Y}$ to represents multiplication between any matrixes $\mathbf{X}$ and $\mathbf{Y}$ in ciphertext. $\boxtimes$ denotes the elewent-wise homomorphic multiplication between two ciphertexts. In Section IV-A2, we describe how to utilize the parallel homomorphic multiplication described above to boost the generation of authenticated convolution triples.

F. Secret Sharing

- *Additive Secret Sharing:* Given any $x \in \mathbb{F}_p$, a 2-out-of-2 additive secret sharing of x is a pair $(\langle x \rangle_0, \langle x \rangle_1) = (x - r, r) \in \mathbb{F}_p^2$, where r is a random value uniformly selected from $\mathbb{F}_p$, and $x = \langle x \rangle_0 + \langle x \rangle_1$. Additive secret sharing is perfectly hiding, that is, given a share $\langle x \rangle_0$ or $\langle x \rangle_1$, x is perfectly hidden.
- *Authenticated Shares:* Given a random value α (known as the MAC key) uniformly chosen from $\mathbb{F}_p$, for any $x \in \mathbb{F}_p$, the authenticated shares of x on α denote that each party P_b holds $[\![x]\!]_b = \{\langle \alpha \rangle_b, \langle x \rangle_b, \langle \alpha x \rangle_b\}_{b \in \{0,1\}}$,[2] where we have $(\langle \alpha \rangle_0 + \langle \alpha \rangle_1) \times (\langle x \rangle_0 + \langle x \rangle_1) = (\langle \alpha x \rangle_0 + \langle \alpha x \rangle_1)$. While in the general malicious 2 PC setting, α should be generated randomly through interactions between all parties, in our *model holder-malicious* model, α can be picked up by P_1 and secretly shared with P_0. Authenticated sharing provides $\lfloor \log p \rfloor$ bits of statistical security. Informally, if a malicious P_0 tries to forge the shared x to be $x + \beta$, by tampering with its shares $(\langle x \rangle_0, \langle \alpha x \rangle_0)$ to $(\langle x \rangle_0 + \beta, \langle \alpha x \rangle_0 + \beta')$, for non-zero $\{\beta, \beta'\}$, the probability of parties being authenticated to hold the share of $x + \beta$ (*i.e.*, $\alpha x + \beta' = \alpha(x + \beta)$) is at most $2^{-\lfloor \log p \rfloor}$.

G. Authenticated Beaver's Triples

In *VerifyML*, we require the technique of authenticated Beaver's triples to detect possible breaches of the protocol from the malicious model holder. In more detail, authenticated Beaver's multiplication triple is denoted that each P_b holds a tuple $\{[\![x]\!]_b, [\![y]\!]_b, [\![z]\!]_b\}_{b \in \{0,1\}}$, where $x, y, z \in \mathbb{F}_p$, and satisfy $xy = z$. Giving P_0 and P_1 holding authenticated shares of c and d, i.e., $([\![c]\!]_0, [\![d]\!]_0)$, $([\![c]\!]_1, [\![d]\!]_1)$, respectively, to compute the authenticated share of the product of c and d, the parties first reveal $c - x$ and $d - y$, and then each party P_b locally computes the authenticated share of $[\![e = c \cdot d]\!]_b$ as follows:

$$\begin{aligned} \langle e \rangle_b &= (c - x) \cdot (d - y) + \langle x \rangle_b \cdot (d - y) \\ &\quad + (c - x) \cdot \langle y \rangle_b + \langle z \rangle_b \\ \langle \alpha e \rangle_b &= \langle \alpha \rangle_b (c - x) \cdot (d - y) + \langle \alpha x \rangle_b \cdot (d - y) \\ &\quad + (c - x) \cdot \langle \alpha y \rangle_b + \langle \alpha z \rangle_b \end{aligned} \tag{8}$$

Authenticated Beaver's multiplication triple is independent of the user's input in the actual execution of the secure computing protocol, thus can be generated offline (see Section IV) to speed up the performance of online secure multiplication computations. Inspired by existing work to construct custom triples for specific mathematical operations [33] for improving performance, we generalize traditional Beaver's triples to matrix-vector multiplication and convolution domains. We provide the definitions of matrix-vector and convolution triples below and leave the description of generating them to Section IV.

- *Authenticated Matrix-Vector triples:* is denoted that each P_b holds a tuple $\{[\![\mathbf{X}]\!]_b, [\![\mathbf{y}]\!]_b, [\![\mathbf{z}]\!]_b\}_{b \in \{0,1\}}$, where $\mathbf{X}$ is a matrix uniformly chosen from $\mathbb{F}_p^{d_1 \times d_2}$, $\mathbf{y}$ represents a vector selected from $\mathbb{F}_p^{d_2}$, and $\mathbf{z} \in \mathbb{F}_p^{d_1}$ satisfying $\mathbf{X} * \mathbf{y} = \mathbf{z}$,

[2] Sometimes in $[\![x]\!]_b$ we omit $\langle \alpha \rangle_b$ for brevity.

where d_1 and d_2 are determined depending on the ML model architecture.

- *Authenticated Convolution triples* (aka matrix multiplication triples[3]): is denoted that each P_b holds a tuple $\{[\![\mathbf{X}]\!]_b, [\![\mathbf{Y}]\!]_b, [\![\mathbf{Z}]\!]_b\}_{b\in\{0,1\}}$, where $\mathbf{X}$ and $\mathbf{Y}$ are tensors uniformly chosen from $\mathbb{F}_p^{u_w\times u_h\times c_i}$ and $\mathbb{F}_p^{(2l+1)\times(2l+1)\times c_i\times c_o}$, respectively. $\mathbf{Z}\in\mathbb{F}_p^{u'_w\times u'_h\times c_o}$ satisfying convolution $\mathtt{Conv}(\mathbf{X},\mathbf{Y})=\mathbf{Z}$, where u_w, u_w, u'_w, u'_h, l, c_i and c_o are determined depending on the model architecture.

H. Oblivious Transfer

We take OT_n to denote the 1-out-of-2 Oblivious Transfer (OT) [11], [14]. In OT_n, the inputs of the sender (assuming P_0 for convenience) are two strings $s_0, s_1 \in \{0,1\}^n$, and the input of the receiver (P_1) is a bit $b\in\{0,1\}$ for selection. At the end of the OT-execution, P_1 learns s_b while P_0 learns nothing. 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 [23] to implement OT_n^κ with the communication complexity of $\kappa\lambda+2n$ bits.

I. Garbled Circuits

The garbling scheme [8], [38] for boolean circuits parsing arbitrary functions consists of a pair of algorithms ($\mathtt{Garble}$, $\mathtt{GCEval}$) defined as follows:

- $\mathtt{Garble}(1^\lambda, C)\rightarrow(\mathtt{GC}, \{\{\mathtt{lab}_{i,j}^{in}\}_{i\in[n]}, \{\mathtt{lab}_j^{out}\}\}_{j\in\{0,1\}})$. Giving the security parameter λ and an arbitrary Boolean circuit $C:\{0,1\}^n\rightarrow\{0,1\}$, the algorithm $\mathtt{Garble}$ outputs a garbled circuit $\mathtt{GC}$, a set of input labels $\{\mathtt{lab}_{i,j}^{in}\}_{i\in[n],j\in\{0,1\}}$ of this $\mathtt{GC}$, and a set of output labels $\{\mathtt{lab}_j^{out}\}_{j\in\{0,1\}}$, where the size of each label is λ bits. For any $x\in\{0,1\}^n$, we refer to $\{\mathtt{lab}_{i,x[i]}^{in}\}_{i\in[n]}$ as the *garbled input* of x, and $\mathtt{lab}_{C(x)}^{out}$ as the *garbled output* of $C(x)$.
- $\mathtt{GCEval}(\mathtt{GC}, \{\mathtt{lab}_i\}_{i\in[n]})\rightarrow\mathtt{lab}'$. Giving the garbled circuit $\mathtt{GC}$ and a set of input labels $\{\mathtt{lab}_i\}_{i\in[n]}$, the algorithm $\mathtt{GCEval}$ outputs a label $\mathtt{lab}'$.

Let $\mathtt{Garble}(1^\lambda, C)\rightarrow(\mathtt{GC}, \{\{\mathtt{lab}_{i,j}^{in}\}_{i\in[n]}, \{\mathtt{lab}_j^{out}\}\}_{j\in\{0,1\}})$, the above garbled scheme ($\mathtt{Garble}$, $\mathtt{GCEval}$) is required to satisfy the following properties:

- *Correctness:* $\mathtt{GCEval}$ is faithfully performed on the $\mathtt{GC}$ and correctly outputs garbled results when given the garbled input of x. Formally, for any Boolean circuit C and input $x\in\{0,1\}^n$, $\mathtt{GCEval}$ holds that
$$\mathtt{GCEval}(\mathtt{GC}, \{\mathtt{lab}_{i,x[i]}^{in}\}_{i\in[n]})\rightarrow\mathtt{lab}_{C(x)}^{out}$$
- *Security:* Given C, the garbled circuit $\mathtt{GC}$ of C and garbled inputs of any $x\in\{0,1\}^n$ can be simulated by a polynomial probability-time simulator $\mathtt{Sim}$. Formally, for any circuit C and input $x\in\{0,1\}^n$, we have $(\mathtt{GC}, \{\mathtt{lab}_{i,x[i]}^{in}\}_{i\in[n]})\approx\mathtt{Sim}(1^\lambda, C)$, where $\approx$ indicates computational indistinguishability.
- *Authenticity:* This implies that given the garbled input of x and $\mathtt{GC}$, it is infeasible to guess the output label of $1-C(x)$. Formally, for any circuit C and $x\in\{0,1\}^n$, we have $(\mathtt{lab}_{1-C(x)}^{out}|\mathtt{GC}, \{\mathtt{lab}_{i,x[i]}^{in}\}_{i\in[n]})\approx U_\lambda$.

Without loss of generality, the garbled scheme described above can be naturally extended to securely implement Boolean circuits with multi-bit outputs. In *VerifyML*, we utilize state-of-the-art optimization strategies, including point-and-permute [13], free-XOR [25] and half-gates [46] to construct the garbling scheme.

III. Technical Intuition

VerifyML is a 2-party computation (2 PC) protocol designed to operate under the *model holder-malicious* threat model. The protocol enables the client to learn unbiased inference results on a given test set, facilitating a local evaluation of the fairness of the target model. To enhance the performance of the 2 PC protocol execution, we have customized a series of optimization methods that fully exploit the benefits of cryptographic primitives and their inherent ties in the inference process. In the following section, we provide a high-level, technically intuitive overview of the design of *VerifyML*.

A. Offline-Online Paradigm

Following the state-of-the-art work on semi-honest models [32], *VerifyML* is divided into an offline stage and an online stage. During the offline stage, the preprocessing process is independent of the input from the model holders and clients. By performing the majority ($>95\%$) of the computation offline, we aim to minimize the overhead of the online process. Fig. 2 provides an overview of *VerifyML*, outlining the computational components required for the offline and online phases, respectively.

B. Linear Layer Optimization

As shown in Fig. 2, we have moved almost all linear operations to the offline phase of *VerifyML*. This is accomplished through the construction of customized triples for matrix-vector multiplication and convolution, aimed at accelerating linear execution. Specifically, we have implemented two key optimization methods. First, we have designed an efficient construction of matrix-multiplication triples that eliminates the need for generating Beaver's multiplication triples for individual multiplications (see Section IV-A1). Our core insight is a new packed homomorphic multiplication method for matrices and vectors that leverages the inherent connection between secret sharing and homomorphic encryption to remove all rotation operations in parallel homomorphic computation. Second, we have extended the idea of generating matrix multiplicative triples over semi-honest models [33] to the convolution domain under the *model holder-malicious* threat model (see Section IV). The core of our construction is derived from E2DM [19], which proposes a state-of-the-art method for parallel homomorphic multiplication between arbitrary matrices. We have further optimized E2DM to achieve at least a $2\times$ computational speedup compared to naive use.

[3]We can reduce the convolution operation to matrix multiplication by transforming the inputs of convolution appropriately. We provide a detailed description in Section IV.

Offline Phase. This phase the client and model holder pre-compute data in preparation for subsequent online execution, which is independent of input from all parties. That is, *VerifyML* can run this phase without knowing the client's input $\mathbf{x}_0$ and the model holder's input $\mathbf{M}$.

