SIGuard: Guarding Secure Inference with Post Data Privacy

Xinqian Wang*, Xiaoning Liu*, Shangqi Lai†, Xun Yi*,Xingliang Yuan‡
*RMIT University, †CSIRO's Data61, ‡The University of Melbourne

***Abstract*—Secure inference is designed to enable encrypted machine learning model prediction over encrypted data. It will ease privacy concerns when models are deployed in Machine Learning as a Service (MLaaS). For efficiency, most of recent secure inference protocols are constructed using secure multi-party computation (MPC) techniques. They can ensure that MLaaS computes inference without knowing the inputs of users and model owners. However, MPC-based protocols do not hide information revealed from their output. In the context of secure inference, prediction outputs (i.e., inference results of encrypted user inputs) are revealed to the users. As a result, adversaries can compromise output privacy of secure inference, i.e., launching Membership Inference Attacks (MIAs) by querying encrypted models, just like MIAs in plaintext inference.**

We observe that MPC-based secure inference often yields perturbed predictions due to approximations of nonlinear functions like softmax compared to its plaintext version on identical user inputs. Thus, we evaluate whether or not MIAs can still exploit such perturbed predictions on known secure inference protocols. Our results show that secure inference remains vulnerable to MIAs. The adversary can steal membership information with high successful rates comparable to plaintext MIAs.

To tackle this open challenge, we propose SIGuard, a framework to guard the output privacy of secure inference from being exploited by MIAs. SIGuard's protocol can seamlessly be integrated into existing MPC-based secure inference protocols without intruding on their computation. It proceeds with encrypted predictions outputted from secure inference, and then crafts noise for perturbing encrypted predictions without compromising inference accuracy; only the perturbed predictions are revealed to users at the end of protocol execution. SIGuard achieves stringent privacy guarantees via a co-design of MPC techniques and machine learning. We further conduct comprehensive evaluations to find the optimal hyper-parameters for balanced efficiency and defense effectiveness against MIAs. Together, our evaluation shows SIGuard effectively defends against MIAs by reducing the attack accuracy to be around the random guess with overhead (1.1s), occupying ∼24.8% of secure inference (3.29s) on widely used ResNet34 over CIFAR-10.

I. Introduction

Recent advancements in machine learning have spawned numerous Machine-Learning-as-a-Service (MLaaS) offerings for cloud providers. For example, the Google Cloud Vision API [1] provides an intelligent service to interpret, analyze, and derive insights from visual data. In a typical MLaaS workflow, the cloud hosts a pre-trained neural network model and provides API to make predictions. Users feed data to the model through the API and receive an inference result in return. However, the existing implementation of such offerings raises paramount privacy concerns: users' sensitive data are sent to the cloud, and the cloud keeps a copy of the proprietary model from the model owner. Both the privacy of individuals and the lucrative neural network models are jeopardized.

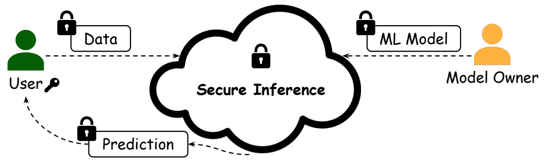


Fig. 1: Secure inference as a service.

To address these privacy concerns, recent advancements focus on constructing secure inference as a service as a prospective solution. As Figure 1 depicts, a user and a model owner upload encrypted data and model to the cloud, which then runs a secure inference protocol over encrypted data. Ideally, such a solution ensures that only the user learns the inference result; neither of them can learn any information about the other's input data. Many recent studies [2], [3], [4], [5] opt for secure multiparty computation (MPC) techniques to realize such guarantees on *input privacy*. Unfortunately, those studies do not protect information that might be revealed from the output of secure inference [3].

Prediction API attacks have emerged to exploit models and underlying training data, only giving access to inference results through black-box queries to the model [6], [7], [8]. Membership inference attack (MIA) [6], [9], [10], [11], [12] is one such threat notably, which aims to determine whether a specific data record belongs to the target model's training dataset. For example, if an MIA adversary infers that a certain patient's data record was used to train a disease classification model, the adversary would know this patient is ill.

The membership inference attacks have posed the possibility of compromising the output privacy of secure inference. This is because those attacks are missed by the MPC's threat model: MPC-based protocols do not hide information inferred from the output of the protocols [3], [13], [14]. Once the inference result is revealed to the users, they can leverage the predictions to carry out MIAs. Such sensitive membership information of the training data could still leak even if the model is encrypted via MPC, which thus makes secure inference vulnerable to membership inference.

The privacy threats on post data have been brought to the attention of the secure multiparty computation community. Over the years, a flurry line of studies has identified achieving output privacy as a desired property of secure inference [3],

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240168
www.ndss-symposium.org

[15], [16]. Defense against MIAs in the plaintext domain can be applied during the training phase and the inference phase. Although models using the specific training approaches in plaintext can be naturally combined with secure inference, this approach inherently reduces inference accuracy [9], [15], [17], [18], [19], [20]. Other works inject carefully crafted noises to perturb the inference outputs [19], [20], [21], yet they are not capable of being adopted in secure inference directly. Effective defending MIAs without compromising inference accuracy remains an open challenge.

Our framework. We present SIGuard (**S**ecure **I**nference **Guard**), a framework for empowering *output privacy* to existing secure inference. SIGuard secretly preserves the membership privacy of the model's training dataset, safeguarding it against exploitation by MIA adversaries. As an off-the-shelf defense module, SIGuard can be seamlessly plugged into any standard MPC-based secure inference without interfering with the existing computation. Our key contributions are as follows:

- We are the first to investigate the threats of MIAs under the context of secure inference. Our intensive evaluations demonstrate that although secure inference commonly leads to perturbed predictions, it still suffers from MIAs.
- We propose SIGuard, the first defense framework against MIAs in secure inference stage. Built using efficient MPC techniques, SIGuard can be readily integrated into secure inference protocols and can securely add perturbations to inference results without affecting accuracy.
- We conduct comprehensive evaluations over various datasets and MIAs for SIGuard, in terms of its running time and communication cost, and effectiveness against MIAs. Our results confirm that SIGuard can effectively defeat MIAs for secure inference without introducing dominant overhead. SIGuard can reduce the attack performance of MIAs to almost 50% (random guess) by only incurring around 6% extra overhead to secure inference. These results suggest that SIGuard is promising for practical deployment in secure inference.

A. Summary of techniques

Evaluating membership inference against secure inference. We observe that MPC-based secure inference often yields perturbed predictions compared to its plaintext version given identical user inputs. Those perturbations largely stem from the approximations of non-linear functions in modern neural networks, such as the softmax. Although the output privacy threat has conveyed increasing awareness of the MPC community, whether the MIA adversaries can still break the output privacy with such perturbed predictions remains unclear. To demonstrate the effectiveness of MIA, we conduct a series of principled MIA attacks against four secure inference protocols, i.e., SecureML [22], CrypTen [23], Piranha [24], and the work proposed by Aly and Smart (AS19) [25]. Our results show the adversary can successfully steal membership information for up to 99% attack accuracy over the Location30 dataset through multiple shadow model MIA [6]. Sometimes, the MIA risk for a model in secure inference can be greater than that in plaintext inference, e.g., the MIA success rate increases by 3.87% on the Texas100 dataset as shown in Section IV.

Protecting the output privacy. To defend against MIAs within secure inference, SIGuard's protocol injects carefully crafted perturbations into the encrypted predictions without harming the inference accuracy. Such a goal is achieved by a synergy of MPC techniques (i.e., based on replicated secret sharing (RSS) [26]) and plaintext defense technique MemGuard [21]. We design two secure components for noise generation by optimizing the loss function introduced by MemGuard. They are termed Secure Noise Optimization (Section V-A) and Secure Noise Validation (Section V-B). The secure noise optimization protocol aims to find an optimal noise vector to perturb the secret-shared confidence vector, thereby reducing the accuracy of MIAs to a level comparable to random guessing. Meanwhile, the noise validation protocol aims to validate that whether the secret-shared noise vector is carefully crafted to preserve the inference accuracy. Finally, the perturbed confidence vector in secret-shared format is returned to the user and opened as the prediction result.

Achieving stringent privacy guarantees with optimal performance. We observe that collusion between MIA adversaries compromising a user and secure inference adversaries compromising a server broadens the attack surface of MIAs. When collusion occurs, MIA adversaries can obtain knowledge of how the secure defense mechanism operates. Specifically, MemGuard (Algorithm 1) recursively solves an optimization problem for generating the noise vector. The whole optimization process is controlled by a `while` loop. The `while` loop terminates when two conditions are met: 1) the generated noise vector no longer changes the original inference results, and 2) it demonstrates effectiveness in defending against MIAs. However, we find that there are distributional differences in the number of iterations between member and non-member data, which the MIA adversary can exploit to increase its success rate. When adapting the `while` loop in MPC, the MIA adversary, by colluding with a server, can naturally count the number of iterations, leading to additional privacy leakage. To address that, we propose two refinements. Refinement I (Section V-C) mitigates the leakage by fixing the iterations for all user inputs. Refinement II (Section V-D) finds the best trade-off that minimizes the number of iterations while maintaining desired defense effectiveness through fine-tuning the hyper-parameters. Refinement II reduces the overhead of SIGuard on CIFAR-10 from looping 300 iterations to 10 iterations, while still maintaining desired defense effectiveness (reduce MIAs to nearly random guess $\sim$50%).

II. Preliminaries

A. Notation

We denote the single value as x, the vector as $\vec{x}$, and the matrix as $\boldsymbol{X}$. The i-th bit of $x \in \mathbb{Z}_{2^\ell}$ is represented as $x[i]$, the i-th element of $\vec{x}$ as $\vec{x}[i]$, and the element at the i-th row and j-th column of $\boldsymbol{X}$ as $\boldsymbol{X}[i][j]$. The L_1 norm of $\vec{x}$ is denoted as $\|\vec{x}\|_1 := \sum_{i=0}^{n-1} |\vec{x}[i]|$. The L_2 norm of $\vec{x}$ is denoted as $\|\vec{x}\|_2 :=$

TABLE I: Notation table.

$\vec{s}$	Original confidence vector.
$\vec{s'}$	Perturbed confidence vector.
$\vec{z}$	$logits$ from the target model, such that $\vec{s} = softmax(\vec{z})$.
$\vec{z'}$	Perturbed $logits$ from the target model.
$\vec{e}$	Noise vector that added to $logits$ $\vec{z}$.
h	The simulated membership classifier.

$\sqrt{\sum_{i=0}^{n-1} \vec{x}[i]^2}$. Table I lists the symbols that consistently retain their specific meanings throughout this paper.

B. Membership inference attacks in neural network inference

Neural network inference can be viewed as a mapping that transforms a data sample into its potential category. Let the model weights be $\{\boldsymbol{W}_i\}_0^{L-1}$, where $\boldsymbol{W}_i$ represents the i-th linear layer's parameters. It works in a layer-by-layer fashion, i.e., a linear layer (e.g., dense layer, convolutional layer) followed by a non-linear layer (e.g., ReLU, softmax). The inference output from neural network inference is a *confidence vector*. It indicates a probability distribution of the input data's possible categories, where each entry is the probability of the input being classified into its corresponding label. The final predicted label is the one with the largest confidence score. Note that, the confidence vector (denoted as $\vec{s}$) is computed by softmax on the input: $logits$ $\vec{z}$.

Membership inference attacks leverage the confidence vector to determine if the query data is in the target model's training dataset. To achieve that, the adversary trains a binary membership classifier, taking the input as a confidence vector and infers whether the data sample is a member of the training dataset or a non-member.

C. MemGuard

The work of Jia *et al.* [21] assumes a pre-trained neural network (target model) providing an inference service that aims to defend against membership inference attacks. To avoid retraining the model, MemGuard adds carefully crafted noise vectors $\vec{e}$ and adds to the target model's $logits$ $\vec{z}$ for each inference. Consequently, when these modified $logits$ are fed into the softmax function, the original confidence vector $\vec{s}$ undergoes perturbation.

We provide the algorithm of MemGuard in Algorithm 1. MemGuard finds a noise vector $\vec{e}$ by iteratively solving an optimization problem that minimizes a composite loss function, $\mathfrak{L}$, with each of its terms weighted by c_i:

$$\begin{aligned} \mathfrak{L} = & c_1 \cdot |h(softmax(\vec{z}+\vec{e}))| && (\mathfrak{L}1)\\ & + c_2 \cdot \text{ReLU}(-\vec{z}[l] - \vec{e}[l] + max_j(\vec{z}+\vec{e})) && (\mathfrak{L}2)\\ & + c_3 \cdot \|softmax(\vec{z}) - softmax(\vec{z}+\vec{e})\|_1 , && (\mathfrak{L}3) \end{aligned}$$

The **first** part $\mathfrak{L}1$ is to defend MIAs. The crafted noise vector should make the adversary's attack performance close to 50%. The 50% refers to the accuracy of random guessing by MIAs, which is the ideal defense against MIAs that all works aim to achieve. To address this, MemGuard trains a membership classifier neural network h to simulate the adversary's membership classifier. To simplify the computation, the architecture of h is set as a fully connected network, it takes a confidence vector $\vec{s}$ as input and outputs $h(\vec{s}) \in \mathbb{R}$. A positive $h(\vec{s})$ indicates the data point comes from the training dataset, while a negative score indicates otherwise. The **second** part $\mathfrak{L}2$ is to preserve the target model's final predicted label $\arg\max_i \{\vec{s}[i]\}$. The **third** part of $\mathfrak{L}3$ is to make the crafted noise as small as possible. We refer readers to Appendix A for more details.

Algorithm 1 MemGuard

```
Input: $\vec{z}$, $h$, $max\_iter$, $c_3$, and $\beta$.
Output: $\vec{e'}$.
1: $\vec{e'} \leftarrow \vec{0}$.
2: $l \leftarrow \arg\max_i \{\vec{z}_i\}$.
3: while $True$ do
4:     $\vec{e} \leftarrow \vec{0}$.
5:     $i \leftarrow 0$.
6:     while $i < max\_iter$ and ($\arg\max_i \{\vec{z}[i] + \vec{e}[i]\} \neq l$ or $h(softmax(\vec{z}+\vec{e})) \cdot h(softmax(\vec{z})) > 0$) do
7:         $\vec{u} \leftarrow \frac{\partial \mathfrak{L}}{\partial \vec{e}}$.
8:         $\vec{u} \leftarrow \vec{u} / \|\vec{u}\|_2$.
9:         $\vec{e} \leftarrow \vec{e} - \beta \cdot \vec{u}$.
10:        $i \leftarrow i + 1$.
11:    end while
12:    if $\arg\max_i \{\vec{z}[i] + \vec{e}[i]\} \neq l$ or $h(softmax(\vec{z} + \vec{e})) \cdot h(softmax(\vec{z})) > 0$ then
13:        return $\vec{e'}$.
14:    end if
15:    $\vec{e'} \leftarrow \vec{e}$.
16:    $c_3 \leftarrow 10 \cdot c_3$.
17: end while
```

D. Building blocks

Replicated secret sharing. SIGuard resorts to 2-out-of-3 replicated secret sharing (RSS) from Araki *et al.* [26]. Let P_1, P_2, P_3 be the three parties involved in the computation. For brevity, P_{i-1}, P_{i+1} denote the prior and the succeeding party of party P_i, where $i \in \{1, 2, 3\}$. Given $x \in \mathbb{Z}_{2^\ell}$ is an integer secret. To *share* a secret, x is split by uniformly sampling three random values $\langle x\rangle_1, \langle x\rangle_2, \langle x\rangle_3 \in \mathbb{Z}_{2^\ell}$ such that $\langle x\rangle_1 + \langle x\rangle_2 + \langle x\rangle_3 \equiv \langle x\rangle \pmod{2^\ell}$. These replicated secret shares are distributed as pairs that P_1 holds $(\langle x\rangle_1, \langle x\rangle_2)$, P_2 holds $(\langle x\rangle_2, \langle x\rangle_3)$, and P_3 holds $(\langle x\rangle_3, \langle x\rangle_1)$. To *reveal* a secret, P_i sends $\langle x\rangle_i$ to P_{i+1}, and each party locally adds the three shares to reconstruct x.

