MUSE: Client-Malicious Secure Inference

Abstract

As machine learning inference has become ubiquitous in applications, there has been a corresponding increase in concerns about the privacy guarantees offered by existing mechanisms for inference. Such concerns have motivated the construction of efficient *secure inference* protocols which allow parties to perform inference without leaking any sensitive information. Recently, there has been a proliferation of such proposals, improving efficiency at a fast pace. However, a majority of these protocols assume that the client is semi-honest, that is, they do not deviate from the protocol; in practice, clients are many, have varying incentives, and can behave arbitrarily.

To demonstrate that a malicious client can completely break the security of semi-honest protocols, we first develop a new *model-extraction attack* against many state-of-the-art secure inference protocols. Our attack enables a malicious client to learn model weights with 22×–312× fewer queries than the best black-box model-extraction attack [Car+20].

Motivated by the severity of our attack, we design and implement MUSE, an efficient two-party secure inference protocol secure against *malicious clients*. MUSE pushes a majority of the cryptographic overhead into a pre-processing phase, making its online phase only 1.6× slower and use 1.4× more communication than that of the equivalent *semi-honest* protocol (which is close to state-of-the-art).

1 Introduction

The past few years have seen increasing deployment of neural network inference in popular applications such as image classification [Liu+17b]. However, the use of inference in such applications raises privacy concerns: existing implementations either require clients to send potentially sensitive data to remote servers for classification, or require the model owner to store their proprietary neural network model on the client's device. Both of these solutions are unsatisfactory: the former harms the privacy of the client, while the latter reveals information about the training data and model weights, both of which can be sensitive.

To resolve this tension, the community has focused increasing attention on constructing specialized protocols for *secure inference*, as we depict in Section 1. A secure inference protocol enables users and model owners to interact so that the user obtains the prediction result, while ensuring that neither party learns any other information about the user input or the model weights. Many of these works implement these guarantees with secure two-party computation [Gil+16; Moh+17; Hes+17; Liu+17a; Bru+18; Cha+17; Cho+18; San+18; Juv+18; Lou+19; Bou+18; Rou+18; Bal+19; Ria+19; Dat+19; Mis+20; Rat+20]. As we can see from Section 1, these works assume that both the client and the server follow the protocol rules, that is, that they are *semi-honest*.

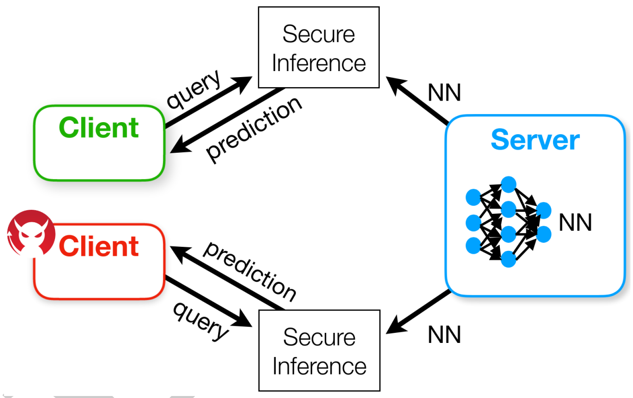


Figure 1: MUSE serves both honest and malicious clients.

While it is common in the literature to assume a semi-honest server, it is fundamentally less likely that *all* clients will behave correctly. The server is hosted at a single service provider, and existing cloud providers deploy competent intrusion detection systems, rigid access control, physical measures, and logging/tracking of the software installed [Goo17]. It is highly non-trivial to bypass these protections. Additionally, if a service provider is caught acting maliciously, the consequences may be dire due to public accountability.

In contrast, clients are many, and it suffices for just a single one of them to misbehave. The incentives for a client to cheat are high, and penalties low: service providers expend vast amounts of effort and money to accumulate data, clean it, design model architectures, and to train the final model. If a client wishes to obtain a similar model, it would be much easier to steal the server's than to try and train an equivalent one. This makes *model-extraction attacks* an attractive proposition; such an attack allows a client to learn the server's model by making a large number of inference queries. Furthermore, the consequences for getting caught are small: the client can simply spin up a new device and perform queries from there.

To illustrate the threat of a malicious client, in Section 2 we demonstrate a new model-extraction attack against semi-honest secure inference protocols whereby a malicious client can learn the server's entire model in a linear number of queries. This attack *greatly* outperforms the best model-extraction attacks for plaintext inference [Jag+20; Car+20],

			Vulnerable to malicious clients	Requires Network Modification [4]
2PC	HE	CHET [Dat+19], CryptoDL [Hes+17], LoLa [Bru+18], [Cha+17], Faster CryptoNets [Cho+18], TAPAS [San+18], FHE-DiNN [Bou+18]	🔴[2]	●
2PC	GC	DeepSecure [Rou+18], [Bal+19]	🔴[2]	◐
2PC	GC	XONN [Ria+19]	🔴[2]	●
2PC	Mixed	SecureML [Moh+17]	🔴[1]	●
2PC	Mixed	Gazelle [Juv+18], MiniONN [Liu+17a], CrypTFlow2 [Rat+20]	🔴[1]	○
2PC	Mixed	Delphi [Mis+20]	🔴[1]	◐
2PC	Mixed	**Muse**	○	○
3PC	Mixed	Chameleon [Ria+18]	🔴[1]	○
3PC	Mixed	ABY^3 [Moh+18]	○[3]	○
3PC	SS	SecureNN [Wag+19], Falcon [Wag+21], CrypTFlow [Kum+20]	○[3]	○

Table 1: Related work on secure convolutional neural network (CNN) inference. See Section 7 for more details. This table compares specialized secure inference protocols, not generic frameworks for MPC. We compare against generic 2PC frameworks in Section 6. HE = Homomorphic Encryption, GC = Garbled Circuits, SS = Secret Sharing. ◐ Network modifications are optional [1]See Section 2.1 [2] See Remark 2.1 [3]Requires that two of the three parties act honestly [4]Polynomial activations or binarized/discretized weights—may reduce network accuracy

and demonstrates that using semi-honest secure inference protocols can *significantly amplify* a malicious client's ability to steal a model.

A natural approach to defend against such an amplification is to leverage state-of-the-art generic secure computation tools providing *malicious security*. This guarantees that if *either* party acts maliciously, they will be caught and the protocol aborted, preserving privacy. However, such methods for achieving malicious security add a large overhead due to the use of heavy cryptographic primitives (e.g. zero-knowledge proofs [Gol+89] and cut-and-choose [Lin+15; Zhu+16]. In Section 6.4 we compare against such techniques).

To reduce this overhead, we propose Muse, a secure inference protocol that works in a slightly weaker *client-malicious* threat model. In this model, the server is presumed to behave semi-honestly, but the client is allowed to deviate arbitrarily from the protocol description. As we will show in Section 6, working in this model enables Muse to achieve much better performance than a fully malicious baseline.

Our contributions. To summarize, in this paper we make the following contributions:

- We devise a novel model-extraction attack against secure inference protocols that rely on additive secret sharing. This attack allows a malicious client to *perfectly extract all* the weights of a model with 22×–312× fewer queries than the state-of-the-art [Car+20]. The complexity of our attack depends *only* on the number of parameters, and not on other factors like the depth of the network.
- We design Muse, an efficient two-party cryptographic inference protocol that is secure against malicious clients. Muse identifies the fundamental barriers to achieving client-malicious secure inference and solves them with minimal overhead. Our implementation of Muse is able to achieve an online phase that is only 1.6× slower and uses 1.4× more communication than Delphi [Mis+20] which is close to state-of-the-art for *semi-honest* inference.
- As part of our system implementation, we formulate a number of techniques to optimize generic MPC frameworks in the client-malicious setting that improve performance by as much as 7×–42×.

Remark 1.1. While Muse's online phase is competitive with some of the best semi-honest protocols [Mis+20; Rat+20], the communication cost of preprocessing is up to 10× higher than in these semi-honest protocols. Hence, we view Muse as a first step in constructing secure inference protocols that achieve client-malicious security, and anticipate that future works will rapidly lower this cost (the same has occurred for semi-honest secure inference protocols). (Muse already improves performance over current techniques for client-malicious inference by more than of 7× (see Section 6.4)).

We now give a high-level overview of our contributions.

1.1 Our attack

What can a malicious client do? We start off by examining the power of malicious clients in secure inference protocols that rely on secret sharing. Specifically, we consider protocols that "evaluate" the neural network in a layer-by-layer fashion, so that at the end of each layer, the client and the server both hold 2-out-of-2 secret shares of the output of that layer. At the end of the protocol, the server sends its share of the final output to the client, who uses it to reconstruct the final output. This class of protocols contains many protocols of interest, such as [Moh+17; Juv+18; Liu+17a; Mis+20; Rat+20].

Our attack relies on the following crucial observation: because the shares at the end of a layer are not authenticated, a malicious client can *additively malleate* them without detection. In more detail, let $\langle m \rangle_C$ be the client's share, and $\langle m \rangle_S$ be the server's share of a message $m \in \mathbb{F}$, so that

$\langle m \rangle_C + \langle m \rangle_S = m$. Then, a malicious client can add an arbitrary shift r to a secret share to change the shared value from m to $m+r$. In Section 2, we show how one can leverage this malleability to learn the model weights.

1.2 Our protocol

We now explain how MUSE protects against malicious client attacks. We begin by describing our starting point: the semi-honest secure inference protocol DELPHI [Mis+20].

Starting point: DELPHI. We design MUSE by following the paradigm laid out in DELPHI [Mis+20]: since a convolutional neural network consists of alternating linear and non-linear layers, one should use subprotocols that are efficient for computing each type of layer, and then translate the output of one subprotocols to the input of the next subprotocol. DELPHI instantiates these subprotocols by using additive secret sharing to evaluate linear layers privately, and garbled circuits to evaluate non-linear layers.

Attempt 1: Preventing malleability via MACs. The key insight in our attack in Sections 1.1 and 2 is that the client can malleate shares undetectably. To prevent this, one can try to use standard techniques for authenticating the client's share via *information-theoretic homomorphic message authentication codes (MACs)*. This technique is employed by the state-of-the-art protocols for malicious security [Kel+18; Che+20; Esc+20]. However, trying to apply this technique directly to DELPHI runs into problems. In more detail, when switching between secret shares and garbled circuits, the server must ensure that the labels obtained by the client correspond to the authenticated secret share, and not to a different share. Doing this in a straightforward manner entails checking the MAC inside the garbled circuit, which is expensive. Furthermore, this check would need to be done in the online phase, which is undesirable.

Attempt 2: Separating authentication from computation. To remedy this, we make the following observation: garbled circuits already achieve malicious security against GC evaluators (clients in our setting). This means that, if we had a specialized protocol that could output labels for the client's secret shares *only if* the corresponding MACs were valid, then we could compose this protocol with the GC to achieve an end-to-end client-malicious secure inference protocol. In Section 5.1, we design exactly such a protocol for "conditional disclosure of secrets" (CDS) by optimizing existing malicious MPC frameworks [Kel+18] for the simpler client-malicious setting. While the resulting protocol is much more efficient than checking MACs inside GCs, it still imposes a significant cost on the online phase.

Our protocol. To remedy this, our final insight in MUSE is that the secret shares and MAC that the client feeds into the CDS protocol *do not depend* on the client's input in the online phase. This allows us to move the execution of the CDS protocol *entirely to the preprocessing phase*, resulting in an online phase that is almost identical to that of DELPHI. For details, see Section 5

2 Attacks on secure inference protocols based on secret-sharing

We now describe how a dishonest client can learn a server's secret model by behaving maliciously in semi-honest secure inference protocols that rely on additive secret sharing. As noted in Section 1.1, our attack relies on the fact that an additive secret share is malleable. We now describe how a malicious client can leverage this malleability to mount an attack that perfectly recovers the weights of the neural network. We begin by describing the kinds of protocols that are vulnerable to our attack in Section 2.1, and then in Section 2.2 we provide a detailed overview of our attack. Finally, we discuss the theoretical and empirical query complexity achieved by our attack in Section 2.3.

2.1 Vulnerable protocols

Our attack works against all semi-honest secure inference protocols that have the following properties. First, the protocol should evaluate the neural network by iteratively applying subprotocols for evaluating linear and non-linear layers. Next, the client and server inputs to each such layer's subprotocol should be additive secret shares of the actual layer input. Finally, the client's result should be the output of the final linear layer. A number of two-party and multi-party secure inference protocols have these properties [Moh+17; Liu+17a; Ria+18; Juv+18; Cha+19; Mis+20], including the state of the art CrypTFlow2 [Rat+20].

Remark 2.1 (Other semi-honest protocols)**.** Our attack does not affect some popular kinds of semi-honest secure inference protocols, including those based on fully-homomorphic encryption (FHE) [Gil+16; Cho+18; Bru+18; San+18; Dat+19], and those based on garbled circuits (GC) [Rou+18; Bal+19; Ria+19]. However, this does not immunize these protocols against other kinds of malicious client attacks:

- *FHE-based protocols* use *noise flooding* [Gen09a] to hide the server's model. This technique is inherently semi-honest as it requires the pre-existing noise to be honestly bounded; if this does not hold, the noise term can reveal information about the server's model *despite noise flooding*.
- *GC-based protocols* use oblivious transfer (OT) [Rab81] to transfer labels for the client's input. However, if this OT is only semi-honest secure, a malicious client can attack it to learn both labels for the same input wire, which breaks the privacy guarantees of the garbled circuit, and leads to a leak of the server's model.

2.2 Attack strategy

Notation. Let NN be a n-layer network convolutional neural network that classifies an image into one of m classes. That is, NN consists of n matrices $M_1, \ldots M_n$ so that $\mathsf{NN}(x) = M_n(\mathsf{ReLU}(\ldots M_2(\mathsf{ReLU}(M_1(x)))))$ where the image of M_n is $\mathbb{R}^m$. We denote by $\mathsf{NN}_i(x)$ the partial evaluation of NN up to the i-th linear layer. That is, $\mathsf{NN}_i(x) := M_i(\mathsf{ReLU}(\ldots M_2(\mathsf{ReLU}(M_1(x)))))$. Below we denote by $\mathbf{e}_j$ the j-th unit vector (the vector whose j-th entry is 1, and other entries are 0). Finally, for simplicity we assume that biases are zero.[1]

2.2.1 Recovering the last layer

Our attack proceeds in a bottom-up fashion: the client first recovers the parameters of the last linear layer $M_n \in \mathbb{R}^{m\times t}$, and then iteratively recovers previous layers. At a high level, to recover M_n, the client proceeds column-by-column as follows: for each $j \in [t]$, the client malleates their share of the input to M_n, so that the input to M_n become $\mathbf{e}_j$. This means that result $M_n \cdot \mathbf{e}_j$ is the j-th column of M_n. We illustrate this graphically for the first column below.

$$\overbrace{\begin{bmatrix}0\\0\end{bmatrix}}^{\mathbf{x}_{n-1}} \xrightarrow{\text{malleate}} \overbrace{\begin{bmatrix}1\\0\end{bmatrix}}^{\mathbf{x}'_{n-1}} \xrightarrow[M_n]{\text{query}} \overbrace{\begin{bmatrix}-0.1 & 0.2\\-1.1 & 1.2\end{bmatrix}}^{M_n}\begin{bmatrix}1\\0\end{bmatrix} = \overbrace{\begin{bmatrix}-0.1\\-1.1\end{bmatrix}}^{\text{first column of } M_n}$$

2.2.2 Recovering intermediate layers

The foregoing algorithm works for recovering the last layer because the client can directly read off M_n column-by-column by "solving" a linear system. However, this approach does not work as is for recovering the weights of intermediate linear layers, as we now demonstrate by considering the case of recovering the $n-1$-th linear layer M_{n-1}. We then describe how to resolve the issues that arise. (The case of the remaining layers follows similarly).

Problem 1: Intervening ReLUs are lossy and non-linear. ReLUs between the $n-1$-th layer and n-th layer disrupt linearity of the system, preventing the use of linear system solvers.

Solution 1: Force ReLUs to behave linearly. To resolve this issue, we recall the fact that ReLU behaves like the identity function on inputs that are positive. We use malleability to exploit this property and *force* the remaining $M_n(\mathsf{ReLU}(\cdot))$ computation to behave linearly, which means that we can once again solve a linear system to learn information about M_{n-1}.

In more detail, let $\langle \mathbf{y}_{n-2}\rangle_C$ be the client's share after applying M_{n-1}. The client malleates $\langle \mathbf{y}_{n-2}\rangle_C$ by setting it to $\langle \mathbf{y}'_{n-2}\rangle_C := \langle \mathbf{y}_{n-2} + \delta\rangle_C$, where δ is a constant vector whose elements are all greater than the magnitude of the largest element in $\mathbf{y}_{n-2}$.[2] This forces all entries of $\mathbf{y}'_{n-2}$ to be positive, which means ReLU acts like the identity function. Then, after evaluating the ReLU and obtaining $\langle \mathbf{x}'_{n-1}\rangle_C$, the client "undoes" the malleation by subtracting δ. The following equation provides a graphical illustration of this process.

