

VerifyML: Obviously Checking Model Fairness Resilient to Malicious Model Holder

Guowen Xu, Xingshuo Han, Gelei Deng, Tianwei Zhang, Shengmin Xu, Jianting Ning, Anjia Yang, Hongwei Li

Abstract—In this paper, 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 (2PC) 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 the 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 $4\times\text{--}42\times$ faster and has $> 48\times$ lesser communication than existing 2PC protocol against malicious parties. In fact, *VerifyML* even beats the state-of-the-art semi-honest ML secure inference system! We provide formal theoretical analysis for *VerifyML* security and demonstrate its performance superiority on mainstream ML models including ResNet-18 and LeNet.

Keywords—Privacy Protection, Deep Learning, Cryptography.



1 INTRODUCTION

Machine learning (ML) systems are increasingly being used to inform and influence people’s decisions, leading to algorithmic outcomes that have powerful implications for individuals and society. For example, most personal loan default risks are calculated by automated ML tools. This approach greatly speeds up the decision-making process, but as with any decision-making algorithm, there is a tendency to provide accurate results for the majority, leaving certain individuals and minority groups disadvantaged [1], [41]. This problem is widely defined as the unfairness of the ML model. It often stems from the underlying inherent human bias in the training samples, and a trained ML model amplifies this bias to the point of causing discriminatory decisions about certain groups and individuals.

Actually, the unfairness of ML model entangles in every corner of society, not only being spied on in financial risk control. A prime example comes from COMPAS [18], an automated software used in US courts to assess the

probability of criminals reoffending. A investigation of the software reveals a bias against African-Americans, i.e., COMPAS having a higher false positive rate for African-American offenders than white criminals, owing to incorrectly estimating their risk of reoffending. Similar model decision biases pervade other real-world applications including childcare systems [7], employment matching [33], AI chatbots, and ad serving algorithms [16]. As mentioned earlier, these resulting unfair decisions stem from neglected biases and discrimination hidden in data and algorithms.

To alleviate the above problems, a series of recent works [4], [24], [31], [32], [34] 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 [36] 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 verifi-

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

cation 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. 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 2PC tools [6], [20], [22] 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 [11], tedious computational authentication and interaction [15] (see Section 3 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× 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 [29], 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× computation speedup and gains at least 10.7× less communication overhead for linear layer computation. For non-linear layers, *VerifyML* is also 4×–42× faster and has > 48× lesser communication than existing 2PC protocol against malicious parties. Meanwhile, *VerifyML* demonstrates an encouraging online runtime boost by 32.6× and 32.2× over existing works on LeNet and ResNet-18, respectively, and at least an order of magnitude communication cost reduction.

2 PRELIMINARIES

2.1 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 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.

2.2 Notations

We use λ and σ to denote the computational security parameter and the statistical security parameter, respectively. $[k]$ represents the set $\{1, 2, \dots, k\}$ for $k > 0$. In our *VerifyML*, all the arithmetic operations are calculated in the field \mathbb{F}_p , where p is 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 $f : \mathbb{F}_p \rightarrow \mathbb{F}_p$, $f(\mathbf{a})$ 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 $\mathbf{L}_1, \dots, \mathbf{L}_m$ and the nonlinear layer is f_1, \dots, 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 $\mathbf{v}_m = \mathbf{L}_m \mathbf{x}_{m-1} = \mathbf{M}(\mathbf{x}_0)$.

2.3 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) [38], 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:

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

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 $\Psi = \{(\mathbf{x}^{(1)}, y^{(1)}, o^{(1)}), \dots, (\mathbf{x}^{(t)}, y^{(t)}, o^{(t)})\} \sim \mathcal{D}^t$, the *empirical conditional risk* is defined as follows:

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

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,

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

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

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

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:

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

In practice, we usually replace FG in Eqn.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 Eqn.5.

2.4 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:

- $\text{KeyGen}(1^\lambda) \rightarrow (pk, sk)$. 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.
- $\text{Enc}(pk, x) \rightarrow c$. Given pk and a plaintext $x \in \mathbb{F}_p$, the algorithm Enc outputs a ciphertext c encrypting x .
- $\text{Dec}(sk, c) \rightarrow x$. Taking sk and a ciphertext c as input, Dec decrypts c and outputs the corresponding plaintext x .
- $\text{Eval}(pk, c_1, c_2, F) \rightarrow c'$. 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 privacy¹. In *VerifyML*, we use the SEAL library [37] to implement the fully homomorphic encryption. In addition, we utilize ciphertext packing technology (CPT) [39] to encrypt multiple plaintexts into a single ciphertext, thus enabling homomorphic computation in a SIMD manner. Specifically, given two plaintext vectors $\mathbf{x} = (x_1, \dots, x_n)$ and $\mathbf{x}' = (x'_1, \dots, x'_n)$, 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 $\mathbf{x} = (x_0, \dots, x_n)$ is encrypted into a single ciphertext c , $\text{Rotation}(pk, c, j)$ transforms c into another ciphertext c' whose encrypted plaintext vector is $x' = (x_{j+1}, x_{j+2}, \dots, x_1, \dots, x_j)$. 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 computationally expensive compared to homomorphic addition and multiplication operations. Therefore, the optimization criterion for homomorphic SIMD operations is to minimize the number of *rotation* operations.

2.5 Parallel Matrix Homomorphic Multiplication

We review the parallel homomorphic multiplication method between arbitrary matrices proposed by *Jiang et al.* [17], 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 $\mathbf{X} = (x_{i,j})_{0 \leq i,j < d}$, 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:

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

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

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

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' .

matrices \mathbf{X} and \mathbf{Y} , the multiplication of the two under the ciphertext is calculated as follows:

$$\mathbf{c}_X \otimes \mathbf{c}_Y = \sum_{k=0}^{d-1} (\phi^k(\text{Enc}_{pk}(\sigma(\mathbf{X})))) \boxtimes (\varphi^k(\text{Enc}_{pk}(\tau(\mathbf{Y})))) \quad (7)$$

In the following sections, we will use $\mathbf{c}_X \otimes \mathbf{c}_Y$ to represent multiplication between any matrices \mathbf{X} and \mathbf{Y} in ciphertext. \boxtimes denotes the element-wise homomorphic multiplication between two ciphertexts. In Section 4.1.2, we describe how to utilize the parallel homomorphic multiplication described above to boost the generation of authenticated convolution triples.

2.6 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 $[\![x]\!]_b = \{\langle \alpha \rangle_b, \langle x \rangle_b, \langle \alpha x \rangle_b\}_{b \in \{0,1\}}$, 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 2PC 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 $2^{-\lfloor \log p \rfloor}$.

2.7 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:

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

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

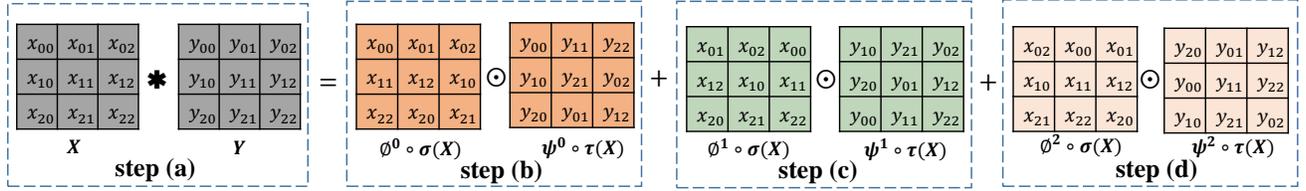


Fig. 1: Parallel matrix multiplication

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 4) to speed up the performance of online secure multiplication computations. Inspired by existing work to construct custom triples for specific mathematical operations [30] 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 4.

- **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 $\mathbf{X} * \mathbf{y} = \mathbf{z}$, where d_1 and d_2 are determined depending on the ML model architecture.
- **Authenticated Convolution triples** (aka matrix multiplication triples³): 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 $\text{Conv}(\mathbf{X}, \mathbf{Y}) = \mathbf{Z}$, where $u_w, u_h, u'_w, u'_h, l, c_i$ and c_o are determined depending on the model architecture.

2.8 Oblivious Transfer

We take OT_n to denote the 1-out-of-2 Oblivious Transfer (OT) [10], [13]. 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 [21] to implement OT_n^κ with the communication complexity of $\kappa\lambda + 2n$ bits.

2.9 Garbled Circuits

The garbling scheme [8], [35] for boolean circuits parsing arbitrary functions consists of a pair of algorithms

3. We can reduce the convolution operation to matrix multiplication by transforming the inputs of convolution appropriately. We provide a detailed description in Section 4.