Arithmetic operations are evaluated by RSS as follows. Given two secret values x and y are shared among three parties. Addition $\langle z\rangle_i = \langle x\rangle_i + \langle y\rangle_i \pmod{2^\ell}$ can be locally computed by each party $\langle z\rangle = \langle x + y\rangle := (\langle x\rangle_1 + \langle y\rangle_1, \langle x\rangle_2 + \langle y\rangle_2, \langle x\rangle_3 + \langle y\rangle_3)$.

Multiplication over two shares $\langle z\rangle = \langle x\rangle \cdot \langle y\rangle \pmod{2^\ell}$ are computed interactively as follows. Each party P_i first locally computes a 3-out-of-3 secret shares $\hat{z}_i = \langle x\rangle_i\langle y\rangle_i + \langle x\rangle_{i+1}\langle y\rangle_i + \langle x\rangle_i\langle y\rangle_{i+1}$ such that $\hat{z}_1 + \hat{z}_2 + \hat{z}_3 = xy$. Then, the parties jointly perform a *reshare* scheme to maintain the invariance of 2-out-of-3 sharing. Given $\alpha_1, \alpha_2, \alpha_3 \in \mathbb{Z}_{2^\ell}$ are

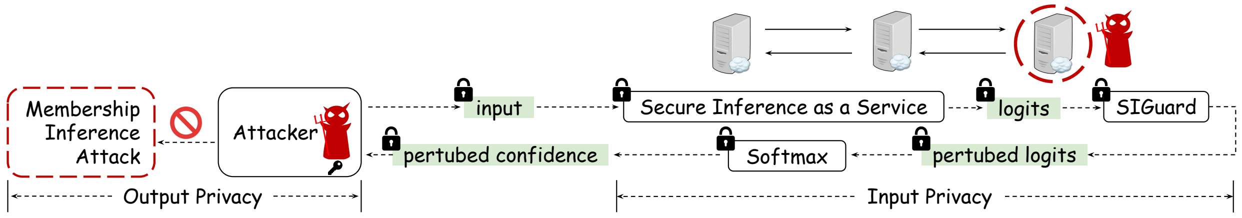


Fig. 2: Workflow of how SIGuard protect **post-data privacy** in the context of three-party computation.

zero sharing terms subject to $\alpha_1 + \alpha_2 + \alpha_3 = 0$. Each party P_i uses α_i to mask $\hat{z}_i$ as $\langle z \rangle_i = \hat{z}_i + \alpha_i$. P_i then sends $\langle z \rangle_i$ to P_{i-1}. Parties obtain their shares as $(\langle z \rangle_i, \langle z \rangle_{i+1})$.

When $\ell = 2$, we have the RSS for Boolean secret shares $[\![x]\!] \equiv [\![x]\!]_1 \oplus [\![x]\!]_2 \oplus [\![x]\!]_3$, where XOR $\oplus$ and AND $\wedge$ is computed equivalent to addition and multiplication over $\mathbb{Z}_2$.

Fixed point representation and truncation. Normally, a float-point number x represented by fixed-point representation with l bits is $\lfloor x \cdot 2^{\ell_D} \rceil$, where $\lfloor \cdot \rceil$ denotes rounding to the nearest integer over ring $\mathbb{Z}_{2^\ell}$ and ℓ_D denotes as the fraction part length. Note that the multiplication of $\lfloor x \cdot 2^{\ell_D} \rceil$ and $\lfloor y \cdot 2^{\ell_D} \rceil$ doubles the length of fraction part. Therefore, there should be a truncation protocol running behind each multiplication [27].

III. System overview

A. System setup

Figure 2 shows our system setup, which contains three servers and users. The three servers invoke a three-party computation (3PC) protocol. The protocol is reactive, invoked continuously upon receiving inputs and computing the outputs. Each user sends private inputs to the servers and receives the outputs directly. The user interacts with the servers via the interface of secure inference, submitting data or retrieving outputs as its requests are processed by the servers.

Components. The protocol sequentially combines three components: *secure inference as a service*, SIGuard, and secure softmax. The secure inference as a service inputs users' private data and outputs the *logits* vectors. SIGuard inputs the *logits* vectors and outputs the perturbed *logits* vectors. The secure softmax converts the perturbed *logits* vectors into confidence vectors. The confidence vectors are set as the servers' outputs (see Section VIII-D for their importance).

Workflow. Before SIGuard is operational, the model owner deploys a pre-trained model and a simulated membership classifier (a part of SIGuard's input) across the three servers using RSS. Once the service requests are processed, users distribute their private data across the servers using RSS. Then, the servers invoke the protocol. Throughout the computation, the intermediate results between each aforementioned component are kept as secret shares. Upon completion, each server holds shares of the perturbed confidence vectors. Finally, the servers transmit the shares to the users, who reconstruct them locally to obtain the final prediction.

Use case. In medical image analysis, a hospital trains a machine learning model on brain tumor MRI scans to provide diagnostic services [28]. The model is trained using patients' private data. To address privacy concerns, the hospital deploys the model to the cloud using secret sharing. Users resort to secure inference for medical diagnostics while keeping their data confidential. However, users can be compromised by MIA adversaries. The adversaries could reveal some patients' associations with the hospital, increasing the risk of re-identification and causing societal harm. By plugging SIGuard into secure inference, the threat of MIAs is largely reduced.

B. Threat model

SIGuard's threats originate from adversaries who may compromise a single service user, up to one cloud server, or both. SIGuard defends against two types of corruption: 1) independent corruption, where either a semi-honest adversary compromises one of the cloud servers or a membership inference adversary compromises the user; and 2) collusion, where the compromised server colludes with the user.

Independent corruption. The independent corruption can be further distilled into two cases, one compromised server and one compromised user.

The first case derives from previous secure inference works [2], [4]. It assumes the 3PC protocol is invoked in an honest-majority setting where two servers behave honestly and one server is corrupted by a semi-honest adversary. The corrupted server is assumed to strictly follow the protocol without deviating from it. However, the adversary may attempt to infer additional information from the computation to uncover the user's private inputs, the model owner's secure inference model, the model owner's simulated membership classifier, and the generated noise vector.

The second case derives from previous studies of MIAs [6], [21]. It grants the adversary unlimited query access to the inference service. The adversary inputs data to the inference service and receives the corresponding outputs. Based on the outputs, the adversary trains a membership classifier to infer which data was part of the model's training set.

Collusion. The collusion between a semi-honest cloud server and a single service user targets on: 1) the data privacy of other users and model owners, and 2) the membership privacy of the training data. The collusion can result in a stronger membership inference adversary by broadening the attack surface compared to standard MIA assumptions. With access to the corrupted server, the adversary can observe the protocol's execution, learn its functionality, and measure the execution time of each subroutine. By understanding the functionality, the adversary knows the defense mechanism so

Parameters: Parties P_1, P_2, P_3, model owner $\mathcal{O}$ and user $\mathcal{U}$.
Uploading Data: On input $(\langle\vec{z}\rangle_i, \langle\vec{z}\rangle_{i+1})$ from P_i (i.e., shares of $logits$). On input $\langle \boldsymbol{W} \rangle := \{\langle \boldsymbol{W}_i \rangle\}_0^{L-1}$ from $\mathcal{O}$ (i.e., shares of simulated membership classifier's model weights).
Computation: On inputs SIGuard and $softmax$ from the three parties, compute the perturbed confidence vector $\vec{s'}$.
Output: Return perturbed confidence vector $\vec{s'}$ to $\mathcal{U}$.

Fig. 3: $\mathcal{F}_{\mathsf{SIGuard}}$ Ideal Functionality.

that it may attempt to reverse its membership inference predictions to bypass the defense (Section VIII-B). Furthermore, the adversary can identify the used non-linear approximations, enabling it to design a membership classifier that mimics the model's behavior by using similar approximations. By measuring the execution time of each subroutine, the adversary gains an advantage in launching MIAs (Section V-C).

Compared to typical black-box MIAs, the knowledge of the used models and defense mechanisms is provided as auxiliary information rather than being inferred by the adversary. Other auxiliary information, e.g. knowledge of the training dataset's distribution, is assumed to be known by the adversary.

In both types of corruption, the compromised server and compromised user attempt to learn information regarding the other uncorrupted service users' input, the model owners' model parameters (i.e., secure inference model and simulated membership classifier), and the generated noise vectors. Additionally, the corrupted user seeks to learn the membership of the training data from the confidence vectors.

C. Security definition

We assume a semi-honest adversary $\mathcal{A}$ that corrupts one user and one of the cloud servers. The security definition of the whole system should ensure that $\mathcal{A}$ only learns the input from the corrupted user, the final prediction, and nothing else. As outlined in Section III-A, the 3PC protocol consists of secure inference as a service, SIGuard, and secure softmax. Since secure inference as a service relies on existing protocols already secured by RSS [26], we consider SIGuard alongside secure softmax, referring to this combination as the system framework of SIGuard, and proceed to prove the security of the framework. Particularly, we demonstrate that except for receiving the perturbed confidence vectors (final predictions), $\mathcal{A}$ does not learn any information about the simulated membership classifier and the generated noise vector.

We formally define the ideal functionality of the SIGuard system framework in Figure 3. Let Π_{SIGuard} be a protocol that realizes $\mathcal{F}_{\mathsf{SIGuard}}$. Then, we define the security of the SIGuard system framework by comparing its real-world functionality with its ideal functionality [29].

Definition 1 (Security Definition): A protocol Π_{SIGuard} securely realizes $\mathcal{F}_{\mathsf{SIGuard}}$ in the context of semi-honest adversaries, if for every PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that, for every $\mathcal{P} \subset \{P_1, P_2, P_3, \mathcal{O}, \mathcal{U}\}$:

$$\{\mathrm{Ideal}_{\mathcal{F}_{\mathsf{SIGuard}}, \mathcal{S}, \mathcal{P}}(\langle \boldsymbol{W} \rangle, \langle \vec{z} \rangle)\} \overset{c}{\equiv} \{\mathrm{Real}_{\Pi_{\mathsf{SIGuard}}, \mathcal{A}, \mathcal{P}}(\langle \boldsymbol{W} \rangle, \langle \vec{z} \rangle)\},$$

where $\mathrm{Ideal}_{\mathcal{F}_{\mathsf{SIGuard}}, \mathcal{S}, \mathcal{P}}(\langle \boldsymbol{W} \rangle, \langle \vec{z} \rangle)$ is the output of $\mathcal{F}_{\mathsf{SIGuard}}$'s interaction in the ideal world and $\mathrm{Real}_{\Pi_{\mathsf{SIGuard}}, \mathcal{A}, \mathcal{P}}(\langle \boldsymbol{W} \rangle, \langle \vec{z} \rangle)$ is the output of Π_{SIGuard}'s interaction in the real world.

Note that MPC does not prevent adversaries from observing the protocol's computational patterns, meaning that reducing the expanded attack surface is beyond what MPC alone can accomplish. Similarly, since MPC does not protect output privacy, mitigating MIAs requires implementing a defense mechanism, which, in our case, is MemGuard.

IV. Membership inference in secure inference

Although the community has asserted that secure inference is inherently vulnerable to the black-box MIAs over the years [3], [15]. Membership inference against secure inference differs from the one in plaintext MLaaS. Such a difference is attributed to the distorted confidence scores stemming from the approximated softmax in secure inference. It is unknown under such a large perturbation caused by softmax approximations, whether MIAs would still perniciously exploit the secure inference. In this section, we investigate that the privacy risks of MIAs in the MPC-based secure inference when using different softmax approximations. Our empirical study shows that secure inference remains as vulnerable as plaintext inference to MIAs, and in some cases, even worse.

Specifically, practical neural networks often involve various complex non-linear functions that are not directly supported by MPC in secure inference. This is because standard MPC primitives only efficiently support arithmetic operations of addition and multiplication. Computing the non-linear functions requires approximations to reformulate the standard function to its MPC-friendly forms [3], [27], [30], [31]. Softmax is one such typical function. Moreover, since softmax lies at the final step in secure inference, it plays a vital role in injecting perturbations to MPC-based secure inference.

Following, we empirically analyze the above question. In Section IV-A, we provide our experiment setup and the widely-adopted softmax approximations. In Section IV-B, we investigate the risk of MIAs in MPC. We analyze scenarios where models are equipped with four different softmax approximations. Given studies of MIAs place significant emphasis on analyzing confidence vectors [32], [33], we particularly focus on analyzing the confidence vectors from secure inference.

A. Experiment setup

Softmax approximations. In SIGuard, we focus on four exemplary approximations: ReLU-based approximation in SecureML [22], limit-based approximation in CrypTen [23], approximation-based on normalization and logistic function in Piranha [24], and logarithmic-based approximation proposed in Aly & Smart [25] (denoted as AS19). Given the input $logits$ $\vec{z}$, these approximations can be computed as follows:

· SecureML [22] substitutes the exponential function with ReLU: $softmax(\vec{z}) = \mathrm{ReLU}(\vec{z}[i])/\Sigma\mathrm{ReLU}(\vec{z}[j])$.
· CrypTen [23], [34] calculates the exponential function with base e by using its definition of the limit (limit approximation): $exp(\vec{z}[i]) = \lim_{n\to\infty}(1 + \vec{z}[i]/2^n)^{2^n}$.

