SIMC: ML Inference Secure Against Malicious Clients at Semi-Honest Cost

Nishanth Chandran, Divya Gupta, and Sai Lakshmi Bhavana Obbattu, *Microsoft Research;* Akash Shah, *UCLA*

https://www.usenix.org/conference/usenixsecurity22/presentation/chandran

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

SIMC: ML Inference Secure Against Malicious Clients at Semi-Honest Cost

Nishanth Chandran
Microsoft Research

Divya Gupta
Microsoft Research

Sai Lakshmi Bhavana Obbattu
Microsoft Research

Akash Shah[*]
UCLA

Abstract

Secure inference allows a model owner (or, the server) and the input owner (or, the client) to perform inference on machine learning model without revealing their private information to each other. A large body of work has shown efficient cryptographic solutions to this problem through secure 2-party computation. However, they assume that both parties are semi-honest, i.e., follow the protocol specification. Recently, Lehmkuhl *et al.* showed that malicious clients can extract the whole model of the server using novel model-extraction attacks. To remedy the situation, they introduced the *client-malicious threat model* and built a secure inference system, MUSE, that provides security guarantees, even when the client is malicious.

In this work, we design and build SIMC, a new cryptographic system for secure inference in the client malicious threat model. On secure inference benchmarks considered by MUSE, SIMC has $23-29\times$ lesser communication and is up to $11.4\times$ faster than MUSE. SIMC obtains these improvements using a novel protocol for non-linear activation functions (such as ReLU) that has $> 28\times$ lesser communication and is up to $43\times$ more performant than MUSE. In fact, SIMC's performance beats the state-of-the-art semi-honest secure inference system!

Finally, similar to MUSE, we show how to push the majority of the cryptographic cost of SIMC to an input independent preprocessing phase. While the cost of the online phase of this protocol, SIMC++, is same as that of MUSE, the overall improvements of SIMC translate to similar improvements to the preprocessing phase of MUSE.

1 Introduction

Extensive use of machine learning in applications, specifically inference using pre-trained models, has made the problem of privacy preserving machine learning and in particular, *secure inference* increasingly important. In secure inference, a server P_0 holds a machine learning (ML) model M whose weights w are private and sensitive, while a client P_1 holds a private input data point x. The goal is for P_1 to learn the output of the model on its input - i.e., for P_1 to learn $M(w,x)$ and nothing else; while P_0 must learn no information about client's private input. Secure inference has many applications - private health diagnosis, and secure machine-learning-as-a-service (MLaaS), to name a few. This problem, in theory, can be solved using the cryptographic primitive of secure 2-party computation (2PC) [22,55] that allows any two parties, to run an interactive protocol, to compute any arbitrary function over their inputs without revealing any other information to each other. Over the last 10 years, much work has gone towards building concretely efficient solutions for secure inference [21, 24, 34, 35, 37, 41, 42]. Being already a challenging problem to solve efficiently, all these works focus on the *semi-honest* adversarial model. In this model, both parties P_0 and P_1 are trusted to follow the specifications of the secure inference protocol faithfully and privacy is only provided against such entities that do so.

Prior work by Lehmkuhl *et al.* [32] (refererred to as MUSE) argued that while in deployments it might be reasonable to assume that a server hosting the ML model is semi-honest, it is far less likely that *all* thousands of clients would be semi-honest. This is because while the server is a fixed, typically reputed entity – the model owner, the client could be any arbitray entity. MUSE showed that if a client behaved maliciously (i.e., deviates from protocol specification) in such semi-honest secure inference protocols, then it can completely break the privacy of server's input. Formally, they develop a model extraction attack against state-of-the-art semi-honest protocols that enables a malicious client to learn *all* the weights of the model in far less number of inference queries compared to best black-box model extraction attack [10]. While this attack can be thwarted in theory by using 2PC protocols secure against malicious adversaries [15, 18, 27, 28], the overhead of doing so is exorbitant. To address this gap between existing works in literature and practice, MUSE proposed the *client-malicious* threat model, where the clients are allowed

[*]Work done at Microsoft Research.

to behave malciously, and the server is still assumed to be semi-honest. In this model, MUSE built a system for secure inference with much lower overheads than those that protect against both malicious servers and clients. However, the concrete communication and computational cost of MUSE leaves much to be desired and are still roughly $15\times$ larger than a similar semi-honest system DELPHI [35] supporting arbitrary activation functions that MUSE builds on.

1.1 Our Contribution

In this work, we present SIMC[1] - a new secure neural network inference system that is secure in the client malicious model and is *at least an order of magnitude* more performant than prior state-of-the-art, MUSE.

Neural networks consist of 2 types of layers or functions - *linear layers* (that include functions such as matrix multiplication, convolutions and so on) and *non-linear layers* (that include functions such as ReLU, ReLU6, Maxpool and so on). In the benchmarks considered by MUSE, nearly 99% of the communication overhead of MUSE (and roughly 80% of the overall performance overhead) was due to the secure computation protocols for non-linear layers. At the core of SIMC is a completely novel protocol for securely computing non-linear layers that is even analytically significantly lighter weight compared to MUSE in both compute and communication. As we explain in Section 1.3, MUSE uses computationally heavy leveled homomorphic encryption [8, 19, 20] as well communication heavy authenticated Beaver triples [15, 28] to realize their non-linear layers. In contrast, SIMC uses cheap oblivious transfer and onetime pad encryptions to achieve the same task. Similar to MUSE, our protocol supports computation of arbitrary non-linear functions, is $28-33\times$ communication frugal and upto $43\times$ more performant than MUSE for popular activation functions such as ReLU and ReLU6 (Section 5.2).

Next, we carefully design our protoocols for linear layers and combine them with non-linear layer protocols using a custom consistency check phase, to obtain end-to-end client malicious security for secure inference tasks. On neural network inference benchmarks considered in MUSE, SIMC is $23-29\times$ more communication efficient than MUSE and is between $4.3-11.4\times$ more performant than it. With all this, we are even cheaper than DELPHI, the state-of-the-art in semi-honest secure inference that supports arbitrary non-linear functions, in both communication and runtime.

Finally, MUSE demonstrated how to split their protocols into 2 phases - an offline, client-input independent phase, and an online input-dependent phase. This split can be done in such a way that $> 99.6\%$ of the cryptographic overhead can be moved to the offline phase. We show how to modify our protocols in a similar manner, at the cost of marginally increasing our overall cost. This protocol, SIMC++ pushes almost all cryptographic overhead into a preprocessing model, and has an online phase identical to MUSE. Furthermore, SIMC++'s preprocessing phase is $15-17\times$ communication frugal compared to MUSE and is upto $7.4\times$ more performant.

[1] short for Secure Inference against Malicious Clients

1.2 Technical Overview

We describe our main technical ideas starting with setting up necessary notation for secure inference and background on authenticated shares and mixed arithmetic and boolean computation. We provide a analytical comparison with the state-of-the-art prior work MUSE [32] in Section 1.2.1 and comparison of techniques later in Section 1.3 along with other relevant works.

Notation. For ease of exposition, consider a neural network (NN) with alternating linear and non-linear layers. Let the specification of linear layers be $\mathbf{M}_1, \cdots, \mathbf{M}_\ell$ and of non-linear layers be $f_1, \cdots, f_{\ell-1}$. Consider an input vector $\mathbf{v_0}$, with elements in $\mathbb{F}_p$ (let $\kappa = \lceil \log p \rceil$). Then, one needs to sequentially compute $\mathbf{s}_i = \mathbf{M}_i \mathbf{v}_{i-1}$ and $\mathbf{v}_i = f_i(\mathbf{s}_i)$ for $i \in \{1, \dots, \ell-1\}$, followed by $\mathbf{s}_\ell = \mathbf{M}_\ell \mathbf{v}_{\ell-1} = \mathsf{NN}(\mathbf{v}_0)$. For the setting of secure inference, as already discussed, the server (P_0) holds $\mathbf{M}_1, \cdots, \mathbf{M}_\ell$ and the client (P_1) holds $\mathbf{v}_0$.

Authentication. A long line of influential work [5, 15, 27, 28] on malicious secure computation for dishonest majority uses information theoretic homomorphic MACs as follows: Parties hold shares of a MAC key $\alpha \in \mathbb{F}_p$. The protocol maintains the invariant that parties start with *authenticated shares* of input $x \in \mathbb{F}_p$ to an arithmetic gate (shares of x along with shares of MAC on x, i.e., αx) and compute authenticated shares of the output of the gate. These authentications, along with a protocol specific consistency check at the end, ensure that any malicious behavior is detected and the honest party aborts. In the client malicious setting, as pointed by [32], the server (semi-honest party) can pick the MAC key.

Mixed Computation. The main challenge for computations such as NN inference is the use of *mixed computation*. In more detail, linear layers such as matrix multiplications and convolutions are best expressed as arithmetic circuits, and non-linear activations such as ReLU, ReLU6, and Maxpool are best expressed as boolean circuits. Applying standard machinery for malicious security to mixed computation is problematic and inefficient. For example, while switching between authenticated shares (for arithmetic compute) and garbled circuits (for boolean compute),one must ensure that labels obtained by the (malicious) client correspond to the correct authenticated share. Recent line of work [2, 16, 18, 46] considers such mixed-computation in the malicious setting with dishonest majority. MUSE, by focusing on mixed compute for the weaker client malicious setting, provides protocols that are much more efficient than standard malicious secure protocols [28]. We now discuss our solution that outperforms the prior work by atleast an order of magnitude, in the client malicious setting.

1.2.1 Our Protocol

We maintain the invariant that P_0 and P_1 hold authenticated shares of s values that are components of the output vectors of linear layers, then the shares of s are input to non-linear layers. We first discuss how we realize non-linear layers in our protocol where the challenge is to design a boolean-friendly protocol for non-linear layers that ensures that a malicious client feeds in the correct share of s, and that the output of the non-linear layer is fed correctly into the subsequent linear layer. This describes our main technical ideas; we follow this by describing the computation of linear layers and finally the consistency check phase.

Non-linear layers. We use standard semi-honest secure garbled circuits to realize the non-linear layers and labels corresponding to the client's share are transferred using receiver malicious oblivious transfer. Note that this step allows for a malicious client to input arbitrarily values, and not necessarily the outputs from the previous linear layer. Our main technical idea to ensure security against a malicious client is to generate a re-authentication on the input s and later verify the equality of two independently generated authentications on s. Below, we describe how we efficiently realize both parts.

We define our functionality for non-linear layers computing a function f as follows: It takes shares of a value s from P_0 and P_1 and returns shares of $u = \alpha s$ and authenticated shares of $v = f(s)$, i.e., shares of $f(s)$ and $\alpha f(s)$. Note that the previous linear layer already outputs authenticated shares of s, i.e., shares of s and shares of $t = \alpha s$. Hence, t and u are MACs on the same s computed by consecutive linear and non-linear layers, respectively. We check for their equality during the final consistency check phase, which we describe later. Now, realizing this functionality in a straightforward manner using garbled circuits requires doing at least 2 field multiplications within the circuit and is quite expensive. Note that doing field multiplication inside a garbled circuit requires $O\left(\kappa^2\lambda\right)$ communication.

To remedy this, we garble a circuit, that given shares of s, only computes s and $f(s)$ (and not their MACs). Now, one of our key technical ideas is to effectively use the output labels of the garbled circuit as one-time pad encryption keys to send the appropriate shares of $\alpha s, f(s), \alpha f(s)$ to the client. Security of this step follows by the authenticity property of garbled circuits (see Section 2.2.3). This requires only $6\kappa^2$ bits of additional communication, where κ is the bitlength used. For further details see Section 3.

Overall, our protocol only requires communication of approximately $2c\lambda + 4\kappa\lambda + 6\kappa^2$, where c is the number of AND gates required to reconstruct shares of s and compute $f(s)$. In contrast, MUSE required communication[2] of at least $2d\lambda + 190\kappa\lambda + 232\kappa^2$ where $d > c$ is number of AND gates required to reconstruct shares of s, compute $f(s)$ and generate shares of $f(s)$. As an example, when computing the popular non-linear function $\mathsf{ReLU}(x) := \max(x, 0)$, with $\lambda = 128$ and $\kappa = 44$, this results in our protocol being roughly $30\times$ more communication efficient than the corresponding protocol in MUSE (see Section 5.2).

[2]Constant factors are determined using existing implementation.

Linear Layers. We compute authenticated shares of $\mathbf{s}_{i+1}$ from authenticated shares of $\mathbf{v}_i = f_i(\mathbf{s}_i)$ using additively homormorphic encryption (AHE). Denote authentication on $\mathbf{v}_i$ as $\mathbf{w}_i$. To prevent P_1 from inputing something different from what it received from the previous non-linear layer, we compute an additional tag $\mathbf{z}_i = \alpha^3\mathbf{v}_i - \alpha^2\mathbf{w}_i$, which is zero if $\mathbf{w}_i = \alpha\mathbf{v}_i$ and non-zero otherwise for $\alpha \neq 0$. We check for $\mathbf{z}_i = 0$ in the final consistency check described below. As discussed in Section 4.1 and validated in Section 5, our cost for linear layer is similar to MUSE.

Consistency check. Let $\mathbf{t}_i$ and $\mathbf{u}_i$ denote authentications on $\mathbf{s}_i$ obtained from the i^{th} linear and non-linear layers respectively. Recall that $\mathbf{z}_i$ is the tag output from linear layers. In this phase, P_0 and P_1 compute a random linear combination of $\mathbf{t}_i - \mathbf{u}_i$ and $\mathbf{z}_i$ and P_0 checks that final result q evaluates to 0. We formally show that if P_1 deviates from the protocol, then $q \neq 0$ with overwhelming probability. This proof crucially relies on the structure of $\mathbf{z}_i$, that is, $\mathbf{z}_i$ using high powers for α, to avoid cancellation between different errors introduced by P_1. This phase is super lightweight.

Preprocessing Model. Plugging our novel protocols and custom consistency checks into the basic protocol design of DELPHI/MUSE, we are able to push bulk of the cryptographic cost of our protocol to an client input-independent preprocessing phase. Our online phase has same cost as MUSE. For details see Section 4.4.

1.3 Related Works

Since MUSE [32] is the only prior work on secure inference in the client-malicious setting, we begin with a detailed comparison of techniques with MUSE pointing out their main performance bottleneck. For non-linear layers, MUSE uses a garbled circuit that takes shares of s as input and outputs shares of $f(s)$. Note that this circuit has higher number of AND gates compared to the circuit used in SIMC. Next, they provide specialized protocols to securely transfer the labels corresponding to the client's share of s when the corresponding MAC is valid. This part of their protocol has the highest complexity among all the building blocks used in MUSE. At a high level, the parties do the following: for every bit of client's share for s, they run a secure multiplication protocol where the parties learn the shares of the label. The server sends its share of the label, only if the check on client's input and MACs succeed. This method, in addition to the garbled circuit communication, requires communication of at least $188\lambda\kappa + 232\kappa^2$ for each s, for security parameter λ. Furthermore, this step is computationally heavy and corresponds to bulk of the compute cost of non-linear layers due to the use of homomorphic encryption (to generate $(\lambda + \kappa)$ authenticated

Beaver triples per s value). We provide an extensive empirical comparison with MUSE in Section 5.