(Garble, GCEval) defined as follows:

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

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

- **Correctness.** GCEval is faithfully performed on the GC and correctly outputs garbled results when given the garbled input of x . Formally, for any Boolean circuit C and input $x \in \{0, 1\}^n$, GCEval holds that

$$\text{GCEval}(\text{GC}, \{\text{lab}_{i,x[i]}^{\text{in}}\}_{i \in [n]}) \rightarrow \text{lab}_{C(x)}^{\text{out}}$$

- **Security.** Given C , the garbled circuit GC of C and garbled inputs of any $x \in \{0, 1\}^n$ can be simulated by a polynomial probability-time simulator Sim. Formally, for any circuit C and input $x \in \{0, 1\}^n$, we have $(\text{GC}, \{\text{lab}_{i,x[i]}^{\text{in}}\}_{i \in [n]}) \approx \text{Sim}(1^\lambda, C)$, where \approx indicates computational indistinguishability.
- **Authenticity.** This implies that given the garbled input of x and 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 $(\text{lab}_{1-C(x)}^{\text{out}} | \text{GC}, \{\text{lab}_{i,x[i]}^{\text{in}}\}_{i \in [n]}) \approx U_\lambda$.

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 [12], free-XOR [23] and half-gates [42] to construct the garbling scheme.

3 TECHNICAL INTUITION

VerifyML is essentially a 2PC protocol over the *model holder-malicious* threat model, where the client unbiasedly learns the inference results on a given test set, thereby faithfully evaluating the fairness of the target model

locally. For boosting the performance of the 2PC protocol execution, we customize a series of optimization methods by fully exploring the advantages of cryptographic primitives and their natural ties in inference process. Below we present a high-level technically intuitive overview of VerifyML’s design.

3.1 Offline-Online Paradigm

Consistent with state-of-the-art work on the setting of semi-honest models [29], *VerifyML* is deconstructed into an offline stage and an online stage, where the preprocessing process of the offline stage is independent of the input of model holders and clients. In this way, the majority ($> 95\%$) of the computation can be performed offline to minimize the overhead of the online process. Figure 2 provides an overview of *VerifyML*, where we describe the computational parts required for the offline and online phase, respectively.

3.2 Linear Layer Optimization

As described in Figure 2, we move almost all linear operations into the offline phase, where we construct customized triples for matrix-vector multiplication and convolution to accelerate linear execution. Specifically, 1) we design an efficient construction of matrix-multiplication triples instead of generating Beaver’s multiplication triples for individual multiplications (see Section 4.1.1). Our core insight is a new packed homomorphic multiplication method for matrices and vectors. We explore the inherent connection between secret sharing and homomorphic encryption to remove all the rotation operation in parallel homomorphic computation. 2) We extend the idea of generating matrix multiplicative triples over semi-honest models [30] into convolution domain over the *model holder-malicious* threat model (see Section 4). The core of our construction is derived from E2DM [17], which proposes a state-of-the-art method for parallel homomorphic multiplication between arbitrary matrices. We further optimize E2DM to achieve at least $2\times$ computational speedup compared to naive use.

Our optimization technique for linear layer computation exhibits superior advantages compared to state-of-the-art existing methods [20], [22]⁴. In more detail, we reduce the communication overhead from cubic to quadratic (both for offline and online phases) compared to Overdrive [22], which is the mainstream tool for generating authenticated multiplicative triples on malicious adversary models (see Section 4 for detailed analysis).

4. Note that several efficient parallel homomorphic computation methods [19], [43] with packed ciphertext have been proposed and run on semi-honest or client-malicious models [5], [26], [29] 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.

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 \mathbf{GC} for circuit \mathbf{C} parsing ReLU. The client sends \mathbf{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 known 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 \mathbf{GC} to the model holder. The model holder evaluates the \mathbf{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.

Figure 2: Overview of the *VerifyML*

3.3 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 authen-

ticated 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 (denoted 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.* Since the model holder is malicious, it must be ensured that the input from the model holder in the GC (i.e. $\langle \mathbf{v}_i \rangle_0$) is consistent with the share obtained by the previous linear layer. In the traditional malicious adversary model [6], [20], [22], a standard approach is to verify the correctness of the authenticated sharing of all inputs from malicious entities in the GC. However, this is very expensive and takes tens of seconds or even minutes to process a ReLU function. It obviously does not meet the practicality of ML model inference because a modern ML model usually contains 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) [26]. 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 4). (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 $1 \leq j \leq \kappa$) in the GC, rather than obtaining the exact arithmetic share of $f_i(\mathbf{v}_i)$ [5]. Moreover, we can parse ReLU function as $ReLU(\mathbf{v}_i) = \mathbf{v}_i \cdot \text{sign}(\mathbf{v}_i)$, where the sign function $\text{sign}(\mathbf{v}_i)$ equals 1 if $t \geq 0$ and 0 otherwise. Hence, we only encapsulate the non-linear part of $ReLU(\mathbf{v}_i)$ (i.e., $\text{sign}(\mathbf{v}_i)$) into the GC, thereby substantially minimizing the number of multiplication operations.

Compared with works [6], [20], [22] 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 $48 \times$ less communication overhead for nonlinear layer computation.

Remark 3.1. 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], [20], [22] 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.

4 THE VERIFYML FRAMEWORK

4.1 Offline Phase

In this section, we describe the technical details of *VerifyML*. As described above, *VerifyML* is divided into offline 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.

4.1.1 Generating matrix-vector multiplication triple

Figure 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 $\langle \mathbf{Y} \rangle_0$ before encryption (step 2 in Figure 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 Figure 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 Figure 3).

Figure 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 Figure 4), \mathbf{y} is first transformed into a matrix \mathbf{Y} by copying, where each row of \mathbf{Y} contains a copy of \mathbf{y} . P_1 then performs element-wise multiplications (step(b) in Figure 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 Figure 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 Figure 4), similarly, P_1 performs the same operation on matrix \mathbf{R} to obtain $\langle \mathbf{z} \rangle_1$.

Remark 4.1: Compared to generating multiplication triples for single multiplication [20], [22], 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

4. 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], [22] for more details.

5. 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.

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 $\{\llbracket \mathbf{X} \rrbracket_b, \llbracket \mathbf{y} \rrbracket_b, \llbracket \mathbf{z} \rrbracket_b\}_{b \in \{0,1\}}$ where $\mathbf{X} * \mathbf{y} = \mathbf{z}$.

Procedure:

- 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.
- 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 $c_1 \leftarrow \text{Enc}(pk, \langle \mathbf{X} \rangle_0)$ and $c_2 \leftarrow \text{Enc}(pk, \langle \mathbf{Y} \rangle_0)$ to P_1 along with zero-knowledge (ZK) proofs of plaintext knowledge of the two ciphertexts⁴.
- 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 $c_3 = \text{Enc}_{pk}(\alpha(\langle \mathbf{X} \rangle_1 + \langle \mathbf{X} \rangle_0) - \langle \alpha \mathbf{X} \rangle_1)$, $c_4 = \text{Enc}_{pk}(\alpha(\langle \mathbf{Y} \rangle_1 + \langle \mathbf{Y} \rangle_0) - \langle \alpha \mathbf{Y} \rangle_1)$, $c_5 = \text{Enc}_{pk}(\alpha(\langle \mathbf{X} \rangle_1 \odot \langle \mathbf{Y} \rangle_1) - \langle \alpha \mathbf{Z} \rangle_1)$, and $c_6 = \text{Enc}_{pk}(\langle \mathbf{X} \rangle_1 \odot \langle \mathbf{Y} \rangle_1 - \langle \mathbf{Z} \rangle_1)$ to P_0 .
- 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 $\langle \alpha \mathbf{Y} \rangle_0$, $\langle \alpha \mathbf{Z} \rangle_0$ 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 $\langle \mathbf{z} \rangle_1$.
- P_b outputs $\{\llbracket \mathbf{X} \rrbracket_b, \llbracket \mathbf{y} \rrbracket_b, \llbracket \mathbf{z} \rrbracket_b\}_{b \in \{0,1\}}$, where $\mathbf{X} * \mathbf{y} = \mathbf{z}$.

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

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 [14], [17], 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. 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 Figure 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.

Theorem 4.1. *Let the fully homomorphic encryption used in $\pi_{Mtriple}$ have the properties defined in Section 2.4. $\pi_{Mtriple}$*

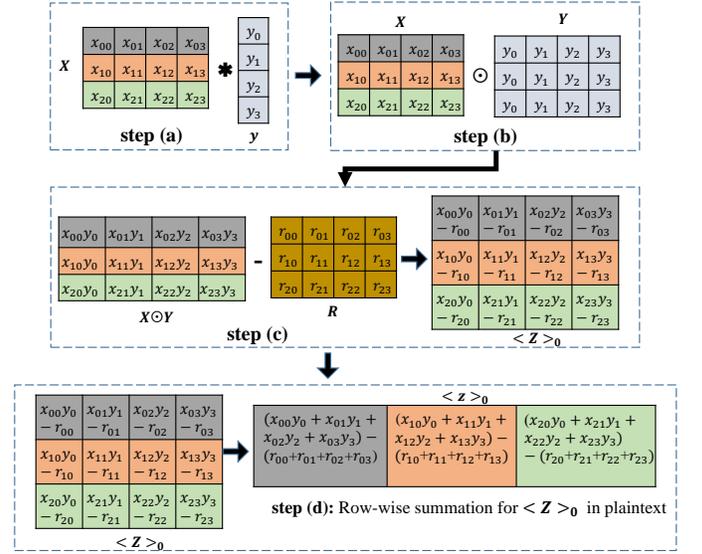


Fig. 4: Matrix-vector multiplication

is secure against the malicious model holder P_0 and the semi-honest client P_1 .