- *Preprocessing for the linear layer*. The Client interacts with the model holder to generate authenticated triples for matrix-vector multiplication and convolution.
- *Preprocessing for the nonlinear layer*. The client constructs a garbled circuit GC for circuit C parsing ReLU. The client sends GC and a set of ciphertexts to the model holder for generating the authenticated shares of ReLU's results.

Online Phase. This phase is divided into following parts.

- *Preamble*. The client secretly shares its input $\mathbf{x}_0$ with the model holder, and similarly, the model holder shares the model parameter $\mathbf{M}$ with the client. Thus both the model holder and the client hold an authenticated share of $\mathbf{x}_0$ and $\mathbf{M}$. Note that the sharing of $\mathbf{M}$ can be done offline, if the model to be verified is knowed in advance.
- *Layer evaluation*. Let $\mathbf{x}_i$ be the result of evaluating the first i layers of model $\mathbf{M}$ on $\mathbf{x}_0$. At the beginning of the $i+1$-th layer, both the client and the model holder hold an authenticated share about $\mathbf{x}_i$ and the i+1-th layer parameter $\mathbf{L}_{i+1}$, i.e., parties $P_{b\in\{0,1\}}$ hold $(\llbracket \mathbf{x}_i \rrbracket_b, \llbracket \mathbf{L}_{i+1} \rrbracket_b)$.
  1. *Linear layer* . The client interacts with the model holder to perform the authenticated shares of $\mathbf{v}_{i+1} = \mathbf{L}_{i+1}\mathbf{x}_{i+1}$, where both parties securely compute matrix-vector multiplication and convolution operations with the aid of triples generated in the precomputing process.
  2. *Nonlinear layer*. After the linear layer, the two parties hold the authenticated shares of $\mathbf{v}_{i+1}$. The client and the model holder invoke the OT to send the garbled input of GC to the model holder. The model holder evaluates the GC, and eventually the two parties get authenticated shares of the ReLU result.
- *Consistency check*. The client interacts with the model holder to check any malicious behavior of the model holder during the entire inference process. The client uses the properties of the authenticated sharing to construct the consistency check protocol. If consistency passes, the client locally computes the fairness of the target model, otherwise the client outputs abort.

Fig. 2. Overview of the *VerifyML*.

Our optimization technique for linear layer computation exhibits superior advantages compared to state-of-the-art existing methods [22], [24][4]. In more detail, we reduce the communication overhead from cubic to quadratic (both for offline and online phases) compared to Overdrive [24], which is the mainstream tool for generating authenticated multiplicative triples on malicious adversary models(see Section IV for detailed analysis).

[4]Note that several efficient parallel homomorphic computation methods [21], [47] with packed ciphertext have been proposed and run on semi-honest or client-malicious models [5], [28], [29], [32] for secure inference. It may be possible to transfer these techniques to our method to speed up triple's generation, but this is certainly non-trivial and we leave it for future work.

C. Non-Linear Layer Optimization

We use the garbled circuit to achieve secure computation of nonlinear functions (mainly ReLU) in ML models. Specifically, assumed that P_0 and P_1 learn the authenticated sharing about $\mathbf{v}_i = \mathbf{L}_i\mathbf{x}_{i-1}$ after executing the i-th linear layer, that is, each party P_b holds $\llbracket \mathbf{v}_i \rrbracket_b = \{\langle \alpha \rangle_b, \langle \mathbf{v}_i \rangle_b, \langle \alpha \mathbf{v}_i \rangle_b\}_{b\in\{0,1\}}$. Then, $\{\langle \mathbf{v}_i \rangle_b\}_{b\in\{0,1\}}$ will be used as the input of ReLU (dented as f_i for brevity) in the i-th nonlinear layer for both parties learning the authentication sharing about $\mathbf{x}_i = f_i(\mathbf{v}_i)$, i.e., $\llbracket \mathbf{x}_i \rrbracket_b$. However, constructing such a satisfactory garbling scheme has the following intractable problems.

- *How to validate input from the malicious model holder:* As the model holder is untrustworthy, it is necessary to ensure that the input from the model holder in the GC (i.e., $\langle \mathbf{v}_i \rangle_0$) matches the share obtained by the previous linear layer. In the traditional malicious adversary model [6], [22], [24], it is common to authenticate the sharing of all inputs from malicious entities in the GC to ensure correctness. However, this approach is costly and takes tens of seconds or even minutes to process a ReLU function. It is clearly impractical for ML model inference, as a modern ML model may contain thousands of ReLU functions.
- *How to minimize the number of multiplication encapsulated into* GC*:* For the i-th nonlinear layer, we need to compute the authenticated shares of the ReLU output, i.e., $\llbracket \mathbf{x}_i \rrbracket_b = \{\langle \alpha \rangle_b, \langle \mathbf{x}_i \rangle_b, \langle \alpha \mathbf{x}_i \rangle_b\}_{b\in\{0,1\}}$. This requires at least two multiplications on the field, if all computations are encapsulated into the GC. Note that performing arithmetic multiplication operations in the GC is expensive and requires at least $O(\kappa^2\lambda)$ communication overhead.

We design novel protocols to remedy the above problems through the following insights: (1) garbled circuits already achieve malicious security against garbled circuit evaluators (i.e., the model holder in our setting) [28]. This means that we only need to construct a lightweight method to check the consistency between the input of the malicious adversary in the nonlinear layer and the results obtained by the previous linear layer. Then, this method can be integrated with GC to achieve end-to-end nonlinear secure computing (see Section IV). (2) It is enough to calculate the output label for each bit of $f_i(\mathbf{v}_i)$'s share (i.e., $f_i(\mathbf{v}_i)[j]$, for $1 \le j \le \kappa$) in the GC, rather than obtaining the exact arithmetic share of $f_i(\mathbf{v}_i)$ [5]. Moreover, we can parse ReLU function as $ReLU(\mathbf{v}_i) = \mathbf{v}_i \cdot sign(\mathbf{v}_i)$, where the sign function $sign(\mathbf{v}_i)$ equals 1 if $t \ge 0$ and 0 otherwise. Hence, we only encapsulate the non-linear part of $ReLU(\mathbf{v}_i)$ (*i.e.*, $sign(\mathbf{v}_i)$) into the GC, thereby substantially minimizing the number of multiplication operations.

Compared with works [6], [22], [24] with malicious adversary, *VerifyML* reduces the communication overhead of each ReLU function from $2c\lambda + 190\kappa\lambda + 232\kappa^2$ to $2d\lambda + 4\kappa\lambda + 6\kappa^2$, where $d \ll c$. Our experiments show that *VerifyML* achieves $4\times$-$42\times$ computation speedup and gains

Input: $\{P_b\}_{b\in\{0,1\}}$ holds $\langle \mathbf{X}\rangle_b$ uniformly chosen from $\mathbb{F}_p^{d_1\times d_2}$, and $\langle \mathbf{y}\rangle_b$ uniformly chosen from $\mathbb{F}_p^{d_2}$. In addition, P_1 hold a MAC key α uniformly chosen from $\mathbb{F}_p$.
Output: P_b obtains $\{[\![\mathbf{X}]\!]_b, [\![\mathbf{y}]\!]_b, [\![\mathbf{z}]\!]_b\}_{b\in\{0,1\}}$ where $\mathbf{X} * \mathbf{y} = \mathbf{z}$.
Procedure:

1. P_0 and P_1 participate in a secure two-party computation such that P_0 obtains an FHE public secret key pair (pk, sk) while P_1 obtains the public key pk. This process is performed only once.
2. P_0 first converts $\langle \mathbf{y}\rangle_0$ into a $d_1 \times d_2$-dimensional matrix $\langle \mathbf{Y}\rangle_0$ where each row constitutes a copy of $\langle \mathbf{y}\rangle_0$. Then, P_0 send the encryptions $c_1 \leftarrow \mathtt{Enc}(pk, \langle \mathbf{X}\rangle_0)$ and $c_2 \leftarrow \mathtt{Enc}(pk, \langle \mathbf{Y}\rangle_0)$ to P_1 along with zero-knowledge (ZK) proofs of plaintext knowledge of the two ciphertexts [4].
3. P_1 also converts $\langle \mathbf{y}\rangle_1$ into a $d_1 \times d_2$-dimensional matrix $\langle \mathbf{Y}\rangle_1$ where each row constitutes a copy of $\langle \mathbf{y}\rangle_1$. Then it samples $(\langle \alpha\mathbf{X}\rangle_1, \langle \alpha\mathbf{Y}\rangle_1, \langle \alpha\mathbf{Z}\rangle_1, \langle \mathbf{Z}\rangle_1)$ from $\mathbb{F}_p^{4\times(d_1\times d_2)}$. P_1 sends $c_3 = \mathtt{Enc}_{pk}(\alpha(\langle \mathbf{X}\rangle_1 + \langle \mathbf{X}\rangle_0) - \langle \alpha\mathbf{X}\rangle_1)$, $c_4 = \mathtt{Enc}_{pk}(\alpha(\langle \mathbf{Y}\rangle_1 + \langle \mathbf{Y}\rangle_0) - \langle \alpha\mathbf{Y}\rangle_1)$, $c_5 = \mathtt{Enc}_{pk}(\alpha(\mathbf{X}\odot\mathbf{Y}) - \langle \alpha\mathbf{Z}\rangle_1)$, and $c_6 = \mathtt{Enc}_{pk}((\mathbf{X}\odot\mathbf{Y}) - \langle \mathbf{Z}\rangle_1)$ to P_0.
4. P_0 decrypts c_3, c_4, c_5 and c_6 to obtain $(\langle \alpha\mathbf{X}\rangle_0, \langle \alpha\mathbf{Y}\rangle_0, \langle \alpha\mathbf{Z}\rangle_0, \langle \mathbf{Z}\rangle_0)$, respectively. Then, it sums the elements of each row of the matrices $\langle \alpha\mathbf{Y}\rangle_0$[5], $\langle \alpha\mathbf{Z}\rangle_0$ and $\langle \mathbf{Z}\rangle_0$ to form the vectors $\langle \alpha\mathbf{y}\rangle_0$, $\langle \alpha\mathbf{z}\rangle_0$ and $\langle \mathbf{z}\rangle_0$. P_1 does the same for $(\langle \alpha\mathbf{Y}\rangle_1, \langle \alpha\mathbf{Z}\rangle_1, \langle \mathbf{Z}\rangle_1)$ to obtain $\langle \alpha\mathbf{y}\rangle_1$, $\langle \alpha\mathbf{z}\rangle_1$ and $\langle \mathbf{z}\rangle_1$.
5. P_b outputs $\{[\![\mathbf{X}]\!]_b, [\![\mathbf{y}]\!]_b, [\![\mathbf{z}]\!]_b\}_{b\in\{0,1\}}$, where $\mathbf{X} * \mathbf{y} = \mathbf{z}$.

Fig. 3. Algorithm $\pi_{Mtriple}$ for generating authenticated matrix-vector multiplication triple.

48× less communication overhead for nonlinear layer computation.

Remark III.1: Beyond the above optimization strategies, we also do a series of strategies to reduce the overhead in the implementation process, including removing the reliance on distributed decryption primitives in previous works [6], [22], [24] and minimizing the number of calls to zero-knowledge proofs of ciphertexts. In the following section, we provide a comprehensive technical description of the proposed method.

IV. The VerifyML Framework

A. Offline Phase

In this section, we describe the technical details of *VerifyML*. As described above, *VerifyML* is divided intoofffline and online phases. We first describe the operations that need to be precomputed in the offline phase, including generating matrix-vector multiplications and triples for convolution, and garbled circuits for constructing the objective function. Then, we introduce the technical details of the online phase.