TABLE II: Comparison of membership inference attacks in secure inference under different softmax approximations and datasets (D1=CIFAR-10, D2=CIFAR-100, D3=CH-MINIST, D4=Location30, and D5=Texas100).

	plaintext		SecureML [22]		CrypTen [23]		Piranha [24]		AS19 [25]	
	lowest	**highest**	lowest	**highest**	lowest	**highest**	lowest	**highest**	lowest	**highest**
D1	57.51% (NN-M)	60.48% (LiRA)	50.11% (entr)	86.56% (LiRA)	56.71% (entr)	61.58% (LiRA)	26.88% (LiRA)	59.11% (conf)	57.15% (entr)	60.76% (LiRA)
D2	70.10% (entr)	82.53% (LiRA)	52.15% (entr)	90.90% (LiRA)	71.43% (NN-M)	82.58% (LiRA)	60.40% (LiRA)	75.73% (Mentr)	66.40% (entr)	82.30% (LiRA)
D3	68.50% (LiRA)	77.95% (conf)	58.20% (entr)	74.70% (NN-M)	69.05% (LiRA)	77.75% (NN-M)	35.35% (LiRA)	67.75% (conf)	69.60% (LiRA)	76.95% (NN-M)
D4	72.00% (LiRA)	99.26% (NN-M)	67.26% (entr)	88.66% (NN-M)	72.06% (LiRA)	99.06% (NN-M)	54.13% (LiRA)	86.40% (Mentr)	71.73% (LiRA)	98.93% (NN-M)
D5	73.58% (LiRA)	77.38% (NN-M)	54.58% (NN-M)	75.80% (Mentr)	73.61% (entr)	77.23% (NN-M)	53.88% (LiRA)	81.25% (Mentr)	72.88% (entr)	78.36% (NN-M)

· Piranha [24] combines normalization with an approximation of logistic function [22]:

$$exp(\vec{z}[i] - \vec{z}_{max}) \approx \begin{cases} 0.5(\vec{z}[i] - \vec{z}_{max}) + 1 & \text{if } \vec{z}[i] - \vec{z}_{max} \geq -2 \\ 0 & \text{otherwise} \end{cases}$$

· AS19 [25] use logarithmic identities to convert e^x into $2^{x \cdot \log_2 e}$. Denote $y = x \cdot \log_2 e$ and we further express y as the sum of its integer part and fractional part, such that $y = \lfloor y \rfloor + \{y\}$. Then the problem is decomposed into calculating $2^{\lfloor y \rfloor}$ and $2^{\{y\}}$.

$$exp(\vec{z}[i]) = 2^{\vec{z}[i] \cdot \log_2 e} = 2^y = 2^{\lfloor y \rfloor + \{y\}},$$
$$2^{\{y\}} \approx 1 + (ln(2)/1!)\{y\} + \cdots + (ln(2)^n/n!)\{y\}^n.$$

Datasets. We select six real-world datasets to conduct our experiments, CIFAR-10 [35], CIFAR-100 [35], CH-MNIST [36], Location30 [6], and Texas100 [6]. They are widely employed in the analysis of MIAs [6], [11], [12], [21]. We use the CINIC-10 dataset (a drop-in replacement for CIFAR-10) for a more comprehensive evaluation when comparing our approach to training-stage defense algorithms.

MIAs. We selected five MIAs encompassing almost all categories of MIAs in literature, Multiple shadow model attack (NN-M) [6], Confidence-based attack (conf) [18], Entropy-based attack (entr) [18], Modified entropy-based attack (Mentr) [18], and Likelihood Ratio Attack (LiRA) [12].

For NN-M, based on the auxiliary dataset, the adversary can train a binary membership classifier. For conf, the adversary finds a confidence threshold for each classification label to differentiate between member and non-member data. For entr, the adversary finds an entropy threshold for each classification label to differentiate between member and non-member data. For Mentr, the adversary distinctively treats the classification label associated with the input confidence vectors during entropy computation. For LiRA, the adversary treats inferring the membership information as conducting a hypothesis test of whether the target model has trained on a specific data sample or not. LiRA presents an online attack and an offline attack. Although the online attack exhibits marginally better performance of TPR at low FPR than the offline attack, the latter demonstrates superior efficiency. In our paper, we implement the **offline** attack version of LiRA, it is calculated as a one-sided hypothesis.

Evaluation metrics. The evaluation of MIAs hinges upon Balanced Accuracy (Bal. Acc.) and TPR at low FPR, where TPR at low FPR is computed from the receiver operating characteristic (ROC) curve. Bal. Acc. reflects the success rate of MIAs on a dataset composed of 50% member samples and 50% non-member samples, reflecting the average-case risk. The ROC curve illustrates how MIAs balance the true positive rate (TPR) and false positive rate (FPR), where a perfect attack would achieve a TPR of 1.0 and an FPR of 0.0 at a specific threshold. Specifically, the TPR at low FPR measures the extreme case of whether the attack can confidently identify any members of the training set.

Experimental procedures. We train target models on each dataset and deploy these models for MPC-based secure inference. For each model, we execute secure inference equipped with four different softmax approximations, as described earlier, and obtain the resulting confidence vectors. Finally, the confidence vectors are fed into the membership classifiers to assess the privacy risk.

B. Analyze the risk of MIAs on secure inference

Table II lists the highest and the lowest Bal. Acc. achieved by MIAs on both plaintext inference and secure inference. Table II illustrates the risk of membership inference against secure inference remains significant, and in some cases, is even higher. For example, applying LiRA to plaintext inference on CIFAR-10 achieves a maximum Bal. Acc. of 60.48%. However, when applied to secure inference on the same dataset using SecureML, the Bal. Acc. increases to 86.56%. Other MIAs' effectiveness can be found in Table X.

Table II also shows that when the approximations are accurate, the risk of MIAs will be comparable to that of plaintext inference; otherwise, it will differ from plaintext inference. The approximations of Crypten and AS19 exhibit the closest resemblance to the true softmax function; hence, their Bal. Acc. of MIAs tend to be closer. The approximations by SecureML and Piranha are less precise compared to those by Crypten and AS19. SecureML preserves only positive $logits$; however, in certain scenarios, it suffers from more severe MIAs compared to other approximations. Piranha represents the softmax approximation with the smallest effective range, often exhibiting the lowest MIA performance among all softmax approximations in most cases. Even the entries with the lowest MIA risk often reach up to 60%, indicating that the risk of MIA in MPC cannot be overlooked.

V. The SIGuard protocol

The overarching objective of SIGuard is to secretly generate noise vectors to perturb the final predictions of secure inference, specifically the confidence vectors. SIGuard is required to maintain both efficiency and defense effectiveness for practical purposes. We identify two core components from MemGuard (i.e., noise optimization and noise validation) and demonstrate how we use MPC to implement them, particularly for computing the non-linear operations within these functions.

Section V-A presents Secure Noise Optimization OPT for secretly computing the derivatives of the loss function $\mathfrak{L}$. Section V-B presents Secure Noise Validation VAL for secretly validating the acceptability of the currently generated noise vectors. We find additional leakage caused by the direct implementation of MemGuard via MPC. To address the leakage, we present our finding and corresponding mitigation in Refinement I, Section V-C. We strike a balance between the defense effectiveness and efficiency of SIGuard through empirical evaluation in Refinement II, Section V-D. We integrate various softmax approximations into SIGuard to evaluate its defense performance in Section V-E. Finally, we provide SIGuard's secure protocol in Section V-F.

A. Secure noise optimization

The secure noise optimization protocol OPT aims to craft a noise vector by iteratively optimizing a composite loss function $\mathfrak{L}$. It mainly takes the replicated secret-shared *logits* $\langle\vec{z}\rangle$ and noise vector $\langle\vec{e}\rangle$ as input and securely calculates the loss functions' derivatives $\langle\vec{u}\rangle$, such that $\vec{u} = \frac{\partial \mathfrak{L}}{\partial \vec{e}}$. The primary goal of OPT is to securely optimize the loss function $\mathfrak{L}$, which composes three terms: the *membership loss* $\mathfrak{L}1$, the *utility loss* $\mathfrak{L}2$, and the *perturbation loss* $\mathfrak{L}3$.

1) Differentiate secure membership loss function: The gradient of the membership loss $\mathfrak{L}1$ can be decomposed into the product of a scalar, the derivative of an absolute value function, the derivative of the simulated membership classifier (backward pass), and the derivative of the softmax function:

$$\frac{\partial \mathfrak{L}1}{\partial \vec{e}} = \underbrace{c_1}_{\text{scalar}} \cdot \underbrace{\frac{\partial \left|h(\vec{s'})\right|}{\partial h(\vec{s'})}}_{\text{absolute value}} \cdot \underbrace{\frac{\partial h(\vec{s'})}{\partial \vec{s'}}}_{\text{backward pass}} \cdot \underbrace{\frac{\partial \vec{s'}}{\partial \vec{e}}}_{\text{softmax}}, \qquad (1)$$

where $\vec{s'} := softmax(\vec{z}+\vec{e})$ stands for the confidence vector that is perturbed by adding the noise vector $\vec{e}$.

In what follows, we illustrate how to securely compute Eq. 1 in SIGuard. There are mainly three derivatives needed to securely compute for differentiating the secure membership loss function: secure absolution, secure membership classifier, and secure softmax.

Secure absolution. The derivative of an absolute value function is equivalent to the sign of its input, except at zero. Note that we omit handling the case of zero during computation as it rarely occurs. To extract the sign of a secret-shared value $\langle x\rangle$, we resort to the standard secure bit decomposition mechanism Π_{A2B} for the most significant bit (MSB) extraction [2], [27]. Π_{A2B} outputs a list of boolean shares, from which we retain only the MSB $[\![\text{b}]\!]$. To compute the arithmetic shared sign bit $\langle\text{sign}\rangle = 1-2\cdot[\![\text{b}]\!]$ using the boolean shared MSB $[\![\text{b}]\!]$, we need to compute a public value multiplying a boolean shared value. We follow the ABY3 framework [2] for a direct multiplication, which is more efficient than first converting the boolean to its arithmetic share and performing the multiplication.

Protocol 1: Secure Noise Optimization OPT

Input: Replicated secret-shared *logits* $\langle\vec{z}\rangle$, original confidence vector $\langle\vec{s}\rangle$, noise vector $\langle\vec{e}\rangle$, neural network h with model weights $\{\langle \boldsymbol{W}_i\rangle\}_0^{L-1}$; the hyper-parameters c_1, c_2 of loss function $\mathfrak{L}$'s and its replicated secret-shared trainable weight $\langle c_3\rangle$.
Output: Parties hold gradient $\langle\vec{u}\rangle$.

```
1: $\langle\vec{z'}\rangle \leftarrow \langle\vec{e}\rangle + \langle\vec{z}\rangle$.
2: $\langle\vec{s'}\rangle \leftarrow \Pi_{\text{Softmax}}(\langle\vec{z'}\rangle)$.
3: $\langle\boldsymbol{J}\rangle \leftarrow \Pi_{\text{DSoftmax}}(\langle\vec{s'}\rangle)$.
Secure Membership Loss $\mathfrak{L}1$
4: $\{\langle\vec{g'}_i\rangle\}_0^{L-2}, \langle m\rangle \leftarrow \Pi_{\text{Forward}}(h, \{\langle\boldsymbol{W}_i\rangle\}_0^{L-1}, \langle\vec{s'}\rangle)$.
5: $\langle\text{sign}_\text{m}\rangle \leftarrow \Pi_{\text{ABS}}(\langle m\rangle)$.
6: $\langle\vec{g}\rangle \leftarrow \Pi_{\text{Backward}}(h, \{\langle\vec{g}_i\rangle\}_0^{L-2})$.
7: $\langle\vec{u_1}\rangle \leftarrow \langle\text{sign}_\text{m}\rangle \cdot \langle\boldsymbol{J}\rangle \cdot \langle\vec{g}\rangle$.
Secure Utility Loss $\mathfrak{L}2$
8: $\langle\vec{\sigma}_l\rangle \leftarrow \Pi_{\text{1AGM}}(\langle\vec{s}\rangle)$.
9: $\langle\vec{\sigma}_{l'}\rangle \leftarrow \Pi_{\text{1AGM}}(\langle\vec{s'}\rangle)$.
10: $\langle\vec{u_2}\rangle \leftarrow \langle\vec{\sigma}_{l'}\rangle - \langle\vec{\sigma}_l\rangle$.
Secure Perturbation Loss $\mathfrak{L}3$
11: $([\![\text{b}_0]\!]\cdots[\![\text{b}_{\text{n}-1}]\!]) \leftarrow \Pi_{\text{A2B}}(\langle\vec{s'}\rangle - \langle\vec{s}\rangle)$.
12: $\langle\vec{sign}\rangle \leftarrow (1-2\cdot[\![\text{b}_0]\!]\cdots 1-2\cdot[\![\text{b}_{\text{n}-1}]\!])$.
13: $\langle\vec{u_3}\rangle \leftarrow \langle\boldsymbol{J}\rangle \cdot \langle\vec{sign}\rangle$.
14: return $\langle\vec{u}\rangle \leftarrow c_1\cdot\langle\vec{u_1}\rangle + c_2\cdot\langle\vec{u_2}\rangle + \langle c_3\rangle\cdot\langle\vec{u_3}\rangle$.
```

Given above, on input shared value $\langle x\rangle$, our secure absolution Π_{ABS} securely computes $\langle\text{sign}\rangle$ as follows:

1) P_1, P_2, P_3 run $[\![x[0]]\!], \cdots, [\![x[\ell-1]]\!] \leftarrow \Pi_{\text{A2B}}(\langle x\rangle)$ to decompose arithmetic share to its boolean shared bit string.
2) P_i sets $[\![\text{b}]\!] \leftarrow [\![x[\ell-1]]\!]$ as shared MSB.
3) P_1, P_3 sample random value $c_1 \xleftarrow{\$} \mathbb{Z}_{2^\ell}$.
4) P_2, P_3 sample random value $c_3 \xleftarrow{\$} \mathbb{Z}_{2^\ell}$.
5) P_3 set $m_0 \leftarrow 1-2\cdot(0\oplus[\![\text{b}]\!]_1\oplus[\![\text{b}]\!]_3) - c_1 - c_3$.
6) P_3 set $m_1 \leftarrow 1-2\cdot(1\oplus[\![\text{b}]\!]_1\oplus[\![\text{b}]\!]_3) - c_1 - c_3$.
7) P_3 (sender) and P_2 (receiver) invoke Oblivious Transfer, with P_3 inputs m_0, m_1 and P_2 inputs $[\![\text{b}]\!]_2$. In the end, P_2 gets $c_2 \leftarrow m_{[\![\text{b}]\!]_2}$.
8) P_2 sends c_2 to P_1.
9) The $\langle\text{sign}\rangle$ is set as P_1 holds (c_1, c_2), P_2 holds (c_2, c_3), P_3 holds (c_3, c_1).

Secure membership classifier. The derivative of the simulated membership classifier h can be computed layer by layer, using a backward pass that starts from the last layer and progresses to the first layer. The structure of h is set to be a fully connected network that only comprises linear layers and ReLU layers.

To compute the backward pass, we must first calculate the forward pass of h to obtain the derivative of each ReLU layer [37]. For the linear layer, the forward pass involves secure matrix multiplication (we omit the bias term for simplicity), and its derivative is simply the matrix parameters themselves. For the ReLU layer, its forward pass requires computing $max(0, x)$ for each element x in the input and its derivative is denoted as 1 if the element x remains greater than zero, and 0 otherwise. Here, we resort to ABY3 [2] to securely realize the forward pass of ReLU. Specifically, the ReLU layer's forward pass involves using a share of $[\![\mathrm{b}]\!]$ to select between a secret value $\langle x\rangle$ and 0 by computing $[\![\mathrm{b}]\!]\cdot(0-\langle x\rangle)+\langle x\rangle$. To compute $[\![\mathrm{b}]\!]\cdot(0-\langle x\rangle)$, parties invoke Protocol Π_{BMS}, which computes boolean share multiplying secret share [2]. For the rest operations, we omit the notation of Π_{BMS} and directly represent the multiplication of any arithmetic share $\langle x\rangle$ by a Boolean share $[\![\mathrm{b}]\!]$ as $[\![\mathrm{b}]\!]\cdot\langle x\rangle$.