4.1.2 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 $\tilde{\mathbf{X}}_{ijk}$, where $1 \leq i \leq u_w$ and $1 \leq j \leq u_h$ are spatial coordinates, and $1 \leq k \leq c_i$ 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 $-l \leq \Delta_i, \Delta_j \leq l$ are shifts of the spatial coordinates, $1 \leq k \leq c_i$ and $1 \leq k' \leq c_o$ are the channels and kernels, respectively. The convolution between \mathbf{X} and \mathbf{Y} (i.e., $\mathbf{Z} = \text{Conv}(\mathbf{X}, \mathbf{Y})$) is defined as below:

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

The resulting tensor $\mathbf{Z}_{ijk'}$ has $u'_w \times u'_h$ spatial coordinates and c_o 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 [25]. 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 Eqn.(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 $\mathbf{Y}'_{(i,j)k'} = \mathbf{Y}_{\Delta_i, \Delta_j, k'}$. 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, we provide a detailed example to implement the above transformation.

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 $\mathbb{F}_p^{(2l+1)^2 \times (2l+1) \times c_i \times c_o}$. In addition, p_1 holds a MAC key α uniformly chosen from \mathbb{F}_p .

Output: P_b obtains $\{\llbracket \mathbf{X} \rrbracket_b, \llbracket \mathbf{Y} \rrbracket_b, \llbracket \mathbf{Z} \rrbracket_b\}_{b \in \{0,1\}}$, where $\mathbf{Z} = \text{Conv}(\mathbf{X}, \mathbf{Y})$.

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 $\langle \mathbf{Y}' \rangle_0$, 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 $c_1 \leftarrow \text{Enc}(pk, \langle \mathbf{X}' \rangle_0)$ and $c_2 \leftarrow \text{Enc}(pk, \langle \mathbf{Y}' \rangle_0)$ 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 $\langle \mathbf{Y}' \rangle_1$. 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 $c_3 = \text{Enc}_{pk}(\alpha(\langle \mathbf{X}' \rangle_1 + \langle \mathbf{X}' \rangle_0) - \langle \alpha \mathbf{X}' \rangle_1)$, $c_4 = \text{Enc}_{pk}(\alpha(\langle \mathbf{Y}' \rangle_1 + \langle \mathbf{Y}' \rangle_0) - \langle \alpha \mathbf{Y}' \rangle_1)$, $c_5 = \alpha \boxtimes (\mathbf{c}_{\mathbf{X}'} \circledast \mathbf{c}_{\mathbf{Y}'}) - \text{Enc}_{pk}(\langle \alpha \mathbf{Z}' \rangle_1)$, and $c_6 = (\mathbf{c}_{\mathbf{X}'} \circledast \mathbf{c}_{\mathbf{Y}'}) - \text{Enc}_{pk}(\langle \mathbf{Z}' \rangle_1)$. 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 $\{\llbracket \mathbf{X} \rrbracket_b, \llbracket \mathbf{Y} \rrbracket_b, \llbracket \mathbf{Z} \rrbracket_b\}_{b \in \{0,1\}}$, where $\mathbf{Z} = \text{Conv}(\mathbf{X}, \mathbf{Y})$.

Figure 4: Algorithm π_{Ctriplet} for generating authenticated convolution triple

③ **Generating convolution triple.** Figure 4 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 Figure 4). 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 Figure 4). 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 Figure 4). Finally, P_b obtains

$\{\llbracket \mathbf{X} \rrbracket_b, \llbracket \mathbf{Y} \rrbracket_b, \llbracket \mathbf{Z} \rrbracket_b\}_{b \in \{0,1\}}$, where $\mathbf{Z} = \text{Conv}(\mathbf{X}, \mathbf{Y})$.

Remark 4.2: We utilize the method in [17] 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 [14]. Besides, [17] requires only one ciphertext to represent a single matrix whereas existing work [14] 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 [20], [22], 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 4.3: 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 3.2 in [17] 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, π_{Ctriplet} , is secure against the malicious model holder P_0 and the semi-honest client P_1 . We provide the following theorem.

Theorem 4.2. *Let the fully homomorphic encryption used in π_{Ctriplet} have the properties defined in Section 2.4. π_{Ctriplet} 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 4.1**, we omit it for brevity. \square

4.1.3 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. $\text{sign}(\mathbf{v})$ given an arbitrary input \mathbf{v} . We first define a truncation function $\text{Trun}_h : \{0, 1\}^\lambda \rightarrow \{0, 1\}^h$, which outputs the last h bits of the input, where λ satisfies $\lambda \geq 2\kappa$. 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 $\text{Garble}(1^\lambda, \text{booln}^C) \rightarrow (\text{GC}, \{\{\text{lab}_{i,j}^{\text{in}}\}_{i \in [2\kappa]}, \{\text{lab}_{i,j}^{\text{out}}\}_{i \in [2\kappa]}\}_{j \in \{0,1\}})$, where GC is the garbled circuit of booln^C , $\{\{\text{lab}_{i,j}^{\text{in}}\}_{i \in [2\kappa]}, \{\text{lab}_{i,j}^{\text{out}}\}_{i \in [2\kappa]}\}_{j \in \{0,1\}}$ 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 $\{\text{lab}_{i,j}^{\text{out}}\}$ as $\varsigma_{i,j} \parallel \vartheta_{i,j}$ 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 $ct_{i,\varsigma_{i,j}} = \iota_{i,j} \oplus \text{Trun}_{\kappa}(\vartheta_{i,j})$ and $\hat{ct}_{i,\varsigma_{i+\kappa,j}} = (\eta_{i,j} \parallel \gamma_{i,j}) \oplus \text{Trun}_{2\kappa}(\vartheta_{i+\kappa,j})$.

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 2.9, P_0 cannot obtain the plaintext corresponding to the ciphertext sent by the client.

4.2 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.

4.2.1 Perform linear layers in the online phase

Figure 5 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 Figure 5). 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], [20], [22]. 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 Figure 5).

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 4.3. *Let triples used in π_{OLin} are generated from $\pi_{M\text{triple}}$ and $\pi_{C\text{triple}}$. π_{OLin} is secure against the malicious model holder P_0 and the semi-honest client P_1 .*

Preamble: Consider a neural network (NN) consists of m linear layers and $m - 1$ nonlinear layers. Let the specification of the linear layer is $\mathbf{L}_1, \mathbf{L}_1, \dots, \mathbf{L}_m$ and the non-linear layer is f_1, \dots, 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 $\llbracket \mathbf{v}_i = \mathbf{L}_i \mathbf{x}_{i-1} \rrbracket_b$ 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 $\llbracket \mathbf{R}_i \rrbracket$ of the same dimension as \mathbf{L}_i .
2. $\llbracket \mathbf{R}_i \rrbracket$ 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 $\llbracket \mathbf{L}_i \rrbracket_b = \llbracket \mathbf{R}_i \rrbracket_b + \varpi_i$ 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 $\llbracket \mathbf{v}_0 \rrbracket_0 = (\mathbf{v}_0 - \xi, \alpha \mathbf{v}_0 - \zeta)$ to P_0 . P_1 sets $\llbracket \mathbf{v}_0 \rrbracket_1 = (\xi, \zeta)$.
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 $\{\llbracket \mathbf{X} \rrbracket_b, \llbracket \mathbf{y} \rrbracket_b, \llbracket \mathbf{z} \rrbracket_b\}_{b \in \{0,1\}}$ 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 $\llbracket \mathbf{A} * \mathbf{b} \rrbracket_b$ 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 $\{\llbracket \mathbf{X} \rrbracket_b, \llbracket \mathbf{Y} \rrbracket_b, \llbracket \mathbf{Z} \rrbracket_b\}_{b \in \{0,1\}}$ 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 $\llbracket \text{Conv}(\mathbf{A}, \mathbf{B}) \rrbracket_b$ based on Eqn.(8).

6. P_b obtains $\llbracket \mathbf{v}_i = \mathbf{L}_i \mathbf{x}_{i-1} \rrbracket_b$ for $i \in [m]$ and $b = \{0, 1\}$.

Figure 5: Online linear layers protocol π_{OLin}

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

4.2.2 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 non-linear function in deep neural networks. As shown in Figure 5, 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 $f_i(\mathbf{v}_i)$

Input: P_0 holds $\llbracket \mathbf{v}_i \rrbracket_0$ and P_1 holds $\llbracket \mathbf{v}_i \rrbracket_1$ for $i \in [m]$ and $b = \{0, 1\}$. In addition, P_1 holds the MAC key α .