1) Generating Matrix-Vector Multiplication Triple: Fig. 3 depicts the interaction between the model holder P_0 and the client P_1 to generate triples of matrix-vector multiplications. Succinctly, P_0 first uniformly selects $\langle \mathbf{X}\rangle_0$ and $\langle \mathbf{y}\rangle_0$ and sends their encryption to P_1, along with zero-knowledge proofs about these ciphertexts, where $\langle \mathbf{y}\rangle_0$ need to be transformed into matrix

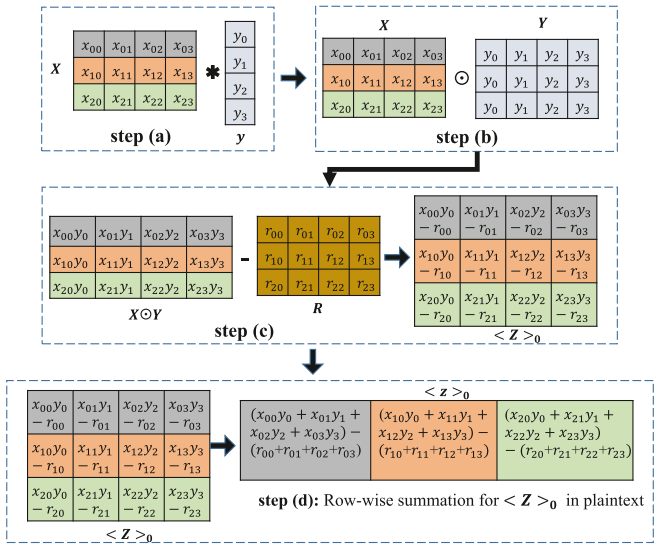


Fig. 4. Matrix-vector multiplication.

$\langle \mathbf{Y}\rangle_0$ before encryption (step 2 in Fig. 3). P_1 recovers $\mathbf{X}$ and $\mathbf{Y}$ in ciphertext and then computes $(\langle \alpha\mathbf{X}\rangle_0, \langle \alpha\mathbf{Y}\rangle_0, \langle \alpha\mathbf{Z}\rangle_0, \langle \mathbf{Z}\rangle_0)$ (step 3 in Fig. 3). Then it returns the corresponding ciphertexts to P_0. P_0 decrypts them and computes $\langle \alpha\mathbf{y}\rangle_1$, $\langle \alpha\mathbf{z}\rangle_1$ and $\langle \mathbf{z}\rangle_1$ (step 4 in Fig. 3). [5][6]

Fig. 4 provides an example of the multiplication of a 3×4-dimensional matrix $\mathbf{X}$ and a 4-dimensional vector $\mathbf{y}$ to facilitate understanding. To compute the additive sharing of $\mathbf{z} = \mathbf{X} * \mathbf{y}$ (step(a) in Fig. 4), $\mathbf{y}$ is first transformed into a matrix $\mathbf{X}$ by copying, where each row of $\mathbf{Y}$ contains a copy of $\mathbf{y}$. P_1 then performs element-wise multiplications (step(b) in Fig. 4) for $\mathbf{X}$ and $\mathbf{Y}$ under the ciphertext. To construct the additive sharing of $\mathbf{z} = \mathbf{X} * \mathbf{y}$, P_1 uniformly chooses a random matrix $\mathbf{R} \in \mathbb{F}_p^{3\times 4}$ and computes $\langle \mathbf{Z}\rangle_0 = \mathbf{X}\odot\mathbf{Y} - \mathbf{R}$ (step(c) in Fig. 4). P_1 sends the ciphertext result to P_0. P_0 decrypts it and sums each row in plaintext to obtain vector $\langle \mathbf{z}\rangle_0$ (step(d) in Fig. 4), similarly, P_1 performs the same operation on matrix $\mathbf{R}$ to obtain $\langle \mathbf{z}\rangle_1$.

Remark IV.1: Compared to generating multiplication triples for single multiplication [22], [24], our constructed matrix-multiplication triples enable the communication overhead to be independent of the number of multiplications, only related to the size of the input. This reduces the amount of data that needs to be exchanged between P_0 and P_1. In addition, we move the majority of the computation to be executed by the semi-honest party, which avoids the need for distributed decryption and frequent zero-knowledge proofs in malicious adversary settings. Compared to existing parallel homomorphic computation methods [15], [19], our matrix-vector multiplication does not involve any rotation operation, which is very computationally expensive compared to other homomorphic operations. This stems from our observation of the inner tie between HE and secret sharing.

[5]A ZK proof of knowledge for ciphertexts is used to state that c_1 and c_2 are valid ciphertexts generated from the given FHE cryptosystem. Readers can refer to [6], [24] for more details.

[6]Note that for $\langle \alpha\mathbf{Y}\rangle_0$, we only take the all elements in the first row as $\langle \alpha\mathbf{y}\rangle_0$ by default.The operation for $\langle \alpha\mathbf{Y}\rangle_1$ is the same as above.

Since the final ciphertext result needs to be secretly shared to P_0 and P_1, we can first perform the secret sharing under the ciphertext (see step(c) and step(d) in Fig. 4), and then perform all rotation and summation operations under the plaintext.

Security: Our protocol for generating matrix-vector multiplication triples, $\pi_{Mtriple}$, is secure against the malicious model holder P_0 and the semi-honest client P_1. We provide the following theorem and prove it in Appendix B, available online.

Theorem IV.1: Let the fully homomorphic encryption used in $\pi_{Mtriple}$ have the properties defined in Section II-D. $\pi_{Mtriple}$ is secure against the malicious model holder P_0 and the semi-honest client P_1.

2) Generating Convolution Triple: We describe the technical details of generating authenticated triples for convolution. Briefly, for a given convolution operation, we first convert it to equivalent matrix multiplications, and then generate triples for the matrix multiplications. We start by reviewing the definition of convolution and how to translate it into the equivalent matrix multiplication. Then, we explain how to generate authenticated triples.

① *Convolution:* Assuming an input tensor of size $u_w \times u_h$ with c_i channels, denoted as $\mathbf{X}_{ijk}$, where $1 \le i \le u_w$ and $1 \le j \le u_h$ are spatial coordinates, and $1 \le k \le c_i$ is the channel. Let c_o kernels with a size of $(2l+1) \times (2l+1) \times c_i$ denote as tensor $\mathbf{Y}_{\Delta_i,\Delta_j,k,k'}$, where $-l \le \Delta_i, \Delta_j \le l$ are shifts of the spatial coordinates, $1 \le k \le c_i$ and $1 \le k' \le c_o$ are the channels and kernels, respectively. The convolution between $\mathbf{X}$ and $\mathbf{Y}$ (i.e., $\mathbf{Z} = \texttt{Conv}(\mathbf{X}, \mathbf{Y})$) is defined as below:

$$\mathbf{Z}_{ijk'} = \sum_{\Delta_i,\Delta_j,k} \mathbf{X}_{i+\Delta_i,j+\Delta_j,k} \cdot \mathbf{Y}_{\Delta_i,\Delta_j,k'} \tag{9}$$

The resulting tensor $\mathbf{Z}_{ijk'}$ has $u'_w \times u'_h$ spatial coordinates and c_0 channels. We have $u'_w = (u_w - (2l+1) + 2p)/s + 1$ and $u'_h = (u_h - (2l+1) + 2p)/s + 1$, where p represents the number of turns to zero-pad the input, and s represents the stride size of the kernel movement [27]. Note that the entries of $\mathbf{X}$ to be zero if $i + \Delta_i$ or $j + \Delta_j$ are outside of the ranges $[1; u'_w]$ and $[1; u'_h]$, respectively.

② *Conversion between convolution and matrix multiplication:* Based on (9), we can easily convert convolution into an equivalent matrix multiplication. Specifically, we construct a matrix $\mathbf{X}'$ with dimension $u'_w u'_h \times (2l+1)^2 \cdot c_i$, where $\mathbf{X}'_{(i,j)(\Delta_i,\Delta_j,k)} = \mathbf{X}_{i+\Delta_i,j+\Delta_j,k}$. Similarly, we construct a matrix $\mathbf{Y}'$ of dimension $(2l+1)^2 \cdot c_i \times c_o$ such that $\mathbf{Y}'(\Delta_i, \Delta_j, k)k' = \mathbf{Y}_{\Delta_i,\Delta_j,k'}$. Then, the original convolution operation is transformed into $\mathbf{Z}' = \mathbf{X}' * \mathbf{Y}'$, where $\mathbf{Z}'_{(ij)k'} = \mathbf{Z}_{ijk'}$. In Appendix C, available online, we provide a detailed example to implement the above transformation.

③ *Generating convolution triple:* Fig. 5 depicts the interaction between the model holder P_0 and the client P_1 to generate triples of convolution. Succinctly, P_0 first uniformly selects $\langle \mathbf{X}' \rangle_0$ and $\langle \mathbf{Y}' \rangle_0$ and sends their encryption to P_1, along with zero-knowledge proofs about these ciphertexts (step 2 in Fig. 5). P_1 recovers $\mathbf{X}'$ and $\mathbf{Y}'$ under the ciphertext and then computes $(\langle \alpha\mathbf{X}' \rangle_0, \langle \alpha\mathbf{Y}' \rangle_0, \langle \alpha\mathbf{Z}' \rangle_0, \langle \mathbf{Z}' \rangle_0)$ (step 3 in Fig. 5). Then it returns the corresponding ciphertexts to P_0. P_0 decrypts these ciphertexts and computes $\langle \alpha\mathbf{X} \rangle_0, \langle \alpha\mathbf{Y} \rangle_0, \langle \alpha\mathbf{Z} \rangle_0$ and $\langle \mathbf{Z} \rangle_0$ (step 4 in Fig. 5). Finally, P_b obtains $\{[\![\mathbf{X}]\!]_b, [\![\mathbf{Y}]\!]_b, [\![\mathbf{Z}]\!]_b\}_{b\in\{0,1\}}$, where $\mathbf{Z} = \texttt{Conv}(\mathbf{X}, \mathbf{Y})$.

Input: $\{P_b\}_{b\in\{0,1\}}$ holds $\langle \mathbf{X} \rangle_b$ uniformly chosen from $\mathbb{F}_p^{u_w \times u_h \times c_i}$, and $\langle \mathbf{Y} \rangle_b$ uniformly chosen from $\mathbb{F}_p^{(2l+1)\times(2l+1)\times c_i \times c_o}$. In addition, p_1 holds a MAC key α uniformly chosen from $\mathbb{F}_p$
Output: P_b obtains $\{[\![\mathbf{X}]\!]_b, [\![\mathbf{Y}]\!]_b, [\![\mathbf{Z}]\!]_b\}_{b\in\{0,1\}}$, where $\mathbf{Z} = \texttt{Conv}(\mathbf{X}, \mathbf{Y})$.
Procedure:

1. P_0 and P_1 participate in a secure two-party computation such that P_0 obtains an FHE public-secret key pair (pk, sk) while P_1 obtains the public key pk. This process is performed only once.
2. P_0 first converts $\langle \mathbf{X} \rangle_0$ and $\langle \mathbf{Y} \rangle_0$ into equivalent matrixes $\langle \mathbf{X}' \rangle_0$ and and $\langle \mathbf{Y}' \rangle_0$, where $\langle \mathbf{X}' \rangle_0 \in \mathbb{F}_p^{u'_w u'_h \times (2l+1)^2 \cdot c_i}$ while $\langle \mathbf{Y}' \rangle_0 \in \mathbb{F}_p^{(2l+1)^2 \cdot c_i \times c_o}$. Then, P_0 sends the encryptions $c_1 \leftarrow \texttt{Enc}(pk, \langle \mathbf{X}' \rangle_0)$ and $c_2 \leftarrow \texttt{Enc}(pk, \langle \mathbf{Y}' \rangle_0)$ to P_1 along with zero-knowledge (ZK) proofs of plaintext knowledge of the two ciphertexts.
3. P_1 also converts $\langle \mathbf{X} \rangle_1$ and $\langle \mathbf{Y} \rangle_1$ into equivalent matrixes $\langle \mathbf{X}' \rangle_1$ and and $\langle \mathbf{Y}' \rangle_1$. Then it samples $(\langle \alpha\mathbf{X}' \rangle_1, \langle \alpha\mathbf{Y}' \rangle_1, \langle \alpha\mathbf{Z}' \rangle_1, \langle \mathbf{Z}' \rangle_1)$, and computes $c_3 = \texttt{Enc}_{pk}(\alpha(\langle \mathbf{X}' \rangle_1 + \langle \mathbf{X}' \rangle_0) - \langle \alpha\mathbf{X}' \rangle_1)$, $c_4 = \texttt{Enc}_{pk}(\alpha(\langle \mathbf{Y}' \rangle_1 + \langle \mathbf{Y}' \rangle_0) - \langle \alpha\mathbf{Y} \rangle_1)$, $c_5 = \alpha \boxtimes (\mathbf{c}_{\mathbf{X}'} \circledast \mathbf{c}_{\mathbf{Y}'}) - \texttt{Enc}_{pk}(\langle \alpha\mathbf{Z}' \rangle_1)$, and $c_6 = (\mathbf{c}_{\mathbf{X}'} \circledast \mathbf{c}_{\mathbf{Y}'}) - \texttt{Enc}_{pk}(\langle \mathbf{Z} \rangle_1)$. P_1 sends c_3, c_4, c_5 and c_6 to P_0.
4. P_0 decrypts c_3, c_4, c_5 and c_6 to obtain $(\langle \alpha\mathbf{X}' \rangle_0, \langle \alpha\mathbf{Y}' \rangle_0, \langle \alpha\mathbf{Z}' \rangle_0, \langle \mathbf{Z}' \rangle_0)$, respectively. Then, Both P_0 and P_1 converts these matrices back into tensors to get $(\langle \alpha\mathbf{X} \rangle_b, \langle \alpha\mathbf{Y} \rangle_b, \langle \alpha\mathbf{Z} \rangle_b, \langle \mathbf{Z} \rangle_b)$ for $b = \{0, 1\}$.
5. P_b outputs $\{[\![\mathbf{X}]\!]_b, [\![\mathbf{Y}]\!]_b, [\![\mathbf{Z}]\!]_b\}_{b\in\{0,1\}}$, where $\mathbf{Z} = \texttt{Conv}(\mathbf{X}, \mathbf{Y})$.