Secure softmax and its derivative. Given the approximation of AS19 has the greatest defense strength and is more accurate than others, we select it to construct our SIGuard. On input shared $logits$ $\langle\vec{z}\rangle \in \mathbb{Z}_{2^\ell}^{n\times 1}$, our secure softmax with AS19 Π_{Softmax} securely computes shared confidence vector $\langle\vec{s}\rangle \in \mathbb{Z}_{2^\ell}^{n\times 1}$. The confidence value $\vec{s}[i]$ is computed from $\frac{e^{\vec{z}[i]}}{\sum e^{\vec{z}[j]}}$, which involves computing two non-linear operations, secure exponential Π_{Exp} [25] and secure reciprocal Π_{Rec} [30]. Our secure softmax protocol Π_{Softmax} works as follow:

```
1) $P_i$ sets $\vec{s} := (0,\cdots,0) \in \mathbb{R}^{n\times 1}$.
2) $P_1, P_2, P_3$ invoke $\langle\vec{s}\rangle \leftarrow \Pi_{\mathrm{Shr}}(\vec{s})$. //Share public value.
3) for $i \in \{0,\cdots,n-1\}$
4) $P_1, P_2, P_3$ invoke $\langle\vec{s}[i]\rangle \leftarrow \Pi_{\mathrm{Exp}}(\langle\vec{z}[i]\rangle)$.
5) end for.
6) $P_1, P_2, P_3$ invoke $\langle\Sigma\rangle \leftarrow \sum_{j=0}^{n-1}\langle\vec{s}[j]\rangle$.
7) $P_1, P_2, P_3$ invoke $\langle\Sigma^{-1}\rangle \leftarrow \Pi_{\mathrm{Rec}}(\langle\Sigma\rangle)$.
8) for $i \in \{0,\cdots,n-1\}$
9) $P_1, P_2, P_3$ invoke $\langle\vec{s}[i]\rangle \leftarrow \langle\Sigma^{-1}\rangle\cdot\langle\vec{s}[i]\rangle$.
10) end for.
```

The derivative of the softmax function can be represented by a Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{n\times n}$ ($n := len(\vec{s})$). Each element of this matrix is given by:

$$\frac{\partial\vec{s}[i]}{\partial\vec{e}[j]} = \boldsymbol{J}[i][j] = \vec{s}[i]\cdot(\delta_{ij}-\vec{s}[j]), \tag{2}$$

where $\delta_{ij} = 1$ if and only if $i = j$. We define the secure derivative of the softmax function as Π_{DSoftmax} (Protocol 4).

2) Differentiate secure utility loss function: The gradients of the utility loss $\mathfrak{L}2$ can be computed as below:

$$\begin{aligned}\frac{\partial\mathfrak{L}2}{\partial\vec{e}} &= c_2\cdot\frac{\partial\mathrm{ReLU}(max_{i,i\neq l}(\vec{z}[i]+\vec{e}[i])-\vec{z}[l]-\vec{e}[l])}{\partial\vec{e}}\\ &=\vec{\sigma}_{l'}-\vec{\sigma}_l,\end{aligned} \tag{3}$$

where $l' := arg\max_i(\vec{z}[i]+\vec{e}[i])$ and $\vec{\sigma}_l := (0,0,\cdots,+1,\cdots,0)$. $\vec{\sigma}_l$ is an one-hot vector with the l-th entry equals to 1. If the perturbed confidence vector's prediction label is identical to the original predicted label l, the loss and its gradients will be equal to $\vec{0}$. Otherwise, the protocol outputs a vector of $\vec{\sigma}_{l'}-\vec{\sigma}_l = (0,0,+1,\cdots,-1,\cdots,0)$, with the index of l to be -1 and the index of l to be 1. To secretly compute $\vec{\sigma}_{l'}-\vec{\sigma}_l$, the indices of l and l' should not be known by any party, so it is impossible to directly set $+1$ and -1 to the index we want. Thus, we propose Secure One-hot Argmax Π_{1AGM} to compute the operation.

Secure one-hot argmax. On input of $\langle\vec{x}\rangle$, Π_{1AGM} outputs the one-hot vector in secret $\vec{\sigma}_{l'}$, where $l' = arg\max_i(\vec{x}[i])$ is the index of the maximum value in $\vec{x}$. Π_{1AGM} iteratively compares each value to maintain the maximum value. Simultaneously, it preserves a one-hot vector where the index corresponding to the current maximum value is set to one. Then, parties extract the MSB from the secret shares. By utilizing a boolean share, parties invoke Π_{BMS} to compute $[\![\mathrm{b}]\!]\cdot(\langle\vec{x}[j]\rangle-\langle max\rangle)$. Our secure one-hot argmax protocol Π_{1AGM} works as follows:

```
1) $P_i$ sets $\langle max\rangle \leftarrow \langle\vec{x}[0]\rangle$ as the first element of $\langle\vec{x}\rangle$.
2) $P_i$ sets $\vec{\sigma_0} := (1,\cdots,0) \in \mathbb{R}^{n\times 1}$ with all entries zero.
3) $P_i$ shares $\langle\vec{y}\rangle \leftarrow \Pi_{\mathrm{Shr}}(\vec{\sigma_0})$ as RSS.
4) for $j \in \{1,\cdots,n-1\}$
5) $P_1, P_2, P_3$ invoke $[\![\mathrm{b}]\!] \leftarrow \Pi_{\mathrm{A2B}}(\langle max\rangle-\langle\vec{x}[j]\rangle)$ to extract
   the MSB $[\![\mathrm{b}]\!]$. Here, $\mathrm{b}=0$ indicates that $max \geq \vec{x}[j]$, and
   $\mathrm{b}=1$ otherwise.
6) $P_1, P_2, P_3$ invoke $\langle max\rangle \leftarrow [\![\mathrm{b}]\!]\cdot(\langle\vec{x}[j]\rangle-\langle max\rangle)+\langle max\rangle$.
7) $P_1, P_2, P_3$ invoke $\langle\vec{y}\rangle \leftarrow [\![\mathrm{b}]\!]\cdot(\langle\vec{\sigma_j}\rangle-\langle\vec{y}\rangle)+\langle\vec{y}\rangle$ to get the
   resulting one-hot max value vector $\langle\vec{y}\rangle$.
8) end for.
```

3) Differentiate secure perturbation loss function: The derivative of the perturbation loss $\mathfrak{L}3$ can be expanded as:

$$\frac{\partial\mathfrak{L}3}{\partial\vec{e}} = c_3\cdot\frac{\partial\left\|\vec{s'}-\vec{s}\right\|_1}{\partial\vec{s'}-\vec{s}}\cdot\frac{\partial\vec{s'}-\vec{s}}{\partial\vec{e}}. \tag{4}$$

It breaks down the derivative into the product of derivatives associated with the L_1 norm and the softmax function. The derivative of the L_1 norm can be computed in Π_{ABS} for each entry of $\vec{s'}-\vec{s}$ and the softmax function's gradient can be computed with Equation 2.

4) Protocol: Protocol 1 securely computes the gradient $\langle\vec{u}\rangle$ of the loss function $\mathfrak{L}$. On input replicated secret-shared $logits$ $\langle\vec{z}\rangle$, the original confidence vector $\langle\vec{s}\rangle$, noise vector $\langle\vec{e}\rangle$, a fully connected neural network h with its linear layer's weights $\{\langle\boldsymbol{W}_i\rangle\}_0^{L-1}$, and $\mathfrak{L}$'s weights $c_1, c_2, \langle c_3\rangle$, the secure noise optimization OPT outputs RSS gradient $\langle\vec{u}\rangle$.

Parties first add noise vector to the $logits$ $\langle\vec{z}\rangle$. In Steps 2-3, parties invoke Π_{Softmax} and Π_{DSoftmax} to get the perturbed confidence vector $\langle\vec{s'}\rangle$ and Jacobian matrix $\langle\boldsymbol{J}\rangle$. In Step 4, parties invoke Π_{Forward} on membership classifier h to get ReLU's derivative $\{\langle\vec{g'}_i\rangle\}_0^{L-2}$ and the inference result $\langle m\rangle$ such that $h(\vec{s'}) = m$ (Protocol 5). In Step 5, parties invoke Π_{ABS} on $\langle m\rangle$ to get the sign of m, $\langle\mathrm{sign_m}\rangle$ In Step 6, parties run Π_{Backward} to get h's derivative $\langle\vec{g}\rangle$ (Protocol 6). In Step 7, parties compute the gradients of $\mathfrak{L}1$ as $\langle\vec{u}_1\rangle$. In Steps 8-9, parties invoke Π_{1AGM} on normal confidence vector $\langle\vec{s}\rangle$ and perturbed one $\langle\vec{s'}\rangle$ to get $\langle\vec{\sigma}_l\rangle$ and $\langle\vec{\sigma}_{l'}\rangle$. In Step 10, parties compute the gradients of $\mathfrak{L}2$ as $\langle\vec{u}_2\rangle$. In Step 11, parties invoke

Π_{A2B} on each element of $\langle\vec{s'}\rangle - \langle\vec{s}\rangle$ to its MSB, denoted as $[\![\mathrm{b}_0]\!]\cdots[\![\mathrm{b}_{\mathrm{n}-1}]\!]$. In Step 12, parties compute the sign on each element. In Step 13, parties compute the gradients of $\mathfrak{L}3$ as $\langle\vec{u}_3\rangle$. In Step 14, parties output the composite gradient $\langle\vec{u}\rangle$.

B. Secure noise validation

VAL takes the shares of noise vector $\langle\vec{e}\rangle$ as input and outputs a bit $[\![b]\!]$ indicates that whether $\langle\vec{e}\rangle$ is valid to be added to the *logits* $\vec{z}$ or not. VAL performs the final step before MemGuard returns the crafted noise vectors, as outlined in Step 12 of Algorithm 1. VAL outputs $[\![b]\!]$ where $b = 1$ iff two conditions are satisfied: 1) whether the share of crafted noise $\langle\vec{e}\rangle$ do not change the original prediction; 2) whether the share of crafted noise $\langle\vec{e}\rangle$ change the neural network h's membership prediction. The first condition requires to compute the equality by parties invoking Π_{EQZ} [38]. The second condition is equivalent to computing its sign, where parties can invoke Π_{LTZ} to compute.

Secure equal to zero. To determine if two RSSs, $\langle a\rangle$ and $\langle b\rangle$, are equal, the problem is reduced to evaluating whether $\langle a\rangle - \langle b\rangle$ equals zero. We use Π_{EQZ} [38] for the secure computation of this equality. Parties first invoke Π_{A2B} to convert the result of $\langle a\rangle - \langle b\rangle$ into bits. Then, they perform successive OR operations $\bigvee$ on each bit [39]. The result of these operations is XORed with 1 and returned. On input of $\langle x\rangle$, Π_{EQZ} [38] securely computes whether a secret value equals to zero as follows:

1) $[\![x[0]]\!], \cdots, [\![x[\ell-1]]\!] \leftarrow \Pi_{\mathrm{A2B}}(\langle x\rangle)$.
2) $[\![b]\!] \leftarrow 1 \oplus \bigvee_{i=0}^{\ell-1}([\![x[i]]\!])$.

1) Protocol: Protocol 2 securely validates whether the generated noise vector is effective. At the beginning, parties hold *logits* $\langle\vec{z}\rangle \in \mathbb{Z}_{2^\ell}^{n\times1}$, original confidence vector $\langle\vec{s}\rangle \in \mathbb{Z}_{2^\ell}^{n\times1}$, generated noise vector $\langle\vec{e}\rangle$, original predicted label $\langle l\rangle$, original neural network's output $\langle m\rangle$, and neural network h with its linear layer's weights $\{\langle\boldsymbol{W}_i\rangle\}_0^{L-1}$.

Parties first compute perturbed *logits* $\langle\vec{z'}\rangle$. In Step 2, parties invoke Π_{Softmax} on $\langle\vec{z'}\rangle$ to get $\langle\vec{s'}\rangle$. In Step 3, parties invoke Π_{Argmax} [40] on $\langle\vec{s'}\rangle$ to get $\langle l'\rangle$. In Step 4, parties invoke Π_{EQZ} on $\langle l'\rangle - \langle l\rangle$ to get $[\![b_1]\!]$. In Step 5, parties invoke Π_{Forward} on h, $\{\langle\boldsymbol{W}_i\rangle\}_0^{L-1}$, $\langle\vec{s'}\rangle$ to get $\langle m\rangle$. In Step 6, parties invokes Π_{LTZ} (Less Than Zero [38]) on $\langle m\rangle\cdot\langle m'\rangle$ to get $[\![b_2]\!]$. Since $m \cdot m' < 0$ indicates a shift in the neural network h's prediction from positive to negative or vice versa, it implies that the noise vector has caused the prediction's membership information to change. In Step 7, parties output $[\![b]\!]$.

Protocol 2: Secure Noise Validation VAL

Input: Parties hold *logits* $\langle\vec{z}\rangle \in \mathbb{Z}_{2^\ell}^{n\times1}$, original confidence vector $\langle\vec{s}\rangle \in \mathbb{Z}_{2^\ell}^{n\times1}$, noise vector $\langle\vec{e}\rangle$, original predicted label $\langle l\rangle$, original neural network's output $\langle m\rangle$, and neural network h with its linear layer's weights $\{\langle\boldsymbol{W}_i\rangle\}_0^{L-1}$.
Output: Parties hold validation result $[\![b]\!]$.

```
1: $\langle\vec{z'}\rangle \leftarrow \langle\vec{e}\rangle + \langle\vec{z}\rangle$.
2: $\langle\vec{s'}\rangle \leftarrow \Pi_{\mathrm{Softmax}}(\langle\vec{z'}\rangle)$.
3: $\langle l'\rangle \leftarrow \Pi_{\mathrm{Argmax}}(\langle\vec{s'}\rangle)$.
4: $[\![b_1]\!] \leftarrow \Pi_{\mathrm{EQZ}}(\langle l'\rangle - \langle l\rangle)$.
5: $\langle m'\rangle \leftarrow \Pi_{\mathrm{Forward}}(h, \{\langle\boldsymbol{W}_i\rangle\}_0^{L-1}, \langle\vec{s'}\rangle)$.
6: $[\![b_2]\!] \leftarrow \Pi_{\mathrm{LTZ}}(\langle m\rangle\cdot\langle m'\rangle)$.
7: return $[\![b]\!] \leftarrow [\![b_1]\!] \wedge [\![b_2]\!]$.
```

C. Refinement I: mitigating potential leakages from iterations

Applying RSS to a single secure branching operation is relatively straightforward. A standard approach is to use a secret-shared selection bit and its Two's complement to multiply the secret-shared condition results and obliviously choose between them [41]. For example, parties compute $[\![b]\!]\cdot\langle x\rangle + (1\oplus[\![b]\!])\cdot\langle y\rangle$ to securely branch between $\langle x\rangle$ and $\langle y\rangle$.