Output: P_b obtains $\llbracket \mathbf{x}_i = \text{ReLU}(\mathbf{v}_i) \rrbracket_b$ 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 $\text{OT}_{\lambda}^{\kappa}$ (see Section 2.8), where P_1 's inputs are $\{\text{lab}_{j,0}^{in}, \text{lab}_{j,1}^{in}\}_{j \in \{\kappa+1, \dots, 2\kappa\}}$ while P_0 's input is $\langle \mathbf{v}_i \rangle_0$. Hence, P_0 learns $\{\tilde{\text{lab}}_j^{in}\}_{j \in \{\kappa+1, \dots, 2\kappa\}}$. Also, P_1 sends its garbled inputs $\{\{\tilde{\text{lab}}_j^{in} = \text{lab}_{j, \langle \mathbf{v}_i \rangle_1[j]}\}_{j \in [\kappa]}$ to P_0 .
 - With GC and $\{\tilde{\text{lab}}_j^{in}\}_{j \in [2\kappa]}$, P_0 evaluates $\text{GCEval}(\text{GC}, \{\tilde{\text{lab}}_j^{in}\}_{j \in [2\kappa]}) \rightarrow \{\tilde{\text{lab}}_j^{out}\}_{j \in [2\kappa]}$.
2. Authentication Phase 1:
 - P_0 parses $\tilde{\text{lab}}_j^{out}$ as $\tilde{c}_j || \tilde{\vartheta}_j$ where $\tilde{c}_j \in \{0, 1\}$ and $\tilde{\vartheta}_j \in \{0, 1\}^{\lambda-1}$ for every $j \in [2\kappa]$.
 - P_0 computes $c_j = \text{ct}_{j, \tilde{c}_j} \oplus \text{Trun}_{\kappa}(\tilde{\vartheta}_j)$ and $(d_j || e_j) = \hat{\text{ct}}_{j, \tilde{c}_j + \kappa} \oplus \text{Trun}_{2\kappa}(\tilde{\vartheta}_{j+\kappa})$ 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 $\{\llbracket x \rrbracket_b, \llbracket y \rrbracket_b, \llbracket z \rrbracket_b\}_{b \in \{0,1\}}$.
 - All parties reveal $\mathbf{v}_i - x$ and $g_2 - y$ to each other, and then locally compute $\langle z_2 \rangle_b = \langle \mathbf{v}_i \cdot \text{sign}(\mathbf{v}_i) \rangle_b$ and $\langle z_3 \rangle_b = \langle \alpha \mathbf{v}_i \cdot \text{sign}(\mathbf{v}_i) \rangle_b$ based on Eqn.(8).
 - P_b obtains $\llbracket \mathbf{x}_i = \text{ReLU}(\mathbf{v}_i) \rrbracket_b = (\langle z_2 \rangle_b, \langle z_3 \rangle_b)$ and $\langle \alpha \mathbf{v}_i \rangle_b = \langle g_1 \rangle_b$.

Figure 6: Online non-linear layers protocol π_{ONLin}

and share it to the model holder and client in the authenticated sharing manner. We describe details in Figure 6.

Garbled Circuit Phase. As described in Section 4.1.3, in the offline phase, P_1 constructs a GC for the nonlinear part of ReLU (i.e., $\text{sign}(\mathbf{v}_i)$ 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 $\text{OT}_{\lambda}^{\kappa}$, where P_1 as the sender whose inputs are $\{\text{lab}_{j,0}^{in}, \text{lab}_{j,1}^{in}\}_{j \in \{\kappa+1, \dots, 2\kappa\}}$ 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 GC. Then, P_0 evaluates GC with garbled inputs of \mathbf{v}_i and learns the set of output labels for the bits of \mathbf{v}_i and $\text{sign}(\mathbf{v}_i)$.

Authentication Phase 1. This phase aims to calculate the share of the authentication of each bit of \mathbf{v}_i , i.e., $\text{sign}(\mathbf{v}_i)[j]$, $\alpha \text{sign}(\mathbf{v}_i)[j]$, 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 $\text{lab}_{j,0}^{out}$ and $\text{lab}_{j,1}^{out}$ 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 $\text{lab}_{j,1}^{out}$ and encrypts $\iota_j + \alpha$ as $\text{lab}_{j,0}^{out}$. 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 $\text{lab}_{j,0}^{out}$ in the previous phase, it can definitely decrypt it and obtain its own share of $\alpha \mathbf{v}_i[j]$. Computation of $\text{sign}(\mathbf{v}_i)[j]$ and $\alpha \text{sign}(\mathbf{v}_i)[j]$ 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 $\text{sign}(\mathbf{v}_i)$, $\alpha \text{sign}(\mathbf{v}_i)$, 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 $\text{sign}(\mathbf{v}_i)$ and $\alpha \text{sign}(\mathbf{v}_i)$ in a similar manner.

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

Remark 4.4. We adopt two methods to minimize the number of multiplication operations involved in the GC. One is to compute the garbled output of per-bit of $\text{sign}(\mathbf{v}_i)$ in GC. Another is to encapsulate only the nonlinear part of ReLU into GC. In this way, we avoid computing $\alpha \text{ReLU}(\mathbf{v}_i)$ and $\text{ReLU}(\mathbf{v}_i)$ in GC, which is multiply operation intensive. Compared with works [6], [20], [22] 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 4.5. 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 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 4.3 for more details).

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

$(d_j || e_j) = \hat{c}t_{j, c_j + \kappa, \text{sign}(\mathbf{v}_i)[j]} \oplus \mathbf{Trun}_{2\kappa}(\vartheta_{j+\kappa, \text{sign}(\mathbf{v}_i)[j]}) = \eta_{j, \text{sign}(\mathbf{v}_i)[j]} || \gamma_{j, \text{sign}(\mathbf{v}_i)[j]}$. Based on these, we have

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

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

Theorem 4.4. *Let (Garble, GCEval) be a garbling scheme with the properties defined in Section 2.9. Authenticated shares have the properties defined in Section 2.6. Then our protocol π_{ONlin} is secure against the malicious model holder P_0 and the semi-honest client P_1 .*

4.3 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 Eqn.(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 Figure 5) and open them for computation. Assume that the set of all opened elements is $(a_1, a_2 \dots, a_t)$, and P_b holds $\langle \rho_i \rangle_b = \langle \alpha a_i \rangle_b$ as well as $\langle \tau_i \rangle_b = \langle a_i \rangle_b$, we need to perform a consistency check to verify $\rho_i - \alpha \tau_i = 0$. Beside, For executing each nonlinear layer $\{f_i\}_{i \in [m-1]}$, the inputs of π_{ONlin} are shares of \mathbf{v}_i and $\tau_i = \alpha \mathbf{v}_i$. 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 $\sum_{i=1}^m \tau_i - \xi_i = \mathbf{0}$.

Figure 7 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 $\rho_j - \alpha \tau_j = (\langle \rho_j \rangle_0 - \alpha_0 a_j + \langle \rho_j \rangle_1 - \alpha_1 a_j) = 0$ for every linear layer $\{\mathbf{L}_j\}_{j \in [m]}$. By correctness of π_{ONlin} , we have $\tau_i - \xi_i = (\langle \tau_i \rangle_0 - \langle \xi_i \rangle_0) + (\langle \tau_i \rangle_1 - \langle \xi_i \rangle_1) = 0$ for all nonlinear layers. Hence, we have $\langle q \rangle_0 + \langle q \rangle_1 = \sum_{j \in [t]} \mathbf{r}_j (\rho_j - \alpha \tau_j) + \sum_{i \in [m-1]} \mathbf{r}_i (\tau_i - \xi_i) = \mathbf{0}$.

Input: P_b $b \in \{0, 1\}$ holds $\langle \tau_i \rangle_b$, $\langle \xi_i \rangle_b$ and $\llbracket a_j \rrbracket_b$ 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 $\langle q \rangle_0 + \langle q \rangle_1 \neq 0 \pmod p$. Else, P_1 locally evaluates the fairness of the ML model based on Eqn.(2) by reconstructing $\mathbf{M}(\mathbf{x}_0)$.

Figure 7: 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.

Theorem 4.5. *In real execution, if P_0 tampers with its input, then P_1 aborts with probability at least $1 - 1/p$.*

5 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* [22]⁵ 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.

5. 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.

5.1 Implementation details

VerifyML is implemented through the C++ language and provides 128 bits of computational security and 40 bits of statistical security. The entire system operates on the 44-bit prime field. We utilize the SEAL homomorphic encryption library [37] 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 the EMP toolkit [40] (with the OT protocol that resists active adversaries). Zero-knowledge proofs of plaintext knowledge are implemented based on MUSE [26]. 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 1.4GHz CPUs with 32 threads on 16 cores and 32GB RAM. The server workstation has Intel(R) Xeon(R) E5-2697 v3 2.6GHz CPUs with 28 threads on 14 cores and 64GB RAM. The WAN setting is based on a connection between a local PC and an Amazon AWS server with an average bandwidth of 963Mbps and running time of around 14ms.