We note that the works of [2,16,18,46] study mixed computation and provide malicious security in the dishonest majority setting. [53] discusses secure inference in the zero knowledge setting. All these works use boolean MACs as a building block for authenticated boolean computation and hence incur higher cost than SIMC. In our protocol, to provide security against a malicious client we do not use any building block for authenticated boolean computation and instead leverage the authenticity (of the semi-honest secure) garbled circuits to detect a cheating client, thus avoiding boolean MACs. Our techniques differ in this fundamental way.

Many prior works also consider secure inference in the honest majority setting [7,13,30,36,44,50,51], or rely on trusted hardware [38, 49]. In this work, we focus on the stronger threat model of 2PC and provide formal cryptographic security guarantees. Other works include those that considered malicious adversaries [12, 23, 57] (for simpler ML models like linear models, regression, and polynomials) as well as specialized DNNs with 1 or 2 bit weights [1,43,47].

1.4 Organisation

We begin by describing the various cryptographic building blocks used by our protocol and the threat model in Section 2. Section 3 presents our novel protocol for non-linear functions along with its security proof. In Section 4, we describe how to securely compute linear layers as well as our end-to-end protocol for secure inference and its proof. In the same section we provide details of our secure inference protocol in the preprocessing model. We provide implementation details and empirical results in Section 5. We conclude in Section 6.

2 Threat Model and Building Blocks

2.1 Threat Model and Security

Threat Model. We consider the two party setting with a server P_0 and a client P_1. The adversary can corrupt the server, but is restricted to be semi-honest, i.e., is guaranteed to follow the protocol specification. Or, the adversary can corrupt the client and can behave maliciously, i.e., is allowed to deviate from the protocol arbitrarily. The network architecture is assumed to be known to both P_0 and P_1. Our goal is to design a protocol that allows the client to learn the inference result on the model owned by the server; the client must learn no other information and the server must learn no information through this interaction. Our formal definition, provided for completeness in Appendix B captures this.

Hybrid Model. Our protocols sometimes invoke multiple sub-protocols and we describe these using the hybrid model. This is similar to a real interaction, except that sub-protocols are replaced by the invocations of instances of corresponding functionalities. A protocol invoking a functionality $\mathcal{F}$ is said to be in "$\mathcal{F}$-hybrid model."

2.2 Building Blocks

Notation. λ is the computational security parameter. σ is the statistical security parameter. For $n > 0$, $[n]$ denotes the set $\{1,2,\cdots,n\}$. In this paper, all arithmetic additions and multiplications are over a field $\mathbb{F}_p$, where p is a prime and $\kappa = \lceil \log p \rceil$. We assume natural mapping of elements in $\mathbb{F}_p$ to $\{0,1\}^\kappa$ and $a[i]$ denotes the i^{th} bit of this map for $a \in \mathbb{F}_p$ (i.e $a = \sum_{i\in[\kappa]} a[i] \cdot 2^{i-1}$). For two vectors $\mathbf{a}$ and $\mathbf{b}$, $\mathbf{a}+\mathbf{b}$ represents their component wise addition. For an element $\alpha \in \mathbb{F}_p$ and a vector $\mathbf{a}$ over $\mathbb{F}_p$, $\alpha + \mathbf{a}$ and $\alpha\mathbf{a}$ denote addition and multiplication of each component in $\mathbf{a}$ with α respectively. Inner product of vectors $\mathbf{a}$ and $\mathbf{b}$ is denoted by $\mathbf{a}*\mathbf{b}$. For any function $f : \mathbb{F}_p \to \mathbb{F}_p$, $f(\mathbf{a})$ denotes evaluation of f on each component of $\mathbf{a}$. We use a similar notation for functionalities and $\mathcal{F}(\mathbf{a})$ denotes invocation of $\mathcal{F}$ on each component of $\mathbf{a}$. $a||b$ denotes concatenation of a and b. We denote uniform distribution on the set $\{0,1\}^n$ by U_n for any $n > 0$. For any two distributions A and B, $A \approx B$ denotes computational indistinguishability of A and B.

Additive Secret Sharing. For $x \in \mathbb{F}_p$, $\langle x\rangle_0 \in \mathbb{F}_p$ and $\langle x\rangle_1 \in \mathbb{F}_p$ denote additive shares of x, i.e., $x = (\langle x\rangle_0 + \langle x\rangle_1) \mod p$.

Authenticated Shares [15]. For $\alpha \in \mathbb{F}_p$ chosen uniformly at random (known as the MAC key) and any $x \in \mathbb{F}_p$, authenticated shares of x on α denote that P_b holds the shares $\langle x\rangle_b$ and $\langle \alpha x\rangle_b$ for $b \in \{0,1\}$. While fully malicious protocols [15, 27, 28] require α to be uniform and secret shared amongst all participating parties, in our client malicious setting, similar to [32], we have P_0 pick α. Note that these authenticated shares offer $\lfloor \log p \rfloor$-bit statistical security, which means the probability that a malicious P_1 forges the shared value x to $x+\delta$ (by tampering the shares $(\langle x\rangle_1, \langle \alpha x\rangle_1)$ to $(\langle x\rangle_1 + \delta, \langle \alpha x\rangle_1 + \delta')$ for a non-zero $(\delta,\delta') \in \mathbb{F}_p^2$) such that the shares are authenticated on $x+\delta$ (i.e $(\alpha x + \delta') = \alpha(x+\delta)$) is atmost $2^{-\lfloor \log p \rfloor}$.

2.2.1 Additive Homomorphic Encryption

An additive homomorphic encryption scheme [17, 39, 45] $\mathsf{AHE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ is a public key encryption scheme that additionally supports linear homomorphic operations on the ciphertexts. We use $\mathbb{F}_p$ as the message space.

- $\mathsf{KeyGen} \to (\mathsf{pk}, \mathsf{sk})$. KeyGen is a randomised algorithm that samples a public key pk and a corresponding secret key sk.
- $\mathsf{Enc}(\mathsf{pk}, m) \to c$. Enc takes a public key pk and a message $m \in \mathbb{F}_p$ to output a ciphertext c.
- $\mathsf{Dec}(\mathsf{sk}, c) \to m$. Dec takes a secret key sk and a ciphertext c to output the message m encrypted in c.
- $\mathsf{Eval}(\mathsf{pk}, c_1, c_2, L) \to c'$. Eval takes a public key pk and two ciphertexts c_1 and c_2 encrypting messages $m_1, m_2 \in \mathbb{F}_p$ and

a linear function L[3] to output a ciphertext c' encrypting $L(m_1, m_2)$.

We require AHE to satisfy correctness, semantic security and additive homomorphism along with *function privacy*[4]. Concretely, we use the SEAL library [48] that implements the more versatile fully homomorphic scheme [20] from [8, 19].

2.2.2 Oblivious Transfer

The oblivious transfer (OT) functionality [40] over strings of length n, denoted by OT_n, takes as input $s_0, s_1 \in \{0,1\}^n$ from P_0 (the sender) and a choice bit $c \in \{0,1\}$ from P_1 (the receiver) and outputs s_c to P_1. We require instantiation of OT_n that is secure against a semi-honest sender and a malicious receiver. Finally, we use OT_n^k to denote k instances of OT_n. We use the instantiation from [26] that has communication complexity of $k(\lambda + 2n)$-bits.

2.2.3 Garbled Circuits

A garbling scheme for boolean circuits [4, 55] consists of a pair of algorithms $(\mathsf{Garble}, \mathsf{GCEval})$ defined as:

- $\mathsf{Garble}(1^\lambda, C) \rightarrow (\mathsf{GC}, \{\{\mathsf{lab}^{\mathsf{in}}_{i,j}\}_{i\in[n]}, \{\mathsf{lab}^{\mathsf{out}}_{j}\}\}_{j\in\{0,1\}})$. Garble on input the security parameter λ and a boolean circuit $C : \{0,1\}^n \rightarrow \{0,1\}$ outputs a garbled circuit GC, a collection of input labels $\{\mathsf{lab}^{\mathsf{in}}_{i,j}\}_{i\in[n],j\in\{0,1\}}$ and a collection of output labels $\{\mathsf{lab}^{\mathsf{out}}_{j}\}_{j\in\{0,1\}}$ where each label is of λ-bits. For any $x \in \{0,1\}^n$, the labels $\{\mathsf{lab}^{\mathsf{in}}_{i,x[i]}\}_{i\in[n]}$ are referred to as the *garbled input* for x and the label $\mathsf{lab}^{\mathsf{out}}_{C(x)}$ is referred to as the *garbled output* for $C(x)$.
- $\mathsf{GCEval}(\mathsf{GC}, \{\mathsf{lab}_i\}_{i\in[n]}) \rightarrow \mathsf{lab}'$. GCEval on input a garbled circuit GC and a set of labels $\{\mathsf{lab}_i\}_{i\in[n]}$ outputs a label lab'.

and is required to satisfy the following properties. Let $\mathsf{Garble}(1^\lambda, C) \rightarrow (\mathsf{GC}, \{\{\mathsf{lab}^{\mathsf{in}}_{i,j}\}_{i\in[n]}, \{\mathsf{lab}^{\mathsf{out}}_{j}\}\}_{j\in\{0,1\}})$.

- **Correctness** is the guarantee that evaluation of GCEval on GC and garbled input of x gives the garbled output for $C(x)$. Formally, for any circuit C and $x \in \{0,1\}^n$, it holds that $\mathsf{GCEval}(\mathsf{GC}, \{\mathsf{lab}^{\mathsf{in}}_{i,x[i]}\}_{i\in[n]}) = \mathsf{lab}^{\mathsf{out}}_{C(x)}$.
- **Security** is the guarantee that GC and garbled input for any x is simulatable given C. Formally, there exists a simulator Sim such that for any circuit C and $x \in \{0,1\}^n$, $(\mathsf{GC}, \{\mathsf{lab}^{\mathsf{in}}_{i,x[i]}\}_{i\in[n]}) \approx \mathsf{GCSim}(1^\lambda, C)$.
- **Authenticity** implies that given GC and a garbled input of x, it is infeasible to guess the output label for $1 - C(x)$. Formally, for any circuit C and any x, it holds that $\left(\mathsf{lab}^{\mathsf{out}}_{(1-C(x))} \middle| \mathsf{GC}, \{\mathsf{lab}^{\mathsf{in}}_{i,x[i]}\}_{i\in[n]}\right) \approx U_\lambda$.

[3]L maps (m_1, m_2) to $am_1 + m_2$ for some $a \in \mathbb{F}_p$

[4]Function privacy informally guarantees that a ciphertext c encrypting a share of $L(m_1, m_2)$, obtained as a result of homomorphically computing L, is indistinguishable from a ciphertext encrypting a share of $L'(m_1, m_2)$, for any other L', even given sk.

Note that this definition naturally extends to boolean circuits with multi-bit outputs. Garbling scheme instantiations use two labels $\{\mathsf{lab}_{w,0}, \mathsf{lab}_{w,1}\}$ for every wire w in C and the initial constructions of garbled circuit comprises of a set of four ciphertexts for each gate in C, one for each pair of input wire labels. Our protocols use the instantiation of garbling schemes with point-and-permute described below.

Point-and-permute [3]: Consider any gate g in C. Given a label for each input wire of g, earlier instantiations of garbled circuits required to decrypt all the four ciphertexts (using the labels as keys) during the evaluation, to obtain a label for the output wire of g. For every wire w in C, the point-and-permute optimization allows to choose a permutation bit b and for $j \in \{0,1\}$, the bit $b \oplus j$ is prepended to $\mathsf{lab}_{w,j}$. The four ciphertexts of g are then cleverly permuted according to the permutation bits of g's input wires such that given a label for each input wire of the gate the ciphertext to be decrypted is uniquely pointed. The property that we use from point-and-permute is that, for all the wires, the exclusive-or of the first bits of the two labels is 1.

We use the state-of-the-art constructions of garbled circuits that use point-and-permute and other optimizations such as free-XOR [29] and half-gates [56]. For these instantiations, the size of the garbled ciruit is $2c\lambda$, where c is the number of AND gates in the circuit C, that is being garbled.

Secure computation using garbled circuits. As background, we provide high-level description on how garbled circuits along with OTs are typically used for 2-party secure computation [55]. A semi-honest P_0 and a malicious P_1 with inputs x and y respectively, can securely evaluate[5] any boolean circuit C on these inputs to learn $C(x,y)$ as follows. P_0 garbles the circuit C to learn a garbled circuit GC and a collection of input and output labels. Then, both the parties invoke the OT functionality, where the server (acting as the sender) inputs the collection of input labels corresponding to the input wires of P_1 and P_1(acting as the receiver) inputs y to learn the garbled input for y. P_0 additionally sends to P_1 the garbled input for x, GC and a pair of ciphertexts for every output wire w (of C) encrypting the bits 0 and 1 using the labels $\mathsf{lab}_{w,0}$ and $\mathsf{lab}_{w,1}$ respectively, as encryption keys. P_1 evaluates the garbled circuit using the garbled inputs for (x,y) to learn the garbled output for $C(x,y)$. From this garbled output and the pair of ciphertexts given for each output wire P_1 learns $C(x,y)$. P_1 sends $C(x,y)$ along with the (hash of) garbled output to P_0 who would accept $C(x,y)$ upon verifying that the (hash of) garbled output sent by P_1 corresponds to $C(x,y)$.

3 Nonlinear Functions

We begin by formally describing the functionality $\mathcal{F}^{f}_{\mathsf{Non\text{-}lin}}$ for non-linear layers in Figure 1 required by our protocol

[5]See Appendix B for formal interpretation of secure evaluation.

for secure inference in Section 4.2. For ease of exposition, we first consider elementary or single-input functions, i.e., the functionality is parameterized by a function $f : \mathbb{F}_p \rightarrow \mathbb{F}_p$ and defer the discussion on multi-input functions such as Maxpool to Remark 3.1. $\mathcal{F}^{f}_{\text{Non-lin}}$ takes shares of $s \in \mathbb{F}_p$ from P_0 and P_1 and a MAC key α (also in $\mathbb{F}_p$) from P_0 as input. It outputs authenticated shares of $f(s)$ (i.e., shares of $f(s)$ and $\alpha f(s)$), along with the shares of authentication on s (i.e. αs) to both parties.

Function $f : \mathbb{F}_p \rightarrow \mathbb{F}_p$.
Input: P_0 sends $\langle s \rangle_0, \alpha \in \mathbb{F}_p$. P_1 sends $\langle s \rangle_1 \in \mathbb{F}_p$.
Output: P_b learns $\langle \alpha s \rangle_b, \langle f(s) \rangle_b, \langle \alpha f(s) \rangle_b$ for $b \in \{0,1\}$.

Figure 1: Functionality $\mathcal{F}^{f}_{\text{Non-lin}}$.