[1]One can handle biases via a simple extension of our techniques.

[2]Note that since model weights are usually small (in the range $[-1,1]$), we can set δ to be a large value (say, ~ 10) to ensure that all entries of $\mathbf{y}'_{n-2}$ are positive.

$$\overbrace{\begin{bmatrix}-0.1 & 0.2\\-1.1 & 1.2\end{bmatrix}}^{M_{n-1}}\begin{bmatrix}1\\0\end{bmatrix} = \overbrace{\begin{bmatrix}-0.1\\-1.1\end{bmatrix}}^{y_{n-2}} \xrightarrow[\delta:=10]{\text{malleate}} \overbrace{\begin{bmatrix}9.9\\8.9\end{bmatrix}}^{y'_{n-2}} \xrightarrow{\text{ReLU}} \overbrace{\begin{bmatrix}9.9\\8.9\end{bmatrix}}^{x'_{n-1}} \xrightarrow{\text{unmalleate}} \overbrace{\begin{bmatrix}-0.1\\-1.1\end{bmatrix}}^{x_{n-1}}$$

Problem 2: Underconstrained linear system. While the foregoing technique enables us to force the network to behave like a linear function, we have no guarantees that the resulting linear system is solvable. Indeed, neural networks necessarily map a high-dimensional feature to a low-dimensional classification, and so the resulting "linearized" neural network *must* be lossy. The following figure illustrates this graphically:

$$\overbrace{\begin{bmatrix}1 & 3\\2 & 4\end{bmatrix}}^{M_2}\cdot\overbrace{\begin{bmatrix}a_1 & a_2 & a_3\\b_1 & b_2 & b_3\end{bmatrix}}^{M_1} = \overbrace{\begin{bmatrix}a_1+3b_1 & a_2+3b_2 & a_3+3b_3\\2a_1+4b_1 & 2a_2+4b_2 & 2a_3+4b_3\end{bmatrix}}^{M_3}$$

Here, M_1 and M_2 are the first and last layers of the network, respectively. We have used the technique in Section 2.2.1 to recover M_2, and now must recover M_1. If we try to do this by querying M_3, we get three equations for six variables:

$$M_3\mathbf{e}_1 = \begin{bmatrix}3a_1+6b_1\\0\\0\end{bmatrix} \text{ and } M_3\mathbf{e}_2 = \begin{bmatrix}0\\3a_2+6b_2\\0\end{bmatrix} \text{ and } M_3\mathbf{e}_3 = \begin{bmatrix}0\\0\\3a_3+6b_3\end{bmatrix}$$

Solution 2: Masking variables. The issue is that M_3 *does* contain sufficient information to recover M_1, but querying it naively loses that information. To resolve this, we use malleability again: the client uses the intervening ReLUs to "zero" out all but m entries of intermediate state, as follows:

$$M_1\cdot\begin{bmatrix}1\\1\\0\end{bmatrix} = \begin{bmatrix}a_1+a_2\\b_1+b_2\end{bmatrix} \xrightarrow[\text{+ mask}]{\text{malleate}} \begin{bmatrix}a_1+a_2+\delta\\b_1+b_2-\delta\end{bmatrix} \xrightarrow[\text{unmalleate}]{\text{ReLU +}} \begin{bmatrix}a_1+a_2\\0\end{bmatrix}$$

Now, the client can obtain $M_2 \cdot [a_1+a_2, 0]$, and can solve the resulting equations to learn a_1 and a_2. It can then repeat this process with different queries and "masks" to learn all of M_1.

For a detailed description of our algorithm, see Appendix C.

2.3 Efficiency and evaluation

Efficiency. Given a neural network where the i-th linear layer has dimension $m_i \times t_i$, and the number of classes is $m_n = m$, the foregoing algorithm learns the model parameters in just $\sum_{i=1}^{n}\lceil \frac{m_i}{m}\rceil \cdot t_i$ queries. Furthermore, the complexity of our attack depends *only on the number of parameters*, and not on other factors such as the depth. (This is not the case for other model-extraction attacks, which fail at larger depths)

Evaluation. In Table 2, we compare our work to the state-of-the-art prior work on model extraction [Car+20], which does not rely on the existence of a secure inference protocol (and hence does not exploit properties of such protocols). Our experiments match the query complexity derived above.

network dimensions	# parameters	# queries		speedup
		us	[Car+20]	
784-128-1	100,480	100,480	$2^{21.5}$	29.5×
784-32-1	25,120	25,120	$2^{19.2}$	24×
10-10-10-1	210	210	2^{16}	312×
10-20-20-1	620	620	$2^{17.1}$	226.5×
40-20-10-10-1	1,110	1,110	$2^{21.5}$	205×
80-40-20-1	4,020	4,020	$2^{17.1}$	92×
80×5-40-20-1	29,620	29,620	—	n/a

Table 2: Query complexity of our attack vs. that of [Car+20].

3 Building blocks

MUSE uses the following cryptographic building blocks.

Garbling Scheme. A *garbling scheme* [Yao86; Bel+12] is a tuple of algorithms $\mathsf{GS} = (\mathsf{Garble}, \mathsf{Eval})$ with the following syntax:

- $\mathsf{Garble}(1^\lambda, C, \{\mathsf{lab}_{i,0}, \mathsf{lab}_{i,1}\}_{i\in[n]}) \to \tilde{C}$. On input the security parameter, a boolean circuit C (with n input wires) and a set of labels $\{\mathsf{lab}_{i,0}, \mathsf{lab}_{i,1}\}_{i\in[n]}$, Garble outputs a *garbled circuit* $\tilde{C}$. Here $\mathsf{lab}_{i,b}$ represents assigning the value $b \in \{0,1\}$ to the i-th input wire.
- $\mathsf{Eval}(\tilde{C}, \{\mathsf{lab}_{i,x_i}\}) \to y$. On input a garbled circuit $\tilde{C}$ and labels $\{\mathsf{lab}_{i,x_i}\}$ corresponding to an input $x \in \{0,1\}^n$, Eval outputs a string $y = C(X)$.

We provide a formal definition in Appendix A.1, and briefly describe here the key properties satisfied by garbling schemes. First, GS must be *correct*: the output of Eval must equal $C(x)$. Second, it must be *private*: given $\tilde{C}$ and $\{\mathsf{lab}_{i,x_i}\}$, the evaluator should not learn anything about C or x except the size of $|C|$ (denoted by $1^{|C|}$) and the output $C(x)$.

Linearly homomorphic public-key encryption. A *linearly homomorphic* encryption scheme $\mathsf{HE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ [Elg85; Pai99] is a public key encryption scheme that additionally supports homomorphically evaluating linear functions on encrypted messages. Formally, HE satisfies the following syntax and properties:

- $\mathsf{KeyGen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$: On input a security parameter, KeyGen outputs a public key pk and a secret key sk.
- $\mathsf{Enc}(\mathsf{pk}, m) \to c$: On input the public key pk and a message m, the encryption algorithm Enc outputs a ciphertext c. We assume that the message space is $\mathbb{Z}_p$ for some prime p.
- $\mathsf{Dec}(\mathsf{sk}, c) \to m$: On input a secret key sk and a ciphertext c, the decryption algorithm Dec outputs a message m.
- $\mathsf{Eval}(\mathsf{pk}, c_1, c_2, L) \to c'$: On input a public key pk, ciphertexts c_1 and c_2 encrypting m_1 and m_2 respectively, and a linear function L,[3] Eval outputs a new ciphertext c'.

Besides the standard correctness and semantic security properties, we require HE to satisfy the following properties:

- *Homomorphism.* If $c_1 := \mathsf{Enc}(\mathsf{pk}, m_1)$, $c_2 := \mathsf{Enc}(\mathsf{pk}, m_2)$, and $c := \mathsf{Eval}(\mathsf{pk}, c_1, c_2, L)$, then $\mathsf{Dec}(\mathsf{sk}, c) = L(m_1, m_2)$.
- *Function privacy.* Given a ciphertext c, no attacker can tell what homomorphic operations led to c.

See Appendix A.2 for more formal definitions.

Additive secret sharing. Let p be a prime. A 2-of-2 *additive secret sharing* of $x \in \mathbb{Z}_p$ is a pair $(\langle x\rangle_1, \langle x\rangle_2) = (x - r, r) \in \mathbb{Z}_p^2$ for a random $r \in \mathbb{Z}_p$ such that $x = \langle x\rangle_1 + \langle x\rangle_2$. Additive secret sharing is perfectly hiding, i.e., given a share $\langle x\rangle_1$ or $\langle x\rangle_2$, the value x is perfectly hidden.

Message authentication codes. A *message authentication code (MAC)* is a tuple of algorithms $\mathsf{MAC} = (\mathsf{KeyGen}, \mathsf{Tag}, \mathsf{Verify})$ with the following syntax:

- $\mathsf{KeyGen}(1^\lambda) \to \alpha$: On input the security parameter, KeyGen outputs a MAC key α.
- $\mathsf{Tag}(\alpha, m) \to \sigma$: On input a key α and message m, Tag outputs a tag σ and a secret state st.
- $\mathsf{Verify}(\alpha, \mathsf{st}, m, \sigma) \to \{0,1\}$: On input a key α, secret state st, message m and tag σ, Verify outputs 0 or 1.

We require MAC to satisfy the following properties:

- *Correctness.* For any message m, $\alpha \leftarrow \mathsf{KeyGen}(1^\lambda)$, and $(\sigma, \mathsf{st}) \leftarrow \mathsf{Tag}(\alpha, m)$, $\mathsf{Verify}(\alpha, \mathsf{st}, m, \sigma) = 1$.
- *One-time Security.* Given a valid message-tag pair, no adversary can forge a different, valid message-tag pair.

See Appendix A.3 for formal definitions. In this work, we will use the following construction of MACs:

1. The message space is $\mathbb{Z}_p^n$ for some $p^n \geq 2^\lambda$.
2. KeyGen samples a uniform element $\alpha \leftarrow \mathbb{Z}_p$.
3. $\mathsf{Tag}(\alpha, m)$ outputs $\sigma = \langle \alpha \cdot m\rangle_1$ and $\mathsf{st} = \langle \alpha \cdot m\rangle_2$.
4. $\mathsf{Verify}(\alpha, \mathsf{st}, m, \sigma)$ checks if $\sigma + \mathsf{st} = \alpha \cdot m$.

We prove the one-time security of this scheme in Lemma A.1.

Beaver's multiplicative triples. A multiplication triple is a triple $(a, b, c) \in \mathbb{Z}_p^3$ such that $ab = c$. A triple generation procedure is a two-party protocol that outputs secret shares of a triple (a, b, c) to two parties.

Authenticated secret shares. For any prime p, an element $x \in \mathbb{Z}_p$, and a MAC key $\delta \in \mathbb{Z}_p$ an *authenticated share* of x is a tuple $(\varepsilon, [\![x]\!]_1, [\![x]\!]_2) := (\varepsilon, (\langle x\rangle_1, \langle \delta \cdot x\rangle_1), (\langle x\rangle_2, \langle \delta \cdot x\rangle_2))$. An authenticated share naturally supports local evaluation of addition and multiplication by public constants, as well as addition with another authenticated share. To multiply two authenticated shares, one needs to use multiplication triples. For simplicity of exposition, in the rest of the paper we omit ε, as it is merely used for bookkeeping when adding public constants.

Zero-knowledge proofs. Let $\mathcal{R}$ be any NP relation. A *zero-knowledge proof* for $\mathcal{R}$ is a protocol between a prover P and a verifier V that both have a common input x, where P tries to convince V that it "knows" a secret witness w such that $(x, w) \in \mathcal{R}$. At the end of the protocol, V should have learnt no additional information about w. We want our zero-knowledge proof system to satisfy the standard definitions

[3] L maps (m_1, m_2) to $am_1 + bm_2 + c$ for some $a, b, c \in \mathbb{Z}_p$.

of *completeness*, *soundness*, *proof of knowledge*, and *zero-knowledge*. See Appendix A.4 for formal definitions of these properties.

4 Threat model and privacy goals

In our system, there are two parties: the client and the service provider (or server). The server holds a neural network model, and the client holds some data that it wants classified by the server's model. To achieve this goal, the two parties interact via a protocol for *secure inference*. This protocol takes as input the server's model and the client's data, and computes the classification so that neither party learns any information except this final classification. Below we clarify the security guarantees we aim for when designing our secure inference protocol MUSE.

4.1 Threat model

There are two standard notions of security for multiparty computation: security against semi-honest adversaries, and security against malicious adversaries. A semi-honest adversary follows the protocol perfectly but inspects messages it receives to learn information about other parties' inputs. A malicious adversary, on the other hand, may arbitrarily deviate from the protocol.

In this work, we introduce a new threat model called "security against *fixed-subset malicious* adversaries". In a fixed-subset malicious threat model, the adversary can only maliciously corrupt a subset of parties that is known at protocol design time. In particular, in this work we consider the restriction of this model to the two-party setting, where the fixed-subset consists of the client. We refer to this setting as the *client-malicious* threat model.

4.2 Privacy goals

MUSE's goal is to enable the client to learn at most the following information: the architecture of the neural network, and the result of the inference; all other information about the client's private inputs and the parameters of the server's neural network model should be hidden. Concretely, we aim to achieve a strong simulation-based definition of security; see Definition B.1. Like most prior work, MUSE does not hide information that is revealed by the result of the prediction. See Section 7.1 for a discussion of attacks that leverage this information, as well as potential mitigations.

5 The MUSE protocol

In this section, we describe MUSE, our secure inference protocol that is secure against a malicious client and a semi-honest server. Like our starting point DELPHI (see Fig. 2) [Mis+20], MUSE's protocol consists of two phases: an offline preprocessing phase, and an online inference phase. The offline preprocessing phase is independent of the client's input (which regularly changes), but assumes that the server's model is static; if this model changes, then both parties have to re-run the preprocessing phase. After preprocessing, during the online inference phase, the client provides its input to our specialized secure two-party computation protocol, and eventually learns the inference result. Below, we expand on the high level overview in Section 1.2 and provide a detailed description of both phases of our protocol.

Notation. The server holds a model $\mathbf{M}$ consisting of ℓ linear layers $\mathbf{M}_1, \ldots, \mathbf{M}_\ell$ and the client holds an input vector $\mathbf{x} \in \mathbb{Z}_p^n$. We use $\mathsf{NN}(\mathbf{M}, \mathbf{x})$ to denote the output of the neural network when the server's input is $\mathbf{M}$ and the client's input is $\mathbf{x}$. We assume that the algorithm computing NN is public and is known to both the client and the server.

Preprocessing phase. During preprocessing, the client and the server precompute data that can be used during the online execution. This phase can be executed independent of the input values, i.e., DELPHI can run this phase before either party's input is known.

1. *Linear correlations generator:* The client and server interact with a functionality that, for each $i \in [\ell]$, outputs to them secret shares of $M_i \mathbf{r}_i$, where $\mathbf{r}_i$ is a random masking vector.
2. *Preprocessing for ReLUs:* The server construct a garbled circuit $\widetilde{C}$ for a circuit C computing ReLU. It sends $\widetilde{C}$ to the client and then uses OT send to the client the input wires corresponding to $\mathbf{r}_{i+1}$ and $\mathbf{M}_i \cdot \mathbf{r}_i - \mathbf{s}_i$.

Online phase. The online phase is divided into a two stages:
1. *Preamble:* On input $\mathbf{x}$, the client sends $\mathbf{x} - \mathbf{r}_1$ to the server. The server and the client now hold an additive secret sharing of $\mathbf{x}$.
2. *Layer evaluation:* Let $\mathbf{x}_i$ be the result of evaluating the first $(i-1)$ layers of the neural network on $\mathbf{x}$. At the beginning of the i-th layer, the client holds $\mathbf{r}_i$, and the server holds $\mathbf{x}_i - \mathbf{r}_i$, which means that they possess secret shares of $\mathbf{x}_i$.
   - *Linear layer:* The server computes $M_i \cdot (\mathbf{x}_i - \mathbf{r}_i)$, which means that the client and the server hold an additive secret sharing of $M_i \mathbf{x}_i$.
   - *ReLU layer:* After the linear layer, the client and server hold secret shares of $M_i \mathbf{x}_i$. The server sends to the client the labels corresponding to its secrete share, and the client then evaluates the GC to obtain a secret share of the ReLU output.

Figure 2: High-level overview of the DELPHI protocol [Mis+20].

5.1 Preprocessing phase

In the preprocessing phase, the client and the server precompute data that can be used during the online execution. This phase is independent of the client's input values, and can be run before the client's input is known.

5.1.1 Intuition

As explained in Section 1.2, MUSE follows the approach of DELPHI, and uses different cryptographic primitives to produce preprocessed material for linear and non-linear layers. Below we describe these primitives at a high level.