5.2 Performance of offline phase

5.2.1 Cost of generating matrix-vector multiplication triple

TABLE I: Cost of generating the matrix-vector multiplication triple

Dimension	Comm.cost (MB)		Running time (s)			
	Overdrive	<i>VerifyML</i> (Reduction)	Overdrive		<i>VerifyML</i> (speedup)	
			LAN	WAN	LAN	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 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× reduction in communication overhead and at least 1.5× 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.

5.2.2 Cost of generating convolution triple

TABLE II: Cost of generating the convolution triple

Input	Kernel	Comm.cost (GB)		Running time (s)			
		Overdrive	<i>VerifyML</i>	Overdrive		<i>VerifyML</i> (Speedup)	
				LAN	WAN	LAN	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×)

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× in communication cost and a speedup of at least 1.6× 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 [17] 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.

5.3 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.

5.3.1 Performance of executing linear layers

TABLE III: Comparison of the communication overhead for executing convolution in the online phase

Input	Kernel	Comm.cost (MB)	
		Overdrive	<i>VerifyML</i> (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×)

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 computational overhead. Therefore, we focus on analyzing the difference in communication overhead between the two of executing convolution. TABLE III depicts the communication overhead of *VerifyML* and *Overdrive* for 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.

5.3.2 Performance of executing nonlinear layers

Figure 6 provides the comparison of the cost between *Overdrive* and *VerifyML*. We observe that *VerifyML* outperforms *Overdrive* by 4 – 42× in runtime on LAN Setting and 3 – 16× in WAN Setting. For example, *Overdrive* takes 165.4s and 1283.5s to compute 2^{15} ReLUs on LAN and WAN setting, respectively. Whereas, *VerifyML* took just 5.1s and 110.2s in the respective network settings. For communication overhead, we observed that *Overdrive* required 401KB of traffic to perform a single ReLU while we only need 8.33KB, which is at least a 48× improvement. This is mainly due to the fact that our optimized GC substantially reduces the multiplication

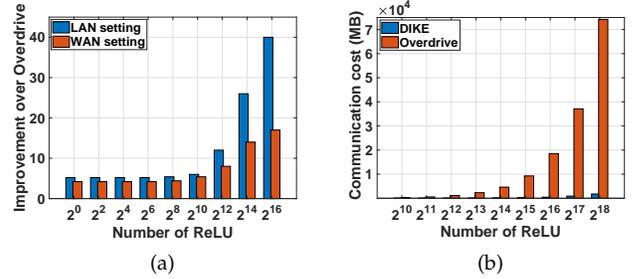


Fig. 6: Comparison of the overhead for executing nonlinear layers. ((a) Running time improvement of *VerifyML* over *Overdrive*. The y-axis shows $\frac{\text{Overdrive time}}{\text{VerifyML time}}$ (b) Comparison of the communication overhead.

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.

5.4 Performance of end-to-end secure inference

TABLE IV: Cost of end-to-end secure inference

	Phases	Comm.cost (MB)		Running time (s)			
		Overdrive	<i>VerifyML</i>	Overdrive		<i>VerifyML</i> (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×)
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	25717.1	238951.2	240331.1	114008.6 (2.1×)	115096.7 (2.1×)

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 [28] 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× and 32.2× over existing works on LeNet and ResNet-18, respectively, and at least an order of magnitude communication cost reduction. In online phase, *Overdrive* takes 32.8s and 177s to compute single query on LeNet and ResNet-18, respectively. Whereas, *VerifyML* took just 1s and 5.5s in the respective network settings. Consistent with the previous analysis,

this stems from the customized optimization mechanism we designed for *VerifyML*.

5.5 Comparison with other works

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 [29] under the 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 32KB of traffic to perform a single ReLU while we only need 8.33KB.

Compared with MUSE and SIMC. We note that several works such as MUSE [26] 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 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 [43] and GAZELLE [19]. 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 [26] 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 [26] 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.

6 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] Finastra Adam Lieberman. How data scientists can create a more inclusive financial services landscape, 2022.
- [2] Martin Arjovsky, Léon Bottou, Ishaan Gulrajani, and David Lopez-Paz. Invariant risk minimization. *arXiv preprint arXiv:1907.02893*, 2019.
- [3] Rachel KE Bellamy, Kuntal Dey, Michael Hind, Samuel C Hoffman, Stephanie Houde, Kalapriya Kannan, Pranay Lohia, Jacquelyn Martino, Sameep Mehta, Aleksandra Mojsilovic, et al. Ai fairness 360: An extensible toolkit for detecting, understanding, and mitigating unwanted algorithmic bias. *arXiv preprint arXiv:1810.01943*, 2018.
- [4] Sumon Biswas and Hriday Rajan. Do the machine learning models on a crowd sourced platform exhibit bias? an empirical study on model fairness. In *Proceedings of ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering (ESEC/FSE)*, pages 642–653, 2020.
- [5] Nishanth Chandran, Divya Gupta, Sai Lakshmi Bhavana Obbattu, and Akash Shah. Simc: Ml inference secure against malicious clients at semi-honest cost. *Cryptology ePrint Archive*, 2021.
- [6] Hao Chen, Miran Kim, Ilya Razenshteyn, Dragos Rotaru, Yongsoo Song, and Sameer Wagh. Maliciously secure matrix multiplication with applications to private deep learning. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, pages 31–59. Springer, 2020.
- [7] Alexandra Chouldechova, Diana Benavides-Prado, Oleksandr Fialko, and Rhema Vaithianathan. A case study of algorithm-assisted decision making in child maltreatment hotline screening decisions. In *Conference on Fairness, Accountability and Transparency*, pages 134–148. PMLR, 2018.
- [8] Michele Ciampi, Vipul Goyal, and Rafail Ostrovsky. Threshold garbled circuits and ad hoc secure computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 64–93. Springer, 2021.
- [9] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference (CRYPTO)*, pages 643–662. Springer, 2012.
- [10] Nico Döttling, Sanjam Garg, Mohammad Hajiabadi, Daniel Masny, and Daniel Wichs. Two-round oblivious transfer from cdh or lpn. *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 12106:768, 2020.
- [11] Uriel Feige, Amos Fiat, and Adi Shamir. Zero-knowledge proofs of identity. *Journal of cryptology*, 1(2):77–94, 1988.

- [12] Tore Kasper Frederiksen, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. Minilego: Efficient secure two-party computation from general assumptions. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 537–556. Springer, 2013.
- [13] Alex B Grilo, Huijia Lin, Fang Song, and Vinod Vaikuntanathan. Oblivious transfer is in minicrypt. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 531–561. Springer, 2021.
- [14] Shai Halevi and Victor Shoup. Algorithms in helib. In *Annual Cryptology Conference*, pages 554–571. Springer, 2014.
- [15] Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Concretely efficient large-scale mpc with active security (or, tinykeys for tinyot). In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, pages 86–117. Springer, 2018.
- [16] Ayanna Howard and Jason Borenstein. The ugly truth about ourselves and our robot creations: the problem of bias and social inequity. *Science and engineering ethics*, 24(5):1521–1536, 2018.
- [17] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In *Proceedings of the ACM SIGSAC conference on computer and communications security (CCS)*, pages 1209–1222, 2018.
- [18] Surya Mattu Julia Angwin, Jeff Larson and ProPublica Lauren Kirchner. Machine bias, 2016.
- [19] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakan. {GAZELLE}: A low latency framework for secure neural network inference. In *USENIX Security Symposium (USENIX Security 18)*, pages 1651–1669, 2018.
- [20] Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. In *Proceedings of ACM SIGSAC conference on computer and communications security (CCS)*, pages 1575–1590, 2020.
- [21] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure ot extension with optimal overhead. In *Annual Cryptology Conference (CRYPTO)*, pages 724–741. Springer, 2015.
- [22] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: making spdz great again. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 158–189. Springer, 2018.
- [23] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. Flexor: Flexible garbling for xor gates that beats free-xor. In *Annual Cryptology Conference (CRYPTO)*, pages 440–457. Springer, 2014.
- [24] Preethi Lahoti, Alex Beutel, Jilin Chen, Kang Lee, Flavien Prost, Nithum Thain, Xuezhong Wang, and Ed Chi. Fairness without demographics through adversarially reweighted learning. *Advances in neural information processing systems (NeurIPS)*, 33:728–740, 2020.
- [25] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.
- [26] Ryan Lehmkuhl, Pratyush Mishra, Akshayaram Srinivasan, and Raluca Ada Popa. Muse: Secure inference resilient to malicious clients. In *USENIX Security Symposium (USENIX Security 21)*, pages 2201–2218, 2021.
- [27] Yehuda Lindell. How to simulate it—a tutorial on the simulation proof technique. *Tutorials on the Foundations of Cryptography*, pages 277–346, 2017.
- [28] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of the IEEE international conference on computer vision*, pages 3730–3738, 2015.
- [29] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *USENIX Security Symposium*, pages 2505–2522, 2020.
- [30] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *IEEE symposium on security and privacy (S&P)*, pages 19–38. IEEE, 2017.
- [31] Debarghya Mukherjee, Mikhail Yurochkin, Moulinath Banerjee, and Yuekai Sun. Two simple ways to learn individual fairness metrics from data. In *International Conference on Machine Learning (ICML)*, pages 7097–7107. PMLR, 2020.
- [32] Luca Oneto and Silvia Chiappa. Fairness in machine learning. In *Recent Trends in Learning From Data*, pages 155–196. Springer, 2020.
- [33] Osonde A Osoba and William Welser IV. *An intelligence in our image: The risks of bias and errors in artificial intelligence*. Rand Corporation, 2017.
- [34] Flavien Prost, Pranjal Awasthi, Nick Blumm, Aditee Kumthekar, Trevor Potter, Li Wei, Xuezhong Wang, Ed H Chi, Jilin Chen, and Alex Beutel. Measuring model fairness under noisy covariates: A theoretical perspective. In *Proceedings of AAAI/ACM Conference on AI, Ethics, and Society (AIES)*, pages 873–883, 2021.
- [35] Mike Rosulek and Lawrence Roy. Three halves make a whole? beating the half-gates lower bound for garbled circuits. In *Annual International Cryptology Conference (CRYPTO)*, pages 94–124. Springer, 2021.
- [36] Pedro Saleiro, Benedict Kuester, Loren Hinkson, Jesse London, Abby Stevens, Ari Anisfeld, Kit T Rodolfa, and Rayid Ghani. Aequitas: A bias and fairness audit toolkit. *arXiv preprint arXiv:1811.05577*, 2018.
- [37] Microsoft SEAL (release 4.0). <https://github.com/Microsoft/SEAL>, March 2022. Microsoft Research, Redmond, WA.
- [38] Shahar Segal, Yossi Adi, Benny Pinkas, Carsten Baum, Chaya Ganesh, and Joseph Keshet. Fairness in the eyes of the data: Certifying machine-learning models. In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society (AIES)*, pages 926–935, 2021.
- [39] Nigel P Smart and Frederik Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71(1):57–81, 2014.
- [40] Xiao Wang, Alex J Malozemoff, and Jonathan Katz. Emp-toolkit: Efficient multiparty computation toolkit. <https://github.com/emp-toolkit>, 2016.
- [41] Paul Weiss, Rifkind, Wharton, and Garrison LLP. Breaking new ground, cfpb will pursue discrimination as an “unfair” practice across the range of consumer financial services, 2022.
- [42] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 220–250. Springer, 2015.
- [43] Qiao Zhang, Chunsheng Xin, and Hongyi Wu. Gala: Greedy computation for linear algebra in privacy-preserved neural networks. In *Proceedings of the Network and Distributed System Security (NDSS)*, 2021.