The above functionality can be realized naïvely using garbled circuits. In slightly more detail, P_0 can create a garbled circuit that computes the above functionality - i.e., reconstructs s from its shares, computes $\alpha s, f(s), \alpha f(s)$, and outputs shares of these values to P_0 and P_1 respectively. However, this approach has the drawback that field multiplication (while computing $\alpha f(s)$ and αs) must be performed within the garbled circuit. Field multiplication would require roughly κ^2 AND gates and hence, this would lead to a protocol with high communication. In our protocol that realizes this functionality, described in the following subsection, we show how to avoid performing field multiplications within the garbled circuit and consequently, how to obtain a protocol with much lower cost. Moreover, our technique is also more efficient in generating additive shares of $f(s)$ after computing $f(s)$.

3.1 Protocol

Our protocol proceeds in three main phases - Garbled Circuit phase, Authentication phase, and Local Computation phase. Below, we provide a high level overview of our protocol. Recall that P_0 inputs $(\langle s \rangle_0, \alpha)$ and P_1 inputs $\langle s \rangle_1$.

- **Garbled Circuit phase.** The goal of this phase is for P_0 to hold a pair of "labels" for each bit of s and $f(s)$ and P_1 to learn the "correct labels" depending on the value of the bits of s and $f(s)$. To enable this, P_0 creates a garbled circuit for the boolean circuit Comp^f – this circuit takes shares of $a \in \mathbb{F}_p$, reconstructs a and outputs $(a, f(a))$. Recall that a garbled circuit has 2 output labels encoded in it for each of the output bits, which in this case are the bits of a and $f(a)$. P_1 evaluates this garbled circuit on $\langle s \rangle_0$ and $\langle s \rangle_1$ after learning the correct input labels using an OT protocol. After this phase, P_1 learns the set of output labels corresponding to the bits of s and $f(s)$.
- **Authentication phase.** We make two observations. First, "output labels" of a garbled circuit can be used as one-time pads for encryption[6]. Second, to compute shares of αs, it suffices to compute shares of $\alpha(s[i])$ (written shortly as $\alpha s[i]$) for every bit $s[i]$ of s (a similar observation holds for computing $\alpha f(s)$ and hence we focus the rest of this discussion on computing shares of αs). Now, shares of $\alpha s[i]$ are either shares of 0 or α depending on whether $s[i]$ is 0 or 1. Recall that the garbled circuit used in the previous phase had 2 output labels corresponding to every $s[i]$ (one each for $s[i] = 0$ and 1); we denote these labels by $\mathsf{lab}^{\mathsf{out}}_{i,0}$ and $\mathsf{lab}^{\mathsf{out}}_{i,1}$ for $s[i] = 0$ and $s[i] = 1$, respectively. Now, to compute shares of $\alpha s[i]$, P_0 picks a random $\nu_i \in \mathbb{F}_p$ and "encrypts" ν_i with $\mathsf{lab}^{\mathsf{out}}_{i,0}$ and $\nu_i + \alpha$ with $\mathsf{lab}^{\mathsf{out}}_{i,1}$. P_0 sends these 2 ciphertexts to P_1. P_0 sets it's share of $\alpha s[i]$ as $-\nu_i$. Now, since from the first phase P_1 received $\mathsf{lab}^{\mathsf{out}}_{i,s[i]}$, P_1 can decrypt exactly one of these 2 ciphertexts[7] and learn its share of $\alpha s[i]$. Computation of $\alpha f(s)[i]$ values for every i are done in a similar manner using the output labels for $f(s)$. We use a similar logic to compute shares of $f(s)[i]$ using output labels for $f(s)$.
- **Local Computation phase.** As the last step, the parties can locally compute shares of αs, $f(s)$ and $\alpha f(s)$ from shares of $\{\alpha s[i]\}$, $f(s)[i]$ and $\{\alpha f(s)[i]\}$ as follows: Each party locally multiplies the share of $\alpha s[i]$ with 2^{i-1} and sums all the resultant values to obtain share of αs. The other outputs are computed in a similar manner.

We describe our protocol for realising $\mathcal{F}^{f}_{\text{Non-lin}}$ formally in Figure 2. Note that our protocol only incurs additional communication of $6\kappa^2$ bits (as ciphertexts) in addition to garbled circuits and has much lower communication compared to the naïve approach. For comparison of our non-linear layers with MUSE, refer to Section 1.2 for (rough) asymptotic comparison and Section 5.2 for concrete comparison of cost (both communication and runtime).

Remark 3.1 (Multi-input non-linear functions)**.** Our techniques extend easily for non-elementary functions, i.e., functions that take multiple inputs and produce one or more outputs, such as Maxpool. For this, our garbled circuit will take shares of all inputs $\{s_1, \ldots, s_k\}$ to f and output the reconstructed input values $\{s_1, \ldots, s_k\}$ and $f(s_1, \ldots, s_k)$. We generate authentications on both of these using exactly the same ideas. Moreover, when the Maxpool windows are overlapping, we create a big garbled circuit for the entire non-linear layer so that our cost is linear in input size and output size (instead of input-size×filter-size).

[6] Strictly speaking, in order to be compatible with the point-and-permute optimization, the label is parsed as two components and a part of it is can be used as a one-time pad.

[7] We make use of the point-and-permute optimization to determine which of the two ciphertexts should be decrypted by P_1.

Preamble: The function f is such that $f : \mathbb{F}_p \rightarrow \mathbb{F}_p$. Consider a boolean circuit Comp^f that takes additive shares of $a \in \mathbb{F}_p$, i.e., $\langle a\rangle_0, \langle a\rangle_1 \in \mathbb{F}_p$, as input and outputs $(a, f(a))$. Let $\mathsf{Trim}_n : \{0,1\}^\lambda \rightarrow \{0,1\}^n$ be a function that outputs the last n bits of input.
Input: P_0 inputs $\langle s\rangle_0 \in \mathbb{F}_p^n$ and $\alpha \in \mathbb{F}_p$. P_1 inputs $\langle s\rangle_1 \in \mathbb{F}_p^n$.
Output: P_b learns $\langle \alpha s\rangle_b, \langle f(s)\rangle_b, \langle \alpha f(s)\rangle_b$ for $b \in \{0,1\}$.
Protocol:

1. *Garbled Circuit Phase*:
   - P_0 computes $(\mathsf{GC}, \{\{\mathsf{lab}^{\mathsf{in}}_{i,j}\}_{i\in[2\kappa]}, \{\mathsf{lab}^{\mathsf{out}}_{i,j}\}_{i\in[2\kappa]}\}_{j\in\{0,1\}}) \leftarrow \mathsf{Garble}(1^\lambda, \mathsf{Comp}^f)$
   - P_0 and P_1 invoke $\mathsf{OT}^\kappa_\lambda$ where P_0 is the sender and P_1 is the receiver with inputs $\{\mathsf{lab}^{\mathsf{in}}_{i,0}, \mathsf{lab}^{\mathsf{in}}_{i,1}\}_{i\in\{\kappa+1,\cdots,2\kappa\}}$ and $\langle s\rangle_1$, respectively. P_1 learns $\{\widetilde{\mathsf{lab}}^{\mathsf{in}}_i\}_{i\in\{\kappa+1,\cdots,2\kappa\}}$. P_0 sends GC and $\{\widetilde{\mathsf{lab}}^{\mathsf{in}}_i = \mathsf{lab}^{\mathsf{in}}_{i,\langle s\rangle_0[i]}\}_{i\in[\kappa]}$ to P_1.
   - P_1 computes $\{\widetilde{\mathsf{lab}}^{\mathsf{out}}_i\}_{i\in[2\kappa]} \leftarrow \mathsf{GCEval}(\mathsf{GC}, \{\widetilde{\mathsf{lab}}^{\mathsf{in}}_i\}_{i\in[2\kappa]})$
2. *Authentication Phase*:
   - For $i \in [\kappa]$, P_0 chooses $\eta_{i,0}, \delta_{i,0}, \nu_{i,0} \in_R \mathbb{F}_p$ and sets $(\eta_{i,1}, \delta_{i,1}, \nu_{i,1}) = (1+\eta_{i,0}, \alpha+\delta_{i,0}, \alpha+\nu_{i,0})$.
   - For $i \in [2\kappa]$ and $j \in \{0,1\}$, P_0 parses $\mathsf{lab}^{\mathsf{out}}_{i,j}$ as $p_{i,j}||k_{i,j}$ where $p_{i,j} \in \{0,1\}$ and $k_{i,j} \in \{0,1\}^{\lambda-1}$.
   - For $i \in [\kappa], j \in \{0,1\}$, P_0 sends $\mathsf{ct}_{i,p_{i,j}} = \nu_{i,j} \oplus \mathsf{Trim}_\kappa(k_{i,j})$ and $\hat{\mathsf{ct}}_{i,p_{i+\kappa,j}} = (\eta_{i,j}||\delta_{i,j}) \oplus \mathsf{Trim}_{2\kappa}(k_{i+\kappa,j})$.
   - For $i \in [2\kappa]$, P_1 parses $\widetilde{\mathsf{lab}}^{\mathsf{out}}_i$ as $\tilde{p}_i||\tilde{k}_i$ where $\tilde{p}_i \in \{0,1\}$ and $\tilde{k}_i \in \{0,1\}^{\lambda-1}$.
   - For $i \in [\kappa]$, P_1 computes $c_i = \mathsf{ct}_{i,\tilde{p}_i} \oplus \mathsf{Trim}_\kappa(\tilde{k}_i)$ and $(d_i||e_i) = \hat{\mathsf{ct}}_{i,\tilde{p}_{i+\kappa}} \oplus \mathsf{Trim}_{2\kappa}(\tilde{k}_{i+\kappa})$.
3. *Local Computation Phase*:
   - P_0 outputs $\langle z_1\rangle_0 = \left(-\sum_{i\in[\kappa]} \nu_{i,0} 2^{i-1}\right)$, $\langle z_2\rangle_0 = \left(-\sum_{i\in[\kappa]} \eta_{i,0} 2^{i-1}\right)$ and $\langle z_3\rangle_0 = \left(-\sum_{i\in[\kappa]} \delta_{i,0} 2^{i-1}\right)$.
   - P_1 outputs $\langle z_1\rangle_1 = \left(\sum_{i\in[\kappa]} c_i 2^{i-1}\right)$, $\langle z_2\rangle_1 = \left(\sum_{i\in[\kappa]} d_i 2^{i-1}\right)$ and $\langle z_3\rangle_1 = \left(\sum_{i\in[\kappa]} e_i 2^{i-1}\right)$.

Figure 2: Protocol $\pi^f_{\mathsf{Non\text{-}lin}}$

Remark 3.2. Note that in Step 2 of our protocol, P_0 uses (parts of) output labels of garbled circuit (i.e., λ-bit strings) to one-time pad values of length κ bits and 2κ bits. For our benchmarks, it holds that $\lambda \geqslant 2\kappa$. In case this condition is not met, i.e., $\lambda < 2\kappa$, P_0 first applies a pseudorandom generator (PRG) to the output label to expand it to an appropriate length string and then uses the PRG output as an one-time pad.

Remark 3.3 (Checking well-formedness of client input)**.** Note that a malicious client can input a value $x \in \{0,1\}^\kappa$ such that $x \notin \mathbb{F}_p$ as its share of the input to Comp^f. In our implementation, the garbled circuit first checks if $x < p$ and outputs a label corresponding to this bit. In the end of evaluation phase of our main protocol, the client will send a hash of all such labels to the server, who will check that it is equal to the hash of labels corresponding to the bit 1. This will be a part of the consistency check phase (see Section 4.2).

Remark 3.4 (Optimization for semi-honest setting)**.** Our novel method for generating shares of $f(s)$ is beneficial even for semi-honest secure inference. All prior works [24, 34, 35] generated arithmetic shares of $f(s)$ within the garbled circuit. When working over prime fields, this computation requires at least 3κ additional AND gates, and hence, $6\kappa\lambda$ bits of communication. Our method for generating shares of $f(s)$ using garbled circuit output labels requires sending only 2κ encryptions of length κ bits each. For $\kappa = 44$, $\lambda = 128$, our method gives roughly $9\times$ lower communication for generating shares of $f(s)$ from $f(s)$.

Theorem 1. *Let* $(\mathsf{Garble}, \mathsf{GCEval})$ *be a garbling scheme for boolean circuits satisfying the properties defined in Section 2.2.3. Then, the protocol* $\pi^f_{\mathsf{Non\text{-}lin}}$ *(in Figure 2) securely realizes the functionality* $\mathcal{F}^f_{\mathsf{Non\text{-}lin}}$ *in the* $\mathsf{OT}^\kappa_\lambda$*-hybrid model against a malicious client (P_1) and a semi-honest server (P_0).*

Proof. We first prove correctness of the protocol followed by security.
Correctness. By correctness of $\mathsf{OT}^\kappa_\lambda$, for all $i \in \{\kappa+1, \cdots, 2\kappa\}$, $\widetilde{\mathsf{lab}}^{\mathsf{in}}_i = \mathsf{lab}^{\mathsf{in}}_{i,\langle s\rangle_1[i]}$. Using $\widetilde{\mathsf{lab}}^{\mathsf{in}}_i = \mathsf{lab}^{\mathsf{in}}_{i,\langle s\rangle_0[i]}$ for $i \in [\kappa]$ and correctness of $(\mathsf{Garble}, \mathsf{GCEval})$ for Comp^f, it holds that $\widetilde{\mathsf{lab}}^{\mathsf{out}}_i = \mathsf{lab}^{\mathsf{out}}_{i,s[i]}$ and $\widetilde{\mathsf{lab}}^{\mathsf{out}}_{i+\kappa} = \mathsf{lab}^{\mathsf{out}}_{i+\kappa,f(s)[i]}$, for $i \in [\kappa]$. Therefore, $\tilde{p}_i||\tilde{k}_i = (p_{i,s[i]}||k_{i,s[i]})$ and $\tilde{p}_{i+\kappa}||\tilde{k}_{i+\kappa} = (p_{i+\kappa,f(s)[i]}||k_{i+\kappa,f(s)[i]})$ for $i \in [\kappa]$. Using this we get, for each $i \in [\kappa]$ it holds that $c_i = \mathsf{ct}_{i,p_{i,s[i]}} \oplus \mathsf{Trim}_\kappa(k_{i,s[i]}) = \nu_{i,s[i]}$ and $(d_i||e_i) = \hat{\mathsf{ct}}_{i,p_{i+\kappa,f(s)[i]}} \oplus \mathsf{Trim}_{2\kappa}(k_{i+\kappa,f(s)[i]}) = (\eta_{i,f(s)[i]}||\delta_{i,f(s)[i]})$. With this, we have:

- $z_1 = \sum_{i\in[\kappa]}(c_i - \nu_{i,0})2^{i-1} = \sum_{i\in[\kappa]} \alpha(s[i])2^{i-1} = \alpha s$
- $z_2 = \sum_{i\in[\kappa]}(d_i - \eta_{i,0})2^{i-1} = \sum_{i\in[\kappa]} f(s)[i]2^{i-1} = f(s)$

- $z_3 = \sum_{i\in[\kappa]}(e_i - \delta_{i,0})2^{i-1} = \sum_{i\in[\kappa]} \alpha(f(s)[i])2^{i-1} = \alpha f(s)$.

This concludes the correctness proof.