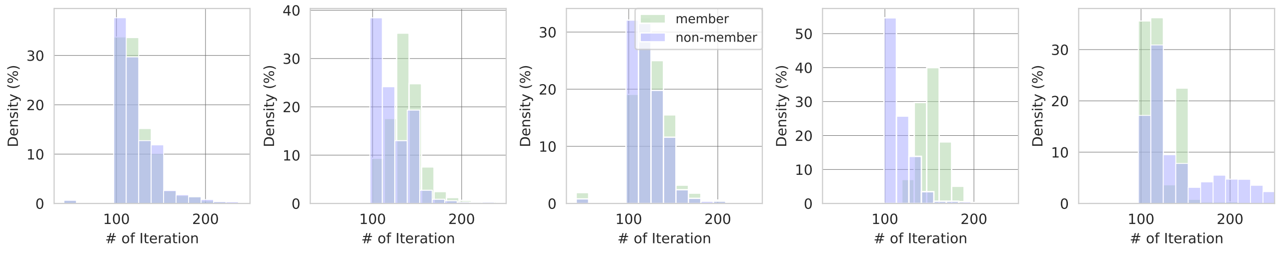
However, MemGuard, as shown in Algorithm 1, involves two `while` loops to recursively optimize the loss functions. If directly adapted to MPC, the conditions for evaluating `while` loops are in the encrypted domain. Those secret conditions cannot be used to determine when to terminate the loop [41]. Thus, parties have to open the secret conditions. Furthermore, the number of loops could also reveal certain information about the `while` conditions. For instance, if the loop terminates after the first loop, the adversary should infer that the share of the vector with zero entries is returned, feeling confident to make MIAs on that data sample.

Considering that the adversary can corrupt a single party, it naturally becomes aware of the defense mechanism. In general cases, different data samples should make SIGuard to terminate at different optimization stages. In Figure 4, we count the number of iterations in the while loop for each data sample during the execution of SIGuard. The experiment demonstrates there exist differences in iteration counts across data samples are linked to their membership information. For instance, there is no member data when the number of iterations is greater than 160 in Figure 4e.

We emphasize that such leakage issues inherently exist in recursive-based algorithms. When these algorithms are applied in MPC, careful attention must be paid to their iteration count. If the issue is placed in the plaintext configuration of black-box membership inference attacks (MIAs), the adversary can still measure the iteration by counting the service's running time; however, network latency or other factors may introduce some level of ambiguity. If the issue is placed in our threat model, the adversary can count the iteration sample by sample.

For intuition, we think the different distribution of total iteration number between member and non-member data is caused by each sample's optimization hardness. In the optimization of MemGuard, some data samples make it easy to find noise vectors. In such cases, MemGuard keeps increasing c_3 and begins a new round of iteration. Conversely, for data samples that are difficult to find noise vectors or smaller noise vectors, MemGuard terminates immediately.

With our insight of SIGuard's optimization mechanism, we make the following refinement: we substitute the while loops in the MemGuard (Steps 3, 6 of Algorithm 1) with two fixed `for` loops, parameterized by *outer_loop* and *inner_loop*.



(a) CIFAR-10 (b) CIFAR-100 (c) CH-MNIST (d) Location30 (e) Texas100

Fig. 4: Comparison of iteration frequency between member and non-member samples.

Protocol 3: SIGuard

Input: Parties hold *logits* $\langle \vec{z} \rangle \in \mathbb{Z}_{2^\ell}^{n\times 1}$, neural network h with its linear layer's weights $\{\langle \boldsymbol{W}_i \rangle\}_0^{L-1}$, learning rate $\langle \beta \rangle$, loss function $\mathfrak{L}$'s weights $c_1, c_2, \langle c_3 \rangle$, and iteration parameters *outer_loop*, *inner_loop*.

Output: Parties hold crafted noise vector $\langle \vec{e'} \rangle \in \mathbb{Z}_{2^\ell}^{n\times 1}$.

```
1:  $\langle \vec{e'} \rangle \leftarrow \Pi_{\text{Shr}}(0, \cdots, 0)$.
2:  $\langle \vec{s} \rangle \leftarrow \Pi_{\text{Softmax}}(\langle \vec{z} \rangle)$.
3:  $\_, \langle m \rangle \leftarrow \Pi_{\text{Forward}}(h, \{\langle \boldsymbol{W}_i \rangle\}_0^{L-1}, \langle \vec{s} \rangle)$.
4:  for $i \leftarrow 0$ to $outer\_loop$ do
5:      $\langle \vec{e} \rangle \leftarrow \Pi_{\text{Shr}}(0, \cdots, 0)$.
6:      for $j \leftarrow 0$ to $inner\_loop$ do
7:          $\langle \vec{u} \rangle \leftarrow \text{OPT}(\langle \vec{z} \rangle, \langle \vec{s} \rangle, \langle \vec{e} \rangle, h, c_1, c_2, \langle c_3 \rangle)$.
8:          $\langle \vec{u_{sq}} \rangle \leftarrow \langle \vec{u} \rangle \cdot \langle \vec{u} \rangle$.
9:          $\langle \Sigma \rangle \leftarrow \sum_{j=0}^{n-1} \langle \vec{u_{sq}}[j] \rangle$.
10:         $\langle \vec{u} \rangle \leftarrow \Pi_{\text{InvSqrt}}(\langle \Sigma \rangle) \cdot \langle \vec{u} \rangle$.
11:         $\langle \vec{e} \rangle \leftarrow \langle \vec{e} \rangle - \beta \cdot \langle \vec{u} \rangle$.
12:     end for
13:     $[\![b]\!] \leftarrow \text{VAL}(\langle \vec{z} \rangle, \langle \vec{s} \rangle, \langle \vec{e} \rangle, \langle l \rangle, \langle m \rangle, h)$.
14:     $\langle \vec{e'} \rangle \leftarrow [\![b]\!] \cdot (\langle \vec{e} \rangle - \langle \vec{e'} \rangle) + \langle \vec{e'} \rangle$.
15:     $\langle c_3 \rangle \leftarrow [\![b]\!] \cdot (\langle 10 \cdot c_3 \rangle - \langle c_3 \rangle) + \langle c_3 \rangle$.
16: end for
17: return $\langle \vec{e'} \rangle$.
```

D. Refinement II: balance efficiency & accuracy

MemGuard's hyper-parameters for maximum iterations are no longer practical in SIGuard, as it crafts noise vectors using a fixed number of iterations as refined in Section V-C. According to MemGuard's code[1], it sets maximum iterations for the `while` loops in Steps 3 and 6 of Algorithm 1 to 6 and 300, respectively. However, in MPC, such a setting is impractical due to the significantly higher runtime cost compared to the plaintext computation. MemGuard's iteration limits of 6 and 300 serve only as the upper bound within the while loop, meaning that each data sample's optimization process varies. In SIGuard, using fixed iterations means that some data, which can find smaller noise vectors when the weights of $\mathfrak{L}3$ are increased, will fully utilize all the iterations. Conversely, data that cannot find smaller noise vectors or any feasible solutions will waste computation by looping without progress. This is a side-effect of setting iterations fixed, and we certainly do not want this scenario to significantly harm efficiency. Therefore, we need to balance the total number of iterations (efficiency) and defense performance (accuracy).

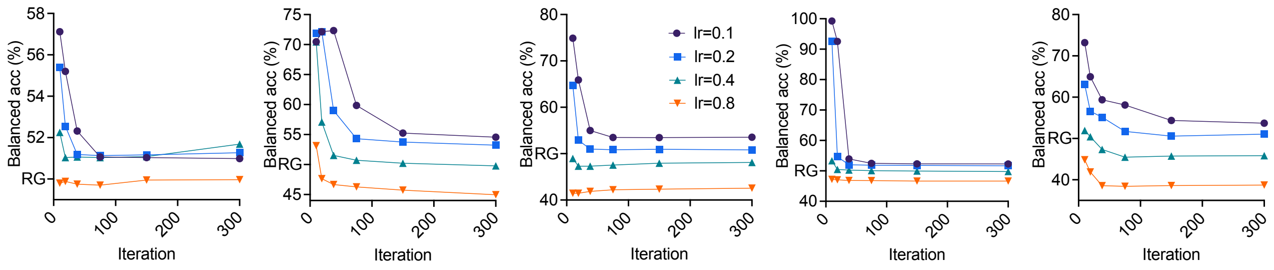
[1]https://github.com/jinyuan-jia/MemGuard

We need to demonstrate that through parameter selection, our solution should not only reduce iterations to improve efficiency but also maintain SIGuard′s utility. This means that even with reduced iterations, SIGuard can still achieve a performance level close to that of a random guess (50%). Therefore, for each choice of *inner_loop* (the iteration count from the inner loop in Protocol 3), we adjust the learning rates β. The results of these runs, for various iteration counts and learning rates, are displayed in Figure 5. We evaluate defense performance by launching the MIA of NN-M. In summary, these results indicate that optimization with fewer iterations but a higher learning rate achieves defense performance comparable to that of more iterations with a lower learning rate. For example, in the dataset of CIFAR-10, setting the iteration as 19 and the learning rate as 0.4 can achieve a better performance than setting the iteration as 300 and the learning rate as 0.1 Meanwhile, we reduce *outer_loop*, which is mainly used to control the norm size of the generated noise, from 6 to 3, as MemGuard's evaluation shows $outer\_loop = 3$ and $outer\_loop = 6$ achieve nearly the same performances [21].

Thus, based on our empirical studies of performing selections of the hyper-parameters, we make the following refinement: We reduce the number of iterations *inner_loop* and increase the learning rate β. Based on our empirical studies, for each dataset, we can find proper hyper-parameters to balance the efficiency and utility.

E. Analysis of SIGuard on softmax approximations

Securely computing the softmax and its derivatives is essential in SIGuard's computation. This is because the underlying defense mechanism adopted by our system is MemGuard [21], which crafts noise vectors to perturb confidence scores and then optimizes the noise vectors to reduce the accuracy loss using a series of loss functions (i.e., Equation 1). Such computations of $\mathfrak{L}1$ and $\mathfrak{L}3$ extensively involve softmax. An improper softmax approximation might lead to incorrect convergence of those loss functions in the optimization [27]. Moreover, since the adversary's membership classifiers can be derived from the



(a) CIFAR-10 (b) CIFAR-100 (c) CH-MNIST (d) Location30 (e) Texas100

Fig. 5: Comparison of SIGuard's defense performance under different learning rates and iterations ($inner\_loop \in \{10, 19, 38, 75, 150, 300\}$).

actual softmax function, inaccurate approximations may fail to capture these classifiers effectively. In a nutshell, how the choice of softmax approximations would impact the defense strength is still unknown.

Secure softmax selection. As shown in Table X, we adapt different softmax approximations into SIGuard to evaluate how different approximations affect the defense performance of SIGuard. We observe that using CrypTen and AS19 will give us better performance, compared to using Piranha and SecureML. The use of the softmax approximation with SecureML shows that SIGuard struggles to achieve defense degradation across CIFAR-10 and CIFAR-100. The softmax approximation using Piranha outperforms SecureML but does not compare favorably to the approximations of CrypTen and AS19. The defense performance of SIGuard using CrypTen and AS19 is nearly equivalent, effectively addressing MIAs.

F. Wrapping up

Protocol 3 presents the final protocol for SIGuard. SIGuard's protocol comprises two recursive optimizations: an inner loop (Steps 6-12) and an outer loop (Steps 4-16). The inner loop is to find an optimal noise vector in the encrypted domain that can be used to perturb the secret-shared logits $\langle \vec{z} \rangle$ such that the MIAs attack accuracy is around the random guess ($\sim$50%). In Step 10, to securely compute the inverse square root, parties invoke Π_{InvSqrt} [31]. The outer loop is to validate whether the current secret-shared noise vector $\langle \vec{e} \rangle$ is as small as possible to preserve the inference accuracy. Once the above two conditions of loops are achieved, all cloud servers obtain the secret shares of crafted noise vector $\langle \vec{e'} \rangle \in \mathbb{Z}_{2^\ell}^{n\times 1}$, which can then be added to the encrypted confidence vector, and opened to the user as the final prediction result.

VI. Security Proof

***Theorem** 1:* The protocol of SIGuard securely realizes the ideal functionality $\mathcal{F}_{\mathsf{SIGuard}}$ using Definition 1.

Proof. To prove Theorem 1, we let $\mathcal{S}$ simulate the joint distribution of $\mathcal{F}_{\mathsf{SIGuard}}$'s inputs and outputs in the ideal world, and then demonstrate that the simulated distribution is computationally indistinguishable from the real-world distribution. We construct $\mathcal{S}$ as follows:

Simulation for Corrupted Model Owner. Model owner $\mathcal{O}$ does not participate in the computation but contributes to the parameters of the simulated membership classifier. In the ideal world, the simulator $\mathcal{S}$ plays as the corrupted model owner by inputting the parameters $\boldsymbol{W}$ of the simulated membership classifier to the ideal functionality $\mathcal{F}_{\mathsf{SIGuard}}$. The remaining protocol Π_{SIGuard} is invoked by three honest parties. Finally, only the share of the perturbed confidence vector $\langle \vec{s'} \rangle$ is returned to the user $\mathcal{U}$, while nothing is returned to $\mathcal{O}$. In the simulation, $\mathcal{S}$ can simply output $\boldsymbol{W}$ in a dummy way. The output of $\mathcal{S}$ is identically distributed to the view of $\mathcal{O}$, making the real and ideal worlds indistinguishable.

Simulation for corrupted user. The user $\mathcal{U}$ does not participate in the computation but contributes its data, denoted as $\vec{x}$, by partitioning $\vec{x}$ into replicated secret shares $\langle \vec{x} \rangle$ and distributing them to each party. In the ideal world, the simulator $\mathcal{S}$ plays as the corrupted user by inputting the $\vec{x}$ to the ideal function that computes the secure inference and $\mathcal{F}_{\mathsf{SIGuard}}$. Upon receiving the inference result $\vec{s'}$ from $\mathcal{F}_{\mathsf{SIGuard}}$, $\mathcal{S}$ samples two random vectors $\vec{r}_1, \vec{r}_2$ from $\mathbb{Z}_{2^\ell}$ and computes $\vec{r}_1, \vec{r}_2, \vec{r}_3 := \vec{s'} - \vec{r}_1 - \vec{r}_2$. Then $\mathcal{S}$ outputs $\vec{r}_1, \vec{r}_2$, and $\vec{r}_3$. The security follows from the fact that the replicated secret shares formed by $\vec{r}_1, \vec{r}_2$, and $\vec{r}_3$ are uniformly random values in $\mathbb{Z}2^\ell$ and are identically distributed to the output in the real world. Since the inputs are the same, the inputs and outputs in both the real and ideal worlds are indistinguishable.