Input: P_0 holds $\langle \mathbf{X} \rangle_0$ uniformly chosen from $\mathbb{F}_p^{d_1 \times d_2}$ and $\langle \mathbf{y} \rangle_0$ uniformly chosen from $\mathbb{F}_p^{d_2}$. P_1 holds $\langle \mathbf{X} \rangle_1$ uniformly chosen from $\mathbb{F}_p^{d_1 \times d_2}$, and $\langle \mathbf{y} \rangle_1$ uniformly chosen from $\mathbb{F}_p^{d_2}$ and 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 $\mathbf{X} * \mathbf{y} = \mathbf{z}$.

Figure 8: Functionality of $\mathcal{F}_{Mtriple}$

APPENDIX

APPENDIX A THREAT MODEL

We formalize the threat model involved in *VerifyML* with the simulated paradigm [27]. We define two interactions to capture security: a real interaction by P_0 and P_1 in the presence of adversary \mathbf{A} and an environment Z , and an ideal interaction where parties send their respective inputs to a trusted entity that computes functionally faithfully. Security requires that for any adversary \mathbf{A} in real interaction, there exists a simulator \mathbf{S} in ideal interaction, such that no environment Z can distinguish real interaction from ideal interaction. Specifically, let $f = (f_0, f_1)$ be the two-party functionality such that P_0 and P_1 invoke f on inputs a and b to obtain $f_0(a, b)$ and $f_1(a, b)$, respectively. We say a protocol π securely implements f if it holds the following properties.

- **Correctness:** If P_0 and P_1 are both honest, then P_0 gets $f_0(a, b)$ and P_1 gets $f_1(a, b)$ from the execution of π on the inputs a and b , respectively.
- **Semi-honest Client Security:** For a semi-honest adversary \mathbf{A} that compromises P_1 , there exists a simulator \mathbf{S} such that for any input (a, b) , we have

$$\text{View}_{\mathbf{A}}^{\pi}(a, b) \approx \mathbf{S}(b, f_1(a, b))$$

where $\text{View}_{\mathbf{A}}^{\pi}(a, b)$ represents the view of \mathbf{A} during the execution of π , and a and b are the inputs of P_0 and P_1 , respectively. $\mathbf{S}(b, f_1(a, b))$ represents the view simulated by \mathbf{S} when it is given access to b and $f_1(a, b)$. \approx indicates computational indistinguishability of two distributions $\text{View}_{\mathbf{A}}^{\pi}(a, b)$ and $\mathbf{S}(b, f_1(a, b))$.

- **Malicious Model Holder Security:** For the malicious adversary \mathbf{A} that compromises P_0 , there exists a simulator \mathbf{S} , such that for any input b from P_1 , we have

$$\text{Out}_{P_1}, \text{View}_{\mathbf{A}}^{\pi}(b, \cdot) \approx \hat{\text{Out}}, \mathbf{S}^{f(b, \cdot)}$$

where $\text{View}_{\mathbf{A}}^{\pi}(b, \cdot)$ denotes \mathbf{A} 's view during the execution of π with S_1 's input b . Out_{P_1} indicates the output of P_1 in the real protocol execution. Similarly, $\hat{\text{Out}}$ and $\mathbf{S}^{f(b, \cdot)}$ represents the output of P_1 and the simulated view in the ideal interaction.

APPENDIX B PROOF OF THEOREM 1

Proof: Let $\mathcal{F}_{Mtriple}$ shown in Figure 8 be the function-

ality of generating matrix-vector multiplication triple. We first prove security for semi-honest clients and then demonstrate security against malicious model holders.

Semi-honest client security. The simulator Sim_c samples $(pk, sk) \leftarrow \text{KeyGen}(1^\lambda)$. The simulator and the semi-honest client run a secure two-party protocol to generate the public and secret keys for homomorphic encryption. When the simulator accesses the ideal functionality, it provides pk as output. In addition, the Sim_c sends $\text{Enc}_{pk}(\mathbf{0})$ to the client along with the simulated zero-knowledge proof of well-formedness of ciphertexts. We now show the indistinguishability between real and simulated views by the following hybrid arguments.

- **Hyb₁:** This corresponds to the real execution of the protocol.
- **Hyb₂:** The simulator Sim_c runs the two-party computation protocol with the semi-honest client to generate the public and secret keys for homomorphic encryption. When the simulator accesses the ideal functionality, we sample $(pk, sk) \leftarrow \text{KeyGen}(1^\lambda)$ and send pk to the semi-honest client. This hybrid is computationally indistinguishable to **hyb₁**.
- **Hyb₃:** In this hybrid, instead of sending the encryptions $c_1 \leftarrow \text{Enc}(pk, \langle \mathbf{X} \rangle_0)$ and $c_2 \leftarrow \text{Enc}(pk, \langle \mathbf{Y} \rangle_0)$ to P_1 , Sim_c sends ciphertexts with all 0s (i.e., $\text{Enc}_{pk}(\mathbf{0})$) to the client. Sim_c also provides a zero-knowledge (ZK) proof of plaintext knowledge of the ciphertexts. For any two plaintexts, FHE ensures that an adversary cannot distinguish them from their ciphertexts. In addition, zero-knowledge proofs also guarantee the indistinguishability of two ciphertexts. Therefore, this hybrid is indistinguishable from the previous one.

Malicious model holder security. The simulator Sim_m samples $(pk, sk) \leftarrow \text{KeyGen}(1^\lambda)$. The simulator and the semi-honest client run a secure two-party protocol to generate the public and secret keys for homomorphic encryption. When the simulator accesses the ideal functionality, it provides (pk, sk) as outputs. Once P_0 sends $c_1 \leftarrow \text{Enc}(pk, \langle \mathbf{X} \rangle_0)$ and $c_2 \leftarrow \text{Enc}(pk, \langle \mathbf{Y} \rangle_0)$, Sim_m verifies the validity of the ciphertext from the client. If the verification is passed, Sim_m extracts $\langle \mathbf{X} \rangle_0$ and $\langle \mathbf{Y} \rangle_0$ and the randomness used for generating these ciphertexts, since it has access to the client's input. Then, Sim_m samples $\langle \mathbf{X} \rangle_1$ and $\langle \mathbf{Y} \rangle_1$, and queries the ideal functionalities on the input $\langle \mathbf{X} \rangle_0, \langle \mathbf{X} \rangle_1, \langle \mathbf{Y} \rangle_0$ and $\langle \mathbf{Y} \rangle_1$ to obtain $(\langle \alpha \mathbf{X} \rangle_0, \langle \alpha \mathbf{y} \rangle_0, \langle \alpha \mathbf{z} \rangle_0, \langle \mathbf{z} \rangle_0)$. Then, Sim_m uses these outputs and the randomness used to generate the initial ciphertexts to construct the four simulated ciphertexts. It sends the simulated ciphertexts to the client.