Security. Security of $\pi^f_{\text{Non-lin}}$ against any semi-honest adversary $\mathcal{A}$ controlling the server P_0 is immediate from the protocol description. This is because P_0 gets no output from OT^κ_λ and receives no message from P_1. Now, we prove security against any malicious adversary $\mathcal{A}$ controlling P_1.

Claim 1. *Let* $(\text{Garble}, \text{GCEval})$ *be a garbling scheme with the properties defined in Section 2.2.3. Then, in the* OT^κ_λ*-hybrid model,* $\pi^f_{\text{Non-lin}}$ *is secure against any malicious adversary* $\mathcal{A}$ *corrupting the client* P_1.

Proof. Let Real denote the protocol execution $\pi^f_{\text{Non-lin}}$ between P_0 and an adversarially controlled P_1. We argue simulation based security against a malicious P_1, i.e., we will show that the view of $\mathcal{A}$ in Real is indistinguishable from the view of $\mathcal{A}$ in a simulated execution Sim via a standard hybrid argument. In particular, we define two intermediate hybrid executions Hyb_1 and Hyb_2 and argue indistinguishability of the views of the adversary in consecutive executions.

Hybrid execution Hyb_1: Hyb_1 is identical to Real except in the authentication phase, where $\mathcal{S}$ uses the labels $\widehat{\text{lab}}^{\text{out}}_{i,j}$ (instead of $\text{lab}^{\text{out}}_{i,j}$ used in Real) where $\widehat{\text{lab}}^{\text{out}}_{i,j}$ is defined as follows. Note that $\mathcal{S}$ in Hyb_1 has access to honest P_0's inputs $\langle s\rangle_0$ and α. Define $s = \langle s\rangle_0 + \langle s\rangle_1$. For $t = (s||f(s))$ and $i \in [2\kappa]$, if $j = t[i]$, $\widehat{\text{lab}}^{\text{out}}_{i,j} = \text{lab}^{\text{out}}_{i,j}$, else (the "other" label) $\widehat{\text{lab}}^{\text{out}}_{i,1-t[i]}$ is chosen uniformly from $\{0,1\}^\lambda$ such that the first bit[8] is $1 - p_{i,t[i]}$. The formal description of Hyb_1 is provided below. The indistinguishability of views of $\mathcal{A}$ in Real and Hyb_1 executions directly follows from the authenticity of the garbled circuit (Section 2.2.3).

1. $\mathcal{S}$ receives $\langle s\rangle_1$ from $\mathcal{A}$ as its input to OT^κ_λ.
2. *Garbled Circuit Phase:*
   - $\mathcal{S}$ computes $(\text{GC}, \{\{\text{lab}^{\text{in}}_{i,j}\}_{i\in[2\kappa]}, \{\text{lab}^{\text{out}}_{i,j}\}_{i\in[2\kappa]}\}_{j\in\{0,1\}}) \leftarrow \text{Garble}(1^\lambda, \text{Comp}^f)$ and sends $\{\widetilde{\text{lab}}^{\text{in}}_{i+\kappa} = \text{lab}^{\text{in}}_{i+\kappa,\langle s\rangle_1[i]}\}_{i\in[\kappa]}$ to $\mathcal{A}$ as the output of OT^κ_λ. It also sends GC and $\{\widetilde{\text{lab}}^{\text{in}}_i = \text{lab}^{\text{in}}_{i,\langle s\rangle_0[i]}\}_{i\in[\kappa]}$ to $\mathcal{A}$.
3. *Authentication Phase:*
   - $\mathcal{S}$ sets $t = (s||f(s))$.
   - For $i \in [2\kappa]$, $\mathcal{S}$ sets $\widehat{\text{lab}}^{\text{out}}_{i,t[i]} = \text{lab}^{\text{out}}_{i,t[i]}$.
   - For $i \in [2\kappa]$, $\mathcal{S}$ samples $\widehat{\text{lab}}^{\text{out}}_{i,1-t[i]} \in_R \{0,1\}^\lambda$ such that first bit of $\widehat{\text{lab}}^{\text{out}}_{i,1-t[i]}$ equals $1 - p_{i,t[i]}$
   - $\mathcal{S}$ computes and sends $\{\text{ct}_{i,j}, \hat{\text{ct}}_{i,j}\}_{i\in[\kappa],j\in\{0,1\}}$ to $\mathcal{A}$ using $\{\widehat{\text{lab}}^{\text{out}}_{i,j}\}_{i\in[2\kappa],j\in\{0,1\}}$ (similar to how P_0 computes in the real execution using $\{\text{lab}^{\text{out}}_{i,j}\}_{i\in[2\kappa],j\in\{0,1\}}$).

[8] This restriction on the first bit is done, so as to be consistent with the point and permute optimization. Recall that for any $i \in [2\kappa]$, $p_{i,0} \oplus p_{i,1} = 1$.

Hybrid execution Hyb_2: We make four changes to Hyb_1 to obtain Hyb_2 and argue that the views of the adversary in the two hybrids are identical distributions. Let $\{\widetilde{\text{lab}}^{\text{out}}_i = (\widetilde{p}_i||\widetilde{k}_i)\}_{i\in[2\kappa]} \leftarrow \text{GCEval}(\text{GC}, \{\widetilde{\text{lab}}^{\text{in}}_i\}_{i\in[2\kappa]})$. First, by correctness of garbled circuits, $\{\widetilde{\text{lab}}^{\text{out}}_i = \text{lab}^{\text{out}}_{i,t[i]}\}_{i\in[2\kappa]}$. Second, since, for $i \in [\kappa]$, the ciphertexts $\text{ct}_{i,1-\widetilde{p}_i}$ and $\hat{\text{ct}}_{i,1-\widetilde{p}_{i+\kappa}}$ are computed using the "other" set of output labels (as onetime pads) picked uniformly in Hyb_1, $\mathcal{S}$ can directly sample them uniformly at random. With this change, the randomness $\eta_{i,0}, \delta_{i,0}, \nu_{i,0}$ is only used once to generate one set of ciphertexts. It can be shown that the underlying messages of the ciphertexts $\text{ct}_{i,\widetilde{p}_i}$ and $\hat{\text{ct}}_{i,\widetilde{p}_{i+\kappa}}$, namely, c_i, $d_i||e_i$ are uniformly random subject to the only constraint that $\left(\sum_{i\in[\kappa]} c_i 2^{i-1}\right) = \langle\alpha s\rangle_1$, $\left(\sum_{i\in[\kappa]} d_i 2^{i-1}\right) = \langle f(s)\rangle_1$ and $\left(\sum_{i\in[\kappa]} e_i 2^{i-1}\right) = \langle\alpha f(s)\rangle_1$. Hence, as the third change, $\mathcal{S}$ correctly picks $c_i, d_i||e_i$ from this distribution. Finally, one can observe that $\langle\alpha s\rangle_1, \langle f(s)\rangle_1 \langle\alpha f(s)\rangle_1$ are the outputs of P_1 from the functionality $\mathcal{F}^f_{\text{Non-lin}}$ on input $\langle s\rangle_1$. Hence, as the fourth change, $\mathcal{S}$ obtains these as outputs from $\mathcal{F}^f_{\text{Non-lin}}$. Observe that with these changes, $\mathcal{S}$ no longer needs to know one of the inputs of P_0, i.e., α. A formal description of Hyb_2 follows:

1. $\mathcal{S}$ receives $\langle s\rangle_1$ from $\mathcal{A}$ as its input to OT^κ_λ.
2. *Garbled Circuit Phase:* Same as Hyb_1.
3. *Authentication Phase:* $\mathcal{S}$ uses $(\langle s\rangle_1, \text{GC}, \{\widetilde{\text{lab}}^{\text{in}}_i\}_{i\in[2\kappa]})$ as inputs to this phase.
   - $\mathcal{S}$ runs $\{\widetilde{\text{lab}}^{\text{out}}_i\}_{i\in[2\kappa]} \leftarrow \text{GCEval}(\text{GC}, \{\widetilde{\text{lab}}^{\text{in}}_i\}_{i\in[2\kappa]})$.
   - For each $i \in [2\kappa]$, $\mathcal{S}$ parses $\widetilde{\text{lab}}^{\text{out}}_i$ as $(\widetilde{p}_i||\widetilde{k}_i)$.
   - $\mathcal{S}$ sends $\langle s\rangle_1$ to $\mathcal{F}^f_{\text{Non-lin}}$ to learn $(\langle\alpha s\rangle_1, \langle f(s)\rangle_1, \langle\alpha f(s)\rangle_1)$.
   - For each $i \in [\kappa]$, $\mathcal{S}$ samples c_i, d_i, e_i each uniformly from $\mathbb{F}_p$ such that $\left(\sum_{i\in[\kappa]} c_i 2^{i-1}\right) = \langle\alpha s\rangle_1$, $\left(\sum_{i\in[\kappa]} d_i 2^{i-1}\right) = \langle f(s)\rangle_1$ and $\left(\sum_{i\in[\kappa]} e_i 2^{i-1}\right) = \langle\alpha f(s)\rangle_1$.
   - For each $i \in [\kappa]$, $\mathcal{S}$ computes $\text{ct}_{i,\widetilde{p}_i} = c_i \oplus \text{Trim}_\kappa(\widetilde{k}_i)$ and $\hat{\text{ct}}_{i,\widetilde{p}_{i+\kappa}} = (d_i||e_i) \oplus \text{Trim}_{2\kappa}(\widetilde{k}_{i+\kappa})$, and samples $\text{ct}_{i,1-\widetilde{p}_i}$ and $\hat{\text{ct}}_{i,1-\widetilde{p}_{i+\kappa}}$ uniformly from $\{0,1\}^\kappa$ and $\{0,1\}^{2\kappa}$ resp.
   - $\mathcal{S}$ sends $\{\text{ct}_{i,j}, \hat{\text{ct}}_{i,j}\}_{i\in[\kappa],j\in\{0,1\}}$ to $\mathcal{A}$.

Simulated execution Sim: We remove the dependence on P_0's input $\langle s\rangle_0$ by invoking the simulator of the garbling scheme in the Garbled Circuit Phase. The indistinguishability of Sim and Hyb_2 directly follows by the security of garbled circuits. The formal description of Sim is given below.

1. $\mathcal{S}$ receives $\langle s\rangle_1$ from $\mathcal{A}$ as its input to OT^κ_λ.
2. *Garbled Circuit Phase:*

- $\mathcal{S}$ samples $(\widetilde{\mathsf{GC}}, \{\hat{\mathsf{lab}}_i^{\mathsf{in}}\}_{i\in[2\kappa]}) \leftarrow \mathsf{GCSim}(1^\lambda, \mathsf{Comp}^f)$ and sends $\{\hat{\mathsf{lab}}_i\}_{i\in\{\kappa+1,\cdots,2\kappa\}}$ to $\mathcal{A}$ as the output of $\mathsf{OT}_\lambda^\kappa$. It also sends $\widetilde{\mathsf{GC}}$ and $\{\hat{\mathsf{lab}}_i^{in}\}_{i\in[\kappa]}$ to $\mathcal{A}$.

3. *Authentication Phase:* Same as Hyb_2, where $\mathcal{S}$ uses $(\langle s\rangle_1, \widetilde{\mathsf{GC}}, \{\hat{\mathsf{lab}}_i^{\mathsf{in}}\}_{i\in[2\kappa]})$ as its input in the phase.

$\square$

4 Secure Inference

Neural network inference algorithms typically consist of two types of layers - *linear* and *non-linear*. Linear layers include functions such as matrix multiplications (fully connected layers) and convolutions, while non-linear layers consists of functions such as $\mathsf{ReLU}, \mathsf{ReLU6}$ and $\mathsf{Maxpool}$. In this section, we first discuss how to securely compute the linear layers – the functionality and the protocol to realize it– in Section 4.1. Then, in Section 4.2 we describe our complete protocol for secure inference in the client malicious setting. This protocol, by combining our protocols from Section 4.1 and Section 3.1, together with a consistency check phase, allows for the secure inference of any neural network that uses any combination of linear and non-linear layers. In Section 4.4 we discuss our secure inference protocol in the preprocessing model.

4.1 Linear Layers

We describe the functionality $\mathcal{F}_{\mathsf{Lin}}$ formally in Figure 3 used by our protocol for secure inference in Section 4.2. The functionality can be invoked in two ways. The argument **InitLin** is used for invoking the functionality for the first linear layer of the neural network. **InitLin** is invoked exactly once and takes as input a matrix $\mathbf{M}$ and a MAC key α from P_0 and $\mathbf{x}$ from P_1. It outputs authenticated shares of $\mathbf{Mx}$ to both parties, i.e., shares of $\mathbf{Mx}$ and shares of $\alpha\mathbf{Mx}$. Second, the argument **Lin** is used for all subsequent linear layers in the neural network. **Lin** is invoked on shares of $\mathbf{v}$ and $\mathbf{w}$ from P_0 and P_1 and a matrix $\mathbf{M}$ and a MAC key α from P_0. It outputs shares of $\mathbf{Mv}$, $\mathbf{Mw}$ and $\alpha^3\mathbf{v} - \alpha^2\mathbf{w}$ to both the parties. Looking ahead, in an honest execution, $\mathbf{w} = \alpha\mathbf{v}$ and hence, this functionality takes in authenticated shares of input and produces authenticated shares of output along with $\alpha^3\mathbf{v} - \alpha^2\mathbf{w}$ that would be used later to check that a malicious P_1 indeed fed in correct authenticated shares[9].

We note here that our novelty in linear layers is not how we realize the functionality in Figure 3, but rather defining the functionality itself such that it is both efficiently realizable and allows for cheap consistency checks against a malicious client when we put different pieces together in the overall protocol (Section 4.2).

[9]We discuss the use of higher powers of α in consistency check later.

InitLin: On input $(\mathbf{InitLin}, \mathbf{M}, \alpha)$ from P_0 and $(\mathbf{InitLin}, \mathbf{x})$ from P_1 (where $\mathbf{M} \in \mathbb{F}_p^{m\times n}$, $\alpha \in \mathbb{F}_p$ and $\mathbf{x} \in \mathbb{F}_p^n$), P_b learns $\langle\mathbf{Mx}\rangle_b$ and $\langle\alpha\mathbf{Mx}\rangle_b$ for $b \in \{0,1\}$.

Lin: On input $(\mathbf{Lin}, \langle\mathbf{v}\rangle_0, \langle\mathbf{w}\rangle_0, \mathbf{M}, \alpha)$ from P_0 and $(\mathbf{Lin}, \langle\mathbf{v}\rangle_1, \langle\mathbf{w}\rangle_1)$ from P_1 (where $\mathbf{v}, \mathbf{w} \in \mathbb{F}_p^n$, $\mathbf{M} \in \mathbb{F}_p^{m\times n}$ and $\alpha \in \mathbb{F}_p$), P_b learns $\langle\mathbf{Mv}\rangle_b$, $\langle\mathbf{Mw}\rangle_b$ and $\langle\alpha^3\mathbf{v} - \alpha^2\mathbf{w}\rangle_b$ for $b \in \{0,1\}$.