Simulation for corrupted parties. We demonstrate that the view of the corrupted party from $\mathcal{A}$'s perspective can be simulated by $\mathcal{S}$. The three parties P_1, P_2, P_3 collaboratively invoke Π_{SIGuard} in a symmetric way and each party does not provide their own private inputs or receive any outputs. During the computation of Π_{SIGuard}, the servers interact with each other over replicated secret shares. Given the symmetric nature of replicated secret sharing, it is sufficient for $\mathcal{S}$ to simulate the view of only one party. $\mathcal{S}$ then sequentially simulates the subroutines that have been proven secure in previous works by showing that the transmitted messages are uniformly distributed over $\mathbb{Z}_{2^\ell}$. Furthermore, between each execution of these subroutines, there are no interactive messages. Hence, $\mathcal{S}$ can simulate the views of the parties corrupted by $\mathcal{A}$, implying indistinguishability. The above concludes the proof. $\square$

Secure against MIAs. Assume MemGuard perfectly defends against MIAs, which means that the adversary $\mathcal{A}$, who colludes one of three parties and multiple service users, cannot correctly guess the membership information with a non-negligible advantage more than 50%. In our solution, MPC ensures that no meaningful intermediate results are disclosed to any party. With SIGuard being proved as secure, $\mathcal{A}$ cannot get access to any meaningful intermediate results in the 3PC protocol, such as the generated noise vector. Since $\mathcal{A}$ cannot access any meaningful intermediate results, it only sees the final results, the perturbed confidence vectors. Because MemGuard perfectly defends against MIAs, our solution is secure against MIAs, too.

VII. Evaluation

Implementation. We implemented SIGuard [2] in 2,988 lines of Python code. We leverage the MP-SPDZ [3] [42] framework as the skeleton of SIGuard. MP-SPDZ is a toolkit that includes various secure protocols realized by replicated secret sharing [26]. The neural network models are trained by PyTorch. To evaluate defense performance and efficiency in the LAN setting, we deployed SIGuard on a machine equipped with a 13th Gen Intel(R) Core(TM) i9-13900K, 12 CPUs and 24 GB of RAM. For efficiency in the WAN setting, we deploy SIGuard in an Amazon AWS EC2 workspace equipped with a 4th Gen Intel Xeon Sapphire Rapids processor, featuring 8 CPUs and 32 GB of RAM. The WAN setting was simulated using the Linux tc, configured with a bandwidth of 100 Mbps and a latency of 20 ms across three Docker containers.

A. *Defense performance of SIGuard*

First, we demonstrate our SIGuard effectively reduces the risk of MIAs when evaluated using Bal. Acc. and TPR @ Low FPR. Second, we compare SIGuard with SOTA defense, HAMP [20]. HAMP injects noises(soft labels) during training and applies tailored perturbations to the confidence vector during inference. Due to its inference-stage design, we believe further adaptation is needed for use in MPC.

Balance accuracy mitigation. Table III demonstrates that SIGuard reduces Bal. Acc. across all MIAs. For each dataset, we selected the optimal hyper-parameters for *inner_loop* and learning rate β, as determined by our refinement process. Under the attack of NN-M, SIGuard reduces the Bal. Acc. from 57.60% to 50.30% on CIFAR-10, bringing the attack accuracy close to that of random guessing (50%). For the rest of MIAs, SIGuard can mitigate their effectiveness to a certain degree and sometimes to the level of random guessing.

TPR @ Low FPR mitigation. Figure 6 plots the log-scale ROC curves for LiRA, showing the MIA privacy loss at a TPR of 0.1% FPR. Figure 6 demonstrates that SIGuard can significantly reduce the accuracy of MIAs at low FPR. For example, at an FPR of 0.1%, LiRA achieves a TPR of nearly 10% in Location30, whereas, SIGuard reduces it to to below 1%. Similarly, for the remaining datasets, SIGuard significantly reduces the TPR, with a degradation pattern consistent with that of MemGuard.

[2]https://github.com/Wangxinqian/SIGuard-secure-MIA-defense

[3]https://github.com/data61/MP-SPDZ

TABLE III: Attack performance in PPML w/o SIGuard.

Dataset	Group	NN-M	conf	entr	Mentr	LiRA
CIFAR-10	PPML	57.60	58.35	56.80	58.75	61.95
	SIGuard	50.30	57.35	52.00	57.60	44.05
	Δ	↓ **7.30**	↓ **1.00**	↓ **4.80**	↓ **1.15**	↓ **17.90**
CIFAR-100	PPML	68.85	73.00	65.10	73.00	82.55
	SIGuard	54.45	67.10	53.45	67.85	67.20
	Δ	↓ **14.40**	↓ **5.90**	↓ **11.65**	↓ **5.15**	↓ **15.35**
CH-MINIST	PPML	76.95	72.10	70.50	71.50	69.60
	SIGuard	50.90	54.45	48.90	55.70	61.55
	Δ	↓ **26.05**	↓ **17.65**	↓ **21.60**	↓ **15.80**	↓ **8.05**
Location30	PPML	98.93	92.40	92.40	92.86	71.73
	SIGuard	51.86	50.00	50.00	50.00	71.00
	Δ	↓ **47.07**	↓ **42.40**	↓ **42.40**	↓ **42.86**	↓ **0.73**
Texas100	PPML	78.00	75.85	73.00	75.65	72.70
	SIGuard	53.90	58.40	52.45	59.15	72.55
	Δ	↓ **24.10**	↓ **17.45**	↓ **20.55**	↓ **16.50**	↓ **0.15**

TABLE IV: Model utility and attack performance in PPML under no defense, SIGuard, and HAMP.

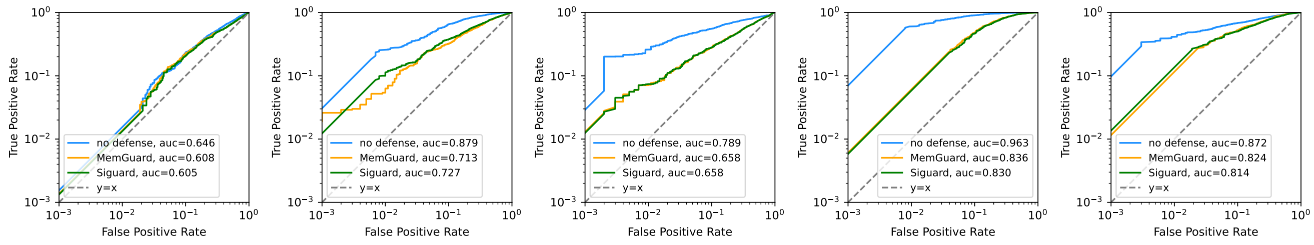
Dataset	Group	Pred Acc	NN-M
CIFAR-10	PPML	90.70%	59.20%
	SIGuard	90.70% (↓ **0%**)	52.53% (↓ **6.67%**)
	HAMP	87.20% (↓ **3.50%**)	50.00% (↓ **19.20%**)
CIFAR-100	PPML	63.15%	73.05%
	SIGuard	63.15% (↓ **0%**)	53.68% (↓ **19.37%**)
	HAMP	62.75% (↓ **0.40%**)	50.00% (↓ **23.05%**)
CINIC-10	PPML	82.95%	52.40%
	SIGuard	82.95% (↓ **0%**)	51.32% (↓ **1.08%**)
	HAMP	81.55% (↓ **1.40%**)	50.00% (↓ **2.40%**)

Comparison to prior art (defend MIAs in training). Table IV shows that our SIGuard and HAMP achieve same-level defense effectiveness on MIAs of NN-M. Additionally, we want to emphasize that our goal is not to develop a better MIA defense algorithm. Table IV also illustrates the trade-off between privacy and utility, highlighting the cost a model must incur to achieve improved defense performance. Although HAMP has a higher Bal. Acc., it results in a slight decrease in model inference accuracy compared to SIGuard, which has no loss in inference accuracy.

B. *Efficiency of SIGuard*

We analyze how different hyper-parameters of SIGuard affect the total running time and communication cost in both WAN and LAN settings. Then, we evaluate the feasibility of SIGuard for practical secure inference services under both WAN and LAN conditions. Last, we compare plaintext model training with HAMP [20] and DP-SGD [17], the defense mechanisms that run in the training stage, to show the extra cost introduced in total running time and inference accuracy when selecting defense in the training stage. Since SIGuard is applied during the secure inference stage, we assume that the model is trained in plaintext before being used in MPC.

LAN setting. Table V examines the communication and computation costs for various iteration counts across all input



(a) CIFAR-10 (b) CIFAR-100 (c) CH-MNIST (d) Location30 (e) Texas100

Fig. 6: Log scale ROC CRUVE of LiRA under no defense, MemGuard, and SIGuard.

TABLE V: Total running time (seconds) and bandwidth cost (MB) of SIGuard over $\mathbb{Z}_{2^{64}}$ with different iteration and input length $\{8, 10, 30, 100\}$. Note that the iteration (in 1st column) represents $outer\_loop \times inner\_loop$.

Iteration	Cost	CH-MINIST	CIFAR-10	Location30	CIFAR-100
3×10	Time(s)	0.99	1.09	1.30	2.41
	Bandwidth (MB)	4.66	5.38	17.52	130.12
3×19	Time(s)	1.83	1.87	2.38	5.13
	Bandwidth (MB)	8.54	9.87	32.16	240.50
3×38	Time(s)	3.55	5.04	4.77	9.06
	Bandwidth (MB)	16.73	19.36	63.64	489.18
3×75	Time(s)	6.91	7.03	9.59	18.02
	Bandwidth (MB)	32.68	37.84	124.96	963.60
3×150	Time(s)	14.41	14.86	19.10	36.78
	Bandwidth (MB)	65.10	75.41	249.29	1925.19
3×300	Time(s)	30.51	32.04	38.87	85.34
	Bandwidth (MB)	129.87	150.46	497.71	3848.34

shapes. The iteration counts are represented as $outer\_loop \times inner\_loop$, following the notation used in Protocol 3. For example, 3×19 means that our hyper-parameters for SIGuard are $outer\_loop = 3$ and $inner\_loop = 19$. Table V shows that the total time (s) and bandwidth costs (MB) of SIGuard increase linearly with the number of iterations. For example, with an input length of 30, when the number of iterations increases from 30 to 900, the running time and bandwidth cost both increase approximately 30 times, specifically by factors of 29.9 and 28.4, respectively.

Note that from Refinement II (Section V-D), we optimize the iterations from $inner\_loop = 300$ to $inner\_loop = 10$. According to Table V, the optimized iteration parameters ensure that for any input length of 100 or less, setting the number of iterations to 3×10 or lower can result in a total prediction time of less than 2.50 seconds, which is at least $29\times$ faster than iterations of 3×300.

Figure 7 further analyzes the communication and computation costs on the optimised iteration setting. It illustrates how the communication and computation costs increase with the length of SIGuard's input under the $inner\_loop = 10$ and 19, respectively. With iteration times set to 3×10, all inputs with lengths up to 400 have execution times under 10 seconds. Also, when the iteration times increase to 3×19, SIGuard still can process 200 inputs within 10 seconds.

Figure 8 illustrates SIGuard's breakdown costs. It shows

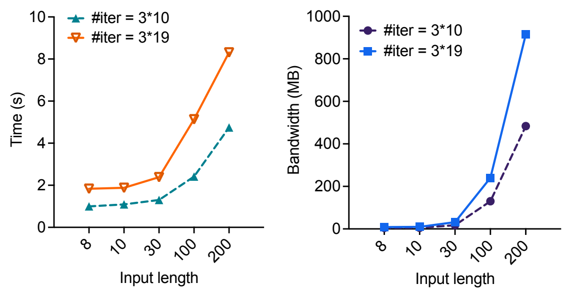


Fig. 7: Total running time (left) and bandwidth cost (right) with different input lengths under optimized $inner\_loop$ settings.

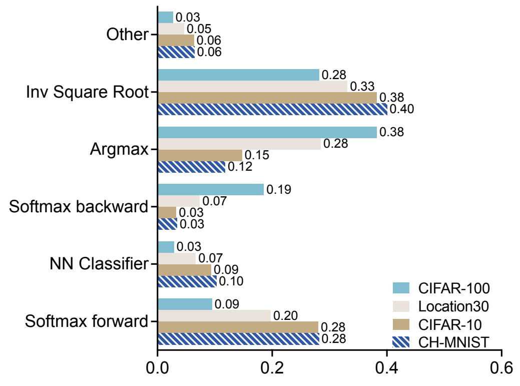


Fig. 8: Breakdown of SIGuard's secure protocols.

the non-linear computations of inverse square root and softmax dominate the execution time.

WAN setting. Table VI evaluates the total time (s) and bandwidth cost (MB) of SIGuard with the iteration of 3×10 and 3×19. Given the bottleneck for MPC protocols is communication/interaction, the end-to-end latency is increased. For instance, with an input length of 8 and 3×10 iterations, the total running time increased from approximately 1 second in the LAN setting to about 114 seconds in the WAN setting. We recommend SIGuard deploying with dedicated networking or cloud data centre networking for low latency.

Feasibility study. Figure 9 and Figure 10 show the time cost ratio for SIGuard in both LAN and WAN settings during secure inference of practical neural networks, using 3×10 iterations with an input length of 10. The overall

TABLE VI: Total running time (seconds) and bandwidth cost (MB) of SIGuard over $\mathbb{Z}_{2^{64}}$ with WAN setting.

Iteration	Cost	CH-MINIST	CIFAR-10	Location30	CIFAR-100
3×10	Time(s)	113.97	126.43	186.89	418.61
	Bandwidth (MB)	4.45	5.17	17.05	129.90
3×19	Time(s)	209.65	225.86	352.57	768.77
	Bandwidth (MB)	8.15	9.48	31.79	245.11

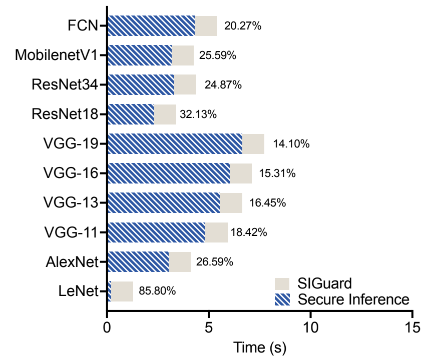


Fig. 9: Percentage of SIGuard in the overall secure inference time with iteration $= 3 \times 10$ and CIFAR-10 with LAN setting.

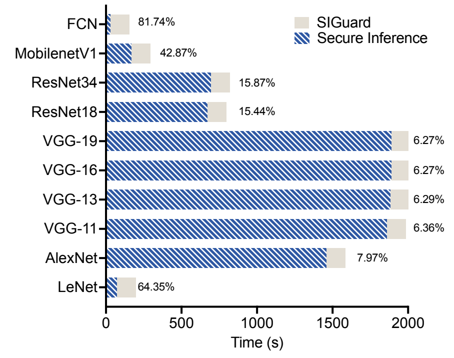


Fig. 10: Percentage of SIGuard in the overall secure inference time with iteration $= 3 \times 10$ and CIFAR-10 with WAN setting.

TABLE VII: Training overhead of HAMP and DP-SGD under different training sample sizes in maintaining the equivalent performance of standard plaintext training on CIFAR-10.

	Plaintext		HAMP		DP-SGD	
# data	Acc	Time (s)	Acc	Time (s)	Acc	Time (s)
5000	71.77%	599	57.56%	4971	37.96%	7868
10000	81.17%	1371	71.18%	8908	45.22%	13180
15000	85.24%	2058	80.83%	11814	47.99%	18565
20000	87.29%	2743	85.42%	15108	50.26%	24091
25000	88.58%	3464	87.59%	3595	52.97%	29188
30000	90.29%	4047	89.30%	4260	52.27%	34397
35000	90.86%	4798	90.06%	4657	54.62%	44377
40000	91.65%	4656	90.99%	4554	57.70%	45161

TABLE VIII: Training overhead of HAMP and DP-SGD under different training sample sizes in maintaining the equivalent performance of standard plaintext training on CIFAR-100.