Fig. 5. Algorithm $\pi_{Ctriple}$ for generating authenticated convolution triple.

Remark IV.2: We utilize the method in [19] to perform the homomorphic multiplication operations involved in generating convolution triples in parallel. Given the multiplication of two $d \times d$-dimensional matrices, it reduces the computational complexity from $O(d^2)$ to $O(d)$, compared with the existing method [15]. Besides, [19] requires only one ciphertext to represent a single matrix whereas existing work [15] requires d ciphertexts (assuming the number of plaintext slots n in FHE is greater than d^2). In addition, compared to generating multiplication triples for single multiplication [22], [24], the communication overhead of our method is independent of the number of multiplications, only related to the size of the input, i.e., reduce the communication cost from cubic to quadratic (both offline and online phases).

Remark IV.3: We further exploit the properties of semi-honest clients to improve the performance of generating convolution triples. Specifically, for the multiplication of matrices $\mathbf{X}$ and $\mathbf{Y}$, the permutations $\sigma(\mathbf{X})$ and $\varphi(\mathbf{Y})$ can be done in plaintext beforehand, which reduces the rotation in half compared to the original method (see Section III-B in [19] for comparison). Moreover, we move the majority of the computation to be executed by the semi-honest party, which avoids the need for

distributed decryption and frequent zero-knowledge proofs in malicious adversary settings.

Security: Our protocol for generating authenticated convolution triples, $\pi_{Ctriple}$, is secure against the malicious model holder P_0 and the semi-honest client P_1. We provide the following theorem.

Theorem IV.2: Let the fully homomorphic encryption used in $\pi_{Ctriple}$ have the properties defined in Section II-D. $\pi_{Ctriple}$ is secure against the malicious model holder P_0 and the semi-honest client P_1.

Proof: The proof logic of this theorem is very similar to **Theorem** IV.1, we omit it for brevity.

3) Preprocessing for the Nonlinear Layer: This process is performed by the client to generate garbled circuits of nonlinear functions for the model holder. Note that we do not generate GC for ReLu but for the nonlinear part of ReLU, i.e., $sign(\mathbf{v})$ given an arbitrary input $\mathbf{v}$. We first define a truncation function $\mathbf{Trun}_h : \{0,1\}^\lambda \to \{0,1\}^h$, 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 $\mathbf{Trun}$. Then, the client is required to generate random ciphertexts and send them to the model holder. as follows.

- Given the security parameter λ, and the boolean circuit $booln^C$ denoted the nonlinear part of ReLU, P_1 computes $\mathtt{Garble}(1^\lambda, booln^C) \to (\mathtt{GC}, \{\{\mathtt{lab}_{i,j}^{in}\}_{i\in[2\kappa]}, \{\mathtt{lab}_{i,j}^{out}\}_{i\in[2\kappa]}\}_{j\in\{0,1\}})$, where GC is the garbled circuit of $booln^C$, $\{\{\mathtt{lab}_{i,j}^{in}\}_{i\in[2\kappa]}, \{\mathtt{lab}_{i,j}^{out}\}_{i\in[2\kappa]}\}_{j\in\{0,1\}}$ represent all possible garbled input and output labels, respectively. P_1 sends GC to the model holder P_0.
- P_1 uniformly selects $\eta_{i,1}$, $\gamma_{i,1}$ and $\iota_{i,1}$ from $\mathbb{F}_p$ for every $i \in [\kappa]$. Then, P_1 sets $(\eta_{i,0}, \gamma_{i,0}, \iota_{i,0}) = (1+\eta_{i,1}, \alpha+\gamma_{i,1}, \alpha+\iota_{i,1})$.
- P_1 parses $\{\mathtt{lab}_{i,j}^{out}\}$ as $\varsigma_{i,j}||\vartheta_{i,j}$ for every $i \in [2\kappa]$ and $j \in \{0,1\}$, where $\varsigma_{i,j} \in \{0,1\}$ and $\vartheta_{i,j} \in \{0,1\}^{\lambda-1}$.
- For every $i \in [\kappa]$ and $j \in \{0,1\}$, P_1 sends $ct_{i,\varsigma_{i,j}}$ and $\hat{ct}_{i,\varsigma_{i+\kappa,j}}$ to P_0, where $ct_{i,\varsigma_{i,j}} = \iota_{i,j} \oplus \mathbf{Trun}_\kappa(\vartheta_{i,j})$ and $\hat{ct}_{i,\varsigma_{i+\kappa,j}} = (\eta_{i,j}||\gamma_{i,j}) \oplus \mathbf{Trun}_{2\kappa}(\vartheta_{i+\kappa,j})$.

Security: We leave the explanation of above ciphertexts sent by P_1 to P_0 to the following sections. Here we briefly describe the security of preprocessing for nonlinear layers. It is easy to infer that the above preprocessing for the nonlinear layer is secure against the semi-honest client P_1 and the malicious model holder P_0. Specifically, for the client P_1, since the entire preprocessing process does not require the participation of the model holder, the client cannot obtain any private information about the model holder. Similarly, for the malicious model holder P_0, since the preprocessing is non-interactive and the generated ciphertext satisfies the GC security defined in Section II-I, P_0]cannot obtain the plaintext corresponding to the ciphertext sent by the client.

B. Online Phase

In this section, we describe the online phase of *VerifyML*. We first explain how *VerifyML* utilizes the triples generated in the offline phase to generate authenticated shares for matrix-vector multiplication and convolution. Then, we describe the technical details of the nonlinear operation.

Preamble: Consider a neural network (NN) consists of m linear layers and $m-1$ nonlinear layers. Let the specification of the linear layer is $\mathbf{L}_1, \mathbf{L}_1, \cdots \mathbf{L}_m$ and the non-linear layer is $f_1, \cdots, f_{m-1}$.
Input: P_0 holds $\{\mathbf{L}_i\}_{i\in[m]}$, *i.e.,* weights for the m linear layers. P_1 holds $\mathbf{x}_0$ as the input of NN, a random MAC key α from $\mathbb{F}_p$ to be used throughout the protocol execution.
Output: P_b obtains $[\![\mathbf{v}_i = \mathbf{L}_i\mathbf{x}_{i-1}]\!]_b$ for $i \in [m]$ and $b = \{0,1\}$.
Procedure:
Input Sharing:

1. To share P_0's input $\{\mathbf{L}_i\}_{i\in[m]}$, all parties pick up a fresh authenticated element $[\![\mathbf{R}_i]\!]$ of the same dimension as $\mathbf{L}_i$.
2. $[\![\mathbf{R}_i]\!]$ is opened to P_0, and then it sends $\varpi_i = \mathbf{L}_i - \mathbf{R}_i$ to P_1.
3. P_b locally computes $[\![\mathbf{L}_i]\!]_b = [\![\mathbf{R}_i]\!]_b + \varpi_i$ for $b = \{0,1\}$.
4. To share P_1's input $\mathbf{v}_0$, P_1 randomly selects two masks ξ and ζ of the same dimension as $\mathbf{v}_0$. Then, it sends $[\![\mathbf{v}_0]\!]_0 = (\mathbf{v}_0 - \xi, \alpha\mathbf{v}_0 - \zeta)$ to P_0. P_1 sets $[\![\mathbf{v}_0]\!]_1 = (\xi, \zeta)$.
5. For each $i \in [m]$,
   - **Matrix-vector Multiplication**: To generate an authenticated triple of multiplications between matrix $\mathbf{A}$ and vector $\mathbf{b}$, where $\mathbf{A}$ and $\mathbf{b}$ are variables generated in the inference process. P_0 and P_1 take a fresh authenticated matrix-vector triple $\{[\![\mathbf{X}]\!]_b, [\![\mathbf{y}]\!]_b, [\![\mathbf{z}]\!]_b\}_{b\in\{0,1\}}$ of dimensions consistent with $\mathbf{A}$ and $\mathbf{b}$. Then, both party open $\mathbf{A}-\mathbf{X}$ and $\mathbf{b}-\mathbf{y}$. Finally, P_b locally computes $[\![\mathbf{A}*\mathbf{b}]\!]_b$ based on Eqn.(8).
   - **Convolution**: To generate an authenticated triple of Convolution between tensors $\mathbf{A}$ and $\mathbf{B}$, where $\mathbf{A}$ and $\mathbf{B}$ are variables generated in the inference process. P_0 and P_1 take a fresh authenticated Convolution triple $\{[\![\mathbf{X}]\!]_b, [\![\mathbf{Y}]\!]_b, [\![\mathbf{Z}]\!]_b\}_{b\in\{0,1\}}$ of dimensions consistent with $\mathbf{A}$ and $\mathbf{B}$. Then, both party open $\mathbf{A}-\mathbf{X}$ and $\mathbf{b}-\mathbf{Y}$. Finally, P_b locally computes $[\![\mathtt{Conv}(\mathbf{A},\mathbf{B})]\!]_b$ based on Eqn.(8).
6. P_b obtains $[\![\mathbf{v}_i = \mathbf{L}_i\mathbf{x}_{i-1}]\!]_b$ for $i \in [m]$ and $b = \{0,1\}$.

Fig. 6. Online linear layers protocol π_{OLin}.

1) Perform Linear Layers in the Online Phase: Fig. 6 depicts the interaction of the model holder and the client to perform linear layer operations in the online phase. Specifically, given the model holder's input $\{\mathbf{L}_i\}_{i\in[m]}$ and the client's input $\mathbf{v}_0$, both parties first generate authenticated shares of their respective inputs (steps 1-4 in Fig. 6). Since the client is considered semi-honest, its input is shared more efficiently than the model holder, i.e., only local computations are required on randomly selected masks, while the sharing process of model holder's input is consistent with the previous malicious settings [6], [22], [24]. After that, the model holder and the client use the triples generated in the offline phase (i.e., matrix-vector multiplication triples and convolution triples) to generate authenticated sharing of linear layer computation results (step 5 in Fig. 6).