Figure 3: Functionality $\mathcal{F}_{\mathsf{Lin}}$

Protocol. We realize this functionality using standard techniques relying on any additive homomorphic encryption (Sec 2.2.1) and zero-knowledge proofs. We formally describe the protocol π_{Lin} in Figure 4. Correctness of π_{Lin} follows from inspection; security follows using arguments similar to existing protocols in literature [14, 15, 28, 32].

Remark 4.1 (Convolutions)**.** For ease of exposition, we only considered the case of matrix multiplication (or fully connected layers) in the above discussion. It is easy to see that a similar functionality and a corresponding protocol can be defined for convolutional layers as well, where the communication complexity of the protocol again depends only on the size of the input and output for that layer.

4.2 Neural Network Inference Protocol

For ease of exposition, similar to [32], we consider a neural network NN with ℓ linear layers (specified by $\mathbf{M}_1, \cdots, \mathbf{M}_\ell$) and $\ell - 1$ non-linear layers evaluating the non-linear functions $f_1, \cdots, f_{\ell-1}$, such that linear and non-linear layers alternate and the first layer is a linear layer. Our protocol can naturally also be extended to arbitrary combinations of linear and non-linear functions (Appendix D). Let $\mathbf{x}$ be the input to NN and the output of inference is denoted by $\mathsf{NN}(\mathbf{x})$. Let $\mathbf{s}_i$ denote the (intermediate) inference vector after the i^{th} linear layer evaluation for $i \in [\ell]$ and $\mathbf{v}_i$ denote the (intermediate) inference vector after the i^{th} non-linear layer evaluation for $i \in [\ell-1]$. Note that, $\mathbf{s}_1 = \mathbf{M}_1\mathbf{x}$, for $i \in [\ell-1]$ it holds that $\mathbf{v}_i = f_i(\mathbf{s}_i)$ and $\mathbf{s}_{i+1} = \mathbf{M}_{i+1}\mathbf{v}_i$, and finally, $\mathbf{s}_\ell = \mathsf{NN}(\mathbf{x})$.

In the setting of secure inference, the server's (P_0's) input is weights of all the linear layers, i.e., $\mathbf{M}_1, \cdots, \mathbf{M}_\ell$ and the client (P_1) holds input $\mathbf{x}$. The goal is for the client to learn $\mathsf{NN}(\mathbf{x})$ where the non-linear layers are as above. We describe our protocol π_{Inf} for this setting formally in Figure 5 that is secure against a semi-honest server and a malicious client. Below, we provide a protocol overview.

At a high level, our protocol has two phases: the evaluation phase and the consistency check phase. The evaluation phase evaluates the alternate linear and non-linear layers with appropriate parameters. After the evaluation phase, the server performs a consistency check on the values computed so far.

Realization of **InitLin** in π_{Lin} :
Input: P_0 holds $\mathbf{M} \in \mathbb{F}_p^{m \times n}$ and $\alpha \in \mathbb{F}_p$ and P_1 holds $\mathbf{x} \in \mathbb{F}_p^n$.
Output: P_b learns $\langle \mathbf{Mx} \rangle_b$, $\langle \alpha\mathbf{Mx} \rangle_b$ for $b \in \{0,1\}$.
Protocol:

- P_0 and P_1 (one time) engage in a two-party computation protocol secure against a semi-honest P_0 and malicious P_1 to sample $(\mathsf{pk}, \mathsf{sk})$ for AHE[a] such that P_1 learns $(\mathsf{pk}, \mathsf{sk})$ and P_0 learns pk. Both parties store these for use in this all subsequent calls to π_{Lin} as well.
- P_1 sends the encryption $\mathbf{c}_1 \leftarrow \mathsf{Enc}(\mathsf{pk}, \mathbf{x})$ to P_0 along with a zero-knowledge (ZK) proof of plaintext knowledge of this ciphertext[b].
- P_0 samples $\langle \mathbf{Mx} \rangle_0, \langle \alpha\mathbf{Mx} \rangle_0 \in_R \mathbb{F}_p^m$.
- P_0 homomorphically evaluates and sends to P_1, the ciphertexts $\mathbf{c}_2 \in \mathsf{Enc}_{\mathsf{pk}}(\mathbf{Mx} - \langle \mathbf{Mx} \rangle_0)$ and $\mathbf{c}_3 \in \mathsf{Enc}_{\mathsf{pk}}(\alpha\mathbf{Mx} - \langle \alpha\mathbf{Mx} \rangle_0)$.
- P_1 sets $\langle \mathbf{Mx} \rangle_1 = \mathsf{Dec}_{\mathsf{sk}}(\mathbf{c}_2)$, $\langle \alpha\mathbf{Mx} \rangle_1 = \mathsf{Dec}_{\mathsf{sk}}(\mathbf{c}_3)$.
- P_b outputs $\langle \mathbf{Mx} \rangle_b$, $\langle \alpha\mathbf{Mx} \rangle_b$ for $b \in \{0,1\}$.

Realization of **Lin** in π_{Lin}:
Input: P_0 holds $\langle \mathbf{v} \rangle_0, \langle \mathbf{w} \rangle_0 \in \mathbb{F}_p^n$, $\mathbf{M} \in \mathbb{F}_p^{m \times n}$ and $\alpha \in \mathbb{F}_p$. P_1 holds $\langle \mathbf{v} \rangle_1, \langle \mathbf{w} \rangle_1 \in \mathbb{F}_p^n$.
Output: P_b learns $\langle \mathbf{Mv} \rangle_b$, $\langle \mathbf{Mw} \rangle_b$ and $\langle \alpha^3\mathbf{v} - \alpha^2\mathbf{w} \rangle_b$ for $b \in \{0,1\}$.
Protocol:

- P_1 sends the encryptions $\mathbf{e}_1 \leftarrow \mathsf{Enc}(\mathsf{pk}, \langle \mathbf{v} \rangle_1)$, $\mathbf{e}_2 \leftarrow \mathsf{Enc}(\mathsf{pk}, \langle \mathbf{w} \rangle_1)$ to P_0 and a ZK proof of plaintext knowledge for both.
- P_0 samples $\langle \mathbf{Mv} \rangle_0, \langle \mathbf{Mw} \rangle_0 \in_R \mathbb{F}_p^m$ and $\langle \alpha^3\mathbf{v} - \alpha^2\mathbf{w} \rangle_0 \in_R \mathbb{F}_p^n$.
- P_0 homomorphically evaluates and sends to P_1, the ciphertexts $\mathbf{e}_3 \in \mathsf{Enc}_{\mathsf{pk}}(\mathbf{Mv} - \langle \mathbf{Mv} \rangle_0)$, $\mathbf{e}_4 \in \mathsf{Enc}_{\mathsf{pk}}(\mathbf{Mw} - \langle \mathbf{Mw} \rangle_0)$ and $\mathbf{e}_5 \in \mathsf{Enc}_{\mathsf{pk}}(\alpha^3\mathbf{v} - \alpha^2\mathbf{w} - \langle \alpha^3\mathbf{v} - \alpha^2\mathbf{w} \rangle_0)$.
- P_1 sets $\langle \mathbf{Mv} \rangle_1 = \mathsf{Dec}_{\mathsf{sk}}(\mathbf{e}_3)$, $\langle \mathbf{Mw} \rangle_1 = \mathsf{Dec}_{\mathsf{sk}}(\mathbf{e}_4)$ and $\langle \alpha^3\mathbf{v} - \alpha^2\mathbf{w} \rangle_1 = \mathsf{Dec}_{\mathsf{sk}}(\mathbf{e}_5)$.
- P_b outputs $\langle \mathbf{Mv} \rangle_b$, $\langle \mathbf{Mw} \rangle_b$ and $\langle \alpha^3\mathbf{v} - \alpha^2\mathbf{w} \rangle_b$ for $b \in \{0,1\}$.

[a] Function privacy of AHE holds only for honestly generated keys.
[b] ZK proof of knowledge for the statement that c is a valid sample from $\mathsf{Enc}_{\mathsf{pk}}(m)$ for an m known to the prover. We refer the reader to [28, 32] for more details.

Figure 4: Protocol π_{Lin}

If the check passes, output is revealed to the client. In more detail, P_0 begins by sampling a MAC key $\alpha \in \mathbb{F}_p$ uniformly that will be used to authenticate all intermediate values. During the protocol, the two parties will hold authenticated shares of all intermediate values where authentications are generated using α along with some additional values generated to aid in consistency checks.

- **Linear Layer Evaluation:** To evaluate the first linear layer, P_0 and P_1 invoke $\mathcal{F}_{\mathsf{Lin}}$ with inputs $(\mathbf{InitLin}, \mathbf{M}_1, \alpha)$ and $(\mathbf{InitLin}, \mathbf{x})$ respectively. They learn authenticated shares of $\mathbf{s}_1$, i.e., shares of $\mathbf{s}_1, \mathbf{t}_1$, where $\mathbf{t}_1$ is authentication on $\mathbf{s}_1$. For evaluation of the i^{th} linear layer $(i > 1)$, P_0 and P_1 invoke $\mathcal{F}_{\mathsf{Lin}}$ with input **Lin** and authenticated shares of the output of the previous non-linear layer, i.e $\mathbf{v}_{i-1}$ and $(\mathbf{M}_i, \alpha)$ from P_0. We denote authentication on $\mathbf{v}_{i-1}$ with $\mathbf{w}_{i-1}$. **Lin** outputs shares of $\mathbf{s}_i = \mathbf{M}_i\mathbf{v}_{i-1}$ and $\mathbf{t}_i = \mathbf{M}_i\mathbf{w}_{i-1}$ and an additional *"tag"* $\mathbf{z}_i = (\alpha^3\mathbf{v}_i - \alpha^2\mathbf{w}_{i-1})$ (that is 0 numerically whenever the inputs to **Lin** were infact authenticated shares, i.e., if $\alpha\mathbf{v}_{i-1} = \mathbf{w}_{i-1}$ and non-zero otherwise (when $\alpha \neq 0$)).
- **Non-linear Layer Evaluation:** To evaluate the i^{th} non-linear layer for $i \in [\ell - 1]$, P_0 and P_1 invoke $\mathcal{F}_{\mathsf{Non\text{-}lin}}^{f_i}$ on shares of $\mathbf{s}_i$ and the input α from P_0 to learn authenticated shares of $\mathbf{v}_i = f(\mathbf{s}_i)$ and *another* set of shares of authentication on $\mathbf{s}_i$, denoted by $\mathbf{u}_i$.
- **Consistency Check Phase:** The server performs the following two sets of checks.
  - For each $i \in \{2, \cdots, \ell\}$, check that the pair of shares input to **Lin** are valid authenticated shares under α by verifying that $\mathbf{z}_i = 0^{n_{i-1}}$.
  - For each $i \in [\ell - 1]$, check that the shares input to $\mathcal{F}_{\mathsf{Non\text{-}lin}}^{f_i}$ were same as the shares output by **Lin** on i^{th} linear layer by verifying if $\mathbf{t}_i - \mathbf{u}_i = 0^{n_i}$.

  Finally, all the above checks can be combined into a single check by using random scalars picked by P_0. If the check fails, P_0 aborts, and sends its final share to P_1 otherwise, who can reconstruct the output.

Remark 4.2. Our protocol works over $\mathbb{F}_p$ that represents fixed-point numbers. We use fixed-point computation to emulate computation on real numbers. Moreover, to maintain precision, we need to truncate intermediate values and our protocol can do this for free within garbled circuits.

4.3 Correctness and Security

Theorem 2. *The protocol π_{Inf} securely realizes the functionality $\mathcal{F}_{\mathsf{Inf}}$ in the $\mathcal{F}$-hybrid model where $\mathcal{F} = (\mathcal{F}_{\mathsf{Lin}}, \mathcal{F}_{\mathsf{Non\text{-}lin}}^{f_1}, \cdots, \mathcal{F}_{\mathsf{Non\text{-}lin}}^{f_{\ell-1}})$ against a semi-honest server P_0 and a malicious client P_1 with probability atleast $1 - 6/p$.*

Proof. **Correctness**. By correctness of $\mathcal{F}_{\mathsf{Lin}}$ on **InitLin**, we have $\mathbf{s}_1 = \mathbf{M}_1\mathbf{x}$ and $\mathbf{t}_1 = \alpha\mathbf{s}_1$. By correctness of $\mathcal{F}_{\mathsf{Lin}}$ on **Lin**, for each $i \in \{2, \cdots, \ell\}$ it holds that $\mathbf{s}_i = \mathbf{M}_i\mathbf{v}_{i-1}$, $\mathbf{t}_i = \mathbf{M}_i\mathbf{w}_{i-1}$ and $\mathbf{z}_i = \alpha^3\mathbf{v}_{i-1} - \alpha^2\mathbf{w}_{i-1}$. By correctness of $\mathcal{F}_{\mathsf{Non\text{-}lin}}^{f_i}$, for each $i \in [\ell - 1]$ it follows that $\mathbf{u}_i = \alpha\mathbf{s}_i$, $\mathbf{v}_i = f_i(\mathbf{s}_i)$ and $\mathbf{w}_i = \alpha f_i(\mathbf{s}_i)$. On substituting, it is easy to see that $q = 0$ since for each

Preamble. A neural network NN with ℓ linear and $\ell-1$ non-linear layers. Let $f_1,\ldots,f_{\ell-1}$ be the elementary functions that need to be computed in $\ell-1$ non-linear layers.
Input: P_0 holds $\{\mathbf{M}_j \in \mathbb{F}_p^{n_j\times n_{j-1}}\}_{j\in[\ell]}$, i.e., weights for the ℓ linear layers. P_1 holds the input $\mathbf{x}\in\mathbb{F}_p^{n_0}$ for NN.
Output: P_1 learns $\mathsf{NN}(\mathbf{x})$.

1. P_0 samples MAC key α uniformly from $\mathbb{F}_p$ to be used throughout the protocol.
2. **First Linear Layer:** P_0 and P_1 invoke $\mathcal{F}_{\mathsf{Lin}}$ with inputs $(\mathbf{InitLin},\mathbf{M}_1,\alpha)$ and $(\mathbf{InitLin},\mathbf{x})$ respectively. For $b\in\{0,1\}$, P_b learns $(\langle\mathbf{s}_1\rangle_b,\langle\mathbf{t}_1\rangle_b)$.
3. For each $j\in[\ell-1]$,

   Non-linear Layer f_j: P_0 and P_1 invoke $\mathcal{F}_{\mathsf{Non\text{-}lin}}^{f_j}$ with inputs $(\langle\mathbf{s}_j\rangle_0,\alpha)$ and $\langle\mathbf{s}_j\rangle_1$ respectively. For $b\in\{0,1\}$, P_b learns $(\langle\mathbf{u}_j\rangle_b,\langle\mathbf{v}_j\rangle_b,\langle\mathbf{w}_j\rangle_b)$.