Linear layers. Like in DELPHI, our goal is to produce shares of $M\mathbf{r}$ for a linear layer M. This enables us to efficiently compute linear layer operations in the online phase. Unlike DELPHI, we additionally need to prevent tampering by malicious clients. To this end, we extend the linear correlations generator (LCG) used in DELPHI (see Fig. 2) to additionally support authentication. We formalize this via functionality $\mathcal{F}_{\mathsf{ACG}}$ for generating *authenticated correlations*. See Fig. 11 for a formal description.

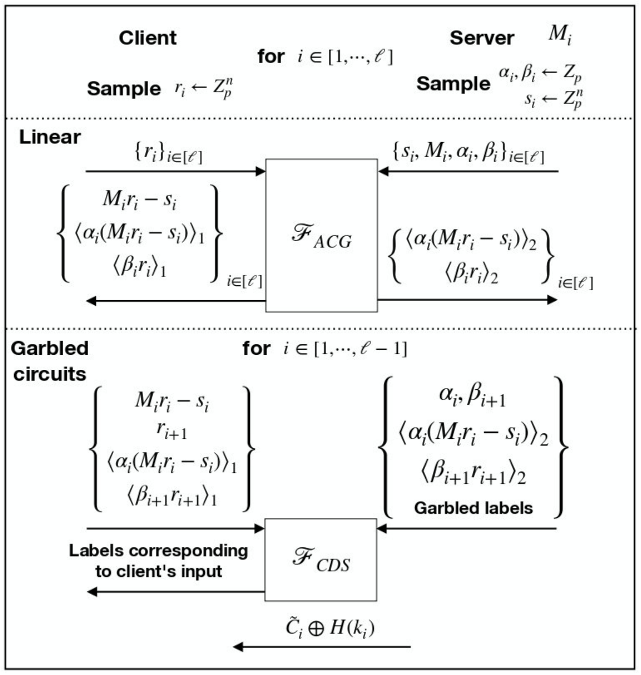

Client
for $i \in [1,\cdots,\ell]$
Server M_i
Sample $r_i \leftarrow Z_p^n$
Sample $\alpha_i, \beta_i \leftarrow Z_p$ $s_i \leftarrow Z_p^n$
Linear
$\{r_i\}_{i\in[\ell]}$
$\{s_i, M_i, \alpha_i, \beta_i\}_{i\in[\ell]}$
$\mathcal{F}_{ACG}$
$\left\{ M_i r_i - s_i, \langle \alpha_i(M_i r_i - s_i)\rangle_1, \langle \beta_i r_i \rangle_1 \right\}_{i\in[\ell]}$
$\left\{ \langle \alpha_i(M_i r_i - s_i)\rangle_2, \langle \beta_i r_i \rangle_2 \right\}_{i\in[\ell]}$
Garbled circuits
for $i \in [1,\cdots,\ell-1]$
$\left\{ M_i r_i - s_i, r_{i+1}, \langle \alpha_i(M_i r_i - s_i)\rangle_1, \langle \beta_{i+1} r_{i+1} \rangle_1 \right\}$
$\left\{ \alpha_i, \beta_{i+1}, \langle \alpha_i(M_i r_i - s_i)\rangle_2, \langle \beta_{i+1} r_{i+1} \rangle_2, \text{Garbled labels} \right\}$
$\mathcal{F}_{CDS}$
Labels corresponding to client's input
$\tilde{C}_i \oplus H(k_i)$

Figure 3: MUSE preprocessing phase.

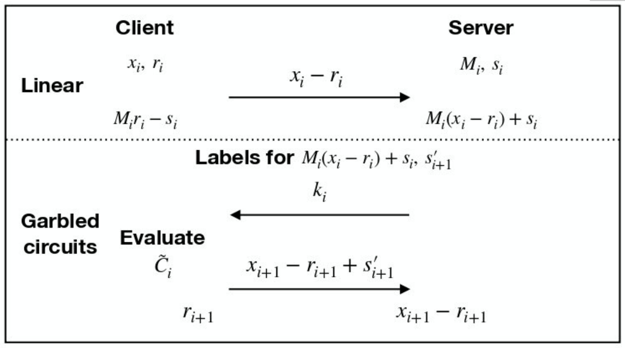


Figure 4: MUSE online phase.

To construct a protocol Π_{ACG} that realizes $\mathcal{F}_{\text{ACG}}$, we extend the techniques based on linearly homomorphic encryption used in DELPHI to additionally authenticate and secret share the relevant ciphertexts (see Fig. 5). We do this by relying on MACs as well as zero knowledge proofs-of-knowledge that assert that every ciphertext used in Π_{ACG} is well-formed.

Non-Linear Layers. Like in DELPHI, we use garbled circuits to efficiently evaluate ReLUs. However, unlike DELPHI, we can no longer use oblivious transfer to send garbled labels to the client, because we have no way to check that the input to OT corresponds to the output from $\mathcal{F}_{\text{ACG}}$. Instead we rely on a functionality which conditionally outputs these labels if the inputs match the output of $\mathcal{F}_{\text{ACG}}$. We call this functionality the *Conditional Disclosure of Secrets* functionality, and denote it by $\mathcal{F}_{\text{CDS}}$.

To construct a protocol Π_{CDS} that realizes $\mathcal{F}_{\text{CDS}}$, we have two options: use 2PC protocols specialized for boolean computation, or 2PC protocols specialized for arithmetic computation. Indeed, because this operation fundamentally reasons about boolean values, it would seem reasonable to use a protocol like garbled circuits. However, checking validity of the client's input requires modular multiplications, which are extremely expensive when expressed as boolean circuits. Since even the simplest neural networks oftentimes have thousands of activations, the resulting communication and computation cost is unacceptable.

Instead, we implement this functionality via MPC for arithmetic circuits, as modular multiplication is cheap here. However, now the boolean operations are expensive. To overcome this, we take further advantage of the client-malicious setting to optimize the MPC protocol we use to securely execute the arithmetic circuit (see Section 5.3).

By using $\mathcal{F}_{\text{ACG}}$ and $\mathcal{F}_{\text{CDS}}$ instead of LCG and OT, the online phase of MUSE remains *identical to that of the semi-honest* DELPHI *protocol.*

5.1.2 Protocol

We now present the full protocol for the preprocessing phase of MUSE (see Fig. 3 for a graphical overview).

1. For every $i \in [\ell]$, denote n_i, m_i as the input and output sizes of the i-th linear layer respectively. The client samples a random layer input mask $\mathbf{r}_i \leftarrow \mathbb{Z}_p^{n_i}$ and the server samples a random layer output mask $\mathbf{s}_i \leftarrow \mathbb{Z}_p^{m_i}$. Additionally, the server samples random MAC keys $\alpha_i, \beta_i \leftarrow \mathbb{Z}_p$.
2. **Authenticated correlations generator:** The client and server invoke functionality $\mathcal{F}_{\text{ACG}}$ with the client input $\{\mathbf{r}_i\}_{i\in[\ell]}$, and with server input $\{\mathbf{s}_i, \mathbf{M}_i, \alpha_i, \beta_i\}_{i\in[\ell]}$. For each $i \in [\ell]$, the client obtains $\{\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i\}_{i\in[\ell]}$ along with $\{\langle \beta_i \cdot \mathbf{r}_i\rangle_1, \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_1\}$ whereas the server receives $\{\langle \beta_i \cdot \mathbf{r}_i\rangle_2, \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_2\}$. In Fig. 11 we describe the ideal functionality in more detail, and give a protocol for achieving it in Fig. 5.
3. For each $i \in [\ell]$, let $\mathsf{inp}_i := (\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i, \mathbf{r}_{i+1})$ denote the client's input to the i-th non-linear layer and $|\mathsf{inp}_i|$ denote its size in bits. The server chooses a set of random garbled circuit input labels $\{\mathsf{lab}^C_{i,k,0}, \mathsf{lab}^C_{i,k,1}\}_{k\in[|\mathsf{inp}_i|]}$.
4. **Conditional disclosure of secrets:** For each $i \in [\ell]$, the client and the server invoke functionality $\mathcal{F}_{\text{CDS}}$ on the client's input inp_i and the MAC shares received from $\mathcal{F}_{\text{ACG}}$. If the client honestly inputs the correct shares, the functionality outputs the garbled input labels $\{\mathsf{lab}^C_{i,k,\mathsf{inp}_i}\}_{k\in[|\mathsf{inp}_i|]}$ corresponding to inp_i to the client. In Fig. 12, we describe the ideal functionality in more detail, and give a protocol for securely computing this functionality in Fig. 6.
5. For each $i \in [\ell]$, the server chooses random labels $\{\mathsf{lab}^S_{i,k,0}, \mathsf{lab}^S_{i,k,1}\}_{k\in[|\mathsf{inp}_i|]}$ for its input to the ith non-linear layer.
6. **Offline garbling:** For each $i \in [\ell]$, the server garbles the circuit C_i (described in Fig. 7) using

$\{\mathsf{lab}^C_{i,k,0}, \mathsf{lab}^C_{i,k,1}, \mathsf{lab}^S_{i,k,0}, \mathsf{lab}^S_{i,k,1}\}_{k \in [|\mathsf{inp}_i|]}$ as the input labels to obtain the garbled circuit $\widetilde{C}_i$. It chooses a key $k_i \leftarrow \{0,1\}^\lambda$ and sends $H(k_i) \oplus \widetilde{C}_i$ to the client where H is a random oracle.

5.2 Online phase

The online phase is divided into a two stages: the *preamble* and the *layer evaluation*. (See Fig. 4 for a graphical overview.)

5.2.1 Preamble

The client sends $(\mathbf{x} - \mathbf{r}_1)$ to the server.

5.2.2 Layer Evaluation

At the beginning of evaluating the i-th layer, the client holds $\mathbf{r}_i$ and the server holds $\mathbf{x}_i - \mathbf{r}_i$ where $\mathbf{x}_i$ is the vector obtained by evaluating the first $(i-1)$ layers of the neural network on input $\mathbf{x}$ (with $\mathbf{x}_1 = \mathbf{x}$). This invariant will be maintained for each layer. We now describe the protocol for evaluating the i-th layer, which consists of linear functions and activation functions:

Linear layer. The server computes $\mathbf{M}_i(\mathbf{x}_i - \mathbf{r}_i) + \mathbf{s}_i$ which ensures that the client and server hold an additive secret share of $\mathbf{M}_i\mathbf{x}_i$

Non-linear layer. After the linear layer, the server holds $\mathbf{M}_i(\mathbf{x}_i - \mathbf{r}_i) + \mathbf{s}_i$ and the client holds $\mathbf{M}_i\mathbf{r}_i - \mathbf{s}_i$. The parties evaluate the non-linear garbled circuit layer as follows:

1. The server chooses a random masking vector $\mathbf{s}'_{i+1}$ and sends the labels from the set $\{\mathsf{lab}^S_{i,k,0}, \mathsf{lab}^S_{i,k,1}\}_{k \in [|\mathsf{inp}_i|]}$ corresponding to its input $\mathbf{M}_i(\mathbf{x}_i - \mathbf{r}_i) + \mathbf{s}_i$ and $\mathbf{s}'_{i+1}$ to the client along with the key k_i.
2. The client uses k_i to unmask $H(k_i) \oplus \widetilde{C}_i$ to obtain $\widetilde{C}_i$. The client evaluates the garbled circuit $\widetilde{C}_i$ using its input labels obtained in the preprocessing phase and labels obtained from the server in the online phase. The client decodes the output labels and sends $\mathbf{x}_{i+1} - \mathbf{r}_{i+1} + \mathbf{s}'_{i+1}$ along with the hash of the output labels to the server.
3. The server checks if the hash computation is correct and recovers $\mathbf{x}_{i+1} - \mathbf{r}_{i+1}$.

Output phase. The server sends $\mathbf{s}'_{\ell+1}$ to the client and the client unmasks the output of the garbled circuit using this to learn the output of the inference $\mathbf{y}$.

Theorem 5.1. *Assuming the security of garbled circuits and the protocols for securely computing $\mathcal{F}_{ACG}$ (see Lemma B.3) and $\mathcal{F}_{CDS}$ (see Lemma B.5), the protocol described above is a private inference protocol against malicious clients and semi-honest servers (see Definition B.2) in the random oracle model.*

Correctness follows from inspection; we prove security in Appendix B.4.

Remark 5.2 (Field modulus and MAC security)**.** The soundness guarantee of our MACs depends on size of our underlying field $\mathbb{Z}_p$. While working with p λ improves soundness,

Protocol Π_{ACG}

1. Both parties engage in a two-party computation protocol with security against malicious clients and semi-honest servers to generate $(\mathsf{pk}, \mathsf{sk})$ for HE. The client learns pk and sk whereas the server only learns pk.
2. The client sends $\{\mathsf{Enc}(\mathsf{pk}, \mathbf{r}_i)\}_{i \in [\ell]}$ to the server along with a zero-knowledge proof of well-formedness of the ciphertext. The server verifies this proof before continuing.
3. For every $i \in [\ell]$,
   (a) The server homomorphically computes $\mathsf{Enc}(\mathsf{pk}, \mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)$, $\mathsf{Enc}(\mathsf{pk}, \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i))$, and $\mathsf{Enc}(\mathsf{pk}, \beta_i \cdot \mathbf{r}_i)$.
   (b) The server randomly samples $\langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_2$ and $\langle \beta_i \cdot \mathbf{r}_i\rangle_2$, homomorphically creates additive shares of the MAC values, and sends $(\mathsf{Enc}(\mathsf{pk}, \mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i), \mathsf{Enc}(\mathsf{pk}, \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_1), \mathsf{Enc}(\mathsf{pk}, \langle \beta_i \cdot \mathbf{r}_i\rangle_1))$ to the client.
   (c) The client decrypts the above ciphertexts and obtains $(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)$, $\langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_1$, and $\langle \beta_i \cdot \mathbf{r}_i\rangle_1$. The server holds $\langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_2$ and $\langle \beta_i \cdot \mathbf{r}_i\rangle_2$.

Figure 5: Our construction of an authenticated correlations generator (ACG).

Protocol Π_{CDS}

Denote the bit decomposition of $\mathsf{inp}_i = (\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i, \mathbf{r}_{i+1})$ as $\{b^i_k\}_{k \in [|\mathsf{inp}_i|]}$.

1. The client and server input securely compute the following function f_{CDS} using a secure two-party computation protocol against malicious clients and semi-honest servers:
   (a) The client's input consists of $(\{b^i_k\}_{k \in [|\mathsf{inp}_i|]}, \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_1, \langle \beta_{i+1} \cdot \mathbf{r}_{i+1}\rangle_1)$ and the server's input consists of $(\alpha_i, \beta_{i+1}, \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_2, \langle \beta_{i+1} \cdot \mathbf{r}_{i+1}\rangle_2, \{\mathsf{lab}^C_{i,k,0}, \mathsf{lab}^C_{i,k,1}\}_{k \in [|\mathsf{inp}_i|]})$ for some $i \in [\ell - 1]$.
   (b) Denote $g_k = \mathsf{lab}^C_{i,k,0} - (\mathsf{lab}^C_{i,k,0} - \mathsf{lab}^C_{i,k,1}) \cdot b^i_k$. Additively secret share g_k into $(\langle g_k\rangle_1, \langle g_k\rangle_2)$.
   (c) Reconstruct $\overline{\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i}$ and $\overline{\mathbf{r}_{i+1}}$ from $\{b^i_k\}_{k \in [|\mathsf{inp}_i|]}$.
   (d) Sample random vectors $\mathbf{c}_1 \leftarrow \mathbb{Z}_p^{|\mathbf{r}_{i+1}|}, \mathbf{c}_2 \leftarrow \mathbb{Z}_p^{|\mathbf{r}_{i+1}|}$
   (e) Denote $\rho = \alpha_i(\mathbf{c}_1^T \cdot \overline{\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i}) - \mathbf{c}_1^T \cdot (\langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_1 + \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_2)$ and $\sigma = \beta_{i+1}(\mathbf{c}_2^T \cdot \overline{\mathbf{r}_{i+1}}) - \mathbf{c}_2^T \cdot (\langle \beta_{i+1} \cdot \mathbf{r}_{i+1}\rangle_1 + \langle \beta_{i+1} \cdot \mathbf{r}_{i+1}\rangle_2)$.
   (f) Output ρ, σ, $\{\langle g_k\rangle_2\}_{k \in [|\mathsf{inp}_i|]}$ to the server and $\{\langle g_k\rangle_1\}_{k \in [|\mathsf{inp}_i|]}$ to the client.
2. If either of them are non-zero, the server aborts the protocol. Else, it sends $\{\langle g_k\rangle_2\}_{k \in [|\mathsf{inp}_i|]}$ to the client. The client reconstructs $\{g_k\}_{k \in [|\mathsf{inp}_i|]}$ and outputs it.

Figure 6: Our protocol for conditional disclosure of secrets.