Input: P_0 holds $[\![\mathbf{v}_i]\!]_0$ and P_1 holds $[\![\mathbf{v}_i]\!]_1$ for $i \in [m]$ and $b = \{0,1\}$. In addition, P_1 holds the MAC key α.
Output: P_b obtains $[\![\mathbf{x}_i = ReLU(\mathbf{v}_i)]\!]_b$ and $\langle \alpha \mathbf{v}_i \rangle_b$ for $i \in [m]$ and $b = \{0,1\}$.
Procedure(take single $\mathbf{v}_i$ as an example):

1. Garbled Circuit Phase:
   - P_0 and P_1 invoke the $\mathrm{OT}_\lambda^\kappa$ (see Section 2.8), where P_1's inputs are $\{\mathtt{lab}_{j,0}^{in}, \mathtt{lab}_{j,1}^{in}\}_{j\in\{\kappa+1,\cdots,2\kappa\}}$ while P_0's input is $\langle \mathbf{v}_i \rangle_0$. Hence, P_0 learns $\{\tilde{\mathtt{lab}}_j^{in}\}_{j\in\{\kappa+1,\cdots,2\kappa\}}$. Also, P_1 sends its garbled inputs $\{\{\tilde{\mathtt{lab}}_j^{in} = \mathtt{lab}_{j,\langle \mathbf{v}_i \rangle_1[j]}\}_{j\in[\kappa]}$ to P_0.
   - With $\mathtt{GC}$ and $\{\tilde{\mathtt{lab}}_j^{in}\}_{j\in[2\kappa]}$, P_0 evaluates $\mathtt{GCEval}(\mathtt{GC}, \{\tilde{\mathtt{lab}}_j^{in}\}_{j\in[2\kappa]}) \rightarrow \{\tilde{\mathtt{lab}}_j^{out}\}_{j\in[2\kappa]}$.
2. Authentication Phase 1:
   - P_0 parses $\tilde{\mathtt{lab}}_j^{out}$ as $\tilde{\varsigma}_j || \tilde{\vartheta}_j$ where $\tilde{\varsigma}_j \in \{0,1\}$ and $\tilde{\vartheta}_j \in \{0,1\}^{\lambda-1}$ for every $j \in [2\kappa]$.
   - P_0 computes $c_j = ct_{j,\tilde{\varsigma}_j} \oplus \mathbf{Trun}_\kappa(\tilde{\vartheta}_j)$ and $(d_j || e_j) = \hat{ct}_{j,\tilde{\varsigma}_{i+\kappa}} \oplus \mathbf{Trun}_{2\kappa}(\tilde{\vartheta}_{j+\kappa})$ for every $j \in [\kappa]$.
3. Local Computation Phase:
   - P_1 outputs $\langle g_1 \rangle_1 = (-\sum_{j\in[\kappa]} \iota_{j,1} 2^{j-1})$, $\langle g_2 \rangle_1 = (-\sum_{j\in[\kappa]} \eta_{j,1} 2^{j-1})$ and $\langle g_3 \rangle_1 = (-\sum_{j\in[\kappa]} \gamma_{j,1} 2^{j-1})$.
   - P_0 outputs $\langle g_1 \rangle_0 = (\sum_{j\in[\kappa]} c_j 2^{j-1})$, $\langle g_2 \rangle_0 = (\sum_{j\in[\kappa]} d_j 2^{j-1})$ and $\langle g_3 \rangle_0 = (\sum_{j\in[\kappa]} e_j 2^{j-1})$.
4. Authentication Phase 2:
   - For every $\mathbf{v}_i$ where $i \in [m]$, P_b randomly select a fresh authenticated triple $\{[\![x]\!]_b, [\![y]\!]_b, [\![z]\!]_b\}_{b\in\{0,1\}}$.
   - All parties reveal $\mathbf{v}_i - x$ and $g_2 - y$ to each other, and then locally compute $\langle z_2 \rangle_b = \langle \mathbf{v}_i \cdot sign(\mathbf{v}_i) \rangle_b$ and $\langle z_3 \rangle_b = \langle \alpha \mathbf{v}_i \cdot sign(\mathbf{v}_i) \rangle_b$ based on Eqn.(8).
   - P_b obtains $[\![\mathbf{x}_i = ReLU(\mathbf{v}_i)]\!]_b = (\langle z_2 \rangle_b, \langle z_3 \rangle_b)$ and $\langle \alpha \mathbf{v}_i \rangle_b = \langle g_1 \rangle_b$.

Fig. 7. Online non-linear layers protocol π_{ONlin}.

Security: Our protocol for performing linear layer operations in the online phase, π_{OLin}, is secure against the malicious model holder P_0 and the semi-honest client P_1. We provide the following theorem.

Theorem IV.3: Let triples used in π_{OLin} are generated from $\pi_{Mtriple}$ and $\pi_{Ctriple}$. π_{OLin} is secure against the malicious model holder P_0 and the semi-honest client P_1.

Proof: The proof logic of this theorem is identical to that of [9]. Interested readers can refer to [9] for more details.

2) Perform Non-Linear Layers in the Online Phase: In this section, we present the technical details of the execution of nonlinear functions in the online phase. We mainly focus on how to securely compute the activation function ReLU, which is the most representative nonlinear function in deep neural networks. As shown in Fig. 6, the result $\mathbf{v}_i$ obtained from each linear layer $\mathbf{L}_i$ is held by both parties in the format of authenticated sharing. Similarly, for the function f_i in the i-th nonlinear layer, the goal of *VerifyML* is to securely compute $f_i(\mathbf{v}_i)$ and share it to the model holder and client in the authenticated sharing manner. We describe details in Fig. 7.

Garbled Circuit Phase: As described in Section IV-A3, in the offline phase, P_1 constructs a $\mathtt{GC}$ for the nonlinear part of ReLU (i.e., $sign(\mathbf{v_i})$ for arbitrary input $\mathbf{v_i} \in \mathbb{F}_p$) and sent it to P_0. In the online phase, P_0 and P_1 invoke the $\mathrm{OT}_\lambda^\kappa$, where P_1 as the sender whose inputs are $\{\mathtt{lab}_{j,0}^{in}, \mathtt{lab}_{j,1}^{in}\}_{j\in\{\kappa+1,\ldots,2\kappa\}}$ while P_0's (as the receiver) input is $\langle \mathbf{v}_i \rangle_0$. As a result, P_0 gets set of garbled inputs of $\mathbf{v_i}$ in $\mathtt{GC}$. Then, P_0 evaluates $\mathtt{GC}$ with garbled inputs of $\mathbf{v_i}$ and learns the set of output labels for the bits of $\mathbf{v}_i$ and $sign(\mathbf{v}_i)$.

Authentication Phase 1: This phase aims to calculate the share of the authentication of each bit of $\mathbf{v_i}$, i.e., $sign(\mathbf{v}_i)[j]$, $\alpha sign(\mathbf{v}_i)[j]$, and $\alpha \mathbf{v}_i[j]$ for $j \in [\kappa]$, based on the previous phase. We take an example of how to calculate $\alpha \mathbf{v}_i$. It is clear that the share of $\alpha \mathbf{v}_i[j]$ is either 0 or α depending on whether $\mathbf{v}_i[j]$ is 0 or 1. Recall that the output of the GC is two output labels corresponding to each $\mathbf{v}_i[j]$ (each one for $\mathbf{v}_i[j] = 0$ and 1). We use the symbol $\mathtt{lab}_{j,0}^{out}$ and $\mathtt{lab}_{j,1}^{out}$ to denote $\mathbf{v}_i[j] = 0$ and $\mathbf{v}_i[j] = 1$, respectively. To calculate the shares of $\alpha \mathbf{v}_i[j]$, P_1 randomly selects $\iota_j \in \mathbb{F}_p$ in the offline phase and encrypts it as $\mathtt{lab}_{j,1}^{out}$ and encrypts $\iota_j + \alpha$ as $\mathtt{lab}_{j,0}^{out}$. P_1 sends the two ciphertexts to P_0 and sets its own share of $\alpha \mathbf{v}_i[j]$ to $-\iota_j$. Since P_0 has obtained $\mathtt{lab}_{j,\mathbf{v}_i[j]}^{out}$ in the previous phase, it can definitely decrypt it and obtain its own share of $\alpha \mathbf{v}_i[j]$. Computation of $sign(\mathbf{v}_i)[j]$ and $\alpha sign(\mathbf{v}_i)[j]$ follows a similar logic, utilizing the random values $\eta_{j,1}, \gamma_{j,1}$ sent by P_1 to P_0 in the offline phase, respectively.

Local Computation Phase: This process is used to calculate the share of $sign(\mathbf{v}_i)$, $\alpha sign(\mathbf{v}_i)$, and $\alpha \mathbf{v}_i$ based on the results learned by all parties in the previous stage. For example, to compute the share of $\alpha \mathbf{v}_i$, each party locally multiplies the share of $\alpha \mathbf{v}_i[j]$ with 2^{j-1} and sums all the resultant values. Each party computes the share of $sign(\mathbf{v}_i)$ and $\alpha sign(\mathbf{v}_i)$ in a similar manner.

Authentication Phase 2: We compute the shares of $ReLU(\mathbf{v}_i) = \mathbf{v}_i sign(\mathbf{v}_i)$, and $\alpha ReLU(\mathbf{v}_i)$. Since each party holds the authenticated shares of $\mathbf{v}_i$ and $sign(\mathbf{v}_i)$, we can achieve this based on (8).

Remark IV.4: We adopt two methods to minimize the number of multiplication operations involved in the $\mathtt{GC}$. One is to compute the garbled output of per-bit of $sign(\mathbf{v}_i)$ in $\mathtt{GC}$. Another is to encapsulate only the nonlinear part of ReLU into $\mathtt{GC}$. In this way, we avoid computing $\alpha ReLU(\mathbf{v}_i)$ and $ReLU(\mathbf{v}_i)$ in $\mathtt{GC}$, which is multiply operation intensive. Compared with works [6], [22], [24] with malicious adversary, *VerifyML* reduces the communication overhead of each ReLU function from $2c\lambda + 190\kappa\lambda + 232\kappa^2$ to $2d\lambda + 4\kappa\lambda + 6\kappa^2$, where $d \ll c$.

Remark IV.5: We devise a lightweight method to check whether the model holder's input at the non-linear layer is consistent with what it has learned at the previous layer. Specifically, at the end of evaluating the $i-1$-th linear layer, both parties learns the share of $\alpha \mathbf{v}_i$. Then, $\mathbf{v}_i$ is used as the input of the i-th nonlinear. To check that P_0 is fed the correct input, We require $\alpha \mathbf{v}_i$ to be recomputed in $\mathtt{GC}$ and share again to both parties. Therefore, after evaluating each nonlinear layer, both parties hold two independent shares of $\alpha \mathbf{v}_i$. This provides a way to determine if P_0 provided the correct input by verifying that the two independent shares are consistent (See Section IV-C for more details).

Correctness: We analyze the correctness of our protocol π_{ONlin} as follows. Based on the correctness of $\text{OT}^{\kappa}_{\lambda}$, the model holder P_0 holds $\{\tilde{\texttt{lab}}^{in}_j = \texttt{lab}_{j,\langle \mathbf{v}_i \rangle_0[j]}\}_{j\in\{\kappa+1,\cdots 2\kappa\}}$. Using $\{\tilde{\texttt{lab}}^{in}_j = \texttt{lab}_{j,\langle \mathbf{v}_i \rangle_1[j]}\}_{j\in[\kappa]}$ for $j \in [\kappa]$, and the correctness of (Garble, GCEval) for circuit $booln^f$, we learn $\tilde{\texttt{lab}}^{out}_j = \texttt{lab}^{out}_{j,\mathbf{v}_i[j]}$ and $\tilde{\texttt{lab}}^{out}_{j+\kappa} = \texttt{lab}^{out}_{j+\kappa,sign(\mathbf{v}_i)[j]}$, for $j \in [\kappa]$. Therefore, for $i \in [k]$, we have $\tilde{\varsigma}_j||\tilde{\vartheta}_j = \varsigma_{j,\mathbf{v}_i[j]}||\vartheta_{j,\mathbf{v}_i[j]}$ and $\tilde{\varsigma}_{j+\kappa}||\tilde{\vartheta}_{j+\kappa} = \varsigma_{j+\kappa,sign(\mathbf{v})_i[j]}||\vartheta_{j+\kappa,sign(\mathbf{v}_i)[j]}$. Hence, $c_j = ct_{j,\varsigma_{j,\mathbf{v}_i[j]}} \oplus \mathbf{Trun}_{\kappa}(\vartheta_{j,\mathbf{v}_i[j]}) = \iota_{j,\mathbf{v}_i[j]}$ and $(d_j||e_j) = \hat{ct}_{j,\varsigma_{j+\kappa,sign(\mathbf{v}_i)[j]}} \oplus \mathbf{Trun}_{2\kappa}(\vartheta_{j+\kappa,sign(\mathbf{v}_i)[j]}) = \eta_{j,sign(\mathbf{v}_i)[j]}||\gamma_{j,sign(\mathbf{v}_i)[j]}$. Based on these, we have

- $g_1 = \sum_{j\in[\kappa]}(c_j - \iota_{j,0})2^{j-1} = \sum_{j\in[\kappa]} \alpha(\mathbf{v}_i[j])2^{j-1} = \alpha\mathbf{v}_i$.
- $g_2 = \sum_{j\in[\kappa]}(d_j - \eta_{j,0})2^{j-1} = \sum_{j\in[\kappa]}(sign(\mathbf{v}_i)[j])2^{j-1} = sign(\mathbf{v}_i)$.
- $g_3 = \sum_{j\in[\kappa]}(e_j - \gamma_{j,0})2^{j-1} = \sum_{j\in[\kappa]} \alpha(sign(\mathbf{v}_i)[j])2^{j-1} = \alpha sign(\mathbf{v}_i)$.