- **Hyb₁:** This corresponds to the real execution of the protocol.
- **Hyb₂:** The simulator Sim_m runs the two-party computation protocol with the malicious model holder to generate the public and secret keys for homomorphic encryption. When the simulator accesses the ideal functionality, we sample $(pk, sk) \leftarrow$

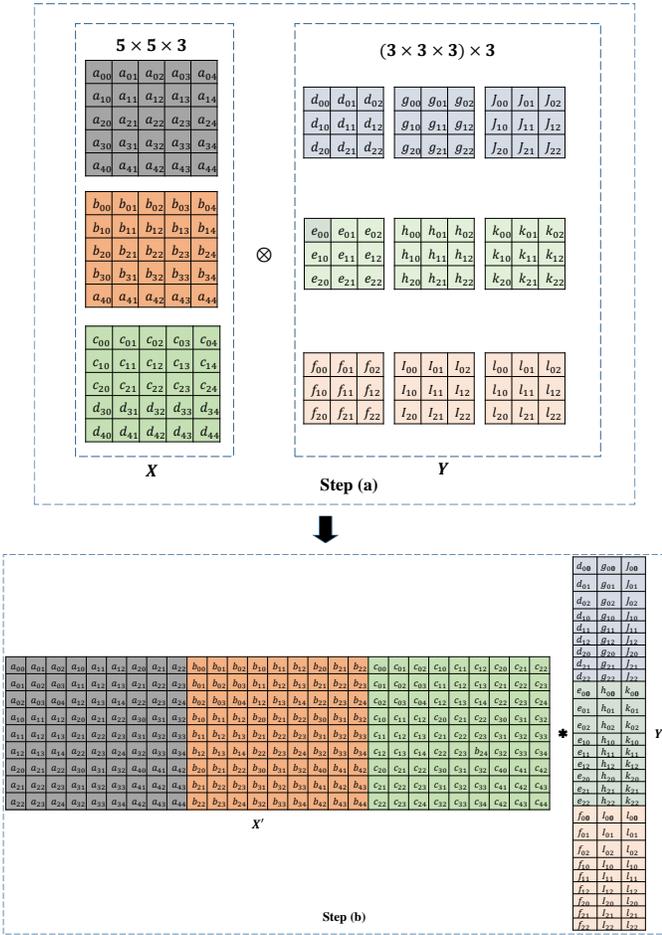


Fig. 9: Conversion between convolution and matrix multiplication

KeyGen(1^λ) and send them to the malicious model holder. This hybrid is computationally indistinguishable to hyb_1 .

- **Hyb₃:** In this hybrid, Sim_m checks the validity of the ciphertext from the client. If the zero-knowledge proofs are valid, Sim_m extracts $\langle \mathbf{X} \rangle_0$ and $\langle \mathbf{Y} \rangle_0$ and the randomness used for generating these ciphertexts, since it has access to the client's input. The properties of zero-knowledge proofs ensure that this hybrid is indistinguishable from the previous one.
- **Hyb₄:** Sim_m exploits the functional privacy of FHE to generate $c_3 = \text{Enc}_{pk}(\alpha(\langle \mathbf{X} \rangle_1 + \langle \mathbf{X} \rangle_0) - \langle \alpha \mathbf{X} \rangle_1)$, $c_4 = \text{Enc}_{pk}(\alpha(\langle \mathbf{Y} \rangle_1 + \langle \mathbf{Y} \rangle_0) - \langle \alpha \mathbf{Y} \rangle_1)$, $c_5 = \text{Enc}_{pk}(\alpha(\mathbf{X} \odot \mathbf{Y}) - \langle \alpha \mathbf{Z} \rangle_1)$, and $c_6 = \text{Enc}_{pk}(\langle \mathbf{X} \odot \mathbf{Y} \rangle_1 - \langle \mathbf{Z} \rangle_1)$. This hybrid is computationally indistinguishable to the previous hybrid from the function privacy of the FHE scheme. Note that view of the model holder in Hyb_4 is identical to the view generated by Sim_m . \square

APPENDIX C

CONVERSION BETWEEN CONVOLUTION AND MATRIX MULTIPLICATION

Figure 9 provides an example to convert a given convolution into the corresponding matrix multiplication. As shown in Figure 9, given input tensor of size 5×5 with 3 channels, denoted as \mathbf{X} , 3 kernels with a size of $(2+1) \times (2+1) \times 3$ denote as tensor \mathbf{Y} , the convolution between \mathbf{X} and \mathbf{Y} are converted an equivalent matrix multiplication \mathbf{X}' and \mathbf{Y}' , where the number of turns to zero-pad is 0, and stride $s = 1$. Specifically, we construct a matrix \mathbf{X}' with dimension (9×27) , 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 (27×3) 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 $\mathbf{Z}'_{(ij)k'} = \mathbf{Z}_{ijk'}$.

APPENDIX D

PROOF OF THEOREM 4

Proof: Semi-honest client security. The security of the protocol π_{ONlin} against the semi-honest client P_1 is evident by observing the execution of the protocol. This stems from the fact that P_1 does obtain output in OT_λ^κ and does not receive any information from P_0 in subsequent executions. Here we focus on the security analysis of π_{ONlin} against malicious model holder P_0 .

Malicious model holder security. We first define the functionality of the protocol π_{ONlin} , denoted as \mathcal{F}_{ONlin} , as shown in Figure 10. We use Real to refer to the view of the real interaction between P_1 and the adversary \mathcal{A} controlling P_0 , and then demonstrate Real indistinguishability from the simulated view interacted by the simulator Sim_m and \mathcal{A} through standard hybrid arguments. In the following we will define three hybrid executions Hyb_1 , Hyb_2 and Hyb_3 . We prove that π_{ONlin} is secure from the malicious model holder P_0 by proving indistinguishability among these hybrid executions.

Function $f: \mathbb{F}_p \rightarrow \mathbb{F}_p$.

Input: P_1 holds $\langle \mathbf{v}_i \rangle_1 \in \mathbb{F}_p$ and a MAC key α uniformly chosen from \mathbb{F}_p . P_0 holds $\langle \mathbf{v}_i \rangle_0 \in \mathbb{F}_p$.

Output: P_b obtains $\{(\langle \alpha \mathbf{v}_i \rangle_b, \langle f(\mathbf{v}_i) \rangle_b, \langle \alpha f(\mathbf{v}_i) \rangle_b)\}$ for $b \in \{0, 1\}$.

Figure 10: Functionality of the nonlinear layer \mathcal{F}_{ONlin}

Hyb₁: This hybrid execution is identical to Real except in the authentication phase. To be precise, in the authentication phase, the simulator Sim_m use labels $\text{lab}_{i,j}^{\text{out}}$ (described below) to replace the labels $\text{lab}_{i,j}^{\text{out}}$ used in Real . Please note that in this hybrid the simulator Sim_m can access P_1 ' input $\langle \mathbf{v}_i \rangle_1$ and α , where $\langle \mathbf{v}_i \rangle_0 + \langle \mathbf{v}_i \rangle_1 = \mathbf{v}_i$. Let $\delta = (\mathbf{v}_i || \text{sign}(\mathbf{v}_i))$. Therefore, for $i \in [2\kappa]$, we set $\text{lab}_{i,j}^{\text{out}} = \text{lab}_{i,1-\delta[i]}^{\text{out}}$ if $j = \delta[i]$, otherwise, $\text{lab}_{i,1-\delta[i]}^{\text{out}}$ (i.e.,

the “other” label) is set to a random value chosen from $\{0,1\}^\lambda$ uniformly, where the first bit of $\text{lab}_{i,1-\delta[i]}^{\text{out}}$ is $1 - \varsigma_{i,\delta[i]}$. We provide the formal description of Hyb_1 as follows, where the indistinguishability between the view of \mathcal{A} in Real and Hyb_1 is directly derived from the authenticity of the garbled circuit.