   Linear Layer $j+1$: P_0 and P_1 invoke $\mathcal{F}_{\mathsf{Lin}}$ with inputs $(\mathbf{Lin},\langle\mathbf{v}_j\rangle_0,\langle\mathbf{w}_j\rangle_0,\mathbf{M}_{j+1},\alpha)$ and $(\mathbf{Lin},\langle\mathbf{v}_j\rangle_1,\langle\mathbf{w}_j\rangle_1)$ respectively. For $b\in\{0,1\}$, P_b learns $(\langle\mathbf{s}_{j+1}\rangle_b,\langle\mathbf{t}_{j+1}\rangle_b,\langle\mathbf{z}_{j+1}\rangle_b)$.
4. **Consistency Check:**
   - For $j\in[\ell-1]$, P_0 samples $\mathbf{r}_j\in_R\mathbb{F}_p^{n_j}$ and $\mathbf{r}'_{j+1}\in_R\mathbb{F}_p^{n_{j+1}}$ and sends $(\mathbf{r}_j,\mathbf{r}'_{j+1})$ to P_1.
   - P_1 computes $\langle q\rangle_1=\sum_{j\in[\ell-1]}\Big((\langle\mathbf{t}_j\rangle_1-\langle\mathbf{u}_j\rangle_1)*\mathbf{r}_j+\langle\mathbf{z}_{j+1}\rangle_1*\mathbf{r}'_{j+1}\Big)$ and sends it to P_0.
   - P_0 computes $\langle q\rangle_0=\sum_{j\in[\ell-1]}\Big((\langle\mathbf{t}_j\rangle_0-\langle\mathbf{u}_j\rangle_0)*\mathbf{r}_j+\langle\mathbf{z}_{j+1}\rangle_0*\mathbf{r}'_{j+1}\Big)$.
   - P_0 aborts if $\langle q\rangle_0+\langle q\rangle_1 \mod p\neq 0$. Else, sends $\langle\mathbf{s}_\ell\rangle_0$ to P_1.
5. **Output Phase:** P_1 outputs $\langle\mathbf{s}_\ell\rangle_0+\langle\mathbf{s}_\ell\rangle_1 \mod p$ if P_0 didn't abort in the previous step.

Figure 5: Secure Inference Protocol π_{Inf}

$i\in[\ell-1]$, $\mathbf{z}_{i+1}=0$, $\mathbf{t}_i=\mathbf{u}_i$. Finally, we conclude correctness by noting that $\mathbf{s}_\ell=\mathsf{NN}(\mathbf{x})$.

Security. We prove that our protocol is secure against a semi-honest server P_0 and a malicious client P_1 using simulation based security. It is easy to see security against a semi-honest adversary corrupting P_0 as follows: During the evaluation phase of the protocol, P_0 only learns one of the shares as output from $\mathcal{F}_{\mathsf{Lin}}/\mathcal{F}_{\mathsf{Non\text{-}lin}}^{f}$ and hence, can be simulated easily by picking uniformly random values from the field. In the consistency check phase, it learns $\langle q\rangle_1$. This value is easy to simulate using $q=0$ and $\langle q\rangle_0$ that can be locally computed. Now, we prove security against a malicious client in the following lemma.

Lemma 1. *The protocol π_{Inf} is secure against a malicious adversary $\mathcal{A}$ controlling the client P_1 in the $\mathcal{F}$-hybrid model where $\mathcal{F}=(\mathcal{F}_{\mathsf{Lin}},\mathcal{F}_{\mathsf{Non\text{-}lin}}^{f_1},\cdots,\mathcal{F}_{\mathsf{Non\text{-}lin}}^{f_{\ell-1}})$.*

Proof. Recall that a malicious $\mathcal{A}$ controlling P_1 can arbitrarily deviate from the protocol specification. Formally, w.r.t. our protocol π_{Inf} that invokes various ideal functionalities, $\mathcal{A}$ can send inconsistent inputs. In particular, $\mathcal{A}$ can do the following: (a) Invoke $\mathcal{F}_{\mathsf{Lin}}(\mathbf{InitLin},\cdot)$ on $\mathbf{x}'\neq\mathbf{x}$ (client's original input), and learns $\langle\mathbf{s}_1\rangle_1$ and $\langle\mathbf{t}_1\rangle_1$; (b) For $i\in[\ell-1]$, $\mathcal{A}$ can invoke $\mathcal{F}_{\mathsf{Non\text{-}lin}}^{f_1}$ on input $\langle\mathbf{s}'_i\rangle_1=\langle\mathbf{s}_i\rangle_1+\Delta_i^1$ to learn $\langle\mathbf{u}_i\rangle_1,\langle\mathbf{v}_i\rangle_1,\langle\mathbf{w}_i\rangle_1$; (c) For $i\in[\ell-1]$, $\mathcal{A}$ can invoke **Lin** on input $\langle\mathbf{v}'_i\rangle_1=\langle\mathbf{v}_i\rangle_1+\Delta_i^2$ and $\langle\mathbf{w}'_i\rangle_1=\langle\mathbf{w}_i\rangle_1+\Delta_i^3$ to learn $\langle\mathbf{s}_{i+1}\rangle_1,\langle\mathbf{t}_{i+1}\rangle_1,\langle\mathbf{z}_{i+1}\rangle_1$; (d) $\mathcal{A}$ can add an error Δ^4 to his share of q and send this errorneous share of q to P_0. Using the above notation, $\mathcal{A}$ behaves honestly and follows the specification if and only if all Δ's are 0. We formally describe the simulator $\mathcal{S}$ simulating the view of malicious $\mathcal{A}$ in Figure 6.

Now, there are two cases to analyse: In the first case, when $\mathcal{A}$ follows the protocol specification, it is easy to see that the views in the real execution of the protocol and simulated execution are identical. In the case, when $\mathcal{A}$ deviates from the protocol execution, i.e., there exists a non-zero Δ, then the simulator sends abort to $\mathcal{A}$ with probability 1. Moreover, it is easy to see, by inspection, that up until the end of the consistency check phase, the views of $\mathcal{A}$ in real and simulated executions are identical and consist of uniformly distributed field elements. Hence, it suffices to argue that in the case of non-zero Δ, P_0 aborts in real execution with all but exponentially low probability (in κ).

Claim 2. *In real execution, if at least one of the Δ's is non-zero, then P_0 aborts with probability at least $1-6/p$.*

Proof. Below q is the value that P_0 reconstructs in the consistency check phase of the real execution. Using notation from above, we have that

$$q=\Delta^4+\sum_{j=1}^{\ell-1}((\mathbf{t}_j-\mathbf{u}_j)*\mathbf{r}_j+\mathbf{z}_{j+1}*\mathbf{r}'_{j+1}) \qquad (1)$$

- **Preamble**: $\mathcal{S}$ interacts with $\mathcal{A}$ controlling P_1 with input $\mathbf{x}$. $\mathcal{S}$ sets a flag bit flag $= 0$.
- **First Linear Layer:** $\mathcal{A}$ invokes $\mathcal{F}_{\mathsf{Lin}}$ on input $(\mathbf{InitLin}, \mathbf{x}')$. $\mathcal{S}$ sends uniform $\langle \mathbf{s}_1 \rangle_1$ and $\langle \mathbf{t}_1 \rangle_1$ to $\mathcal{A}$.
- For $j \in [\ell - 1]$,

  Non-linear Layer f_j**:** $\mathcal{A}$ invokes $\mathcal{F}^{f_j}_{\mathsf{Non\text{-}lin}}$ on input $\langle \mathbf{s}_j{}' \rangle_1 = \langle \mathbf{s}_j \rangle_1 + \Delta^1_j$. $\mathcal{S}$ sends uniform $\langle \mathbf{u}_j \rangle_1, \langle \mathbf{v}_j \rangle_1$ and $\langle \mathbf{w}_j \rangle_1$ to $\mathcal{A}$. Additionally, it sets flag $= 1$, if $\Delta^1_j \neq 0^{n_j}$.

  Linear Layer $j+1$: $\mathcal{A}$ invokes $\mathcal{F}_{\mathsf{Lin}}$ on inputs **Lin**, $\langle \mathbf{v}_j{}' \rangle_1 = \langle \mathbf{v}_j \rangle_1 + \Delta^2_j$ and $\langle \mathbf{w}_j{}' \rangle_1 = \langle \mathbf{w}_j \rangle_1 + \Delta^3_j$. $\mathcal{S}$ sends uniform $\langle \mathbf{s}_j \rangle_1$,$\langle \mathbf{t}_j \rangle_1$ and $\langle \mathbf{z}_j \rangle_1$ to $\mathcal{A}$. Additionally, $\mathcal{S}$ sets flag $= 1$ if $(\Delta^2_j, \Delta^3_j) \neq (0^{n_j}, 0^{n_j})$.
- **Consistency Check:**
  - For $j \in [\ell - 1]$, $\mathcal{S}$ samples $\mathbf{r}_j \in_R \mathbb{F}^{n_j}_p$ and $\mathbf{r}'_{j+1} \in_R \mathbb{F}^{n_{j+1}}_p$ and sends $(\mathbf{r}_j, \mathbf{r}'_{j+1})$ to $\mathcal{A}$.
  - $\mathcal{A}$ sends $\langle q \rangle_1 = \Delta^4 + \sum_{j \in [\ell-1]} (\langle \mathbf{z}_{j+1} \rangle_1 * \mathbf{r}'_{j+1} + (\langle \mathbf{t}_j \rangle_1 - \langle \mathbf{u}_j \rangle_1) * \mathbf{r}_j)$. $\mathcal{S}$ sets flag $= 1$ if $\Delta^4 \neq 0$.
- **Output Phase:** If flag $= 0$, $\mathcal{S}$ queries $\mathcal{F}_{\mathsf{Inf}}$ on $\mathbf{x}'$ to learn $\mathsf{NN}(\mathbf{x}')$ and sends $\mathsf{NN}(\mathbf{x}') - \langle \mathbf{s}_\ell \rangle_1$ to $\mathcal{A}$. Else, it sends abort to both $\mathcal{F}_{\mathsf{Inf}}$ and $\mathcal{A}$.

Figure 6: Simulator against malicious client corresponding to π_{Inf}

Further, from the notation (of Δ's) above and by correctness of the functionalities invoked in π_{Inf}, we have that for each $j \in [\ell-1]$, $\mathbf{w}_j = \alpha \mathbf{v}_j$ and $\mathbf{z}_{j+1} = (\alpha^3(\mathbf{v}_j + \Delta^2_j) - \alpha^2(\mathbf{w}_j + \Delta^3_j))$. Similarly, $\mathbf{t}_1 = \alpha \mathbf{M}_1 \mathbf{x}'$, $\mathbf{u}_1 = \alpha(\mathbf{M}_1 \mathbf{x}' + \Delta^1_1)$ and for each $j \in \{2, \cdots, \ell-1\}$, $\mathbf{t}_j = \mathbf{M}_j(\mathbf{w}_{j-1} + \Delta^3_{j-1})$ and $\mathbf{u}_j = \alpha(\mathbf{M}_j(\mathbf{v}_{j-1} + \Delta^2_{j-1}) + \Delta^1_j)$. On substitution in Equation 1, we get

$$q = \alpha^3 (\sum_{j=1}^{\ell-1} \Delta^2_j * \mathbf{r}'_{j+1}) - \alpha^2 (\sum_{j=1}^{\ell-1} \Delta^3_j * \mathbf{r}'_{j+1})$$
$$- \alpha((\sum_{j=1}^{\ell-1} \Delta^1_j + \sum_{j=2}^{\ell-1} \mathbf{M}_j \Delta^2_{j-1}) * \mathbf{r}_j) + \Delta^4 + \sum_{j=2}^{\ell-1} ((\mathbf{M}_j \Delta^3_{j-1}) * \mathbf{r}_j)$$

The RHS of the above equation is a degree-3 polynomial in α, denoted by $Q(\alpha)$. We argue that $Q(\alpha)$ is a non-zero polynomial whenever $\mathcal{A}$ introduces errors, that is, at least one of Δ's is non-zero. This is because, if either $\Delta^4 \neq 0$ or Δ^i_j is a non-zero vector for any $j \in [\ell-1]$ and $i \in [3]$ atleast one of the coefficients of $Q(\alpha)$ will be non-zero, with probability atleast $1 - 3/p$ over the choice of $\mathbf{r}'_{j+1}$ and $\mathbf{r}_j$ for $j \in [\ell-1]$. Further when $Q(\alpha)$ is a non-zero polynomial, it has at most 3 roots. Hence, over the choice of α, the probability that $Q(\alpha) = 0$ is atmost $3/p$. Therefore, the probability that P_0 aborts is atleast $1 - 6/p$ when $\mathcal{A}$ cheats. ◻

Setting Fieldsize. We choose p to be greater than $2^{\sigma+3}$ for π_{Inf} to be σ-bit statistically secure.

4.4 Secure Inference with Preprocessing

MUSE (similar to DELPHI) considers client input-independent preprocessing model and showed how majority of their cryptographic cost ($> 99\%$) can be pushed to the offline phase. We now briefly outline how our techniques can be extended to obtain a client malicious secure inference protocol in this preprocessing model whose online cost is exactly the same as in MUSE. For concrete communication and runtime, see Section 5.4. We provide the outline for $\pi^{\mathsf{Prep}}_{\mathsf{Inf}}$ below and present the full description in Figure 8 in Appendix C.

We incorporate the novel ideas for non-linear layers and consistency check into the protocol structure of DELPHI/MUSE. At a high level, in the preprocessing phase, parties compute the linear layers on random inputs chosen by the client, and the non-linear layers are set up so that the output share of the client matches the random input chosen for next linear layer. Then, the server adjusts its share of output from linear layer based on output of previous non-linear layer, and preprocessing information. Implementing this, requires additional computation in the garbled circuit as explained below, and that is exactly the additional compute/communication compared to our previous protocol.

In the preprocessing phase, for every linear layer, P_0 holds input $\mathbf{M}_i$ and P_1 picks a uniformly random vector $\mathbf{c}_i$ (of appropriate dimensions). Now, P_0 and P_1 securely compute authenticated shares of $\mathbf{M}_i \mathbf{c}_i$ and shares of authentication on $\mathbf{c}_i$. The online phase for computing linear layers is identical to MUSE (and only involves P_1 sending a share of its input to P_0, followed by local non-cryptographic computations at both ends). For each non-linear layer that computes a function f_i, define a circuit C^{f_i} that takes in shares of $\mathbf{s}_i$, and two random masks $\mathbf{d}_i$ from P_0 and $\mathbf{c}_{i+1}$ from P_1. C^{f_i} outputs $(f_i(\mathbf{s}_i) - \mathbf{d}_i - \mathbf{c}_{i+1}$, $\mathbf{s}_i$, $\mathbf{c}_{i+1})$. The garbled circuit for C^{f_i} and labels for $\mathbf{d}_i$ are sent in the preprocessing phase along with OT computation to transfer labels for $\mathbf{c}_{i+1}$ and client's share of $\mathbf{s}_i$. The labels for P_0's share of $\mathbf{s}_i$ are sent in the online phase. Similar to our protocol $\pi^f_{\mathsf{Non\text{-}lin}}$, we use the output labels of the garbled circuit to generate shares of re-authentications on $\mathbf{s}_i, \mathbf{c}_{i+1}$ and check for consistency in the last phase.

We note that the overall cost of non-linear layers increases by

$< 2\times$ and the cost of linear layers decreases compared to the previous protocol.