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

Security: Our protocol for performing nonlinear layer operations in the online phase, π_{ONlin}, is secure against the malicious model holder P_0 and the semi-honest client P_1. We provide the following theorem and prove it in Appendix D, available online.

Theorem IV.4: Let (Garble, GCEval) be a garbling scheme with the properties defined in Section II-I. Authenticated shares have the properties defined in Section II-F. Then our protocol π_{ONlin} is secure against the malicious model holder P_0 and the semi-honest client P_1.

C. Consistency Check

VerifyML performs π_{OLin} and π_{ONlin} alternately in the online phase to output the inference result $\mathbf{M}(\mathbf{x}_0)$ for a given input $\mathbf{x}_0$, where all intermediate results output by the nonlinear layer and the linear layer are held on P_0 and P_1 in an authenticated sharing manner. To verify the correctness of $\mathbf{M}(\mathbf{x}_0)$, the client needs to perform a consistency check on all computed results. If the verification passes, P_1 locally evaluates the fairness of the ML model based on (2). Otherwise, abort. In more detail, for sharing P_0's input and executing each linear layer $\{\mathbf{L}_i\}_{i\in[m]}$, *VerifyML* needs to pick up a large number of fresh authenticated single elements or triples (see Fig. 6) and open them for computation. Assume that the set of all opened elements is $(a_1, a_2 \cdots, a_t)$, and P_b holds $\langle \boldsymbol{\rho}_i \rangle_b = \langle \alpha a_i \rangle_b$ as well as $\langle \boldsymbol{\tau}_i \rangle_b = \langle a_i \rangle_b$, we need to perform a consistency check to verify $\boldsymbol{\rho}_i - \alpha\boldsymbol{\tau}_i = 0$. Beside, For executing each nonlinear layer $\{f_i\}_{i\in[m-1]}$, the inputs of π_{ONlin} are shares of $\mathbf{v}_i$ and $\boldsymbol{\tau}_i = \alpha\mathbf{v}_i$. To check that P_0 is fed the correct input, We require $\alpha\mathbf{v}_i$ to be recomputed in the GC and share it again to both parties, denoting the new $\alpha\mathbf{v}_i$ as ξ_i. We also need to perform a consistency check to verify $\sum_{i=1}^{i=m} \boldsymbol{\tau}_i - \xi_i = \mathbf{0}$.

Fig. 8 presents the details of consistency check, where we combine all the above checks into a single check by using random scalars picked by P_1. The correctness of π_{Ocheck} can be easily deduced by inspecting the implementation of the protocol. Specifically, By correctness of π_{OLin}, we have $\boldsymbol{\rho}_j - \alpha\boldsymbol{\tau}_j = (\langle \boldsymbol{\rho}_j \rangle_0 - \alpha_0 a_j + \langle \boldsymbol{\rho}_j \rangle_1 - \alpha_1 a_j) = 0$ for every linear layer $\{\mathbf{L}_j\}_{j\in[m]}$. By correctness of π_{ONlin}, we have $\boldsymbol{\tau}_i - \xi_i = (\langle \boldsymbol{\tau}_i \rangle_0 - \langle \xi_i \rangle_0) + (\langle \boldsymbol{\tau}_i \rangle_1 - \langle \xi_i \rangle_1) = 0$ for all nonlinear layers. Hence, we have $\langle q \rangle_0 + \langle q \rangle_1 = \sum_{j\in[t]} \mathbf{r}_j(\boldsymbol{\rho}_j - \alpha\boldsymbol{\tau}_j) + \sum_{i\in[m-1]} \mathbf{r}_i(\boldsymbol{\tau}_i - \xi_i) = 0$.

Input: P_b $b \in \{0,1\}$ holds $\langle \tau_i \rangle_b$, $\langle \xi_i \rangle_b$ and $[\![a_j]\!]_b$ for $i \in [m-1]$ and $j \in [t]$.
Output: P_1 obtains $\mathbf{M}(\mathbf{x}_0)$ if verification passes. Otherwise, abort.
Procedure

- For $i \in [m]$ and $j \in [t]$, P_1 uniformly samples $\mathbf{r}_i$ and $\mathbf{r}_j$ and sends them to P_0.
- P_0 computes $\langle q \rangle_0 = \sum_{j\in[t]} \mathbf{r}_j(\langle \rho_j \rangle_0 - \alpha_0 a_j) + \sum_{i\in[m-1]} \mathbf{r}_i(\langle \tau_i \rangle_0 - \langle \xi_i \rangle_0)$, and sends $\langle q \rangle_0$ to P_1.
- P_1 computes $\langle q \rangle_1 = \sum_{j\in[t]} \mathbf{r}_j(\langle \rho_j \rangle_1 - \alpha_1 a_j) + \sum_{i\in[m-1]} \mathbf{r}_i(\langle \tau_i \rangle_1 - \langle \xi_i \rangle_1)$.
- P_1 aborts if $\langle q \rangle_0 + \langle q \rangle_1 \neq 0 \mod p$. Else, P_1 locally evaluates the fairness of the ML model based on Eqn.(2) by reconstructing $\mathbf{M}(\mathbf{x}_0)$.

Fig. 8. Consistency check protocol π_{Ocheck}.

Security: We demonstrate that the consistency check protocol π_{Ocheck} have an overwhelming probability to abort if P_0 tampered with the input during execution. We provide the following theorem and prove it in Appendix E, available online.

Theorem IV.5: In real execution, if P_0 tampers with its input, then P_1 aborts with probability at least $1 - 1/p$.

V. Performance Evaluation

In this section, we conduct experiments to demonstrate the performance of *VerifyML*. Since there is no secure inference protocol specifically designed for the malicious model holder threat model, we choose the state-of-the-art generic MPC framework Overdrive [24][7] as the baseline. Note that we also consider the client as a semi-honest entity when implementing Overdrive, so that Overdrive can also utilize the properties of semi-honest client to avoid redundant verification and zero-knowledge proof. In this way, we can "purely" discuss the technical advantages of *VerifyML* over Overdrive, while excluding the inherent advantages of *VerifyML* due to the weaker threat model. Specifically, we analyze the performance of *VerifyML* from offline and online phases, respectively, where we discuss the superiority of *VerifyML* over Overdrive in terms of computation and communication cost in performing linear and non-linear layers. In the end, We demonstrate the cost superiority of *VerifyML* compared to Overdrive on mainstream models including ResNet-18 and LeNet.

A. Implementation Details

VerifyML is implemented through the C++ language and provides 128 bits of computational security and 40 bits of

[7]Although work [6] shows better performance compared to Overdrive, it is difficult to compare with [6] because of the unavailability of its code. However, we clearly outperform [6] by constructing a more efficient method to generate triples. In addition, [6] requires fitting nonlinear functions such as ReLU to a quadratic polynomial to facilitate computation, which is also contrary to the motivation of this paper.

TABLE I
COST OF GENERATING THE MATRIX-VECTOR MULTIPLICATION TRIPLE

Dimension	Comm.cost (MB)		Running time (s)			
	Overdrive	*VerifyML*(Reduction)	Overdrive LAN	Overdrive WAN	*VerifyML* (speedup) LAN	*VerifyML* (speedup) WAN
1×4096	27.1	**2.1 (12.9×)**	2.3	17.7	**0.9 (2.6×)**	**12.4 (1.5×)**
16×2048	216.4	**17.6 (12.3×)**	15.3	26.2	**7.6 (2.0×)**	**14.1 (1.6×)**
16×4096	432.8	**34.5 (12.5×)**	30.6	43.4	**15.1 (2.0×)**	**26.9 (1.6×)**
64×2048	865.6	**68.3 (12.7×)**	60.9	72.4	**29.2 (2.1×)**	**40.7 (1.7×)**
64×4096	1326.2	**135.7 (9.8×)**	103.0	114.8	**57.8 (1.8×)**	**68.2 (1.6×)**
128×4096	2247.4	**271.9 (8.3×)**	187.1	199.1	**117.3 (1.6×)**	**128.4 (1.5×)**

TABLE II
COST OF GENERATING THE CONVOLUTION TRIPLE

Input	Kernel	Comm.cost (GB)		Running time (s)			
		Overdrive	*VerifyML*	Overdrive LAN	Overdrive WAN	*VerifyML* (Speedup) LAN	*VerifyML* (Speedup) WAN
16×16 @128	1×1 @128	17.1	**2.1**	1476.1	1494.6	**924.7 (1.6×)**	**938.4 (1.6×)**
16×16 @256	1×1 @256	67.8	**8.2**	6059.3	6059.31	**3568.8 (1.7×)**	**3580.8 (1.7×)**
16×16 @512	3×3 @128	467.5	**56.8**	40753.4	40767.1	**25387.2 (1.6×)**	**25401.5 (1.6×)**
32×32 @2048	5×5 @512	83127.8	**7324.3**	7245056.2	7245068.8	**4521023.3 (1.6×)**	**4521165.6 (1.6×)**

statistical security. The entire system operates on the 44-bit prime field. We utilize the SEAL homomorphic encryption library [40] to perform nonlinear layers including generative matrix-vector multiplication and convolution triples, where we set the maximum number of slots allowed for a single ciphertext as 4096. The garbled circuit for the nonlinear layer is constructed on SIMC [5][8], which implements the evaluation of GC on the basis of the **emp-sh2pc** protocol [9] [43], and also provides the operation mod a prime p [10]. Zero-knowledge proofs of plaintext knowledge are implemented based on MUSE [28]. Our experiments are carried out in both the LAN and 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 Offline Phase

1) Cost of Generating Matrix-Vector Multiplication Triple: TABLE I describes the comparison of the overhead of *VerifyML* and Overdrive in generating matrix-vector multiplication triples in different dimensions. It is clear that *VerifyML* is superior in performance to Overdrive, both in terms of communication overhead and computational overhead. We observe that *VerifyML* achieves more than $8\times$ reduction in communication overhead and at least $1.5\times$ speedup in computation compared to Overdrive. This stems from Overdrive's disadvantage in constructing triples, i.e., constructing triples for only a single multiplication operation (or multiplication between a single row of a matrix and a vector). In addition, the generation process requires frequent interaction between the client and the model holder (for zero-knowledge proofs and preventing breaches by either party). This inevitably incurs substantial computational and communication overhead. Our constructed matrix-multiplication triples enable the communication overhead to be independent of the number of multiplications, only related to the size of the input. This substantially reduces the amount of data that needs to be exchanged between P_0 and P_1. In addition, we move the majority of the computation to be executed by P_1, which avoids the need for distributed decryption and frequent zero-knowledge proofs in malicious adversary settings. Moreover, our matrix-vector multiplication does not involve any rotation operation. As a result, these optimization methods motivate *VerifyML* to exhibit a satisfactory performance overhead in generating triples.

2) Cost of Generating Convolution Triple: TABLE II shows the comparison of the performance of *VerifyML* and Overdrive in generating convolution triples in different dimensions, where input tensor of size $u_w \times u_h$ with c_i channels is denoted as $u_w \times u_h @ c_i$, and the size of corresponding kernel is denoted as $k_w \times k_h @ c_o$. We observe that *VerifyML* is much lower than Overdrive in terms of computational and communication overhead. For instance, *VerifyML* gains a reduction of up to $9\times$ in communication cost and a speedup of at least $1.6\times$ in computation. This is due to the optimization method customized by *VerifyML* for generating convolution triples. Compared to Overdrive, which focuses on constructing authenticated triples for a single multiplication operation, *VerifyML* uses the homomorphic parallel matrix multiplication method constructed in [19] as the underlying structure to construct matrix multiplication triples equivalent to convolution triples. Since a single matrix is regarded as a computational entity, the above method makes the communication overhead between the client and the model holder only related to the size of the matrix, and independent of the number of operations of the multiplication between the two matrices (that is, the communication complexity is reduced from $O(d^3)$ to $O(d)^2$ given the multiplication between the two $d \times d$ matrices). In addition, the optimized parallel matrix multiplication reduces the homomorphic rotation operation from $O(d^2)$ to $O(d)$. This enables *VerifyML* to show significant superiority in computing convolution triples.