	Plaintext		HAMP		DP-SGD	
# data	Acc	Time (s)	Acc	Time (s)	Acc	Time (s)
5000	31.86%	596	28.16%	4893	7.82%	7850
10000	47.00%	1333	43.86%	8740	10.37%	13100
15000	53.85%	2023	52.95%	2272	11.41%	18512
20000	59.62%	2763	59.18%	2904	13.84%	24042
25000	62.80%	3433	61.85%	2356	14.99%	14212
30000	65.58%	1968	64.92%	1281	14.59%	16837
35000	67.24%	2344	66.65%	1502	16.37%	24263
40000	68.14%	2703	68.08%	1662	16.56%	27750

TABLE IX: Comparison of model utility, attack performance, and training overhead under SIGuard, HAMP, and DP-SGD in training with fixed epoch numbers $\{150, 100, 150\}$.

	Evaluation	CIFAR-10	CINIC-10	CIFAR-100
	Train Size	33000	33000	33000
	Test Size	9000	27000	9000
Plaintext	Train Acc	99.33%	84.85%	96.83%
	Test Acc	88.73%	77.04%	62.18%
	Time (s)	3716.89	7387.98	3717.03
HAMP	Train Acc	96.60% (↓ **2.73%**)	82.34% (↓ **2.51%**)	93.50% (↓ **3.33%**)
	Test Acc	83.59% (↓ **5.14%**)	75.80% (↓ **1.24%**)	59.99% (↓ **2.19%**)
	Time (s)	4073.88 (↑ **356.99**)	8088.20 (↑ **700.22**)	4083.21 (↑ **366.18**)
DP-SGD	Train Acc	44.90% (↓ **54.43%**)	42.53% (↓ **42.32%**)	10.33% (↓ **86.50%**)
	Test Acc	44.36% (↓ **39.23%**)	42.08% (↓ **34.96%**)	10.31% (↓ **51.87%**)
	Time (s)	9569.26 (↑ **5849.37**)	19114.76 (↑ **11724.78**)	9477.32 (↑ **5760.29**)

secure inference includes the time for SIGuard. In the LAN setting, except LeNet and ResNet18, all the execution time of SIGuard only accounts for less than 27% of the overall secure inference cost. In the WAN setting, except LeNet, FCN, and MobilenetV1, all the execution time of SIGuard only accounts for less than 16% of the overall secure inference cost. Compared to the WAN and LAN settings, their ratios decrease by 9.00% to 18.62% across most neural network architectures, including AlexNet, the VGG series, and the ResNet series, demonstrating that SIGuard is scalable.

Comparison to prior art (defend MIAs in training). We train our target model using standard plaintext training for 200 epochs, then train the models with HAMP and DP-SGD for an extended period (600 epochs), stopping only if their inference accuracy gap is within 1% of the plaintext accuracy. Table VII and Table VIII illustrate that, given sufficient data, it is possible to train the target model to achieve equivalent performance of normal plaintext training. However, when the dataset size is small, the trained model tends to exhibit degraded performance. For example, Table VII shows that when the data samples are decreased from 25000 to 5000, the gap between HAMP and plaintext normal training is enlarged from 0.99% to 14.21%. Then, we set our training set with a sufficiently large dataset and used different training algorithms to train the target model with fixed epochs. Table IX still shows that using HAMP and DP-SGD training algorithms can lower the model's inference accuracy on the test dataset.

VIII. Discussion

A. Different security definitions

We emphasize that MemGuard and SIGuard are based on fundamentally different security definitions.

MemGuard [21]. MemGuard is tailored to reduce the adversaries' membership advantage, rendering their membership inference accuracy down to nearly 50%. In functionality, MemGuard generates noise vectors to perturb the outputs of confidence vectors, which fools the simulated membership classifier. The inherent reason for fooling a simulated membership classifier to help defend against real-world MIA adversaries is the use of the transferability of the adversarial examples [43]. However, there is limited theoretical analysis explaining why MemGuard works; most of its success is attributed to the empirical results [19].

SIGuard. The trust models of SIGuard and MemGuard differ significantly. MemGuard relies on a single trusted and honest party, ensuring that no privacy concerns arise during computation. In contrast, SIGuard operates across three independent parties, using a distributed trust model. An important security goal of SIGuard is to ensure that the parties running the protocols do not learn anything about the model owner's model parameters and the users' input data in the computation.

B. Defense MIAs in secure inference

We emphasize that defending against MIAs in secure inference is a new challenge. To defend plaintext MLaaS against MIAs, existing solutions rely on plaintext computation during training and inference. When defending secure inference as a service against MIAs, defense mechanisms used for plaintext inference require additional design, while training-stage defenses can be naturally integrated. However, when the model training takes place in an encrypted domain, it is impossible to directly apply plaintext defenses during training. In such scenarios, new solutions are needed to defend against MIAs in secure computation, which is the focus of SIGuard.

C. Adaptive attacks

An MIA adversary compromising a server potentially provides it with insights into the defense mechanism. As a result, the adversary may attempt to repeatedly interact with the server, submitting the same sample to test for membership multiple times to the server. SIGuard is designed to defend against such adaptive attacks. For repeated identical inputs, the perturbed output confidence vector remains stable, and the number of iterations for both member and non-member data is fixed during perturbation generation, ensuring no additional information leakage. Additionally, SIGuard securely implements MemGuard, which also accounts for adaptive attacks, reducing the adversary's inference accuracy to that of random guessing. Even if the adversary attempts to adjust its prediction, the MIA success rate remains close to 50%.

D. Viability of SIGuard

Perturbed confidence vector to defense MIAs. Studies of black-box MIAs place significant emphasis on analyzing confidence vectors. On the one hand, providing confidence vectors as final outputs is a common practice in MLaaS platforms, largely due to concerns of quality control [44]. When users question the predicted labels, confidence vectors can help determine whether to accept the prediction or flag it for human review. For instance, if the final predicted label does not align with the user's expectations, but their expected result is ranked second highest in confidence vectors, the user may still consider the prediction as a useful reference. Only returning a label might not be sufficient for applications with critical requirements on accuracy and analysis. On the other hand, recent studies have shown that confidence vectors are closely linked to the risk of MIAs [32], [33]. The work theoretically shows that the optimal membership inference for each data sample from the training distribution can be derived by evaluating its loss, which is computed from the confidence vectors. Therefore, in secure inference, we should place equivalent emphasis on protecting these vectors.

Non-colluding three servers. Note that the semi-honest assumption with only one corrupted server is realistic, as cloud-based MLaaS providers won't risk their business model and reputation to behave maliciously and deploy competent full-fledged intrusion detection [16]. Non-colluding cloud servers are used in practice (e.g., Facebook's Crypten, Cape's TF-Encrypted, and Mozilla Telemetry), with three servers deployed across clouds in separate trust domains.

E. The limitation of SIGuard

We acknowledge the existence of other privacy attacks, such as attribute inference attack [45], model extraction attack [46], data reconstruction attack [47], and property inference attack [48]. These attacks can be similarly risky in secure inference. However, SIGuard is only designed to defend against membership inference attacks, consequently, SIGuard can not mitigate these other types of privacy attacks. To the best of our knowledge, balancing the mitigation of various privacy attacks is challenging, and few studies have comprehensively addressed this. Effective defense against these privacy attacks should build upon the progress made in existing mitigation efforts. Furthermore, since the perturbations in SIGuard are applied to confidence vectors rather than actual predictions, we acknowledge the existence of label-only MIAs [49]; however, these are beyond the scope of this study.

IX. Related work

A. Privacy-preserving machine learning

Privacy-preserving Machine Learning (PPML) aims to enable encrypted computation over encrypted machine learning models and encrypted data. In essence, PPML protocols are designed to perform linear (e.g., multiplication and addition) and nonlinear (e.g., exponential and comparison) operations in the encrypted domain. In the literature, PPML is designed to support a wide range of machine learning model on inference and/or training, e.g., DNN [3], [22], [34], [37], [50], [51], [52], [53], [54], GNN [55], and Transformers [56], [57].

Most efficient PPML protocols are constructed using secure multi-party computation (MPC) techniques, which can be categorized under party settings. The two-party computation starts at the beginning of MPC-based PPML. The first MPC-based PPML work, CryptoNets [50], utilizes Fully Homomorphic Encryption (FHE) to perform addition and multiplication. For non-linear functions, CryptoNets adopts polynomial approximation to facilitate secure evaluation. SecureML [22] replaces FHE with additive secret sharing for efficiently computing addition and multiplication and utilizes Garbled Circuits (GC) to securely compute non-linear functions such as ReLU. However, it requires the generation of Beaver's triplets (extra communication due to Oblivious Transfer) before each multiplication operation.

For 3PC protocols, ABY^3 [2] employs Replicated Secret Sharing [26] (RSS) to perform multiplications, thereby eliminating the need to generate Beaver's triplets and improving efficiency by reducing one round of communication (55K times faster then SecureML compared in secure machine learning training). Furthermore, ABY^3 [2] supports hybrid operations by facilitating conversions between Arithmetic Secret Sharing, Boolean Secret Sharing, and GC. SecureNN [51], a three-party framework, relies solely on secret sharing, significantly enhancing efficiency. Beside the work mentioned above, considerable efforts [4], [23], [24], [52] have been made to perform various machine learning tasks.

On the other hand, the four-party computation protocols are proposed [58], [59], [60]. As the number of parties involved in the computation increases, the computational cost generally decreases, although the communication cost tends to rise. Furthermore, the system's resilience to adversaries increases. For instance, if two parties are corrupted by adversaries, the remaining two parties can still complete the computation using two-party computation protocols.

B. Output privacy in secure inference

Output privacy refers to any privacy leakage arising from the output of the PPML framework. In our work, we specifically focus on privacy leakage exploited by machine learning attacks, which are membership inference attacks (MIAs). The issue of output privacy was identified in previous studies. Delphi [3] noted that most MPC-based PPML systems cannot conceal information that is eventually disclosed. Namely, the adversary who controls a client can learn the output (label, confidence vector) of a PPML inference protocol and perform MIAs, just like launching MIAs in the plaintext inference scenario. However, there are only a few studies addressing output privacy for PPML. To mitigate this risk, Ruan *et al.* [15] propose sophisticated machine learning training techniques using *Differential Privacy*. However, DP will bring utility loss which has been well known.

C. Membership inference attack and defense

MIAs aim to determine whether a specific data record belongs to the target model's training dataset. Shokri *et al.* [6] propose the first MIA against ML models. After that, one line of work focuses on investigating MIAs over various attack settings, e.g., white-box access [32], [33], black-box access [11], [12], and label-only access [49]. Another line of work attempts to adapt MIAs to diverse ML models, e.g., graph neural networks [61], [62], generative models [63], and recommendation systems [64]. To mitigate MIAs, existing solutions can be categorized into provable defenses and practical defenses [19], [20]. Provable defenses leverage *Differential Privacy* [17] to inject carefully crafted noise for obscuring the gradient information during training. However, such an approach often results in reduced model accuracy [17]. Practical defenses sophisticated advanced training algorithms [9], [18], [19], [20] or introduce carefully crafted noise to inference results [21], effectively defending against MIAs in empirical studies. Compared to defending against MIAs during training, defending against MIAs during inference [21] is more efficient and can be easily implemented without the need for time-consuming retraining. However, those defenses are focused on the inference in plaintext, and cannot be applied to PPML protocols for secure inference.

X. Conclusion

We propose SIGuard, a secure protocol to defend MIAs against secure inference. SIGuard secretly generates noise vectors and adds them to the model's outputs in the encrypted domain without compromising the accuracy of secure inference. We comprehensively evaluate SIGuard across five real-world datasets and various MIAs, selecting optimal hyper-parameters for efficiency and defense effectiveness. Results show SIGuard is practical, cost-effective, and successfully safeguards secure inference.

Acknowledgments

The authors would like to thank all the anonymous reviewers and the shepherd for their invaluable comments. This work is supported in part by Australian Research Council (ARC) Discovery Projects (DP190102835 and DP220102803) and Linkage Project (LP220200649).

References

[1] "Cloud vision api, features list," https://cloud.google.com/vision/docs/features-list, accessed: 2024-09-06.
[2] P. Mohassel and P. Rindal, "Aby^3: A mixed protocol framework for machine learning," in *CCS*, 2018.
[3] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A cryptographic inference service for neural networks," in *USENIX Security*, 2020.
[4] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin, "Falcon: Honest-majority maliciously secure framework for private deep learning," *PoPETs*, 2021.
[5] C. Dong, J. Weng, J. Liu, Y. Zhang, Y. Tong, A. Yang, Y. Cheng, and S. Hu, "Fusion: Efficient and secure inference resilient to malicious servers," in *NDSS*, 2022.
[6] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *S&P*, 2017.
[7] H. Yu, K. Yang, T. Zhang, Y.-Y. Tsai, T.-Y. Ho, and Y. Jin, "Cloudleak: Large-scale deep learning models stealing through adversarial examples," in *NDSS*, 2020.
[8] S. Mehnaz, S. V. Dibbo, E. Kabir, N. Li, and E. Bertino, "Are your sensitive attributes private? novel model inversion attribute inference attacks on classification models," in *USENIX Security*, 2022.
[9] M. Nasr, R. Shokri, and A. Houmansadr, "Machine learning with membership privacy using adversarial regularization," in *CCS*, 2018.
[10] A. Salem, Y. Zhang, M. Humbert, P. Berrang, M. Fritz, and M. Backes, "Ml-leaks: Model and data independent membership inference attacks and defenses on machine learning models," in *NDSS*, 2019.
[11] J. Ye, A. Maddi, S. K. Murakonda, V. Bindschaedler, and R. Shokri, "Enhanced membership inference attacks against machine learning models," in *CCS*, 2022.
[12] N. Carlini, S. Chien, M. Nasr, S. Song, A. Terzis, and F. Tramer, "Membership inference attacks from first principles," in *S&P*, 2022.

[13] O. Goldreich, *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.

[14] Y. Lindell and B. Pinkas, "Privacy preserving data mining," in *CRYPTO*, 2000.

[15] W. Ruan, M. Xu, W. Fang, L. Wang, L. Wang, and W. Han, "Private, efficient, and accurate: Protecting models trained by multi-party learning with differential privacy," in *S&P*, 2023.

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

[17] M. Abadi, A. Chu, I. Goodfellow, B. McMahan, I. Mironov, K. Talwar, and L. Zhang, "Deep learning with differential privacy," in *CCS*, 2016.

[18] L. Song and P. Mittal, "Systematic evaluation of privacy risks of machine learning models," in *USENIX Security*, 2021.

[19] X. Tang, S. Mahloujifar, L. Song, V. Shejwalkar, M. Nasr, A. Houmansadr, and P. Mitta, "Mitigating membership inference attacks by Self-Distillation through a novel ensemble architecture," in *USENIX Security*, 2022.

[20] Z. Chen and K. Pattabiraman, "Overconfidence is a dangerous thing: Mitigating membership inference attacks by enforcing less confident prediction," in *NDSS*, 2024.

[21] J. Jia, A. Salem, M. Backes, Y. Zhang, and N. Z. Gong, "Memguard: Defending against black-box membership inference attacks via adversarial examples," in *CCS*, 2019.