Server's input: $\mathbf{M}_i(\mathbf{x}_i - \mathbf{r}_i) + \mathbf{s}_i, \mathbf{s}'_{i+1}$.
Client's input: $\mathbf{r}_{i+1}$, $\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i$

1. Compute $\mathbf{M}_i(\mathbf{x}_i) = \mathbf{M}_i(\mathbf{x}_i - \mathbf{r}_i) + \mathbf{s}_i + \mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)$.
2. Compute $ReLU(\mathbf{M}_i(\mathbf{x}_i))$ to obtain $\mathbf{x}_{i+1}$.
3. Output $\mathbf{x}_{i+1} - \mathbf{r}_{i+1} + \mathbf{s}'_{i+1}$.

Figure 7: Description of circuit C_i.

it leads to a blowup in the size of the garbled circuits in the online phase, as well as a significant slowdown in the homomorphic encryption scheme we use. Thus, we opt to use a smaller prime and instead execute all MAC operations *twice* (with different keys) to achieve λ bits of computational security.

Remark 5.3 (Fixed-point arithmetic in finite fields)**.** Neural networks work over the real numbers, but our cryptographic protocols work over finite prime fields. To emulate real arithmetic, we rely on fixed-point arithmetic. However, to maintain precision, one needs to occasionally *truncate* intermediate values to ensure that the result does not wrap around. Unlike DELPHI, we cannot use the technique of [Moh+17] as it is incompatible with our MACs. Hence, we perform this truncation securely within our garbled circuits.

5.3 An efficient protocol for computing f_{CDS}

To securely compute the function f_{CDS}, MUSE adapts the state-of-the-art arithmetic MPC framework Overdrive [Kel+18] (which achieves malicious security) to the simpler client-malicious 2PC setting. Doing so results in great efficiency improvements, as we now explain.[4]

The heaviest cryptographic costs when using Overdrive for f_{CDS} are due to (a) MAC key generation, (b) triple generation, and (c) authentication of secret client and server inputs. We now describe how we optimize both of these procedures in the client-malicious setting.

MAC key generation. In Overdrive, a MAC key must be secret-shared among the parties, since any party may be malicious and could use knowledge of the key to cheat. In the client-malicious setting, the server will never cheat so they can simply generate and hold the MAC key themselves.

Triple generation. In order to generate multiplication triples in Overdrive, all parties must generate ciphertexts of their shares, prove knowledge of these ciphertexts in zero-knowledge, homomorphically compute a triple from the ciphertexts, and run a distributed decryption algorithm so all parties receive a share of the result. Note that the distributed decryption allows a malicious adversary to inject an authenticated additive shift, so parties must "sacrifice" a triple in order to ensure correctness [Dam+12], which harms performance.

In our setting, we can take advantage of the fact that the server knows the MAC key as follows: the client sends the encryption of their shares directly to the server (along with a zero-knowledge proof of plaintext knowledge). The server homomorphically computes the shares of the triple, and returns it to the client. Since the server performs the computation, correctness is guaranteed and no sacrifice is necessary. We provide benchmarks of our optimized generation in Section 6.5. See Fig. 17 for a full description of Π_{Triple}.

[4]While the protocol of [Che+20] offers better performance than Overdrive, at the time of writing the source code for it was unavailable, and so we could not build upon it. Our optimizations in this section apply also to the [Che+20] protocol, so it is plausible that in the future MUSE could rely instead on it.

Input authentication. Overdrive [Kel+18] optimizes the input sharing method of [Dam+12], by assuming that the encryption scheme they employ achieves *linear-targeted malleability* (LTM) [Bit+13]. The LTM assumption for an encryption scheme informally states that only affine transformations can be computed on ciphertexts. This assumption is non-falsifiable, and, when applied to the encryption schemes used in Overdrive, has received insufficient scrutiny.

In our protocol, we avoid relying on this strong assumption by observing that the majority of secret inputs originate with the server, and because the server holds the MAC keys, it can easily authenticate its inputs *without cryptography*. In more detail, the protocol proceeds as follows.

- The server shares their inputs by producing a random authenticated share of their input using the MAC key and sends it to the client.
- The client shares their input by following the same methodology as [Dam+12]. Note that generating random authenticated shares can be implemented using our triple generation procedure from above, thus inheriting the same speedups.

We benchmark these techniques in Section 6.5. See Fig. 16 for a full description of $\Pi_{\text{InputAuth}}$.

6 Evaluation

We divide the evaluation into three sections which answer the following questions:

- Section 6.3: What are the latency and communication costs of MUSE's individual components when performing inference and how do they compare to the semi-honest DELPHI?
- Section 6.4: How does MUSE compare to other inference protocols secure against malicious clients?
- Section 6.5: How does our client-malicious Overdrive subprotocol compare to standard Overdrive?

6.1 System Implementation

We implemented MUSE in Rust and C++. We use the SEAL homomorphic encryption library [Sea] to implement HE, the `fancy-garbling` library[5] for garbled circuits, and use MP-SPDZ[6] [Kel20] to approximate the cost of the zero-knowledge proofs in our protocol.

As we show in Section 6.3, the primary overhead of our pre-processing phase is due to triple generation. Thus, in order to minimize the total run time of the pre-processing phase, MUSE intelligently allocates a number of threads to triple generation and uses the remaining threads to run the other components of the pre-processing phase in parallel. This allocation is chosen to minimize the total time required, while also ensuring that network bandwidth is not saturated.

6.2 Evaluation Setup

All experiments were carried out on AWS `c5.9xlarge` instances possessing an Intel Xeon 8000 series CPU at 3.6 GHz.

[5]https://github.com/GaloisInc/fancy-garbling/
[6]https://github.com/data61/MP-SPDZ

					MNIST		MiniONN	
			system	threads	time (s)	comm. (GB)	time (s)	comm. (GB)
Pre-processing	Linear	CG	DELPHI	1	5.30	0.04	40.30	0.14
		ACG	MUSE	2	6.15	0.11	40.85	0.46
	Non-linear	Garbling	DELPHI	2	5.80	0.17	80.54	4.41
		Garbling	MUSE	2	9.31	0.44	134.89	7.15
		OT	DELPHI	1	0.91	0.02	11.67	0.34
		CDS Triple Gen.	MUSE	6	20.10	2.48	322.00	41.45
		CDS Input Auth.	MUSE	2	7.53	0.41	114.00	5.33
		CDS Evalutation	MUSE	2	3.83	0.36	61.86	6.31
Online		Online	DELPHI	8	1.05	0.01	7.14	0.16
		Online	MUSE	8	1.21	0.01	11.24	0.23

Table 3: Latency and communication cost of the individual components of MUSE and DELPHI. See Section 6.2 for more information on the network architectures and number of threads used.

The client and server were executed on two such instances located in the `us-west-1` (Northern California) and `us-west-2` (Oregon) regions respectively with 21 ms round-trip latency. The client and server executions used 8 threads each. We evaluate MUSE on the following datasets and network architectures:

1. *MNIST* is a standardized dataset consisting of (28×28) greyscale images of the digits 0–9. The training set contains 60,000 images, while the test set has 10,000 images. Our experiments use the 2-layer CNN architecture specified in MiniONN [Liu+17a] with average pooling in place of max pooling.
2. *CIFAR-10* is a standardized dataset consisting of (32×32) RGB images separated into 10 classes. The training set contains 50,000 images, while the test set has 10,000 images. Our experiments use the 7-layer CNN architecture specified in MiniONN [Liu+17a].

In all our experiments , MUSE allocates 6 threads for triple generation, 2 threads for the other pre-processing components, and (after preprocessing has completed) 8 threads for the online phase. Note that the current end-to-end numbers for MUSE are *estimates* as we have implemented all of the individual components, but are still in the process of integrating them into a full system. To ensure the accuracy of our estimates, we carefully measured the network throughput of each component during microbenchmarking and ensured that the throughput of our hypothesized architecture never exceeded the total network throughput of $\sim 1\,\text{Gbit/s}$.

Baselines. Since there are no specialized protocols for secure inference with dishonest-majority malicious or fixed-subset malicious security, we chose to use generic MPC frameworks as our baselines to compare MUSE against: maliciously-secure Overdrive [Kel20] *and* Overdrive with our client-malicious optimizations. We used MP-SPDZ's implementation of the CowGear protocol, We exclude the cost of key generation for both protocols since this is a one-time cost, so the covert security does not factor into our comparison. We used microbenchmarks from MUSE's triple generation, MUSE's input authentication, and Overdrive to estimate the total runtime of client-malicious Overdrive. Note that the protocol of [Che+20] claims to offer much better performance than Overdrive, but an implementation was not available at the time of writing, and so we could not compare against it.

Additionally, we use our microbenchmarks to demonstrate the cost of strengthening each individual component of MUSE from semi-honest to client-malicious security. To do so, we choose to compare against DELPHI and not against recent works which offer better performance [Rat+20]. This is because these newer works use different techniques for both linear and non-linear layers, which would make isolating the cost of upgrading security difficult.[7]

6.3 Microbenchmarks

In Table 3 we compare microbenchmarks for MUSE and DELPHI on the MNIST and MiniONN networks to demonstrate the concrete costs of strengthening each component of DELPHI to client-malicious security.

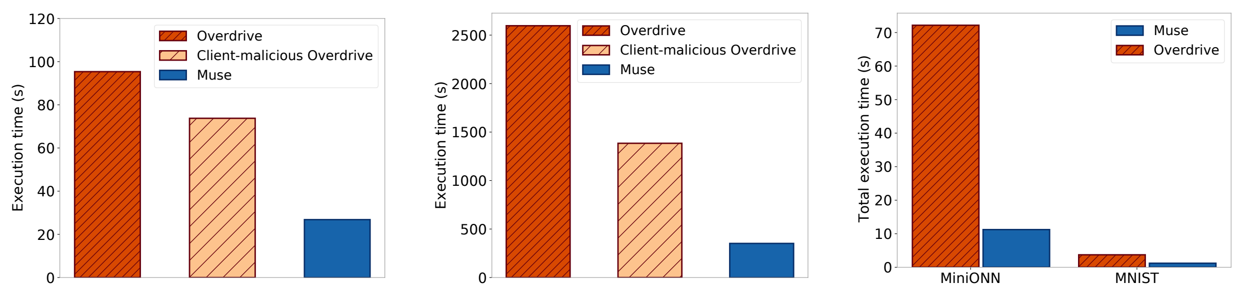
6.3.1 Preprocessing phase

The primary difference between MUSE and DELPHI occurs in the preprocessing phase.

Linear layers. As discussed in Section 5.1, the primary difference in how DELPHI and MUSE pre-process linear layers lies in the fact that the former uses plain correlations generator (CG), while MUSE use a *authenticated correlations generator*. Because these components are simpler in concrete runtime, we should expect the runtimes of DELPHI and MUSE here to be nearly equivalent (since MUSE runs its two ACG iterations in parallel), with an approximately $3\times$ increase in communication to account for the extra ACG iteration, zero knowledge proofs, and additional ciphertexts. In Table 3 we observe that this is precisely the case.

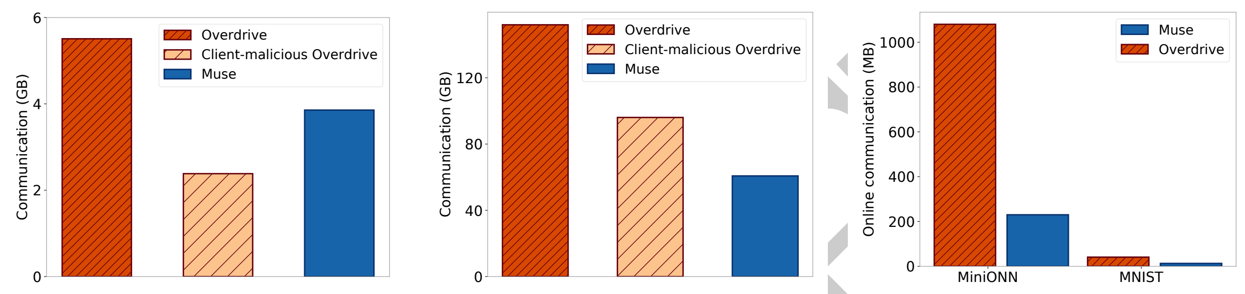
Non-linear layers. To pre-process the non-linear layers in DELPHI, the server garbles a circuit corresponding to ReLU and sends to the client. The two parties than engage in an

[7] It is also unclear how to upgrade their semi-honest protocols to achieve client-malicious security.



(a) MNIST Preprocessing time **(b)** MiniONN Preprocessing time **(c)** Online time

Figure 8: Comparison of execution times between MUSE, Overdrive, and client-malicious Overdrive



(a) MNIST Preprocessing communication **(b)** MiniONN Preprocessing communication **(c)** Online communication

Figure 9: Comparison of communication cost between MUSE, Overdrive, and client-malicious Overdrive

oblivious transfer whereby the client learns the garbled labels corresponding to their input.

In MUSE, a number of modifications to this procedure must be made. First, our protocol pushes some checks from the CDS to the online garbled circuits in This roughly doubles the number of AND gates in the circuit. Secondly, MUSE cannot use simple oblivious transfer and must opt for the much more expensive CDS protocol to ensure the client receives the correct garbled labels.

As a result, we should expect a roughly 2× increase in latency and communication for the garbling in MUSE when compared to DELPHI, and a much higher cost for the CDS compared to oblivious transfer. Table 3 validates these hyoptheses. Furthermore, as expected, we see that CDS is by far the dominant cost due to the large number of required triples.

6.3.2 Online phase

The online phase of DELPHI consists of a round per layer. For each linear layer, the client sends a secret share to the server. For each non-linear layer, the server sends the client labels which are used to evaluate a garbled circuit. At the last layer, the server sends the client their secret share of the output.

MUSE retains the same structure for the online phase but has a few small additions. For repeated linear layers, MUSE requires the additional sending of some MAC shares. For non-linear layers, MUSE requires an extra hash key to be sent, and the circuit being evaluated is roughly twice the size as the one in DELPHI.

As a result, we should expect the garbled circuit evaluation time to be the only difference in online runtime between MUSE and DELPHI, and for MUSE to have slightly higher communication. In Table 3 we see that the difference in online runtime is 1.1×–1.6× and the communication difference is approximately 1.4×.

In conclusion, MUSE is nearly identical to DELPHI in every component *except* the CDS. Consequently, we believe that this component of MUSE has the most room for improvement and discuss ideas for future work in Section 8. To our knowledge, MUSE's online phase outperforms *all previous two-party semi-honest* works besides DELPHI, CrypTFlow2 [Rat+20], and XONN [Ria+19].

6.4 Full system comparisons

Fig. 8 and Fig. 9 demonstrate how MUSE performs against malicious Overdrive and client-malicious Overdrive. Note that our client-malicious optimizations for Overdrive don't affect the online phase which is why we exclude client-malicious Overdrive in the online figures.

In summary, MUSE's pre-processing is 4×–7× faster and reduces communication by 1.5×–3× compared to standard Overdrive. For client-malicious Overdrive, MUSE's preprocessing phase is 3×–4× more efficient. Note that for the smaller MNIST network, the communication cost of MUSE is higher than that of client-malicious Overdrive, but our techniques scale better and achieve a 2× reduction for the larger MiniONN network. For the online phase, we observe a 3×–6.5× latency improvement and 3×–5× communication improvement when comparing MUSE to Overdrive.

6.5 Improvements to Overdrive

In this section we demonstrate the effectiveness of our optimizations to Overdrive in the client-malicious setting. In particular, we show that in client-malicious Overdrive *without the LTME assumption*:

- Triple generation is significantly more efficient.
- Server input authentication is significantly more efficient.
- Client input authentication is of comparable performance.

These improvements are of independent interest and can easily be extended to an n-party fixed-subset malicious setting.

Triple generation. In Fig. 10 we benchmark the generation of triples on a variable number of threads. In summary, client-malicious Overdrive achieves a $5\times$–$6\times$ latency improvement and $2\times$ communication reduction (the latter is not shown in the graph) over standard Overdrive. Beyond 6 threads, we observed little improvement in triple throughput due to maxing out the network throughput of $\sim$ 1Gbit/s.

Input authentication. In Table 4 we show benchmarks for input authentication for the client and server. Note that our protocol for client inputs achieves almost identical performance to Overdrive *without the LTME assumption*. We observe a $42\times$ improvement in latency and $2.25\times$ improvement in communication for server inputs.

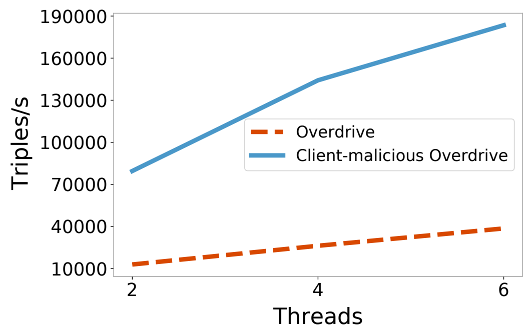


Figure 10: Triple Generation over a 44 bit prime field with 40 bit statistical security amortized over a batch of 10,000,000.