1. Sim_m receives $\langle \mathbf{v}_i \rangle_0$ from \mathcal{A} as the input of OT_λ^κ .
2. Garbled Circuit Phase:
 - For $\text{bool}n^f$, Sim_m first computes $\text{Garble}(1^\lambda, \text{bool}n^f) \rightarrow (\text{GC}, \{\{\text{lab}_{i,j}^{\text{in}}\}, \{\text{lab}_{i,j}^{\text{out}}\}\}_{j \in \{0,1\}})$ for each $i \in [2\kappa]$, and then for $i \in \{\kappa+1, \dots, 2\kappa\}$ sends $\{\text{lab}_j^{\text{in}} = \text{lab}_{j, \langle \mathbf{v}_i \rangle_0[j]}\}$ to \mathcal{A} as the output of OT_λ^κ . In addition, Sim_m sends the garbled circuit GC and its garbled inputs $\{\{\text{lab}_j^{\text{in}} = \text{lab}_{j, \langle \mathbf{v}_i \rangle_1[j]}\}_{j \in [\kappa]}\}$ to \mathcal{A} .
3. Authentication Phase 1:
 - Sim_m sets $\delta = (\mathbf{v}_i || \text{sign}(\mathbf{v}_i))$.
 - For $i \in [2\kappa]$, Sim_m sets $\text{lab}_{i,j}^{\text{out}} = \text{lab}_{i,\delta[i]}^{\text{out}}$ if $j = \delta[i]$.
 - For $i \in [2\kappa]$, if $j = 1 - \delta[i]$, \mathcal{S} sets $\text{lab}_{i,j}^{\text{out}}$ as a random value chosen from $\{0,1\}^\lambda$ uniformly, where first bit of $\text{lab}_{i,1-\delta[i]}^{\text{out}}$ is $1 - \varsigma_{i,\delta[i]}$.
 - Sim_m computes and sends $\{ct_{i,j}, \hat{ct}_{i,j}\}_{i \in [2\kappa], j \in \{0,1\}}$ to \mathcal{A} using $\text{lab}_{i,j}^{\text{out}}$ $i \in [2\kappa], j \in \{0,1\}$. This process is same as in Real execution using $\text{lab}_{i,j}^{\text{out}}$ $i \in [2\kappa], j \in \{0,1\}$.
4. Local Computation Phase: The execution of this phase is indistinguishable from Real since no information needs to be exchanged between Sim_m and \mathcal{A} .
5. Authentication Phase 2:
 - The execution is identical to Real .

Hyb_2 : We will make four changes to Hyb_1 to obtain Hyb_2 , and argue that Hyb_2 is indistinguishable from Hyb_1 from the adversary’s view. To be precise, let $\text{GCEval}(\text{GC}, \{\text{lab}_i^{\text{in}}\}_{i \in [2\kappa]}) \rightarrow \{(\tilde{\varsigma}_i || \tilde{\vartheta}_i)_{i \in [2\kappa]} = \{\text{lab}_i^{\text{out}}\}_{i \in [2\kappa]}\}$. First, we have $\{\text{lab}_i^{\text{out}} = \text{lab}_{i,\delta[i]}^{\text{out}}\}_{i \in [2\kappa]}$ based on the correctness of garbled circuits. Second, we note that ciphertexts $\{ct_{i,1-\varsigma_i}, \hat{ct}_{i,1-\varsigma_i+\kappa}\}_{i \in [\kappa]}$ are computed by exploiting the “other” set of output labels picked uniformly in Hyb_1 . Based on this observation, Sim_m actually can directly sample them uniformly at random. Third, in real execution, 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 , and then P_0 computes c_i, d_i and e_i based on them. To simulate this, Sim_m only needs to uniformly select random values c_i, d_i and e_i which satisfy $\langle \alpha \mathbf{v}_i \rangle_0 = (-\sum_{j \in [\kappa]} c_j 2^{j-1})$, $\langle \text{sign}(\mathbf{v}_i) \rangle_0 = (-\sum_{j \in [\kappa]} d_j 2^{j-1})$ and $\langle \alpha \text{sign}(\mathbf{v}_i) \rangle_0 = (-\sum_{j \in [\kappa]} e_j 2^{j-1})$. Finally, since $\langle \alpha \mathbf{v}_i \rangle_0$, $\langle \text{sign}(\mathbf{v}_i) \rangle_0$ and $\langle \alpha \text{sign}(\mathbf{v}_i) \rangle_0$ are part the outputs of functionality $\mathcal{F}_{\text{ONLin}}$, Sim_m can obtain these as the outputs from $\mathcal{F}_{\text{ONLin}}$. In summary, with the above changes, Sim_m no longer needs α of P_1 . We provide

the formal description of Hyb_2 as follows.

1. Sim_m receives $\langle \mathbf{v}_i \rangle_0$ from \mathcal{A} as the input of OT_λ^κ .
 2. Garbled Circuit Phase: Same as Hyb_1 .
 3. Authentication Phase 1:
 - Sim_m runs $\text{GCEval}(\text{GC}, \{\text{lab}_i^{\text{in}}\}_{i \in [2\kappa]}) \rightarrow \{(\tilde{\varsigma}_i || \tilde{\vartheta}_i)_{i \in [2\kappa]} = \{\text{lab}_i^{\text{out}}\}_{i \in [2\kappa]}\}$.
 - Sim_m learns $\langle \alpha \mathbf{v}_i \rangle_0$, $\langle \text{sign}(\mathbf{v}_i) \rangle_0$ and $\langle \alpha \text{sign}(\mathbf{v}_i) \rangle_0$ by sending $\langle \mathbf{v}_i \rangle_0$ to $\mathcal{F}_{\text{ONLin}}$.
 - For $j \in [\kappa]$, Sim_m uniformly selects random values c_j, d_j and $e_j \in \mathbb{F}_p$ which satisfy $\langle \alpha \mathbf{v}_i \rangle_0 = (-\sum_{j \in [\kappa]} c_j 2^{j-1})$, $\langle \text{sign}(\mathbf{v}_i) \rangle_0 = (-\sum_{j \in [\kappa]} d_j 2^{j-1})$ and $\langle \alpha \text{sign}(\mathbf{v}_i) \rangle_0 = (-\sum_{j \in [\kappa]} e_j 2^{j-1})$.
 - For every $i \in [\kappa]$, Sim_m computes $ct_{i,\varsigma_i} = c_i \oplus \text{Trun}_\kappa(\tilde{\vartheta}_i)$ and $\hat{ct}_{i,\varsigma_i+\kappa} = (d_i || e_i) \oplus \text{Trun}_{2\kappa}(\tilde{\vartheta}_{i+\kappa})$. For ciphertexts $\{ct_{i,1-\varsigma_i}, \hat{ct}_{i,1-\varsigma_i+\kappa}\}_{i \in [\kappa]}$, Sim_m samples them uniformly at random.
 - Sim_m sends $\{ct_{i,j}, \hat{ct}_{i,j}\}_{i \in [\kappa], j \in \{0,1\}}$ to \mathcal{A} .
 4. Local Computation Phase: The execution of this phase is indistinguishable from Real since no information needs to be exchanged between Sim_m and \mathcal{A} .
 5. Authentication Phase 2:
 - The execution is identical to Real .
- Hyb_3 : This hybrid we remove Sim_m ’s dependence on P_1 ’s input $\langle \mathbf{v}_i \rangle_1$. The indistinguishability between Hyb_3 and Hyb_2 stems from the security of the garbled circuit. We provide the formal description of Hyb_3 below.
1. Sim_m receives $\langle \mathbf{v}_i \rangle_0$ from \mathcal{A} as the input of OT_λ^κ .
 2. Garbled Circuit Phase:
 - Sim_m samples $\text{Garble}(1^\lambda, \text{bool}n^f) \rightarrow (\tilde{\text{GC}}, \{\hat{\text{lab}}_i^{\text{in}}\}_{i \in \{\kappa+1, \dots, 2\kappa\}})$ and sends $\{\hat{\text{lab}}_i^{\text{in}}\}_{i \in \{\kappa+1, \dots, 2\kappa\}}$ to \mathcal{A} as the output of OT_λ^κ . \mathcal{S} also sends $\tilde{\text{GC}}$ and $\{\hat{\text{lab}}_i^{\text{in}}\}_{i \in [\kappa]}$ to \mathcal{A} .
 3. Authentication Phase 1:
 4. Local Computation Phase: The execution of this phase is indistinguishable from Real since no information needs to be exchanged between Sim_m and \mathcal{A} .
 5. Authentication Phase 2: Same as Hyb_2 , where Sim_m uses $(\langle \mathbf{v}_i \rangle_0, \tilde{\text{GC}}, \text{ and } \{\hat{\text{lab}}_i^{\text{in}}\}_{i \in [2\kappa]})$ to process this phase for \mathcal{A} .

□

APPENDIX E PROOF OF THEOREM 5

Proof: Assuming that P_0 tampered with any of the inputs it holds during the execution, q can be expressed as follows

$$q = \Delta + \sum_{j \in [t]} \mathbf{r}_j(\rho_j - \alpha \tau_j) + \sum_{i \in [m-1]} \mathbf{r}_i(\tau_i - \xi_i)$$

where Δ refers to the increment caused by P_0 's violation of the protocol. The above formula can be expressed as a 1-degree polynomial function $Q(\alpha)$ with respect to the variable α . It is clear that $Q(\alpha)$ is a non-zero polynomial whenever P_0 introduces errors. Further, when $Q(\alpha)$ is

a non-zero polynomial, it has at most one root. Hence, over the choice of α , the probability that $Q(\alpha) = 0$ is at most $1/p$. Therefore, the probability that P_1 aborts is at least $1 - 1/p$ when P_0 cheats. \square