5 Implementation and Evaluation

We implement SIMC[10] and empirically evaluate its performance. Since the cost of non-linear layers dominate in MUSE [32] ($> 80\%$) and the cost of linear layers in SIMC is similar, our evaluation focusses on three main questions:

- Section 5.2: What are the communication and runtime costs of SIMC on non-linear layers such as ReLU and how do these compare with MUSE?
- Section 5.3: How does SIMC compare with MUSE on end-to-end secure inference tasks?
- Section 5.4: How does SIMC in preprocessing model (denoted by SIMC++) compare with MUSE?

We show that SIMC outperforms MUSE on all parameters. Below, we first discuss our implementation details and evaluation setup followed by evaluation results.

5.1 Implementation and System Setup

Implementation. SIMC is implemented in about 9000 lines of C++ code and provides 128-bit computational security and 40-bit statistical security. Similar to MUSE, our system is implemented over a 44-bit prime. We use the Seal homomorphic encryption library [48] for implementing the AHE scheme for linear layers, and the EMP toolkit [52] for garbled circuits[11] used in non-linear layers. We use AES as a pseudorandom function to generate the randomness $\mathbf{r}_j$ and $\mathbf{r}'_{j+1}$ used in consistency check phase. SHA-256 is used as the hash function to hash all labels output from checking validity of client's inputs in the non-linear protocol (see Remark 3.3). Zero-knowledge (ZK) proofs of plaintext knowledge [11] required in the linear layers, is not implemented in our system due to the lack of a publicly available implementation. As the end-to-end performance analysis of linear layers is somewhat orthogonal to our work, we estimate the performance of these proofs based on [11, Table 1] (see Section 5.3 for more details). MUSE [32] estimates the cost of ZK proofs in their system in a similar way using MP-SPDZ [25]. Finally, to evaluate the performance of SIMC on end-to-end secure inference tasks, we time all individual components of our protocol (linear layers, non-linear layers, and consistency check phase) separately and aggregate them to obtain end-to-end execution times. As there is no cost incurred in connecting the individual components of our protocol, the end-to-end execution time, obtained as described above, provides an accurate estimate.

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

[11]EMP toolkit uses actively secure COTs to send client's labels.

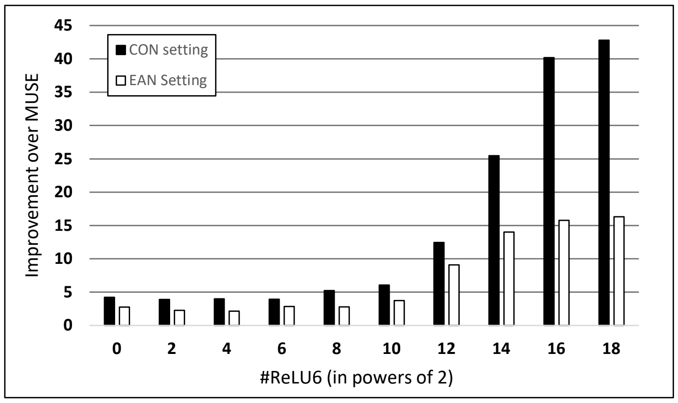

Figure 7: Improvement of SIMC over MUSE as a function of number of ReLU6 instances. The y-axis shows $\frac{\text{MUSE Time}}{\text{SIMC Time}}$.

Evaluation Setup. We carry out our experiments in two network configurations with varying bandwidth and ping latency: In the CON setting, we use the same setup as in MUSE. P_0 and P_1 are two AWS c5.9xlarge instances with Intel Xeon 8000 series CPUs at 3.6GHz running 8 threads each, located in the us-west-1 (Northern California) and us-west-2 (Oregon) regions respectively. In this configuration, the measured bandwidth between the two instances was 584 MBps with 21 ms rtt. For the EAN setting, we use the same machine configuration, but with the machines located in us-west-1 (Northern California) and eu-west-2 (London) respectively. Here, the measured bandwidth was 19.2 MBps with 144 ms rtt. To compare SIMC with MUSE, we use the code available at [31]; all numbers reported are the median values over 5 executions.

5.2 Non-linear Layers performance

We compare communication and latency of SIMC and MUSE for non-linear layers. As representative examples, we choose 2 popular non-linear activation functions – ReLU defined to be $\text{ReLU}(x) = \max(x,0)$; and $\text{ReLU6} := \min(\max(x,0),6)$ – to run our microbenchmarks. While MUSE requires 388 KB and 405 KB of communication to securely compute one instance of ReLU and ReLU6 respectively, SIMC only communicates 11.9 KB and 14.66 KB respectively; thus, SIMC communicates $28-33\times$ less than MUSE.

In typical neural networks, each non-linear layer requires computing large number of instances of the same function, e.g., ReLU6. Hence, we set up our microbenchmarks to study how the performance of SIMC compares with MUSE as the number of instances grow from 1 to 2^{18}. We depict speedups of SIMC over MUSE for the ReLU6 function[12] in the CON and EAN settings in Figure 7. As discussed, SIMC gets rid of

[12]The graph for ReLU looks nearly identical and is omitted here.

	Benchmark A		Benchmark B	
Protocol	MUSE	SIMC	MUSE	SIMC
Linear Layer	0.04	0.05	0.07	0.14
Non-linear Layer	4.14	0.133	67.83	2.24
Total	4.18	0.18	67.90	2.38

Table 1: Communication (in GB) of MUSE and SIMC on benchmarks A and B.

	Benchmark A		Benchmark B	
Protocol	MUSE	SIMC	MUSE	SIMC
Linear Layer	4.40	4.60	40.40	28.30
Non-linear Layer	18.33	0.71	230.02	5.07
Total	22.73	5.31	270.42	33.37

Table 2: Latency (in seconds) of MUSE and SIMC on benchmarks A and B [CON setting].

computationally expensive homomorphic encryption operations from non-linear layers, while reducing communication and this results in significant performance improvements that grow with the number of instances being computed. We observe that SIMC outperforms MUSE by $4-42\times$ in our CON setting and by $2-16.3\times$ in our EAN setting. For instance, MUSE took 313.3s and 2567.7s to compute 2^{18} ReLU6 instances in CON and EAN setting. Whereas, SIMC took just 7.3s and 157.6s in the respective network settings. Furthermore, for 2^{20} instances, the MUSE protocol runs out of memory and crashes, while SIMC computes it in 28.5s in CON and 623.1s EAN settings. Such wide non-linear layers are omnipresent in real-world ML models; ResNet50 on ImageNet dataset has non-linear layers with width upto 2^{20} nodes.

	Benchmark A		Benchmark B	
Protocol	MUSE	SIMC	MUSE	SIMC
Linear Layer	9.40	9.70	54.46	46.43
Non-linear Layer	111.69	9.73	1728.46	110.79
Total	121.09	19.43	1782.92	157.22

Table 3: Latency (in seconds) of MUSE and SIMC on benchmarks A and B [EAN setting].

5.3 Secure Inference Performance

We evaluate SIMC on both neural network architectures considered in MUSE: Benchmark A, 2-layer convolutional neural network (CNN) trained on the MNIST dataset from [34] and Benchmark B, 7-layer CNN architecture for CIFAR-10 provided in [34]. For details on networks, see [32, 34].

As outlined earlier, we implement the components of the protocol (linear layers, non-linear layers, and consistency check) separately and aggregate them to obtain performance estimates. Further, the cost of zero-knowledge proofs required in the linear layers are simulated based on [11, Table 1]. For Benchmark A, the dimensions of input ciphertexts to all the linear-layers are tiny (and in particular much smaller than even the smallest benchmark considered in [11]). Hence, one can accurately estimate that the overhead of zero-knowledge proofs in this benchmark is insignificant ($<$ 100 ms). In Benchmark B, there are 4 layers that require zero-knowledge proofs on sufficiently large vectors – namely linear layers 2 to 5, in which the dimensions of the vectors are 65536, 16384, 16384, and 16384 respectively. Hence, we can estimate that these proofs together would add approximately 1 second to the overall latency (which is still a tiny fraction compared to the overall cost). Since the bulk of the overhead in zero-knowledge proofs is in compute, the above estimates hold true for CON as well as EAN setting.

We compare communication followed by execution time in the two network settings. Table 1 reports total communication as well as the split of communication between linear and non-linear layers of these benchmarks for both MUSE and SIMC. We observe that the total communication of SIMC is $23\times$ and $29\times$ less than that of MUSE for benchmarks A and B respectively. Moreover, SIMC communicates between $21-41\%$ less than even DELPHI[13] [35], that provides only semi-honest security, on the same benchmarks (see Remark 3.4). While linear layers for MUSE and SIMC communicate similar amounts, non-linear layers in SIMC require $\approx 30\times$ lower communication than MUSE for both benchmarks.

Table 2 and Table 3 show the performance of SIMC and MUSE on the two benchmarks in our CON and EAN settings, respectively. As is clear from the split of execution times for MUSE, non-linear layers were the major performance bottleneck (upto 96% of total time). Since our protocol for non-linear layers is significantly lighter weight compared to MUSE, non-linear layers in SIMC outperform those in MUSE by $25.8-45.3\times$ and $11.4-15.7\times$ in the CON and EAN settings, respectively. As discussed in Section 1.2, SIMC significantly improves upon MUSE in both the compute as well communication, leading to high gains in both the high bandwidth setting as well as the low bandwidth settings, where the bottlenecks for MUSE are compute and communication, respectively. Optimizing the performance bottleneck in MUSE, results in significant performance gains for the end-to-end secure inference task as well. In total execution time, SIMC outperforms MUSE by $4.3-8.1\times$ in the CON setting[14] and by $6.2-11.3\times$ in the EAN setting.

5.4 Performance in Preprocessing model

We compare the performance of our protocol in the preprocessing model (SIMC++) with MUSE, on the same benchmarks A and B. As is expected, SIMC++ retains the improvements of SIMC over MUSE. In the preprocessing phase, SIMC++ is $15\times$ and $17\times$ communication frugal compared to MUSE for benchmarks A and B respectively (see Table 4).

[13] We use the communication numbers and runtime of DELPHI in CON setting from [32] as it is identical system setting.

[14] SIMC outperforms DELPHI by $40-82\%$ in the same setting.

	Benchmark A		Benchmark B	
Protocol	MUSE	SIMC++	MUSE	SIMC++
Preprocessing Phase	4.17	0.28	67.67	3.98
Online Phase	0.01	0.01	0.23	0.23
Total	4.18	0.29	67.90	4.21

Table 4: Communication (in GB) of MUSE and SIMC++ on benchmarks A and B in the preprocessing model.

	Benchmark A		Benchmark B	
Protocol	MUSE	SIMC++	MUSE	SIMC++
Preprocessing Phase	21.93	5.51	263.44	36.90
Online Phase	0.8	0.8	7.86	7.86
Total	22.73	6.31	270.42	44.76

Table 5: Latency (in seconds) of MUSE and SIMC++ on benchmarks A and B in preprocessing model [CON setting].

SIMC++'s preprocessing phase is $4-7\times$ and $5-7.4\times$ more performant than MUSE in the CON and EAN setting (see Table 5 and Table 6). As discussed in Section 4.4, our online phase is identical to MUSE, which is reflected from the performance numbers of the two protocols in the online phase.

The communication of SIMC++ is upto $1.8\times$ of SIMC on benchmarks A and B. Compared to SIMC, the end-to-end execution time of SIMC++ is slower by at most $1.3\times$ and $1.7\times$ in CON and EAN setting respectively. This slight increase in overhead comes with the benefit of obtaining an online phase with little cryptographic overhead.

Comparison with other works. We have so far seen that SIMC++ outperforms MUSE by an order of magnitude. We now compare SIMC++ with other related works – namely maliciously-secure Overdrive [28] and the client-malicious version of Overdrive presented in [32], which we refer to as CMOverdrive. As shown in [32], for benchmarks A and B, MUSE's pre-processing phase is $2-3.6\times$ more communication frugal than Overdrive (the improvements in communication inclusive of online phase are also roughly the same). Combining this with SIMC++'s communication presented in Table 4, we see that SIMC++ is $30-61\times$ more communication efficient than Overdrive. [32] also showed their protocol to be $13-21\times$ faster than Overdrive in the CON setting. From Table 5, we can in turn conclude that SIMC++ is $52-147\times$ faster than Overdrive. In a similar manner, one can see that SIMC++ communicates $14-24\times$ lesser bits than CMOverdrive and is $26-49\times$ faster in the CON setting.

6 Conclusion

We consider the problem of client malicious secure inference and build a system, SIMC that is at least an order of magnitude more communication efficient and performant than prior state of the art. Furthermore, our system can also have a very lightweight online phase. Based on our microbenchmarks in Section 5.2, we expect our gains over MUSE to be even higher for larger neural network models, such as ResNet50 on ImageNet, as they have much wider non-linear layers (upto 2^{20} nodes). Moreover, replacing our OT extension based on KOS [26] with Silent-OT extension techniques [6, 54] will further improve the communication, and lead to better performance in low bandwidth settings. Finally, SIMC is along the same lines of work as SecureML [37], Gazelle [24], DELPHI and MUSE that use garbled circuits for their generality to handle all non-linear functions. We leave the exploration of design of specialized non-linear layer protocols in the client malicious setting along the lines of the state-of-the-art work in semi-honest setting, CrypTFlow2 [42], as future work.

References

[1] N. Agrawal, A. S. Shamsabadi, M. J. Kusner, and A. Gascón. QUOTIENT: two-party secure neural network training and prediction. In *CCS*, 2019.

[2] A. Aly, E. Orsini, D. Rotaru, N. P. Smart, and T. Wood. Zaphod: Efficiently combining LSSS and garbled circuits in scale. In *ACM WAHC*, 2019.

[3] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *STOC*, 1990.

[4] M. Bellare, V. T. Hoang, and P. Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In *ASIACRYPT*, 2012.

[5] R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT*, 2012.

[6] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In *CRYPTO*, 2019.

[7] E. Boyle, N. Gilboa, and Y. Ishai. Secure computation with preprocessing via function secret sharing. In *TCC*, 2019.

[8] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *CRYPTO*, 2012.

[9] R. Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptol.*, 13(1):143–202, Jan. 2000.

[10] N. Carlini, M. Jagielski, and I. Mironov. Cryptanalytic extraction of neural network models. In *CRYPTO*, 2020.

[11] H. Chen, M. Kim, I. P. Razenshteyn, D. Rotaru, Y. Song, and S. Wagh. Maliciously secure matrix multiplication with applications to private deep learning. In *ASIACRYPT*, 2020.

[12] V. Chen, V. Pastro, and M. Raykova. Secure computation for machine learning with SPDZ. *CoRR*, abs/1901.00329, 2019.

[13] A. P. K. Dalskov, D. Escudero, and M. Keller. Secure evaluation of quantized neural networks. *PoPETS*, 2020.

[14] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority – or: Breaking the SPDZ limits. In *ESORICS*, 2013.

[15] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, 2012.

[16] I. Damgård, D. Escudero, T. Frederiksen, M. Keller, P. Scholl, and N. Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *IEEE S&P*, 2019.