	Client Inputs		Server Inputs	
Threat Model	time (s)	comm. (MB)	time (s)	comm. (MB)
🟥 Mal.	12.11	90	12.11	90
Mal.-client	12.26	310	0.29	40

Table 4: Input Authentication on 1,000,000 inputs over a 44 bit prime field with 40 bit statistical security using a single thread.
🟥 Relies on the LTME Assumption.

7 Related work

7.1 Model extraction attacks

A number of recent works extract convolutional neural networks (CNNs) [Tra+16; Mil+19; Jag+20; Rol+20; Car+20] given oracle access to a neural network. Like our attack, these works only make black-box use of the neural network. However, unlike our attack, these works do not exploit properties of any secure inference protocol (and indeed do not rely on the existence of these). We compare against the state-of-the-art attack [Car+20] in Section 2.3.

Mitigations. Mitigations fall into two camps: those that inspect prediction queries [Kes+18; Juu+19], and those that try to instead modify the network to make it resilient to extraction [Tra+16; Lee+19]. While the latter kind of defense can be applied independently of secure inference, adapting the first kind of defense to work with secure inference protocols is tricky, because it requires inspecting the client's queries, which can violate privacy guarantees.

7.2 Secure inference protocols

A number of recent works have attempted to design specialized protocols for performing secure inference. These protocols achieve effeciency by combining secure computation techniques such as homomorphic encryption [Gen09b], Yao's garbled circuits [Yao86], and homomorphic secret sharing [Boy+17] with various modifications such as approximating ReLU activations with low-degree polynomials or binarizing (quantizing to one bit of accuracy) network weights (See Section 1 for a high-level overview of these protocols).

While these works have improved on latency and communication costs by orders of magnitude, a majority of them assume a semi-honest adversary. Generally speaking, currently-existing maliciously-secure inference protocols fall into the following categories: 3PC-based protocols, generic MPC frameworks, TEE-based protocols, and GC-based protocols. In the remainder of the section we discuss each of these categories:

3PC-based protocols. Recent works have explored how the addition of a third party can greatly improve efficiency for secure machine learning applications [Moh+17; Ria+18; Moh+18; Wag+19; Wag+21; Kum+20]. Many of these protocols also allow for easy extensions to handle malicious adversaries [Moh+18; Wag+19; Wag+21]. These extensions are made possible by the fact that these works assume only *one* of the parties is corrupted. In other words, these works consider *honest majority* malicious security. On the other hand, MUSE addresses the fundamentally more difficult problem of a dishonest majority.

While having three parties is convenient from a protocol design perspective, it would be difficult to implement in practice. It is the authors' opinion that two-party protocols solve the problem of secure inference in a much more natural manner.

TEE-based protocols. Generally speaking, TEE-based protocols [Tra+19; Top+18; Han+18; App19] provide better efficiency than protocols relying on purely cryptographic techniques. However, this improved efficiency comes at the cost of a weaker threat model that requires trust in hardware vendors and the implementation of the enclave. Furthermore, because the protocol execution occurs in an adversarial environment, any side-channel leakage is more dangerous (since the adversary can carefully manipulate the execution to force this leakage). Indeed, the past few years have seen a number of powerful side-channel attacks [Bra+17; Häh+17; Göt+17; Mog+17; Sch+17; Wan+17; Van+18] against popular enclaves like Intel SGX and ARM TrustZone.

Generic frameworks. Maliciously-secure MPC frameworks exist for computing arithmetic circuits [Dam+12; Kel+18; Che+20], binary circuits [Kat+18], and mixed circuits [Rot+19; Esc+20; Moh+18]. Before MUSE, these were the only existing cryptographic mechanisms for two-party client-malicious secure inference. While [Che+20] is the most efficient of these for inference, an implementation was not available at the time of writing so we compared against [Kel+18] in Section 6.4. From the results of Section 6.4 and the experiments provided in [Che+20], we can roughly estimate that the preprocessing communication of [Che+20] is similar to MUSE, but MUSE is superior on all other accounts.

GC-based protocols. DeepSecure [Rou+18], the protocol of Ball et al. [Bal+19], and XONN [Ria+19], all use circuit garbling schemes to implement constant-round secure inference protocols. While DeepSecure supports general neural networks, the protocol of [Bal+19] operates on discretized neural networks, which have integer weights, while XONN is optimized for binarized neural networks [Cou+15], which have boolean weights. These quantized networks allow for improved performance by avoiding computing expensive fixed-point multiplication in favor of integer multiplication or binary XNOR gates.

While neural network inference is commonly performed on quantized networks [Kri18], in practice quantization is never done below 8-bits since inference accuracy begins to suffer [Ban+18]. To combat this accuracy drop, XONN increases the number of neurons in its linear layers, gaining increased accuracy at the cost of a slower evaluation time. While this technique appears to work well for the datasets XONN evaluates, additional techniques are needed to scale to more difficult datasets like Imagenet as the current best-known quantization techniques for Imagenet requires 2 bit weights and 4 bit activations [Don+19]. Consequently, it is our opinion that it is still important to focus on supporting secure inference for general neural networks even though BNNs appear promising.

Any GC-based protocol can be upgraded to malicious security through a combination of cut-and-choose techniques [Zhu+16] and malicious OT-extension [Kel+15], and client-malicious security for the evaluator by using malicious OT-extension. Thus, it would follow that all of these GC-based inference protocols can be transformed into malicious and fixed-subset malicious protocols. Note that DeepSecure would provide *server-malicious* security since the client garbles the circuit. XONN uses a specialized protocol to evaluate the first layer of the network since the client's input is an un-quantized integer. In order for these malicious/client-malicious transformations to work, XONN would need to evaluate this layer within the more-expensive garbled circuit, instead of their optimized protocol.

MUSE's online speed is already superior to the *semi-honest* version of DeepSecure and the protocol of [Bal+19]. Furthermore, an open-source implementation of XONN was not available at the time of writing, so we could not benchmark a client-malicious version of their protocol on our experimental setup.

8 Future Work

We see a number of exciting directions for future work:

- We only focused on optimizing triple generation and input authentication in Overdrive since we used them in MUSE. What other MPC protocols can be optimized for the fixed-subset malicious threat model?
- Could alternate techniques for the non-linear layers such as polynomial activations as in DELPHI [Mis+20] or the techniques in CrypTFlow2 [Rat+20] have efficient client-malicious extensions?
- The main bottleneck in our CDS protocol (a multiplication of bits) is a binary operation being done in an arithmetic domain. We require the arithmetic domain to do the MAC check, but could techniques to switch between binary and arithmetic domains [Esc+20; Pat+20; Aly+19] result in a faster overall protocol?

References

[Aly+19] A. Aly, E. Orsini, D. Rotaru, N. P. Smart, and T. Wood. “Zaphod: Efficiently Combining LSSS and Garbled Circuits in SCALE”. In: WAHC’19.

[App19] Apple. “iOS Security”. https://www.apple.com/business/docs/site/iOS_Security_Guide.pdf.

[Bal+19] M. Ball, B. Carmer, T. Malkin, M. Rosulek, and N. Schimanski. “Garbled Neural Networks are Practical”. ePrint Report 2019/338.

[Ban+18] R. Banner, I. Hubara, E. Hoffer, and D. Soudry. “Scalable methods for 8-bit training of neural networks”. In: NeurIPS ’18.

[Bel+12] M. Bellare, V. T. Hoang, and P. Rogaway. “Foundations of garbled circuits”. In: CCS ’12.

[Bit+13] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth. “Succinct Non-interactive Arguments via Linear Interactive Proofs”. In: TCC ’13.

[Bou+18] F. Bourse, M. Minelli, M. Minihold, and P. Paillier. “Fast Homomorphic Evaluation of Deep Discretized Neural Networks”. In: CRYPTO ’18.

[Boy+17] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, and M. Orrú. “Homomorphic Secret Sharing: Optimizations and Applications”. In: CCS ’17.

[Bra+17] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. “Software Grand Exposure: SGX Cache Attacks Are Practical”. In: WOOT ’17.

[Bru+18] A. Brutzkus, O. Elisha, and R. Gilad-Bachrach. “Low Latency Privacy Preserving Inference”. ArXiV, cs.CR 1812.10659.

[Car+20] N. Carlini, M. Jagielski, and I. Mironov. “Cryptanalytic Extraction of Neural Network Models”. In: CRYPTO ’20.

[Cha+17] H. Chabanne, A. de Wargny, J. Milgram, C. Morel, and E. Prouff. “Privacy-Preserving Classification on Deep Neural Network”. ePrint Report 2017/035.

[Cha+19] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi. “EzPC: Programmable and Efficient Secure Two-Party Computation for Machine Learning”. In: EuroS&P ’19.

[Che+20] 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 ’20.

[Cho+18] E. Chou, J. Beal, D. Levy, S. Yeung, A. Haque, and L. Fei-Fei. “Faster CryptoNets: Leveraging Sparsity for Real-World Encrypted Inference”. ArXiV, cs.CR 1811.09953.

[Cou+15] M. Courbariaux, Y. Bengio, and J. David. “BinaryConnect: Training Deep Neural Networks with binary weights during propagations”. In: NeurIPS ’18.

[Dam+12] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. “Multiparty Computation from Somewhat Homomorphic Encryption”. In: CRYPTO ’12.

[Dat+19] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. E. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz. “CHET: An optimizing compiler for fully-homomorphic neural-network inferencing”. In: PLDI ’19.

[Don+19] Z. Dong, Z. Yao, A. Gholami, M. Mahoney, and K. Keutzer. “HAWQ: Hessian AWare Quantization of Neural Networks With Mixed-Precision”. In: ICCV ’19.

[Elg85] T. Elgamal. “A public key cryptosystem and a signature scheme based on discrete logarithms”. In: *IEEE Trans. on Inf. Theory* (1985).

[Esc+20] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl. “Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits”. In: CRYPTO ’20.

[Gen09a] C. Gentry. “A Fully Homomorphic Encryption Scheme”. PhD thesis. Stanford University, 2009.

[Gen09b] C. Gentry. “Fully homomorphic encryption using ideal lattices”. In: STOC ’09.

[Gil+16] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing. “CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy”. In: ICML ’16.

[Gol+89] S. Goldwasser, S. Micali, and C. Rackoff. “The Knowledge Complexity of Interactive Proof Systems”. In: *SIAM J. Comput.* (1989).

[Goo17] Google. *Google Infrastructure Security Design Overview*. Tech. rep. 2017.

[Göt+17] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. “Cache Attacks on Intel SGX”. In: EUROSEC ’17.

[Häh+17] M. Hähnel, W. Cui, and M. Peinado. “High-Resolution Side Channels for Untrusted Operating Systems”. In: ATC ’2017.

[Han+18] L. Hanzlik, Y. Zhang, K. Grosse, A. Salem, M. Augustin, M. Backes, and M. Fritz. "ML-Capsule: Guarded Offline Deployment of Machine Learning as a Service". ArXiV, cs.CR 1808.00590.

[Hes+17] E. Hesamifard, H. Takabi, and M. Ghasemi. "CryptoDL: Deep Neural Networks over Encrypted Data". ArXiV, cs.CR 1711.05189.

[Jag+20] M. Jagielski, N. Carlini, D. Berthelot, A. Kurakin, and N. Papernot. "High Accuracy and High Fidelity Extraction of Neural Networks". In: USENIX Security '20.

[Juu+19] M. Juuti, S. Szyller, S. Marchal, and N. Asokan. "PRADA: Protecting Against DNN Model Stealing Attacks". In: EuroS&P '19.

[Juv+18] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. "GAZELLE: A Low Latency Framework for Secure Neural Network Inference". In: USENIX Security '18.

[Kat+18] J. Katz, S. Ranellucci, M. Rosulek, and X. Wang. "Optimizing Authenticated Garbling for Faster Secure Two-Party Computation". In: CRYPTO '18.

[Kel+15] M. Keller, E. Orsini, and P. Scholl. "Actively Secure OT Extension with Optimal Overhead". In: CRYPTO '15.

[Kel+18] M. Keller, V. Pastro, and D. Rotaru. "Overdrive: Making SPDZ Great Again". In: EUROCRYPT '18.

[Kel20] M. Keller. "MP-SPDZ: A Versatile Framework for Multi-Party Computation". In: CCS '20.

[Kes+18] M. Kesarwani, B. Mukhoty, V. Arya, and S. Mehta. "Model Extraction Warning in MLaaS Paradigm". In: ACSAC '18.

[Kri18] R. Krishnamoorthi. "Quantizing deep convolutional networks for efficient inference: A whitepaper". arXiv: 1806.08342 [cs.LG].

[Kum+20] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma. "CrypTFlow: Secure TensorFlow Inference". In: S&P '20.

[Lee+19] T. Lee, B. Edwards, I. Molloy, and D. Su. "Defending Against Neural Network Model Stealing Attacks Using Deceptive Perturbations". In: SP Workshop '19.

[Lin+09] Y. Lindell and B. Pinkas. "A Proof of Security of Yao's Protocol for Two-Party Computation". In: *J. Cryptol.* (2009).

[Lin+15] Y. Lindel and P. Benny. "An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries". In: *J. Cryptol.* (2015).

[Liu+17a] J. Liu, M. Juuti, Y. Lu, and N. Asokan. "Oblivious Neural Network Predictions via MiniONN Transformations". In: CCS '17.

[Liu+17b] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi. "A survey of deep neural network architectures and their applications". In: *Neurocomputing* (2017).

[Lou+19] Q. Lou and L. Jiang. "SHE: A Fast and Accurate Deep Neural Network for Encrypted Data". ArXiV, cs.CR 1906.00148.

[Mil+19] S. Milli, L. Schmidt, A. D. Dragan, and M. Hardt. "Model Reconstruction from Model Explanations". In: FAT* '19.

[Mis+20] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa. "Delphi: A Cryptographic Inference Service for Neural Networks". In: USENIX Security '20.

[Mog+17] A. Moghimi, G. Irazoqui, and T. Eisenbarth. "CacheZoom: How SGX Amplifies the Power of Cache Attacks". In: CHES '17.

[Moh+17] P. Mohassel and Y. Zhang. "SecureML: A System for Scalable Privacy-Preserving Machine Learning". In: S&P '17.

[Moh+18] P. Mohassel and P. Rindal. "ABY^3: A Mixed Protocol Framework for Machine Learning". In: CCS '18.

[Pai99] P. Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: EUROCRYPT '99.

[Pat+20] A. Patra, T. Schneider, A. Suresh, and H. Yalame. "ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation". ePrint Report 2020/1225.

[Rab81] M. O. Rabin. "How To Exchange Secrets with Oblivious Transfer". Harvard University Technical Report 81 (TR-81).

[Rat+20] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma. "CrypTFlow2: Practical 2-Party Secure Inference". In: CCS '20.

[Ria+18] 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 '18.

[Ria+19] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. Lauter, and F. Koushanfar. “XONN: XNOR-based Oblivious Deep Neural Network Inference”. In: USENIX Security ’19.

[Rol+20] D. Rolnick and K. P. Körding. “Reverse-Engineering Deep ReLU Networks”. In: ICML ’20.

[Rot+19] D. Rotaru and T. Wood. “MArBled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security”. In: INDOCRYPT ’19.

[Rou+18] B. D. Rouhani, M. S. Riazi, and F. Koushanfar. “DeepSecure: Scalable Provably-secure Deep Learning”. In: DAC ’18.

[San+18] A. Sanyal, M. Kusner, A. Gascón, and V. Kanade. “TAPAS: Tricks to Accelerate (encrypted) Prediction As a Service”. In: ICML ’18.

[Sch+17] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. “Malware Guard Extension: Using SGX to Conceal Cache Attacks”. In: DIMVA ’17.

[Sea] “Microsoft SEAL (release 3.3)”. https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

[Top+18] S. Tople, K. Grover, S. Shinde, R. Bhagwan, and R. Ramjee. “Privado: Practical and Secure DNN Inference”. ArXiV, cs.CR 1810.00602.

[Tra+16] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. “Stealing Machine Learning Models via Prediction APIs”. In: USENIX Security ’16.

[Tra+19] F. Tramer and D. Boneh. “Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware”. In: ICLR ’19.

[Van+18] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”. In: USENIX Security ’18.

[Wag+19] S. Wagh, D. Gupta, and N. Chandran. “SecureNN: 3-Party Secure Computation for Neural Network Training”. In: *Proc. Priv. Enhancing Technol.* (2019).

[Wag+21] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin. “Falcon: Honest-Majority Maliciously Secure Framework for Private Deep Learning”. In: *Proc. Priv. Enhancing Technol.* (2021).

[Wan+17] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. “Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX”. In: CCS ’17.

[Yao86] A. C. Yao. “How to Generate and Exchange Secrets (Extended Abstract)”. In: FOCS ’86.