C. Performance of Online Phase

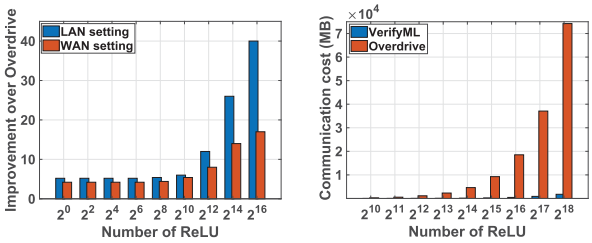
In the online phase, *VerifyML* is required to perform operations at the linear and nonlinear layers alternately. Here we discuss the overhead performance of *VerifyML* compared to Overdrive separately.

1) Performance of Executing Linear Layers: Since both *VerifyML* and Overdrive follow the same computational logic to perform the linear layer in the online phase, i.e., use pre-generated authenticated triples to compute matrix-vector multiplication and convolution, both exhibit similar compu- tational overhe. Therefore, we focus on analyzing the difference in communication overhead between the two of executing convolution.

[8]https://aka.ms/simc
[9]https://github.com/emp-toolkit/emp-sh2pc
[10]https://github.com/shahakash28/simc/blob/master/test/msi_relu_final.cpp#L244

TABLE III
COMPARISON OF THE COMMUNICATION OVERHEAD FOR EXECUTING CONVOLUTION IN THE ONLINE PHASE

Input	Kernel	Comm.cost (MB)	
		Overdrive	*VerifyML* (Reduction)
16×16 @128	1×1 @128	46.1	**0.5 (85.3×)**
16×16 @256	1×1 @256	184.5	**1.4 (128.0×)**
16×16 @512	3×3 @128	1271.7	**15.7 (81.2×)**
32×32 @2048	5×5 @512	226073.0	**1459.8 (154.9×)**



(a) (b)

Fig. 9. Comparison of the overhead for executing nonlinear layers. ((a) Running time improvement of *VerifyML* over Overdrive. The y-axis shows $\frac{\text{Overdrive time}}{\text{VerifyML time}}$ (b) Comparison of the communication overhead.

Table III depicts the communication overhead of *VerifyML* and Overdriveffor computing convolution in different dimensions. It is obvious that *VerifyML* shows superior performance in communication overhead compared to Overdrive. This is mainly due to the fact that Overdrive needs to open a fresh authenticated Beaver's multiplication triple for each multiplication operation, which makes the communication overhead of executing the entire linear layer positively related to the total multiplication operations involved. In contrast, *VerifyML* customizes matrix-vector multiplication and convolution triples, which makes the cost independent of the number of multiplication operations in the linear layer. This substantially reduces the amount of data that needs to be exchanged during the execution.

2) Performance of Executing Nonlinear Layers: Fig. 9 provides the comparison of the cost between Overdrive and *VerifyML*. We observe that *VerifyML* outperforms Overdrive by $4-42\times$ in runtime on LAN Setting and $3-16\times$ in WAN Setting. For example, Overdrive takes 165.4 s and 1283.5 s to compute 2^{15} ReLUs on LAN and WAN setting, respectively. Whereas, *VerifyML* took just 5.1 s and 110.2 s in the respective network settings. For communication overhead, we observed that Overdrive required 401 KB of traffic to perform a single ReLU while we only need 8.33 KB, which is at least a $48\times$ improvement. This is mainly due to the fact that our optimized `GC` substantially reduces the multiplication operations involved in evaluating in the `GC`. Moreover, Overdrive needs to verify the correctness of the input from the model holder in the `GC`, which is very expensive. Conversely, *VerifyML* designs lightweight consistency verification methods to achieve this.

TABLE IV
COST OF END-TO-END SECURE INFERENCE

	Phases	Comm.cost (MB)		Running time (s)			
		Overdrive	*VerifyML*	Overdrive		*VerifyML* (Speedup)	
				LAN	WAN	LAN	WAN
LeNet	Offline	3427.8	**209.6**	235.5	246.8	**92.9 (2.5×)**	**104.6 (2.4×)**
	Online	2543.1	**54.0**	32.8	254.9	**1.0 (32.6×)**	**21.9 (11.6×)**
	Total	5970.9	**263.6**	268.3	501.7	**93.9 (2.9×)**	**126.5 (4.0×)**
	Phases	Comm.cost (MB)		Running time (s)			
		Overdrive	*VerifyML*	Overdrive		*VerifyML* (Speedup)	
				LAN	WAN	LAN	WAN
ResNet18	Offline	2116018.6	**257257.7**	238774.2	238957.4	**114003.1 (2.1×)**	**114978.8 (2.1×)**
	Online	19359.5	**459.4**	177.0	1373.7	**5.5 (32.2×)**	**117.9 (11.7×)**
	Total	2135378.1	**257717.1**	238951.2	240331.1	**114008.6 (2.1×)**	**115096.7 (2.1×)**

D. Performance of End-to-End Secure Inference

We compare the performance of *VerifyML* and Overdrive on real-world ML models. In our experiments, we choose ResNet-18 and LeNet, which are trained on the CelebA [31] and C-MNIST datasets [2] respectively. Note that CelebA and C-MNIST are widely used to check how fair a given trained model is. TABLE IV shows the performance of *VerifyML* and Overdrive in terms of computation and communication overhead. Compared to Overdrive, *VerifyML* demonstrates an encouraging online runtime boost by $32.6\times$ and $32.2\times$ over existing works on LeNet and ResNet-18, respectively, and at least an order of magnitude communication cost reduction. In online phase, Overdrive takes 32.8 s and 177 s to compute single query on LeNet and ReNet-18, respectively. Whereas, *VerifyML* took just 1 s and 5.5 s in the respective network settings. Consistent with the previous analysis, this stems from the customized optimization mechanism we designed for *VerifyML*.

E. Comparison With Other Works

Compared With Fusion: A very recently work Fusion [10], also aims to ensure computation correctness on server-malicious threat model. We would like to clarify that Fusion was published after the completion of our work on VerifyML, and they employ different approaches, each with its own advantages and drawbacks. At a high level, VerifyML utilizes cryptographic primitives as the underlying architecture to achieve computation correctness. Its approach is independent of the machine learning models and application scenarios but incurs non-negligible computational and communication overhead. On the other hand, Fusion seeks to strike a balance between efficiency and accuracy in verifying computation correctness, relying on a stronger assumption compared to VerifyML. Fusion requires a certain amount of public query sample sets that are highly related to the target model to detect server malicious behavior. While this requirement may be unrealistic, it is a mandatory aspect of their design. Unlike VerifyML, Fusion employs a custom method known as the "mix-and-check method" to detect malicious server behavior without involving cryptographic primitives. As a result, Fusion offers significant advantages in terms of computation and communication efficiency compared to VerifyML.

Specifically, In our VerifyML approach, we leverage cryptographic primitives to detect malicious behaviors exhibited by

the cloud server. To achieve this, we employ custom authenticated matrix-vector triples and authenticated convolution triples, alongside zero-knowledge proof protocols [22], [24], enabling us to perform verifiable linear layer operations. To minimize the computational overhead associated with homomorphic operations, particularly rotation operations, we have developed parallel optimization strategies, resulting in significant savings. For the nonlinear layer, we implement verifiable computations based on the evaluation of garbled circuits, combined with authentication sharing, within a semi-honest threat model. Our key insight is that garbled circuits already provide malicious security against garbled circuit evaluators (i.e., the server) [28]. Therefore, we have constructed a lightweight method that solely checks the consistency between the input of the malicious adversary in the nonlinear layer and the results obtained from the previous linear layer.

In contrast, Fusion takes a different approach by bypassing cryptography and instead designing a highly efficient verifiable computing protocol known as the "mix-and-check method." This method draws inspiration from the classic cut-and-choose technique [30] but incorporates non-trivial modifications to suit machine learning inference in a server-malicious threat model. The entire process of verifying the correctness of server calculations in Fusion is exceptionally efficient because it does not rely on any cryptographic primitives. However, compared to VerifyML, Fusion makes an additional assumption that the client has access to a subset of public samples, which are of the same type as its query samples, and knows the outputs of those public samples in advance. This assumption is crucial for Fusion's efficient implementation. Nevertheless, in privacy-critical scenarios, particularly within financial and healthcare domains [45], acquiring a large number of public input-output samples for clients can be challenging. Furthermore, the inherent cryptographic primitives in VerifyML ensure that the server can only deceive the client with a negligible probability (less than 2^{-40}) per query. In contrast, Fusion needs to perform a significant number of queries (usually thousands of times) to achieve the same level of statistical security. Therefore, when compared to VerifyML, Fusion offers an excellent advantage in terms of computing performance. However, VerifyML does not require any assumptions about model tasks and public samples, and the verification operation only requires executing a single query, resulting in better scalability.

Quantitative Comparison: We also conduct experimental comparisons to assess the computational and communication overheads of both schemes in performing ML inference. Following the experimental configurations provided by Fusion [11], we implemented Fusion using Cheetah [18]. Our ML model consisted of a four-layer deep neural network (DNN) with three hidden fully connected layers, each comprising 2000 neurons with quadratic activations. The final layer was fully connected with 183 output neurons. In our experiments, we set the number of query samples to be the same as Fusion. This means the client prepared a mixed dataset containing a total of $1845 \times 5 + 100$ samples, which included 1845 query samples (each with 5 copies) and an additional 100 speech samples used as public samples. Fig 10 illustrates the results of the comparison. Our experimental results clearly demonstrate that Fusion outperforms VerifyML in terms of performance. Fusion exhibits $3.8\times$ lower communication overhead and requires $18\times$ less runtime compared to VerifyML. The primary reason for this performance difference is that Fusion only involves plaintext operations during the verification process of the malicious server's calculation results and does not rely on any cryptographic primitives. On the other hand, VerifyML achieves the same objective through the inclusion of zero-knowledge proof protocols, authentication sharing, and computationally intensive homomorphic multiplication operations.

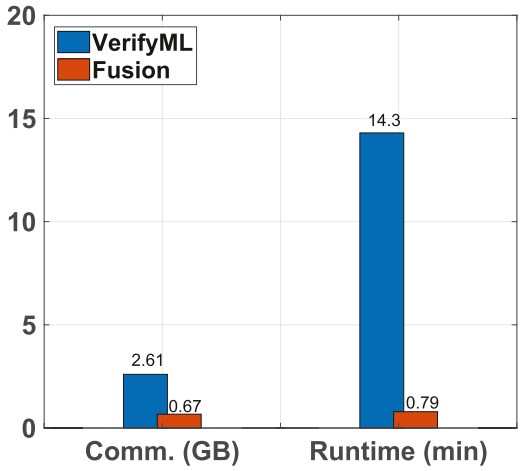


Fig. 10. Comparison between VerifyML and fusion.

However, as mentioned earlier, VerifyML does not impose any assumptions about model tasks and public samples, and the verification operation only requires executing a single query, resulting in better scalability.

Compared With DELPHI: We demonstrate that for the execution of non-linear layers, the communication overhead of *VerifyML* is even lower than the state-of-the-art scheme DELPHI [32] under hte semi-honest threat model. Specifically, for the i-th nonlinear layer, DELPHI needs to calculate shares of $f_i(\mathbf{v_i})$ in GC and share it with two parties. DELPHI requires at least 3κ additional AND gates, which incurs at least $6\kappa\lambda$ bits of communication, compared to only computing each bit of $f_i(\mathbf{v_i})$ in *VerifyML*. In our experiment, For $\kappa = 44, \lambda = 28$, our method gives roughly $9\times$ less communication for generating shares of $f_i(\mathbf{v_i})$, i.e., DELPHI required 32 KB of traffic to perform a single ReLU while we only need 8.33 KB.