[17] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[18] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In *CRYPTO*, 2020.

[19] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2012:144, 2012.

[20] C. Gentry. Fully homomorphic encryption using ideal lattices. In *ACM STOC*, 2009.

[21] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. E. Lauter, M. Naehrig, and J. Wernsing. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *ICML*, 2016.

[22] O. Goldreich, S. Micali, and A. Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *ACM STOC*, 1987.

[23] C. Hazay, Y. Ishai, A. Marcedone, and M. Venkitasubramaniam. Leviosa: Lightweight secure arithmetic computation. In *CCS*, 2019.

[24] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *USENIX Security*, 2018.

[25] M. Keller. MP-SPDZ: A versatile framework for multi-party computation. In *CCS*, 2020.

[26] M. Keller, E. Orsini, and P. Scholl. Actively secure OT extension with optimal overhead. In *CRYPTO*, 2015.

[27] M. Keller, E. Orsini, and P. Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *ACM CCS*, 2016.

[28] M. Keller, V. Pastro, and D. Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT*, 2018.

[29] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*, 2008.

[30] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma. CrypTFlow: Secure TensorFlow Inference. In *IEEE S&P*, 2020.

[31] R. Lehmkuhl and P. Mishra. Muse: a Python, C++, and Rust library for Secure Convolutional Neural Network Inference for Malicious Clients. `https://github.com/ryanleh/muse/`, 2021.

[32] R. Lehmkuhl, P. Mishra, A. Srinivasan, and R. A. Popa. Muse: Secure inference resilient to malicious clients. In *USENIX Security*, 2021.

[33] Y. Lindell. *How to Simulate It – A Tutorial on the Simulation Proof Technique*, pages 277–346. Springer International Publishing, Cham, 2017.

[34] J. Liu, M. Juuti, Y. Lu, and N. Asokan. Oblivious Neural Network Predictions via MiniONN Transformations. In *CCS*, 2017.

[35] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa. Delphi: A Cryptographic Inference Service for Neural Networks. In *USENIX Security*, 2020.

[36] P. Mohassel and P. Rindal. ABY^3: A Mixed Protocol Framework for Machine Learning. In *CCS*, 2018.

[37] P. Mohassel and Y. Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*, 2017.

[38] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, 2016.

[39] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. EUROCRYPT, 1999.

[40] M. O. Rabin. How to exchange secrets with oblivious transfer. Cryptology ePrint Archive, Report 2005/187, 2005. `https://eprint.iacr.org/2005/187`.

[41] D. Rathee, M. Rathee, R. K. K. Goli, D. Gupta, R. Sharma, N. Chandran, and A. Rastogi. SIRNN: A Math Library for Secure RNN Inference. In *IEEE S&P*, 2021.

[42] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma. CrypTFlow2: Practical 2-Party Secure Inference. In *CCS*, 2020.

[43] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. E. Lauter, and F. Koushanfar. XONN: XNOR-based oblivious deep neural network inference. In *USENIX Security*, 2019.

[44] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *AsiaCCS*, 2018.

[45] R. L. Rivest, L. Adleman, and M. L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978.

[46] D. Rotaru and T. Wood. Marbled circuits: Mixing arithmetic and boolean circuits with active security. In *INDOCRYPT*, 2019.

[47] B. D. Rouhani, M. S. Riazi, and F. Koushanfar. Deepsecure: scalable provably-secure deep learning. In *DAC*, 2018.

[48] Microsoft SEAL (release 3.3). https://github.com/Microsoft/SEAL, 2019. Microsoft Research, Redmond, WA.

[49] F. Tramèr and D. Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. In *ICLR*, 2019.

[50] S. Wagh, D. Gupta, and N. Chandran. SecureNN: 3-Party Secure Computation for Neural Network Training. *PoPETs*, 2019.

[51] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *PoPETS*, 2021.

[52] X. Wang, A. J. Malozemoff, and J. Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[53] C. Weng, K. Yang, X. Xie, J. Katz, and X. Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. In *USENIX Security*, 2021.

[54] K. Yang, C. Weng, X. Lan, J. Zhang, and X. Wang. Ferret: Fast extension for correlated OT with small communication. In *CCS*, 2020.

[55] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, 1986.

[56] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *EUROCRYPT*, 2015.

[57] W. Zheng, R. A. Popa, J. E. Gonzalez, and I. Stoica. Helen: Maliciously secure coopetitive learning for linear models. In *IEEE S&P*, 2019.

A Performance in Preprocessing Model

Table 6 shows performance of SIMC++ and MUSE in the EAN setting.

	Benchmark A		Benchmark B	
Protocol	MUSE	SIMC++	MUSE	SIMC++
Preprocessing Phase	117.64	23.95	1758.12	238.76
Online Phase	3.45	3.45	25.21	25.21
Total	121.09	27.40	1782.92	263.97

Table 6: Latency (in seconds), of MUSE and SIMC++ on benchmarks A and B in preprocessing model [EAN setting].

B Threat Model

Formal Security. We formalize security using the simulation paradigm [9,33]. Security is modeled by defining two interactions: a real interaction where P_0 and P_1 execute the protocol in the presence of an adversary $\mathcal{A}$ and the environment $\mathcal{Z}$ and an ideal interaction where the parties send their inputs to a trusted functionality that performs the computation faithfully. Security requires that for every adversary $\mathcal{A}$ in the real interaction, there is an adversary $\mathcal{S}$ (called the simulator) in the ideal interaction, such that no environment $\mathcal{Z}$ can distinguish between real and ideal interactions, which we formally define below. Let $f = (f_0, f_1)$ be a two party functionality such that P_0 and P_1 invoke f on inputs a and b to learn $f_0(a,b)$ and $f_1(a,b)$ respectively. A protocol π securely realizes f in the client malicious paradigm if the following properties hold.

- **Correctness:** If P_0 and P_1 were honest, then P_0 learns $f_0(a,b)$ and P_1 learns $f_1(a,b)$ from the execution of π on inputs a and b respectively.
- **Semi-honest Server Security:** For semi-honest adversary $\mathcal{A}$ controlling P_0, $\exists$ a simulator $\mathcal{S}$ such that for any (a,b),
$$\mathsf{View}^{\pi}_{\mathcal{A}}(a,b) \approx \mathcal{S}(a, f_0(a,b))$$
where $\mathsf{View}^{\pi}_{\mathcal{A}}(a,b)$ denotes view of $\mathcal{A}$ during the execution of the protocol π with P_0's input a and P_1's input b.
- **Malicious Client Security:** For any malicious adversary $\mathcal{A}$ controlling P_1, there exists a simulator $\mathcal{S}$ such that for any input a from P_0,
$$\mathsf{Out}_{P_0}, \mathsf{View}^{\pi}_{\mathcal{A}}(a,\cdot) \approx \hat{\mathsf{Out}}, \mathcal{S}^{f(a,\cdot)}$$
where $\mathsf{View}^{\pi}_{\mathcal{A}}(a,\cdot)$ denotes the view of $\mathcal{A}$ during the execution of the protocol π with P_0's input being a. Out_{P_0}

Preamble. Recall the neural network from Section 4.2. Let $\mathbf{v}_0$ be the input vector of P_1. P_0 holds a MAC key α and $\mathbf{M}_1, \cdots, \mathbf{M}_\ell$. For any non-linear function f, let C^f be the boolean circuit that takes $d, \langle s \rangle_0, \langle s \rangle_1, c \in \mathbb{F}_p$ as input and outputs $(s, c, f(s) - d - c)$. During secure evaluation of C^f in the below protocol, P_0 holds the inputs $(d, \langle s \rangle_0)$ and P_1 holds the inputs $(\langle s \rangle_1, c)$ of C^f. Let $\mathbf{d}_0 = 0^{n_0}$. Let $\mathbf{s}_i$ and $\mathbf{v}_i$ denote the intermediate inference vectors obtained after the i^{th} linear layer and non-linear layer evaluation respectively.

Data from preprocessing phase for:

- *Linear layer $i \in [\ell]$:* P_b learns $\langle \hat{\mathbf{s}}_i \rangle_b, \langle \hat{\mathbf{t}}_i \rangle_b, \langle \mathbf{y}_i \rangle_b$, where $\hat{\mathbf{s}}_i = \mathbf{M}_i \mathbf{c}_i, \hat{\mathbf{t}}_i = \alpha \hat{\mathbf{s}}_i, \mathbf{y}_i = \alpha^2 \mathbf{c}_i$ (for $\mathbf{c}_i$ uniformly chosen from $\mathbb{F}_p^{n_{i-1}}$ by P_1) using additive homomorphic encryption, similar to the functionality $\mathcal{F}_{\mathsf{Lin}}$.
- *Non-linear layer $i \in [\ell - 1]$:* P_0 computes a garbled circuit (for C^{f_i}) GC_i along with pairs of input and outputs labels. P_1 learns labels corresponding to its input $\langle \hat{\mathbf{s}}_i \rangle_1, \mathbf{c}_{i+1}$ (to C^{f_i}) by invoking the oblivious transfer functionality with P_0. P_0 then sends the labels corresponding to its input $\mathbf{d}_i$ (uniformly chosen from $\mathbb{F}_p^{n_i}$) and an encryption of GC_i to P_1.
- *Consistency Check:* For each $i \in [\ell - 1]$, P_0 samples $\langle \mathbf{u}_i \rangle_0, \langle \mathbf{g}_{i+1} \rangle_0$ uniformly (where $\mathbf{u}_i = \alpha \mathbf{s}_i$, $\mathbf{g}_{i+1} = \alpha^2 \mathbf{c}_{i+1}$). For each $i \in [\ell - 1]$, P_1 receives from P_0, the ciphertexts computed using the output labels of GC_i necessary to recover $\langle \mathbf{u}_i \rangle_1, \langle \mathbf{g}_{i+1} \rangle_1$. These ciphertexts are computed by P_0 using techniques similar to the authentication phase of protocol $\pi^f_{\mathsf{Non\text{-}lin}}$ (in Figure 2).

Online Phase:

- *For every linear layer $i \in [\ell]$*: P_1 sends $\mathbf{v}_{i-1} - \mathbf{d}_{i-1} - \mathbf{c}_i$ to P_0 and sets $\langle \mathbf{s}_i \rangle_1 = \langle \hat{\mathbf{s}}_i \rangle_1, \langle \mathbf{t}_i \rangle_1 = \langle \hat{\mathbf{t}}_i \rangle_1$. P_0 computes $\langle \mathbf{s}_i \rangle_0 = \langle \hat{\mathbf{s}}_i \rangle_0 + \mathbf{M}_i(\mathbf{v}_{i-1} - \mathbf{c}_i), \langle \mathbf{t}_i \rangle_0 = \langle \hat{\mathbf{t}}_i \rangle_0 + \alpha \mathbf{M}_i(\mathbf{v}_{i-1} - \mathbf{c}_i)$.
- *For every non-linear layer $i \in [\ell - 1]$*: P_0 sends labels corresponding to its input (to C^{f_i}) $\langle \mathbf{s}_i \rangle_0$ and the decryption key to recover GC_i (from its encryption) to P_1 . P_1 decrypts and evaluates GC_i using the labels it received from P_0 and the labels for the inputs $\mathbf{d}_i, \langle \mathbf{s}_i \rangle_1, \mathbf{c}_{i+1}$ (learnt in the preprocessing phase) to learn $\mathbf{v}_i - \mathbf{d}_i - \mathbf{c}_{i+1}$ and the shares $\langle \mathbf{u}_i \rangle_1, \langle \mathbf{g}_{i+1} \rangle_1$ (recall, $\mathbf{u}_i = \alpha \mathbf{s}_i$, $\mathbf{g}_{i+1} = \alpha^2 \mathbf{c}_{i+1}$ and note that $\mathbf{v}_i = f_i(\mathbf{s}_i)$).
- *Consistency Check Phase*: Both parties locally compute shares of a random linear combination (similar to the consistency check phase of π_{Inf} in Figure 5) of the values $(\mathbf{t}_i - \mathbf{u}_i + \mathbf{y}_{i+1} - \mathbf{g}_{i+1})$ for $i \in [\ell - 1]$, where randomness of this linear combination is chosen by P_0 and sent to P_1. Then both parties reconstruct these shares and P_0 aborts if the reconstruction is non-zero. P_1 also sends a hash of the output labels that let it recover $\mathbf{v}_i - \mathbf{d}_i - \mathbf{c}_{i+1}$ for all $i \in [\ell - 1]$ to P_0. P_0 cross checks this hash with the vectors $\mathbf{v}_i - \mathbf{d}_i - \mathbf{c}_{i+1}$ sent by P_1 in the $(i+1)^{th}$ online linear layer evaluation (for $i \in [\ell - 1]$) and aborts if the check fails.
- *Output Phase:* If the consistency checks pass, P_0 sends $\langle \mathbf{s}_\ell \rangle_0$ to P_1, who reconstructs and outputs $\mathbf{s}_\ell$.

Figure 8: Secure Inference Protocol in the Preprocessing Model $\pi^{\mathsf{Prep}}_{\mathsf{Inf}}$

represents the output of P_0 in the same protocol execution. $\hat{\mathrm{Out}}$ and $\mathcal{S}^{f(a,\cdot)}$ denote the output of P_0 and $\mathcal{S}$ in an ideal interaction with the functionality f, where P_0 inputs a.

C Protocol in the Preprocessing Model

We describe the protocol in Figure 8. Remarks analogous to Remarks 3.1, 3.2, 3.3, 4.1, 4.2 also hold for this protocol.

D Extension for General Neural Networks

Our secure inference protocol π_{Inf} (Figure 5) can be easily tailored for neural networks that don't necessarily have alternate linear and non-linear layers. Though consecutive linear (resp., non-linear) layers can be composed into a single linear (resp., non-linear) layer, for efficiency reasons during the implementation phase it might be preferred to avoid this composition. If say for some i, the linear layers i and $i+1$ are consecutive, consider an imaginary non-linear layer i between them. Then, after computation of shares of $\mathbf{s}_i, \mathbf{t}_i, \mathbf{z}_i$ in the i^{th} linear layer evaluation as per π_{Inf}, parties locally compute shares of $\mathbf{u}_i = \mathbf{t}_i, \mathbf{v}_i = \mathbf{s}_i, \mathbf{w}_i = \mathbf{t}_i$. Next, they input shares of $(\mathbf{v}_i, \mathbf{w}_i)$ to the $(i+1)^{th}$ linear layer evaluation of π_{Inf} and use $\mathbf{u}_i$ in the consistency check phase. If non-linear layers i and $i+1$ are consecutive, consider an imaginary linear layer i between them. After computing shares of $\mathbf{u}_i, \mathbf{v}_i, \mathbf{w}_i$, parties locally compute shares of $\mathbf{s}_{i+1} = \mathbf{v}_i, \mathbf{t}_{i+1} = \mathbf{w}_i$ and $\mathbf{z}_{i+1} = 0$. They use shares of $\mathbf{s}_{i+1}$ as input to the $(i+1)^{th}$ non-linear layer evaluation in π_{Inf} and shares of $\mathbf{z}_{i+1}$ in the consistency check.