[Zhu+16] R. Zhu, Y. Huang, J. Katz, and a. shelat. “The Cut-and-Choose Game and Its Application to Cryptographic Protocols”. In: USENIX Security ’16.

A Security properties of our building blocks

Let λ denote the security parameter. For a finite set S, we denote $x \leftarrow S$ as the process of sampling x uniformly from the set S. We say that a function $\mu : \mathbb{N} \rightarrow [0,1]$ is negligible if for every polynomial $p(\cdot)$, there exists λ_0 such that for all $\lambda > \lambda_0$, $\mu(\lambda) < 1/p(\lambda)$. We denote an unspecified negligible function with negl and an unspecified polynomial with poly. We say that two distribution ensembles $X = \{X_\lambda\}_{\lambda \in \mathbb{N}}$ and $Y = \{Y_\lambda\}_{\lambda \in \mathbb{N}}$ are computationally indistinguishable (denoted by $X \approx_c Y$) if for every polynomial time distinguisher D, there exists a negligible function $\mu(\cdot)$ such that for every λ, we have $|\Pr[D(X_\lambda) = 1] - \Pr[D(Y_\lambda) = 1]| \leq \mu(\lambda)$.

A.1 Garbling Scheme

We would like a garbling scheme GS to satisfy two properties: *correctness* and *security*.

- *Correctness:* The correctness requirement says that the string y output by the Eval algorithm is same as $C(x)$ with probability 1.
- *Security:* The security requirement says that given $\widetilde{C}$ and $\{\mathsf{lab}_{i,x_i}\}$, the evaluator does not learn anything about C or x except the size of the circuit C (denoted by $1^{|C|}$) and the output $C(x)$. Specifically, we require the existence of a simulator $\mathsf{Sim}_{\mathsf{GS}}$ such that for any adversary $\mathcal{A}$,

$$\Pr[\mathsf{Expt}_{\mathsf{GS}}(\mathcal{A}, \mathsf{GS}) = 1] \leq 1/2 + \mathsf{negl}(\lambda)$$

where $\mathsf{Expt}_{\mathsf{GS}}$ is defined below.

```
$(C, x, \{\mathsf{lab}_{i,x_i}\}_{i \in [n]}) \leftarrow \mathcal{A}(1^\lambda)$
$\mathsf{lab}_{i,1-x_i} \leftarrow \{0,1\}^\lambda$ for $i \in [n]$
$b \leftarrow \{0,1\}$
If $b = 0 : \widetilde{C} \leftarrow \mathsf{GS.Garble}(1^\lambda, C, \{\mathsf{lab}_{i,0}, \mathsf{lab}_{i,1}\}_{i \in [n]})$
Else, $\widetilde{C} \leftarrow \mathsf{Sim}_{\mathsf{GS}}(1^\lambda, 1^{|C|}, C(x), \{\mathsf{lab}_{i,x_i}\}_{i \in [n]})$
$b' \leftarrow \mathcal{A}(\widetilde{C})$
Output 1 iff $b = b'$
```

The above requirement where $\mathcal{A}$ chooses the labels corresponding to its input is stronger than the standard definition where in both the labels for each input wire are chosen uniformly. However, we note that the existing constructions of garbled circuits (such as the one described in [Lin+09]) satisfies this stronger definition.

A.2 Linearly Homomorphic Public Key Encryption

We require a linear homomorphic public key encryption to satisfy the following properties:

- *Correctness.* With probability 1, the output of the decryption algorithm Dec on input sk and a ciphertext $c = \mathsf{Enc}(\mathsf{pk}, m)$ is equal to m. Here, the probability is over the random coins of KeyGen and Enc algorithms.
- *Homomorphic Evaluation Correctness.* With probability 1, the output of the decryption algorithm Dec on input sk and a ciphertext $c' = \mathsf{Eval}(\mathsf{pk}, \mathsf{Enc}(\mathsf{pk}, m_1), \mathsf{Enc}(\mathsf{pk}, m_2), L)$ is equal to $L(m_1, m_2)$. Here, the probability is over the random coins of KeyGen and Enc algorithms.
- *Semantic Security.* For any two messages $m, m' \in \mathbb{Z}_p$, we require

$$\{\mathsf{pk}, \mathsf{Enc}(\mathsf{pk}, m)\} \approx_c \{\mathsf{pk}, \mathsf{Enc}(\mathsf{pk}, m')\}$$

where the two distributions are over the random choice of pk and the random coins of the encryption algorithm.
- *Function Privacy.* There exists a simulator $\mathsf{Sim}_{\mathsf{FP}}$ such that for every polynomial time adversary $\mathcal{A}$ and every linear function L, we have:

$$\Pr[\mathsf{Expt}_{\mathsf{FP}}(\mathcal{A}, \mathsf{HE}, L) = 1] \leq 1/2 + \mathsf{negl}(\lambda)$$

where $\mathsf{Expt}_{\mathsf{FP}}$ is defined below.

```
$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda; r)$
$(m_1, r_1), (m_2, r_2) \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{sk})$
$c_i \leftarrow \mathsf{Enc}(\mathsf{pk}, m_i; r_i)$ for $i \in \{1,2\}$
$b \leftarrow \{0,1\}$
If $b = 0 : c' \leftarrow \mathsf{Eval}(\mathsf{pk}, c_1, c_2, L)$
Else, $c' \leftarrow \mathsf{Sim}_{\mathsf{FP}}(\mathsf{pk}, (m_1, r_1), (m_2, r_2))$
$b' \leftarrow \mathcal{A}(c')$
Output 1 iff $b = b'$
```

A.3 Message Authentication Codes

We want the MAC scheme to satisfy the following two properties:

- *Correctness.* For any message m, we want the probability that $\mathsf{Verify}(\alpha, \mathsf{st}, m, \sigma) = 1$ is 1 for $\alpha \leftarrow \mathsf{KeyGen}(1^\lambda)$, $(\sigma, \mathsf{st}) \leftarrow \mathsf{Tag}(\alpha, m)$.
- *One-time Security.* For any message m and for any adversary $\mathcal{A}$, we have:

$$\Pr[\mathsf{Expt}_{\mathsf{MAC}}(\mathcal{A}, m, \mathsf{MAC}) = 1] \leq \mathsf{negl}(\lambda)$$

where $\mathsf{Expt}_{\mathsf{MAC}}$ is defined below.

```
$\alpha \leftarrow \mathsf{KeyGen}(1^\lambda)$
$(\sigma, \mathsf{st}) \leftarrow \mathsf{Tag}(\alpha, m)$
$(m', \sigma') \leftarrow \mathcal{A}(m, \sigma)$
Output 1 iff $(m', \sigma') \neq (m, \sigma) \wedge \mathsf{Verify}(\alpha, \mathsf{st}, m', \sigma') = 1$.
```

Lemma A.1. *The scheme described in Section 3 is a one-time secure MAC.*

Proof. The correctness is easy to observe. Assume for the sake of contradiction that adversary $\mathcal{A}$ produces a (m', σ') such that $(m', \sigma') \neq (m, \sigma) \wedge \mathsf{Verify}(\alpha, \mathsf{st}, m', \sigma') = 1$. This means that $\sigma' + \mathsf{st} = \alpha \cdot m'$. By definition of Tag, it follows that $\sigma + \mathsf{st} = \alpha \cdot m$. Subtracting these two equations, we get $\alpha(m - m') = \sigma - \sigma'$. If $m = m'$, then this equation is satisfied only if $\sigma = \sigma'$ which contradicts $(m', \sigma') \neq (m, \sigma)$. Else, we get $\alpha = (\sigma_i - \sigma'_i)/(m_i - m'_i)$ for some $i \in [n]$ such that $m_i \neq m'_i$ and the probability that α satisfies this equation is $1/p \leq 2^{-\lambda}$. □

A.4 Zero-Knowledge Proofs

We want our zero-knowledge proof system to satisfy the following properties:

- *Completeness.* For any $x \in L$, the prover always convinces the verifier.
- *Soundness.* For any $x \notin L$, the verifier rejects with overwhelming probability.
- *Proof of Knowledge.* For every polynomial time malicious prover P^*, there exists an extractor Ext such that for any statement x, if the verifier accepts the proof on interaction with the prover P^* with probability at least $1/\mathsf{poly}(\lambda)$ then there exists a negligible function μ such that $\Pr[\mathsf{Ext}(x) = w \wedge (x, w) \in L] \geq 1/\mathsf{poly}(\lambda) - \mu(\lambda)$.
- *Zero-Knowledge:* For any $x \in L$ and any polynomial time malicious verifier V^*, there exists a simulator $\mathsf{Sim}_{\mathsf{ZK}}$ such that the transcript of the interaction between the honest P and V^* is computationally indistinguishable to the transcript generated by $\mathsf{Sim}_{\mathsf{ZK}}$.

Functionality $\mathcal{F}_{\text{ACG}}$

1. On client input $\{\mathbf{r}_i\}_{i \in [\ell]}$, server input $\{\mathbf{s}_i, \mathbf{M}_i, \alpha_i, \beta_i\}_{i \in [\ell]}$, compute $\{\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i, \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i), \beta_i \cdot \mathbf{r}_i\}_{i \in [\ell]}$.
2. Secret share $\alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)$ to $\langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i) \rangle_1, \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i) \rangle_2$ and $\beta_i \cdot \mathbf{r}_i$ to $\langle \beta_i \cdot \mathbf{r}_i \rangle_1, \langle \beta_i \cdot \mathbf{r}_i \rangle_2$.
3. Output $(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i, \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i) \rangle_1, \langle \beta_i \cdot \mathbf{r}_i \rangle_1)$ to the client, and $(\langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i) \rangle_2, \langle \beta_i \cdot \mathbf{r}_i \rangle_2)$ to the server.

Figure 11: The ideal functionality for Authenticated Correlations Generator

Functionality $\mathcal{F}_{\text{CDS}}$

1. The client and server input $(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i, \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i) \rangle_1, \mathbf{r}_{i+1}, \langle \beta_{i+1} \cdot \mathbf{r}_{i+1} \rangle_1)$ and $(\alpha_i, \beta_{i+1}, \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i) \rangle_2, \langle \beta_{i+1} \cdot \mathbf{r}_{i+1} \rangle_2, \{\mathsf{lab}^C_{i,k,0}, \mathsf{lab}^C_{i,k,1}\}_{k \in [|\mathsf{inp}_i|]})$ respectively for some $i \in [\ell]$.
2. If $\alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i) = \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i) \rangle_1 + \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i) \rangle_2$ and $\beta_{i+1} \cdot \mathbf{r}_{i+1} = \langle \beta_{i+1} \cdot \mathbf{r}_{i+1} \rangle_1 + \langle \beta_{i+1} \cdot \mathbf{r}_{i+1} \rangle_2$, output the labels $\{\mathsf{lab}^C_{i,k,0}, \mathsf{lab}^C_{i,k,1}\}_{k \in [|\mathsf{inp}_i|]}$ corresponding to $\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i$ and $\mathbf{r}_{i+1}$ to the client. Otherwise, abort.

Figure 12: The ideal functionality for Conditional Disclosure of Secrets

$\mathcal{F}_{\text{InputAuth}}$

- The client's input is $\mathbf{m}_c$ and the server's input is $\mathbf{m}_s$ and a MAC key δ. The client receives $[\![\mathbf{m}_c]\!]_1, [\![\mathbf{m}_s]\!]_1$ and the server receives $[\![\mathbf{m}_c]\!]_2, [\![\mathbf{m}_s]\!]_2$.

Figure 13: Description of $\mathcal{F}_{\text{InputAuth}}$.

$\mathcal{F}_{\text{Rand}}$

- The client's input is $\mathbf{r}_1$ and the server's input is $\mathbf{r}_2$ and a MAC key δ. The client receives $[\![\mathbf{r}_1 + \mathbf{r}_2]\!]_1$ and the server receives $[\![\mathbf{r}_1 + \mathbf{r}_2]\!]_2$.

Figure 14: Description of $\mathcal{F}_{\text{Rand}}$.

B Security proofs

We first give the formal definition of a protocol that is secure against malicious clients and semi-honest server.

Definition B.1. *A protocol Π between a server and a client is said to securely compute a function f against a malicious client and semi-honest server if it satisfies the following properties:*

- ***Correctness.*** *For any server's input $\mathbf{y}$ and client's input $\mathbf{x}$, the probability that at the end of the protocol, the client outputs $f(\mathbf{y}, \mathbf{x})$ is 1.*

- ***Semi-Honest Server Security.*** *For any server S that follows the protocol, there exists a simulator Sim_S such that for any input $\mathbf{y}$ of the server and $\mathbf{x}$ of the client, we have:*

$$\mathsf{view}_S(\mathbf{y}, \mathbf{x}) \approx_c \mathsf{Sim}_S(\mathbf{y})$$

  In other words, Sim_S is able to generate a view of the semi-honest server without knowing the client's private input.

- ***Malicious Client Security.*** *For any malicious client C (that might deviate arbitrarily from the protocol specification), there exists a simulator Sim_C such that for any input $\mathbf{y}$ of the server, we have:*

$$\mathsf{view}_C(\mathbf{y}) \approx_c \mathsf{Sim}_C^{f(\mathbf{y}, \cdot)}$$

  In other words, the Sim_C is able to generate the view of a malicious client with only access to an ideal functionality that accepts a client's input and outputs the result of the function f. This modeling is used in crytographic literature to capture the cases where a malicious client may substitute its actual input with any other input of its choice.

$\mathcal{F}_{\text{Triple}}$

- The client's input is a_1, b_1 and the server's input is a_2, b_2 and a MAC key δ. The client receives $[\![a_1+a_2]\!]_1, [\![b_1+b_2]\!]_1, [\![(a_1+a_2)\cdot(b_1+b_2)]\!]_1$ and the server receives $[\![a_1+a_2]\!]_2, [\![b_1+b_2]\!]_2, [\![(a_1+a_2)\cdot(b_1+b_2)]\!]_2$.

Figure 15: Description of $\mathcal{F}_{\text{Triple}}$.

Protocol $\Pi_{\text{InputAuth}}$

1. Both parties invoke $\mathcal{F}_{\text{Rand}}$ to receive $|\mathbf{m}_c|$ random shares $[\![\mathbf{r}]\!]$
2. $\mathbf{r}$ is privately opened to the client.
3. The client broadcasts $\varepsilon = \mathbf{m}_c - \mathbf{r}$.
4. The server's share is $[\![\mathbf{m}_c]\!]_2 = (\varepsilon, [\![r]\!]_2)$ and the client's share is $[\![\mathbf{m}_c]\!]_1 = (\varepsilon, [\![\mathbf{r}]\!]_1)$.
5. The server chooses two masking vectors $\mathbf{u}, \mathbf{v}$ and sends the client $[\![\mathbf{m}_s]\!]_1 = (\mathbf{m}_s - \mathbf{u}, \delta\cdot\mathbf{m}_s - \mathbf{v})$. The server sets its share $[\![\mathbf{m}_s]\!]_2 = (\mathbf{u}, \mathbf{v})$

Figure 16: The protocol for Input Authentication.

Definition B.2. *We say that Π is a secure inference protocol against malicious clients and semi-honest servers if it securely computes $\mathsf{NN}(\cdot,\cdot)$ with the server input being $\mathbf{M}$ and the client input being $\mathbf{x}$.*

B.1 ACG

Lemma B.3. *Assuming a secure two-party computation protocol against malicious clients and semi-honest servers for* KeyGen*, a linearly homomorphic encryption scheme (see Section 3) and a zero-knowledge proof system for well-formedness of ciphertexts (see Section 3), the protocol described in Fig. 5 securely computes $\mathcal{F}_{ACG}$ against malicious clients and semi-honest servers.*

Proof. The correctness follows from observation and we now prove semi-honest server security and then show security against malicious clients.

Semi-Honest server security. The simulator Sim_S samples $(pk, sk) \leftarrow \mathsf{HE.KeyGen}(1^\lambda)$. It then runs the simulator for the semi-honest server in the two-party computation protocol that generates the public-key, secret-key pair for the homomorphic encryption scheme. When this simulator access the ideal functionality, Sim_S provides pk as the output. For each $i \in [\ell]$, Sim_S sends $\{\mathsf{HE.Enc}(pk, \mathbf{0})\}_{i\in[\ell]}$ and with a simulated proof of well-formedness of ciphertexts. We now show the indistinguishability of the real view from the simulated view using a hybrid argument.