Compared With MUSE and SIMC: We note that several works such as MUSE [28] and SIMC [5] have been proposed to address ML secure inference on the *client malicious* threat model. Such a threat model considers that the server (i.e., the model holder) is semi-honest but the malicious client may arbitrarily violate the protocol to obtain private information. These works intuitively seem to translate to our application scenarios with appropriate modification. However, we argue that this is non-trivial. In more detail, in the *client malicious* model, the client's inputs are encrypted and sent to the semi-honest model holder, which

[11] https://github.com/daisy611/Fusion

performs all linear operations for speeding up the computation. Since the model holder holds the model parameter in the plaintext, executing the linear layer only involves homomorphic operations between the plaintext and the ciphertext. Such type of computation is compatible with mainstream homomorphic optimization methods including GALA [47] and GAZELLE [21]. However, in *VerifyML*, the linear layer operation cannot be done in the model holder because it is considered malicious. One possible approach is to encrypt the model data and perform linear layer operations with two-party interaction. This is essentially performing homomorphic operations between ciphertext and ciphertext, which is not compatible with previous optimization strategies. Therefore, instead of simply fine-tuning MUSE [28] and SIMC [5], we must redesign new parallel homomorphic computation methods to fit this new threat model. On the other hand, we observe that the techniques for nonlinear operations in MUSE [28] and SIMC [5] can clearly be transferred to *VerifyML*. However, our method still outperforms SIMC (an upgraded version of MUSE). This mainly stems from the fact that we only encapsulate the nonlinear part of ReLU into GC to further reduce the number of multiplication operations. Experiments show that our method is about one third of SIMC in terms of computing and communication overhead.

VI. Conclusion

In this paper, we proposed *VerifyML*, the first secure inference framework to check the fairness degree of a given ML model. We designed a series of optimization methods to reduce the overhead of the offline stage. We also presented optimized GC 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 *VerifyML*, to make secure ML inference more suitable for a wider range of practical applications.

References

[1] F. A. Lieberman, "How data scientists can create a more inclusive financial services landscape," 2022. [Online]. Available: https://venturebeat.com/datadecisionmakers/how-data-scientists-can-create-a-more-inclusive-financial-services-landscape/

[2] M. Arjovsky, L. Bottou, I. Gulrajani, and D. Lopez-Paz, "Invariant risk minimization," 2019, *arXiv:1907.02893*.

[3] R. K. E. Bellamy et al., "AI fairness 360: An extensible toolkit for detecting, understanding, and mitigating unwanted algorithmic bias," 2018, *arXiv:1810.01943*.

[4] S. Biswas and H. Rajan, "Do the machine learning models on a crowd sourced platform exhibit bias? An empirical study on model fairness," in *Proc. ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 642–653.

[5] N. Chandran, D. Gupta, S. L. B. Obbattu, and A. Shah, "SIMC: ML inference secure against malicious clients at semi-honest cost," in *Proc. 31st USENIX Secur. Symp.*, 2021, pp. 1361–1378.

[6] H. Chen, M. Kim, I. Razenshteyn, D. Rotaru, Y. Song, and S. Wagh, "Maliciously secure matrix multiplication with applications to private deep learning," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, 2020, pp. 31–59.

[7] A. Chouldechova, D. Benavides-Prado, O. Fialko, and R. Vaithianathan, "A case study of algorithm-assisted decision making in child maltreatment hotline screening decisions," in *Proc. Conf. Fairness Accountability Transparency*, 2018, pp. 134–148.

[8] M. Ciampi, V. Goyal, and R. Ostrovsky, "Threshold garbled circuits and ad hoc secure computation," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2021, pp. 64–93.

[9] I. Damgård, V. Pastro, N. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *Proc. Annu. Cryptol. Conf.*, 2012, pp. 643–662.

[10] C. Dong et al., "Fusion: Efficient and secure inference resilient to malicious server and curious clients," in *Proc. Netw. Distrib. Syst. Secur.*, 2023, pp. 1–18.

[11] N. Döttling, S. Garg, M. Hajiabadi, D. Masny, and D. Wichs, "Two-round oblivious transfer from CDH or LPN," in *Proc. Int. Conf. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2020, pp. 768–797.

[12] U. Feige, A. Fiat, and A. Shamir, "Zero-knowledge proofs of identity," *J. Cryptol.*, vol. 1, no. 2, pp. 77–94, 1988.

[13] T. K. Frederiksen, T. P. Jakobsen, J. B. Nielsen, P. S. Nordholt, and C. Orlandi, "MiniLEGO: Efficient secure two-party computation from general assumptions," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2013, pp. 537–556.

[14] A. B. Grilo, H. Lin, F. Song, and V. Vaikuntanathan, "Oblivious transfer is in MiniQCrypt," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2021, pp. 531–561.

[15] S. Halevi and V. Shoup, "Algorithms in HElib," in *Proc. Annu. Cryptol. Conf.*, 2014, pp. 554–571.

[16] C. Hazay, E. Orsini, P. Scholl, and E. Soria-Vazquez, "Concretely efficient large-scale MPC with active security (or, TinyKeys for TinyOT)," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, 2018, pp. 86–117.

[17] A. Howard and J. Borenstein, "The ugly truth about ourselves and our robot creations: The problem of bias and social inequity," *Sci. Eng. Ethics*, vol. 24, no. 5, pp. 1521–1536, 2018.

[18] Z. Huang, W.-J. Lu, C. Hong, and J. Ding, "Cheetah: Lean and fast secure two-party deep neural network inference," in *Proc. USENIX Secur. Symp.*, 2022, pp. 809–826.

[19] X. Jiang, M. Kim, K. Lauter, and Y. Song, "Secure outsourced matrix computation and application to neural networks," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 1209–1222.

[20] K. Martin, *Ethics of Data and Analytics: Concepts and Cases*. Boca Raton, FL, USA: CRC Press, 2010. [Online]. Available: https://books.google.com.hk/books?id=E51kEAAAQBAJ

[21] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "GAZELLE: A low latency framework for secure neural network inference," in *Proc. USENIX Secur. Symp.*, 2018, pp. 1651–1669.

[22] M. Keller, "MP-SPDZ: A versatile framework for multi-party computation," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2020, pp. 1575–1590.

[23] M. Keller, E. Orsini, and P. Scholl, "Actively secure OT extension with optimal overhead," in *Proc. Annu. Cryptol. Conf.*, 2015, pp. 724–741.

[24] M. Keller, V. Pastro, and D. Rotaru, "Overdrive: Making SPDZ great again," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2018, pp. 158–189.

[25] V. Kolesnikov, P. Mohassel, and M. Rosulek, "FleXOR: Flexible garbling for XOR gates that beats free-XOR," in *Proc. Annu. Cryptol. Conf.*, 2014, pp. 440–457.

[26] P. Lahoti et al., "Fairness without demographics through adversarially reweighted learning," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2020, pp. 728–740.

[27] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[28] R. Lehmkuhl, P. Mishra, A. Srinivasan, and R. A. Popa, "MUSE: Secure inference resilient to malicious clients," in *Proc. USENIX Secur. Symp.*, 2021, pp. 2201–2218.

[29] 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.

[30] Y. Lindell and B. Pinkas, "An efficient protocol for secure two-party computation in the presence of malicious adversaries," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, Springer, 2007, pp. 52–78.

[31] Z. Liu, P. Luo, X. Wang, and X. Tang, "Deep learning face attributes in the wild," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2015, pp. 3730–3738.

[32] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "DELPHI: A cryptographic inference service for neural networks," in *Proc. USENIX Secur. Symp.*, 2020, pp. 2505–2522.

[33] P. Mohassel and Y. Zhang, "SecureML: A system for scalable privacy-preserving machine learning," in *Proc. IEEE Symp. Secur. Privacy*, 2017, pp. 19–38.

[34] D. Mukherjee, M. Yurochkin, M. Banerjee, and Y. Sun, "Two simple ways to learn individual fairness metrics from data," in *Proc. Int. Conf. Mach. Learn.*, 2020, pp. 7097–7107.

[35] L. Oneto and S. Chiappa, "Fairness in machine learning," in *Recent Trends in Learning From Data*. Berlin, Germany: Springer, 2020, pp. 155–196.

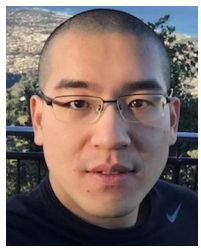
[36] O. A. Osoba, and W. Welser IV, *An Intelligence in Our Image: The Risks of Bias and Errors in Artificial Intelligence*. Santa Monica, CA, USA: Rand Corporation, 2017.
[37] F. Prost et al., "Measuring model fairness under noisy covariates: A theoretical perspective," in *Proc. AAAI/ACM Conf. AI Ethics Soc.*, 2021, pp. 873–883.
[38] M. Rosulek and L. Roy, "Three halves make a whole? Beating the half-gates lower bound for garbled circuits," in *Proc. Annu. Int. Cryptol. Conf.*, 2021, pp. 94–124.
[39] P. Saleiro et al., "Aequitas: A bias and fairness audit toolkit," 2018, *arXiv:1811.05577*.
[40] Microsoft SEAL (release 4.0), Microsoft Research, Redmond, WA, USA, Mar. 2022. [Online]. Available: https://github.com/Microsoft/SEAL
[41] S. Segal, Y. Adi, B. Pinkas, C. Baum, C. Ganesh, and J. Keshet, "Fairness in the eyes of the data: Certifying machine-learning models," in *Proc. AAAI/ACM Conf. AI Ethics Soc.*, 2021, pp. 926–935.
[42] N. P. Smart and F. Vercauteren, "Fully homomorphic SIMD operations," *Des. Codes Cryptogr.*, vol. 71, no. 1, pp. 57–81, 2014.
[43] X. Wang, A. J. Malozemoff, and J. Katz, "EMP-toolkit: Efficient multi-party computation toolkit," 2016. [Online]. Available: https://github.com/emp-toolkit
[44] W. Paul, W. Rifkind, and L. L. P. Garrison, "Breaking new ground, CFPB will pursue discrimination as an "unfair" practice across the range of consumer financial services," 2022. [Online]. Available: https://www.paulweiss.com/practices/litigation/white-collar-regulatory-defense/publications/breaking-new-ground-cfpb-will-pursue-discrimination-as-an-unfair-practice-across-the-range-of-consumer-financial-services?id=42867
[45] X. Yin, Y. Zhu, and J. Hu, "A comprehensive survey of privacy-preserving federated learning: A taxonomy, review, and future directions," *ACM Comput. Surv.*, vol. 54, no. 6, pp. 1–36, 2021.
[46] S. Zahur, M. Rosulek, and D. Evans, "Two halves make a whole," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2015, pp. 220–250.
[47] Q. Zhang, C. Xin, and H. Wu, "GALA: Greedy computation for linear algebra in privacy-preserved neural networks," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2021, pp. 1–16.

Guowen Xu 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 Transactions on Multimedia*, *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.

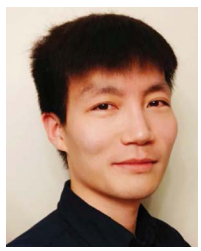
Gelei Deng received the bachelor's degree from the Singapore University of Technology and Design, in 2018. He is currently working toward the PhD degree with Nanyant Technological University, Singapore. His research interests include computer system security and security testing.

Tianwei Zhang received the bachelor's degree from Peking University, in 2011, and the PhD degree from the Princeton University, in 2017. He is an assistant professor with the School of Computer Science and Engineering, Nanyang Technological University. His research interests include 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 received the PhD 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 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 Information Forensics and Security*, *IEEE Transactions on Dependable and Secure Computing*, etc.



Anjia Yang received the PhD degree from the Department of Computer Science, City University of Hong Kong, in 2015. He held a postdoctoral position with the City University of Hong Kong from 2015 to 2016, and in Jinan University from 2016 to 2019, respectively. From 2018 to 2019, he was a visiting scholar in BBCR Group with the University of Waterloo. He is currently an associate professor with Jinan University, Guangzhou. His research interests include security and privacy in Internet of Things, vehicular networks, blockchain and cloud computing, etc. He has published more than 30 international papers including journals and conferences, such as *IEEE Transactions on Dependable and Secure Computing*, *IEEE Transactions on Mobile Computing*, *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Services Computing*, *IEEE Transactions on Cloud Computing*, *IEEE Transactions on Intelligent Transportation Systems*, *IEEE Transactions on Vehicular Technology*, ESORICS, WiSec et al. He served as PC members or organizers for more than 20 international conferences. He also serves as an academic editor for Security and Communication Networks.

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.