Protocol Π_{Triple}

i. The client and the server engage in a two-party computation protocol with security against malicious clients and semi-honest servers to generate the public key, secret key pair for HE. At the end of the protocol, the client learns the public key pk and the secret key sk whereas the server only learns pk.

ii. The client sends $\mathsf{Enc}(\mathsf{pk}, a_1), \mathsf{Enc}(\mathsf{pk}, b_1), \mathsf{Enc}(\mathsf{pk}, a_1\cdot b_1)$ to the server along with a zero-knowledge proof of well-formedness of the three ciphertexts and the fact that the third ciphertext encrypts the product of the two elements encrypted in the first two ciphertexts. The server verifies this proof before continuing.

iii. The server homomorphically computes $\mathsf{Enc}(\mathsf{pk}, a_1+a_2)$, $\mathsf{Enc}(\mathsf{pk}, b_1+b_2)$ and $\mathsf{Enc}(\mathsf{pk}, (a_1+a_2)\cdot(b_1+b_2))$ along with $\mathsf{Enc}(\mathsf{pk}, \delta(a_1+a_2))$, $\mathsf{Enc}(\mathsf{pk}, \delta(b_1+b_2))$ and $\mathsf{Enc}(\mathsf{pk}, \delta\cdot(a_1+a_2)\cdot(b_1+b_2))$.

iv. The server chooses six random masking elements $u_1, v_1, t_1, u_2, v_2, t_2$ and computes $\mathsf{Enc}(\mathsf{pk}, a_1+a_2-u_1)$, $\mathsf{Enc}(\mathsf{pk}, b_1+b_2-v_1)$ and $\mathsf{Enc}(\mathsf{pk}, (a_1+a_2)\cdot(b_1+b_2)-t_1)$ along with $\mathsf{Enc}(\mathsf{pk}, \delta(a_1+a_2)-u_2)$, $\mathsf{Enc}(\mathsf{pk}, \delta(b_1+b_2)-v_2)$ and $\mathsf{Enc}(\mathsf{pk}, \delta\cdot(a_1+a_2)\cdot(b_1+b_2)-t_2)$. It sends these six ciphertexts to the client.

v. The client decrypts the above ciphertexts and obtains $[\![a_1+a_2]\!]_1 = (a_1+a_2-u_1, \delta(a_1+a_2)-u_2), [\![b_1+b_2]\!]_1 = (b_1+b_2-v_1, \delta(b_1+b_2)-v_2), [\![(a_1+a_2)\cdot(b_1+b_2)]\!]_1 = (a_1+a_2)\cdot(b_1+b_2)-t_1, \delta\cdot(a_1+a_2)\cdot(b_1+b_2)-t_2)$. The server outputs $[\![a_1+a_2]\!]_2 = (u_1, u_2), [\![b_1+b_2]\!]_2 = (v_1, v_2), [\![(a_1+a_2)\cdot(b_1+b_2)]\!]_2 = (t_1, t_2)$.

Figure 17: The protocol for Triple Generation.

- Hyb_1 : This corresponds to the real execution of the protocol.
- Hyb_2 : In this hybrid, we use the simulator for the semi-honest server in the two-party computation protocol for generating the public key, secret key pair. When this simulator queries its ideal functionality, we sample a $(pk, sk) \leftarrow \mathsf{HE.KeyGen}(1^\lambda)$ and give pk to it. This hybrid is computationally indistinguishable to Hyb_1 from the security of this protocol.
- Hyb_3 : In this hybrid, we start using the simulator for the zero-knowledge proofs of plaintext knowledge. This hybrid is indistinguishable to the previous one from the zero-knowledge property of these proof systems.
- Hyb_4 : In this hybrid, we switch from sending $\mathsf{HE.Enc}(pk, \mathbf{r}_i)$ for each $i \in [\ell]$ to sending $\mathsf{HE.Enc}(pk, \mathbf{0})$. This hybrid is indistinguishable from the previous one from the semantic security of the homomorphic encryption. Note that view of server in Hyb_4 is identical to the output of Sim_S.

Malicious Client Security. Sim_C samples a $(pk, sk) \leftarrow \mathsf{HE.KeyGen}(1^\lambda)$. Sim_C runs the simulator for the malicious client in the two-party computation protocol that generates the public-key, secret-key pair for the homomorphic encryption scheme. When this simulator access the ideal functionality, Sim_C provides (pk, sk) as the output. On receiving the ciphertext $\{\mathsf{HE.Enc}(pk, \mathbf{r}_i)\}_{i\in[\ell]}$ and the proof of plaintext knowledge from the client, Sim_C first checks if the proof is valid. If yes, Sim_C then runs the knowledge extractor Ext on the ciphertext and the proof to extract $\mathbf{r}_i$ and the randomness used for generating the ciphertext encrypting $\mathbf{r}_i$ for each $i \in [\ell]$. Sim_C queries the ideal functionality $\mathcal{F}_{\mathrm{ACG}}$ on input $\mathbf{r}_i$ and obtains $(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i, \langle\alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_1, \langle\beta_i \cdot \mathbf{r}_i\rangle_1)$. It then runs $\mathsf{Sim}_{\mathsf{FP}}$ on the output $(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i, \langle\alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_1, \langle\beta_i \cdot \mathbf{r}_i\rangle_1)$ along with $\mathbf{r}_i$ and the randomness used to generate the ciphertext and obtains the simulated ciphertexts. It sends the simulated ciphertexts to the client.

We now show the view of the malicious client in the real protocol is computationally indistinguishable to the view generated by Sim_C via a hybrid argument.

- Hyb_1 : This corresponds to the view of the malicious client in the real execution of the protocol.

- Hyb_2 : In this hybrid, we use the simulator for the malicious client in the two-party computation protocol for KeyGen. When this simulator queries its ideal functionality, we sample a $(pk, sk) \leftarrow \mathsf{HE.KeyGen}(1^\lambda)$ and give (pk, sk) to it. This hybrid is computationally indistinguishable to Hyb_1 from the security of this protocol.

- Hyb_3 : In this hybrid, the server checks if the zero-knowledge proofs are valid and in that case, it uses the extractor Ext to extract $\mathbf{r}_i$ and the randomness used for generating the ciphertext encrypting $\mathbf{r}_i$ for each $i \in [\ell]$. Since the verifier in a non-interactive zk proof is deterministic, it follows from the proof of knowledge property that the extractor Ext succeeds in extracting a witness except with negligible probability.

- Hyb_4 : In this hybrid, we use the simulator for the function privacy of the homomorphic encryption in generating $\mathsf{HE.Enc}(\mathsf{pk}, \mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)$, $\mathsf{HE.Enc}(\mathsf{pk}, \langle\alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i)\rangle_1)$ and $\mathsf{HE.Enc}(\mathsf{pk}, \langle\beta_i \cdot \mathbf{r}_i\rangle_1)$ for each $i \in [\ell]$. This hybrid is computationally indistinguishable to the previous hybrid from the function privacy of the HE scheme. Note that view of the client in Hyb_4 is identical to the view generated by Sim_C.

□

B.2 MPC Engine

Lemma B.4. *Assuming a secure two-party computation protocol against malicious clients and semi-honest servers for* KeyGen*, a linearly homomorphic encryption scheme (see Section 3) and a zero-knowledge proof system for* NP *(see Section 3), the protocol described in Fig. 17 securely computes* $\mathcal{F}_{Triple}$ *against malicious clients and semi-honest servers.*

Proof. We now prove semi-honest server security and then show security against malicious clients.

Semi-Honest server security. The simulator Sim_S samples $(pk, sk) \leftarrow \mathsf{HE.KeyGen}(1^\lambda)$. It then runs the simulator for the semi-honest server in the two-party computation protocol that generates the public-key, secret-key pair for the homomorphic encryption scheme. When this simulator access the ideal functionality, Sim_S provides pk as the output. Sim_S sends $\{\mathsf{HE.Enc}(pk, \mathbf{0})\}_{i\in[3]}$ and with a simulated proof of well-formedness of ciphertexts and the simulated zero-knowledge proof for the statement that the third ciphertext encrypts an element that is the product. We now show the indistinguishability of the real view from the simulated view using a hybrid argument.

- Hyb_1 : This corresponds to the real execution of the protocol.

- Hyb_2 : In this hybrid, we use the simulator for the semi-honest server in the two-party computation protocol for generating the public key, secret key pair. When this simulator queries its ideal functionality, we sample a $(pk, sk) \leftarrow \mathsf{HE.KeyGen}(1^\lambda)$ and give pk to it. This hybrid is computationally indistinguishable to Hyb_1 from the security of this protocol.

- Hyb_3 : In this hybrid, we start using the simulator for the zero-knowledge proofs of well-formedness of ciphertexts and the proof for the relation between the third ciphertext and the first two ciphertexts. This hybrid is indistinguishable to the previous one from the zero-knowledge property of these proof systems.

- Hyb_4 : In this hybrid, we switch from sending $\mathsf{HE.Enc}(pk, a_1), \mathsf{Enc}(\mathsf{pk}, b_1), \mathsf{Enc}(\mathsf{pk}, a_1 \cdot b_1)$ to sending $\{\mathsf{HE.Enc}(pk, \mathbf{0})\}_{i\in[3]}$. This hybrid is indistinguishable from the previous one from the semantic security of the homomorphic encryption. Note that view of server in Hyb_4 is identical to the output of Sim_S.

Malicious Client Security. Sim_C samples a $(pk, sk) \leftarrow \mathsf{HE.KeyGen}(1^\lambda)$. Sim_C runs the simulator for the malicious client in the two-party computation protocol that generates the public-key, secret-key pair for the homomorphic encryption scheme. When this simulator access the ideal functionality, Sim_C provides (pk, sk) as the output. On receiving the ciphertext $\mathsf{Enc}(\mathsf{pk}, a_1), \mathsf{Enc}(\mathsf{pk}, b_1), \mathsf{Enc}(\mathsf{pk}, a_1 \cdot b_1)$ and the corresponding zero-knowledge proofs from the client, Sim_C first checks if the proof is valid. If yes, Sim_C then runs the knowledge extractor Ext on the ciphertext and the proof to extract $a_1, b_1, a_1 \cdot b_1$ and the randomness used for generating these ciphertexts. Sim_C queries the ideal functionality $\mathcal{F}_{\mathrm{Triple}}$ on input a_1, b_1 and obtains $[\![a_1 + a_2]\!]_1 = (a_1 + a_2 - u_1, \delta(a_1 +$

$a_2) - u_2), [\![b_1 + b_2]\!]_1 = (b_1 + b_2 - v_1, \delta(b_1 + b_2) - v_2), [\![(a_1 + a_2) \cdot (b_1 + b_2)]\!]_1 = (a_1 + a_2) \cdot (b_1 + b_2) - t_1, \delta \cdot (a_1 + a_2) \cdot (b_1 + b_2) - t_2)$. It then runs $\mathsf{Sim}_{\mathsf{FP}}$ on the six outputs and the randomness used to generate the initial ciphertexts and obtains the six simulated ciphertexts. It sends the six simulated ciphertexts to the client.

We now show the view of the malicious client in the real protocol is computationally indistinguishable to the view generated by Sim_C via a hybrid argument.

- Hyb_1 : This corresponds to the view of the malicious client in the real execution of the protocol.

- Hyb_2 : In this hybrid, we use the simulator for the malicious client in the two-party computation protocol for KeyGen. When this simulator queries its ideal functionality, we sample a $(pk, sk) \leftarrow \mathsf{HE.KeyGen}(1^\lambda)$ and give (pk, sk) to it. This hybrid is computationally indistinguishable to Hyb_1 from the security of this protocol.

- Hyb_3 : In this hybrid, the server checks if the zero-knowledge proofs are valid and in that case, it uses the extractor Ext to extract $a_1, b_1, a_1 \cdot b_1$ and the randomness used for generating these ciphertexts. Since the verifier in a non-interactive zk proof is deterministic, it follows from the proof of knowledge property that the extractor Ext succeeds in extracting a witness except with negligible probability.

- Hyb_4 : In this hybrid, we use the simulator for the function privacy of the homomorphic encryption in generating $\mathsf{Enc}(\mathsf{pk}, a_1 + a_2 - u_1)$, $\mathsf{Enc}(\mathsf{pk}, b_1 + b_2 - v_1)$ and $\mathsf{Enc}(\mathsf{pk}, (a_1 + a_2) \cdot (b_1 + b_2) - t_1)$ along with $\mathsf{Enc}(\mathsf{pk}, \delta(a_1 + a_2) - u_2)$, $\mathsf{Enc}(\mathsf{pk}, \delta(b_1 + b_2) - v_2)$ and $\mathsf{Enc}(\mathsf{pk}, \delta \cdot (a_1 + a_2) \cdot (b_1 + b_2) - t_2)$. This hybrid is computationally indistinguishable to the previous hybrid from the function privacy of the HE scheme. Note that view of the client in Hyb_4 is identical to the view generated by Sim_C.

□

Note that $\mathcal{F}_{\text{Rand}}$ can be implemented using $\mathcal{F}_{\text{Triple}}$ and that security of Π_{Online} follows directly from the security of Beaver's protocol.

B.3 CDS

Lemma B.5. *Assuming a secure two-party computation protocol for the function f_{CDS} (see Fig. 6) against malicious clients and semi-honest servers, the protocol described in Fig. 6 secure computes $\mathcal{F}_{CDS}$ against malicious clients and semi-honest servers.*

Proof. The correctness of the protocol follows from observation and we now show semi-honest server security and then show security against malicious clients.

Semi-honest server security. To show this, we run the simulator for the protocol computing f_{CDS} to generate the client side messages for this protocol. When this simulator queries the ideal functionality, we give $\rho = \sigma = 0$ and an uniform share $\{\langle g_k \rangle_2\}_{k \in [|\mathsf{inp}_i|]}$ as the output. It follows from the security of the protocol that implements f_{CDS} that the above generated view is computationally indistinguishable to the real view.

Malicious Client Security. The simulator Sim_C against malicious client first runs the corresponding simulator for f_{CDS}. When this simulator access the ideal functionality f_{CDS} using the client's input $(\{b_k^i\}_{k \in [|\mathsf{inp}_i|]}, \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i) \rangle_1, \langle \beta_{i+1} \cdot \mathbf{r}_{i+1} \rangle_1)$, the Sim_C queries $\mathcal{F}_{\text{CDS}}$ on input $(\{b_k^i\}_{k \in [|\mathsf{inp}_i|]}, \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i) \rangle_1, \langle \beta_{i+1} \cdot \mathbf{r}_{i+1} \rangle_1)$ and obtains $\{\mathsf{lab}_{i,k,b_k^i}\}_{k \in |\mathsf{inp}_i|}$ or the special symbol abort. Sim_C answers the simulator for f_{CDS} with uniformly chosen shares and random tags as the output of the client. If the answer from $\mathcal{F}_{\text{CDS}}$ was abort, then Sim_C aborts at the end of step (ii). Else, in Step (ii), Sim_C sends $\{\mathsf{lab}_{i,k,b_k^i} - \langle g_k \rangle_1\}_{k \in |\mathsf{inp}_i|}$ as the final round message.

We now show that the view generated by Sim_C is computationally indistinguishable to the malicious client's view in the real protocol using a hybrid argument.

- Hyb_0 : This corresponds to the view of the malicious client in the real protocol execution.

- Hyb_1 : In this hybrid, we use the simulator for f_{CDS} and generate the view of the client in Step (i). When this simulator queries the ideal f_{CDS} functionality, we give uniformly chosen random shares $\{\langle g_k \rangle_1\}_{k \in |\mathsf{inp}_i|}$. We compute the output of the server using its actual inputs and the rest of the steps proceed exactly as in the protocol. This hybrid is computationally indistinguishable to Hyb_0 from the security of the protocol implementing the f_{CDS} functionality against malicious clients.

- Hyb_3 : In this hybrid, we query the $\mathcal{F}_{\text{CDS}}$ functionality on client input $(\{b_k^i\}_{k \in [|\mathsf{inp}_i|]}, \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i) \rangle_1, \langle \beta_{i+1} \cdot \mathbf{r}_{i+1} \rangle_1)$ and obtain the output $\{\mathsf{lab}_{i,k,b_k^i}\}_{k \in |\mathsf{inp}_i|}$ or the special symbol abort. If the output was not abort, then it means that $\sigma = \rho = 0$ and thus, we set $\langle g_k \rangle_2 = \mathsf{lab}_{i,k,b_k^i} - \langle g_k \rangle_1$ and send this as the final round message from the server. Otherwise, we abort at the end of Step (ii).

  Notice that the only difference between Hyb_3 and Hyb_2 is when the output of the $\mathcal{F}_{\text{CDS}}$ functionality is abort. We now argue that with that protocol in Hyb_2 also aborts at the end of Step (ii) with overwhelming probability. If the output of $\mathcal{F}_{\text{CDS}}$ is abort, then it means that either $\alpha_i(\overline{\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i}) \neq \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i) \rangle_1 + \langle \alpha_i(\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i) \rangle_2$ or $\beta_{i+1} \cdot \overline{\mathbf{r}_{i+1}} \neq \langle \beta_{i+1} \cdot \mathbf{r}_{i+1} \rangle_1 + \langle \beta_{i+1} \cdot \mathbf{r}_{i+1} \rangle_2$. Let us assume without loss of generality that $\beta_{i+1} \cdot \overline{\mathbf{r}_{i+1}} \neq \langle \beta_{i+1} \cdot \mathbf{r}_{i+1} \rangle_1 + \langle \beta_{i+1} \cdot \mathbf{r}_{i+1} \rangle_2$. It now follows from Schwartz-Zippel lemma that with overwhelming probability $\sigma = \beta_{i+1}\left\langle \mathbf{c}_2, \overline{\mathbf{r}_{i+1}} \right\rangle - \left\langle \mathbf{c}_2, \langle \beta_{i+1} \cdot \mathbf{r}_{i+1} \rangle_1 + \langle \beta_{i+1} \cdot \mathbf{r}_{i+1} \rangle_2 \right\rangle$ will not be equal to 0 and thus, the honest server will also abort in Hyb_2. We finally observe that the view of malicious client in Hyb_3 is identical to Sim_C.

□

B.4 MUSE Protocol

We first show security against a malicious client and then prove security against a semi-honest server.

Malicious Client. The simulator Sim_C does the following:

- **Offline Phase.** In the offline phase,
  1. **Authenticated Correlations Generator.** Sim_C runs the corresponding simulator for the protocol implementing $\mathcal{F}_{\mathrm{ACG}}$. When this simulator queries the ideal functionality on input $\{\mathbf{r}_i\}_{i\in[\ell]}$, Sim_C stores $\{\mathbf{r}_i\}_{i\in[\ell]}$ and answers the query with randomly chosen values $\{\mathbf{s}_i, \mathbf{t}_i, \mathbf{v}_i\}_{i\in[\ell]}$.
  2. For every $i \in [\ell]$, Sim_C samples random input labels $\{\mathsf{lab}^C_{i,k}\}_{k\in[|\mathsf{inp}_i|]}$.
  3. **Conditional Disclosure of Secrets.** Sim_C runs the corresponding simulator for the protocol implementing $\mathcal{F}_{\mathrm{CDS}}$. For every $i \in [\ell]$, when this simulator queries the ideal functionality on some input, Sim_C checks if this input is equal to $\mathbf{s}_i, \mathbf{t}_i, \mathbf{r}_{i+1}, \mathbf{v}_{i+1}$. If not, Sim_C gives $\perp$ as the answer from ideal functionality. Else, it outputs $\{\mathsf{lab}^C_{i,k}\}_{k\in[\mathsf{inp}_i]}$.
  4. **Offline Garbling.** For every $i \in [\ell]$, Sim_C sends a uniformly chosen random bit string D_i of size $|\widetilde{C}_i|$.
- **Online Phase.** In the online phase, the simulator does the following:
  - **Preamble:** On receiving, $\mathbf{x} - \mathbf{r}_1$, Sim_C extracts $\mathbf{x}$ from this and queries the neural network functionality on input $\mathbf{x}$ and learns the output $\mathbf{y}$.
  - **Layer Evaluation.** For every $i \in [\ell]$, Sim_C does the following:
    - **Non-Linear Layer.**
      1. Sim_C samples random input labels $\{\mathsf{lab}^S_{i,k}\}_{k\in[\mathsf{inp}_i]}$.
      2. The simulator computes $\widetilde{C}_i \leftarrow \mathsf{Sim}_{GC}(\{\mathsf{lab}^C_{i,k}, \mathsf{lab}^S_{i,k}\}_{k\in[|\mathsf{inp}_i|]}, (z_i, \overline{\mathsf{lab}}))$ where $(z_i, \overline{\mathsf{lab}})$ is an uniformly chosen random string from the output space of C_i. It chooses a uniform key $k_i \leftarrow \{0,1\}^\lambda$ and programs the RO to output $D_i \oplus \widetilde{C}_i$ on input k_i.
      3. Sim_C sends $\{k_i, \mathsf{lab}^S_{i,k}\}_{k\in[\mathsf{inp}_i]}$ to the client.
      4. The client evaluates the garbled circuit $\widetilde{C}$ using the labels $\{\mathsf{lab}^C_{i,k}, \mathsf{lab}^S_{i,k}\}_{k\in[\mathsf{inp}_i]}$ and sends (z_i', h).
      5. Sim_C checks if $z_i = z_i'$ and if it is not the case, it aborts. Otherwise, it proceeds to the next layer.
  - **Output Phase.** The simulator sends $z_\ell - \mathbf{y}$ to the client.

Proof of Indistinguishability. To prove that the view of a malicious client is indistinguishable to the above generated simulated view, we use a hybrid argument.

- Hyb_1 : This corresponds to the real execution of the protocol.
- Hyb_2 : In this hybrid, we use the simulator for the protocol implementing $\mathcal{F}_{\mathrm{ACG}}$. When this simulator queries the ideal functionality, we give $\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i$, $\langle \alpha_i(\mathbf{M}_i\mathbf{r}_i - \mathbf{s}_i)\rangle_1$ and $\langle \beta_i \cdot \mathbf{r}_i\rangle_1$ as the output. This hybrid is computationally indistinguishable from Hyb_1 from the security of the sub-protocol implementing $\mathcal{F}_{\mathrm{ACG}}$.
- Hyb_3 : In this hybrid, we run the simulator for the protocol implementing $\mathcal{F}_{\mathrm{CDS}}$. When this simulator queries its ideal functionality, we implement this functionality honestly and output the corresponding labels (or the special symbol $\perp$). This hybrid is computationally indistinguishable to Hyb_2 from the security of this sub-protocol.
- Hyb_4 : In this hybrid, when implementing the ideal functionality for the conditional disclosure of secrets, we check if the inputs provided by the simulator on behalf of the corrupted client are equal to $\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i$, $\langle \alpha_i(\mathbf{M}_i\mathbf{r}_i - \mathbf{s}_i)\rangle_1$, $\mathbf{r}_{i+1}$ and $\langle \beta_{i+1} \cdot \mathbf{r}_{i+1}\rangle_1$ instead of performing the MAC checks. This hybrid is statistically close to the previous one due to the MAC security (see Lemma A.1).
- Hyb_5 : In this hybrid, instead of sending $\widetilde{C}_i \oplus H(k_i)$ in the pre-processing phase, we send a random string D_i and in the online phase, we choose an uniform key $k_i \leftarrow \{0,1\}^\lambda$ and program the random oracle to output $D_i \oplus \widetilde{C}_i$. This hybrid is statistically close to the previous one because the probability that the adversary queries the oracle on an uniformly chosen k_i is negligible.
- Hyb_6 : In this hybrid, we make a syntactic change with respect to the previous hybrid. When the client sends $\mathbf{x} - \mathbf{r}_1$ to the server in the preamble of the online phase, we use the extracted value $\mathbf{r}_1$ and obtain $\mathbf{x}$. As a result, we find the output of the neural network in every layer, namely, $\mathbf{x}_1 = \mathbf{x}, \ldots, \mathbf{x}_{\ell+1} = \mathbf{y}$.
- Hyb_{6+i} : (for every $i \in [\ell]$) In this hybrid, we make the following changes when compared to Hyb_{6+i-1} in the specified order:

- Hyb'_1 : We generate $\widetilde{C}_i$ using the corresponding simulator for the garbled circuits. Specifically, we compute $\widetilde{C}_i \leftarrow \mathsf{Sim}_{GC}(\{\mathsf{lab}^C_{i,k}, \mathsf{lab}^S_{i,k}\}_{k\in[\mathsf{inp}_i]}, (z_i, \overline{\mathsf{lab}}))$ where $z_i, \overline{\mathsf{lab}}$ is the output of C_i on the server's input being equal to $\mathbf{M}_i(\mathbf{x}_i - \mathbf{r}_i) + \mathbf{s}_i, \mathbf{s}'_{i+1}$ and the client's input being equal to $\mathbf{r}_{i+1}$, $\mathbf{M}_i(\mathbf{r}_i) - \mathbf{s}_i$. This hybrid is computationally indistinguishable to Hyb_{6+i-1} from the security of garbled circuits (see Appendix A.1).
- Hyb'_2 : In this hybrid, we sample $z_i, \overline{\mathsf{lab}}$ uniformly at random instead of fixing them to be the output of C_i. When $i = \ell$, we send $z_\ell - \mathbf{y}$ as the final round message from the server to the client. This hybrid is identical to the previous hybrid from the one-time pad security (with the pad s'_{i+1}).
- Hyb'_3 : In this hybrid, on receiving z'_i, h from the client, the server checks if $z'_i = z_i$ and otherwise, it aborts. This hybrid is statistically close to the previous hybrid because for any position where z'_i and z_i differs, the label corresponding to the bit of z'_i in that position is uniformly distributed. Thus, the probability that the malicious client outputs the correct hash value on an uniformly chosen input is negligible.

- $\mathsf{Hyb}_{7+\ell}$: In this hybrid, we give randomly chosen vectors $\{\mathbf{s}_i, \mathbf{t}_i, \mathbf{v}_i\}_{i\in[\ell]}$ as output from $\mathcal{F}_{\text{ACG}}$. This hybrid is identical to the previous hybrid from the one-time pad security.

- $\mathsf{Hyb}_{8+\ell}$: In this hybrid, instead of computing the output $\mathbf{y}$ using the server's input matrices, we query the ideal functionality and obtain $\mathbf{y}$. This change is only syntactic and this hybrid is identical to the previous hybrid. Notice that $\mathsf{Hyb}_{8+\ell}$ is identical to the simulator.

Semi-honest server security. The simulator Sim_S does the following:

- It initializes the server on an uniform random tape.

- **Offline Phase.** In the offline phase,

  1. **Authenticated Correlations Generator.** For each $i \in [\ell]$, Sim_S uses the simulator for the semi-honest server in the protocol implementing $\mathcal{F}_{\text{ACG}}$ by giving randomly chosen shares as the output of the server.
  2. **Conditional Disclosure of Secrets.** Sim_S runs the simulator for the semi-honest server in the two-party computation protocol computing the $\mathcal{F}_{\text{CDS}}$.

- **Online Phase.** In the online phase, the Sim_S does the following:

  - **Preamble:** Send an uniformly chosen random vector $\mathbf{r}_1$ to the server.
  - **Layer Evaluation.** For every $i \in [\ell]$, Sim_S does the following:
    * **Non-Linear Layer.**
      1. Sim_S recovers the output labels of $\widetilde{C}_i$ since it had set the random tape for the semi-honest server.
      2. Sim_S chooses a uniform z'_i and computes the hash of the output labels h'_i corresponding to z'_i. It sends (z'_i, h'_i) to the corrupted server.

Proof of Indistinguishability.

- Hyb_1 : This corresponds to the real execution of the protocol.

- Hyb_2 : In this hybrid, we use the simulator for the protocol implementing the $\mathcal{F}_{\mathsf{ACG}}$ functionality by giving randomly chosen shares as the output of the server. This hybrid is computationally indistinguishable to the previous one from the security of this sub-protocol.

- Hyb_3 : In this hybrid, we use the knowledge of the input of the client, the server and the random tape to compute the correct output of every garbled circuit instead of evaluating them. This change is syntactic and is identical to the previous hybrid.

- Hyb_4 : In this hybrid, we run the simulator for the protocol implementing $\mathcal{F}_{\text{CDS}}$. This hybrid is computationally indistinguishable to Hyb_3 from the security of this protocol.

- Hyb_5 : In this hybrid, instead of using the knowledge of the client's input to compute the output of each garbled circuit, we choose a uniform z'_i and computes the hash of the output labels h'_i corresponding to z'_i. It sends (z'_i, h'_i) to the corrupted server. This hybrid is identically distributed to the previous one from the one-time pad security (with the pad $\mathbf{r}_{i+1}$). Note that Hyb_5 is identical to Sim_S.

C Pseudocode for our attacks from Section 2

```
RecoverLastLayer
1. Denote by $\tilde{M_n}$ the recovered matrix.
2. For each $j \in [t]$:
    (a) Set the initial input to the network to be zero, i.e. $\mathbf{x}_1 := 0$.
    (b) Follow the inference protocol to partially evaluate the network
        up to the $n-1$-th layer: $\mathbf{x}_{n-1} := \mathsf{ReLU}(\mathsf{NN}_i(0)) = 0$.
    (c) Malleate the client's share of $\mathbf{x}_{n-1}$: $\langle \mathbf{x}'_{n-1} \rangle_C := \langle \mathbf{x}_{n-1} \rangle_C + \mathbf{e}_j$.
    (d) Complete the protocol with the server to obtain $\mathbf{x}_n := M_n \mathbf{x}'_{n-1} =$
        $M_n \mathbf{e}_j$.
    (e) Set the $j$-th column of $\tilde{M_n}$ to be $\mathbf{x}_n$.
3. Output $\tilde{M_n}$.
```

```
MaskAndLinearizeReLU($\mathbf{y}_C$):
1. Malleate the client's local share of $\mathbf{y}$ to obtain a share of the
   malleated $\mathbf{y}'$ as follows:
    (a) For all $l \in \{k', \dots, k'+m-1\}$, Set $\langle \mathbf{y}' \rangle_C[l] := \langle \mathbf{y} \rangle_C[l] + c$.
    (b) For all $l \notin \{k', \dots, k'+m-1\}$, Set $\langle \mathbf{y}' \rangle_C[l] := \langle \mathbf{y} \rangle_C[l] - c$.
2. Obtain $\langle \mathbf{x} \rangle_C := \mathsf{LinearizeReLU}(\mathbf{y}')$..
3. Malleate $\langle \mathbf{x} \rangle_C$ to obtain $\langle \mathbf{x}' \rangle_C$ by inverting the operations in
   Item 1a:
       for each $l \in \{k', \dots, k'+m-1\}$, set $\langle \mathbf{x}' \rangle_C[l] := \langle \mathbf{x} \rangle_C[l] - c$
4. Output $\langle \mathbf{x}' \rangle_C$.
```

```
LinearizeReLU($\mathbf{y}_C$):
1. Malleate the client's local share of $\mathbf{y}$ to obtain a share of the
   malleated $\mathbf{y}'$:
       for each $l \in \{k', \dots, k'+m-1\}$, Set $\langle \mathbf{y}' \rangle_C[l] := \langle \mathbf{y} \rangle_C[l] + c$.
2. Interact with the server to obtain the share $\langle \mathbf{x} \rangle_C$, which is the
   client's share of $\mathbf{x} := \mathsf{ReLU}(\mathbf{y}')$.
3. Malleate $\langle \mathbf{x} \rangle_C$ to obtain $\langle \mathbf{x}' \rangle_C$ by inverting the operations in
   Item 1:  for each $l \in \{k', \dots, k'+m-1\}$, set $\langle \mathbf{x}' \rangle_C[l] :=$
   $\langle \mathbf{x} \rangle_C[l] - c$
4. Output $\langle \mathbf{x}' \rangle_C$.
```

```
RecoverIntermediateLayer($i, M_{i+1}, \dots, M_n$) :
1. For each $i \in [n-1, \dots, 1]$:
(a) Let the dimension of the $i$-th linear layer be $s_i \times t_i$.
(b) Let $s'_i := \lfloor s_i/m \rfloor$.
(c) For each $j \in [t_i]$, and for each $k \in [s'_i]$:
     i. Set the initial input to the network to be zero, i.e. $\mathbf{x}_1 := \mathbf{0}$.
    ii. Follow the inference protocol to evaluate the network up
        to the $i-1$-th layer to obtain (a share of) the intermediate
        state $\mathbf{x}_{i-1} := \mathsf{ReLU}(M_{i-1}(\dots \mathsf{ReLU}(M_1 \mathbf{x}_1))) = \mathbf{0}$.
   iii. Construct a query $\mathbf{q}_j := \mathbf{e}_j$.
    iv. Malleate the client's share of $\mathbf{x}_{i-1}$: $\langle \mathbf{x}'_{i-1} \rangle_C := \langle \mathbf{x}_{i-1} \rangle_C +$
        $\mathbf{q}_j$.
     v. Interact with the server to evaluate the $i$-th linear layer to
        obtain a share of $\mathbf{y}_i := M_i \mathbf{x}'_{i-1}$.
    vi. Obtain the input for the next linear layer: $\langle \mathbf{x}_i \rangle_C :=$
        $\mathsf{MaskAndLinearizeReLU}(\mathbf{y}_i)$.
   vii. Let $k' := k\dot{m}$.
  viii. The vector $\mathbf{x}_i$ should now be all-zero, except in locations
        $k', \dots, k'+m-1$, where it should equal the corresponding
        elements of $\mathbf{y}_i$.
    ix. Interact with the server to complete the evaluation of the
        rest of the network, invoking LinearizeReLU to force the
        ReLU to behave linearly.
     x. Define a $t_i \times t_i$ matrix Mask to be the all-zero matrix,
        except at the locations $(l, l)$, where $l \in \{k', \dots, k'+m-$
        $1\}$.
    xi. For each $l \in [m-1]$, construct the $i$-th linear equation
        $\mathbf{x}_n[l] = M_n \cdot M_{n-1} \cdots M_{i+1} \cdot \mathsf{Mask}(M_i q_j[l+k'])$.
(d) Solve all the linear equations to construct and output $\tilde{M}_i$.
```