[22] P. Mohassel and Y. Zhang, "Secureml: A system for scalable privacy-preserving machine learning," in *S&P*, 2017.

[23] B. Knott, S. Venkataraman, A. Hannun, S. Sengupta, M. Ibrahim, and L. van der Maaten, "Crypten: Secure multi-party computation meets machine learning," in *NeurIPS*, 2021.

[24] J.-L. Watson, S. Wagh, and R. A. Popa, "Piranha: A GPU platform for secure computation," in *USENIX Security*, 2022.

[25] A. Aly and N. P. Smart, "Benchmarking privacy preserving scientific operations," in *ACNS*, 2019.

[26] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara, "High-throughput semi-honest secure three-party computation with an honest majority," in *CCS*, 2016.

[27] M. Keller and K. Sun, "Secure quantized training for deep learning," in *ICML*, 2022.

[28] "Project innereye – democratizing medical imaging ai," https://www.microsoft.com/en-us/research/project/medical-image-analysis/, accessed: 2024-08-10.

[29] Y. Lindell, "How to simulate it–a tutorial on the simulation proof technique," *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, 2017.

[30] Catrina, Octavian, and A. Saxena, "Secure computation with fixed-point numbers," in *FC*, 2010.

[31] W. jie Lu, Y. Fang, Z. Huang, C. Hong, C. Chen, H. Qu, Y. Zhou, and K. Ren, "Faster secure multiparty computation of adaptive gradient descent," in *PPMLP*, 2020.

[32] M. Nasr, R. Shokri, and A. Houmansadr, "Comprehensive privacy analysis of deep learning," in *S&P*, 2018.

[33] A. Sablayrolles, M. Douze, Y. Ollivier, C. Schmid, and H. Jégou, "White-box vs black-box: Bayes optimal strategies for membership inference," in *ICML*, 2019.

[34] S. Tan, B. Knott, Y. Tian, and D. J. Wu, "Cryptgpu: Fast privacy-preserving machine learning on the gpu," in *S&P*, 2021.

[35] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[36] "Colorectal histology mnist," https://www.kaggle.com/datasets/kmader/colorectal-histology-mnist, accessed: 2024-07-08.

[37] N. Agrawal, A. S. Shamsabadi, M. J. Kusner, and A. Gascón, "Quotient: Two-party secure neural network training and prediction," in *CCS*, 2019.

[38] O. Catrina and S. de Hoogh, "Improved primitives for secure multiparty integer computation," in *Security and Cryptography for Networks*, 2010.

[39] E. Kiltz, "Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation," *TCC*, 2006.

[40] G. Asharov, K. Hamada, D. Ikarashi, R. Kikuchi, A. Nof, B. Pinkas, K. Takahashi, and J. Tomida, "Efficient secure three-party sorting with applications to data analysis and heavy hitters," in *CCS*, 2022.

[41] J. D. Nielsen and M. I. Schwartzbach, "A domain-specific programming language for secure multiparty computation," in *PLAS*, 2007.

[42] M. Keller, "Mp-spdz: A versatile framework for multi-party computation," in *CCS*, 2020.

[43] S. Ma, Y. Liu, G. Tao, W.-C. Lee, and X. Zhang, "Nic: Detecting adversarial samples with neural network invariant checking," in *NDSS*, 2019.

[44] "Interpret and improve model accuracy and analysis confidence scores," https://learn.microsoft.com/en-us/azure/ai-services/document-intelligence/concept-accuracy-confidence?view=doc-intel-4.0.0, accessed: 2024-04-10.

[45] B. Jayaraman and D. Evans, "Are attribute inference attacks just imputation?" in *CCS*, 2022.

[46] M. Jagielski, N. Carlini, D. Berthelot, A. Kurakin, and N. Papernot, "High accuracy and high fidelity extraction of neural networks," in *USENIX Security*, 2020.

[47] B. Balle, G. Cherubin, and J. Hayes, "Reconstructing training data with informed adversaries," in *S&P*, 2022.

[48] K. Ganju, Q. Wang, W. Yang, C. A. Gunter, and N. Borisov, "Property inference attacks on fully connected neural networks using permutation invariant representations," in *CCS*, 2018.

[49] C. A. Choquette-Choo, F. Tramer, N. Carlini, and N. Papernot, "Label-only membership inference attacks," in *ICML*, 2021.

[50] 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*, 2016.

[51] S. Wagh, D. Gupta, and N. Chandran, "Securenn: 3-party secure computation for neural network training," *PoPETs*, 2018.

[52] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow: Secure tensorflow inference," in *S&P*, 2020.

[53] X. Liu, Y. Zheng, X. Yuan, and X. Yi, "Medisc: Towards secure and lightweight deep learning as a medical diagnostic service," in *ESORICS*, 2021.

[54] ——, "Securely outsourcing neural network inference to the cloud with lightweight techniques," *TDSC*, 2022.

[55] Z. Xu, S. Lai, X. Liu, A. Abuadbba, X. Yuan, and X. Yi, "Oblivgnn: Oblivious inference on transductive and inductive graph neural network," in *USENIX Security*, 2024.

[56] K. Gupta, N. Jawalkar, A. Mukherjee, N. Chandran, D. Gupta, A. Panwar, and R. Sharma, "Sigma: Secure gpt inference with function secret sharing," *PoPETs*, 2024.

[57] Q. Pang, J. Zhu, H. Möllering, W. Zheng, and T. Schneider, "Bolt: Privacy-preserving, accurate and efficient inference for transformers," in *S&P*, 2024.

[58] H. Chaudhari, R. Rachuri, and A. Suresh, "Trident: Efficient 4pc framework for privacy preserving machine learning," in *NDSS*, 2019.

[59] N. Koti, M. Pancholi, A. Patra, and A. Suresh, "SWIFT: Super-fast and robust Privacy-Preserving machine learning," in *USENIX Security*, 2021.

[60] N. Koti, A. Patra, R. Rachuri, and A. Suresh, "Tetrad: Actively secure 4pc for secure training and inference," in *NDSS*, 2021.

[61] B. Wu, X. Yang, S. Pan, and X. Yuan, "Adapting membership inference attacks to gnn for graph classification: Approaches and implications," in *ICDM*, 2021.

[62] B. Wu, H. Zhang, X. Yang, S. Wang, M. Xue, S. Pan, and X. Yuan, "Graphguard: Detecting and counteracting training data misuse in graph neural networks," in *NDSS*, 2024.

[63] D. Chen, N. Yu, Y. Zhang, and M. Fritz, "Gan-leaks: A taxonomy of membership inference attacks against generative models," in *CCS*, 2020.

[64] M. Zhang, Z. Ren, Z. Wang, P. Ren, Z. Chen, P. Hu, and Y. Zhang, "Membership inference attacks against recommender systems," in *CCS*, 2021.

Appendix A
An illustration of MemGuard

The input of Algorithm 1 includes: $\vec{z}$, h, max_iter (maximum iterations), c_1, c_2, c_3 (c_1, c_2 remains as constants), and β (optimization learning rate). Then, it outputs a vector $\vec{e'}$, which is utilized to store the feasible solution of $\vec{e}$ (Step 17, Algorithm 1) through optimization. In Steps 9-11, Algorithm 1 iteratively optimizes the noise vector $\vec{e}$ along the direction of minimizing the loss L. In Steps 8 & 14, Algorithm 1 checks whether the noise vector $\vec{e}$ is validated based on two rules: 1) the noise vector should change the h's prediction (the switch of the h's prediction implies the prediction should be close

TABLE X: Attack performance in PPML w/o SIGuard under different datasets and softmax approximations.

Dataset	Group	SIGuard with SecureML					SIGuard with CrypTen					SIGuard with Piranha					SIGuard with AS19 (**final selection**)				
		NN-M	conf	entr	Mentr	LiRA	NN-M	conf	entr	Mentr	LiRA	NN-M	conf	entr	Mentr	LiRA	NN-M	conf	entr	Mentr	LiRA
CIFAR-10	PPML	51.68	56.95	50.11	57.56	86.56	57.26	58.56	56.71	58.68	61.58	55.75	59.11	55.80	59.08	26.88	57.51	58.81	57.15	59.10	60.76
	SIGuard	51.36	57.10	50.08	57.46	86.41	54.61	58.10	53.63	58.26	46.66	55.70	59.06	55.78	59.03	26.85	51.03	58.31	52.83	58.06	42.06
	Δ	$\downarrow$ **0.32**	$\uparrow$ **0.15**	$\downarrow$ **0.03**	$\downarrow$ **0.10**	$\downarrow$ **0.15**	$\downarrow$ **2.65**	$\downarrow$ **0.46**	$\downarrow$ **3.08**	$\downarrow$ **0.42**	$\downarrow$ **14.92**	$\downarrow$ **0.05**	$\downarrow$ **0.05**	$\downarrow$ **0.02**	$\downarrow$ **0.05**	$\downarrow$ **0.03**	$\downarrow$ **6.48**	$\downarrow$ **0.50**	$\downarrow$ **4.32**	$\downarrow$ **1.04**	$\downarrow$ **18.70**
CIFAR-100	PPML	53.00	65.88	52.15	66.20	90.90	71.43	77.20	73.65	77.36	82.58	68.01	75.68	67.81	75.73	60.40	69.78	74.30	66.40	74.31	82.30
	SIGuard	53.00	65.88	52.15	66.20	90.88	51.85	61.50	52.63	60.08	64.91	59.00	73.81	60.91	73.85	58.46	51.88	69.78	54.88	69.01	64.85
	Δ	$\downarrow$ **0.00**	$\downarrow$ **0.00**	$\downarrow$ **0.00**	$\downarrow$ **0.00**	$\downarrow$ **0.02**	$\downarrow$ **19.58**	$\downarrow$ **15.70**	$\downarrow$ **21.02**	$\downarrow$ **17.28**	$\downarrow$ **17.67**	$\downarrow$ **9.01**	$\downarrow$ **1.87**	$\downarrow$ **6.90**	$\downarrow$ **1.88**	$\downarrow$ **1.94**	$\downarrow$ **17.90**	$\downarrow$ **4.52**	$\downarrow$ **11.52**	$\downarrow$ **5.30**	$\downarrow$ **17.45**
CH-MNIST	PPML	74.70	69.25	58.20	71.90	64.10	77.75	77.60	76.70	76.80	69.05	65.00	67.75	66.35	67.75	35.35	76.95	72.10	70.50	71.50	69.60
	SIGuard	56.15	60.00	48.85	60.65	55.40	48.35	53.30	52.35	53.05	56.50	55.45	64.10	57.10	64.05	34.85	50.90	54.60	48.90	55.65	61.65
	Δ	$\downarrow$ **18.55**	$\downarrow$ **9.25**	$\downarrow$ **9.35**	$\downarrow$ **11.25**	$\downarrow$ **8.70**	$\downarrow$ **29.40**	$\downarrow$ **24.30**	$\downarrow$ **24.35**	$\downarrow$ **23.75**	$\downarrow$ **12.55**	$\downarrow$ **9.55**	$\downarrow$ **3.65**	$\downarrow$ **9.25**	$\downarrow$ **3.70**	$\downarrow$ **0.50**	$\downarrow$ **26.05**	$\downarrow$ **17.50**	$\downarrow$ **21.60**	$\downarrow$ **15.85**	$\downarrow$ **7.95**
Location30	PPML	88.66	86.53	67.26	87.06	72.80	99.06	92.53	92.73	92.13	72.06	81.93	86.40	82.00	86.40	54.13	98.93	92.40	92.40	92.86	71.73
	SIGuard	69.86	66.13	39.66	67.26	71.66	50.26	50.00	50.00	50.00	70.33	72.00	82.33	73.00	82.33	52.73	51.86	50.00	50.00	50.00	70.66
	Δ	$\downarrow$ **18.80**	$\downarrow$ **20.40**	$\downarrow$ **27.60**	$\downarrow$ **19.80**	$\downarrow$ **1.14**	$\downarrow$ **48.80**	$\downarrow$ **42.53**	$\downarrow$ **42.73**	$\downarrow$ **42.13**	$\downarrow$ **1.73**	$\downarrow$ **9.93**	$\downarrow$ **4.07**	$\downarrow$ **9.00**	$\downarrow$ **4.07**	$\downarrow$ **1.40**	$\downarrow$ **47.07**	$\downarrow$ **42.40**	$\downarrow$ **42.40**	$\downarrow$ **42.86**	$\downarrow$ **1.07**
Texas100	PPML	54.58	75.58	58.63	75.80	73.06	77.23	76.50	73.61	76.73	74.38	77.00	81.23	76.80	81.25	53.88	78.36	76.61	72.88	76.56	73.28
	SIGuard	53.98	70.78	57.78	71.03	65.76	47.75	60.16	54.38	59.45	73.53	63.10	76.30	65.88	76.28	50.70	50.78	58.46	52.95	58.36	71.31
	Δ	$\downarrow$ **0.60**	$\downarrow$ **4.80**	$\downarrow$ **0.85**	$\downarrow$ **4.77**	$\downarrow$ **7.30**	$\downarrow$ **29.48**	$\downarrow$ **16.34**	$\downarrow$ **19.23**	$\downarrow$ **17.28**	$\downarrow$ **0.85**	$\downarrow$ **13.90**	$\downarrow$ **4.93**	$\downarrow$ **10.92**	$\downarrow$ **4.97**	$\downarrow$ **3.18**	$\downarrow$ **27.58**	$\downarrow$ **18.15**	$\downarrow$ **19.93**	$\downarrow$ **18.20**	$\downarrow$ **1.97**

Protocol 4: Secure Softmax Differentiation Π_{DSoftmax}

Input: Parties hold confidence vector $\langle \vec{s} \rangle \in \mathbb{Z}_{2^\ell}^{n\times 1}$.
Output: Parties hold Jacobian matrix $\langle \boldsymbol{J} \rangle \in \mathbb{Z}_{2^\ell}^{n\times n}$, where each entry of $\boldsymbol{J}[i][j] = \frac{\partial \vec{s}[i]}{\partial \vec{z}[j]}$.

```
1: $\langle \boldsymbol{J} \rangle \leftarrow \Pi_{\text{Shr}}(\langle \mathbf{0} \rangle)$, where $\boldsymbol{J} \in \mathbb{R}^{n\times n}$.
2: for $i \in \{0, \cdots, n-1\}$ do
3:     for $j \in \{0, \cdots, n-1\}$ do
4:         $\langle \boldsymbol{J}[i][j] \rangle \leftarrow \langle s\vec{[i]} \rangle \cdot (\delta_{ij} - \langle s\vec{[j]} \rangle)$.
5:     end for
6: end for
7: return $\langle \boldsymbol{J} \rangle$.
```

to 50%). 2) the noise vector should not change the target model's final predicted label. In Step 18, once a validated noise vector $\vec{e}$ is found, Algorithm 1 proportionally increases c_3 and subsequently starts a new round of optimization. Finally, if the current optimization does not find a more suitable noise vector with the new increased c_3, the algorithm returns at Step 14.

APPENDIX B
BUILDING BLOCKS

Protocol 4. Π_{DSoftmax} secretly computes the derivative of softmax function. At the beginning, parties confidence vector $\langle \vec{s} \rangle \in \mathbb{Z}_{2^\ell}^{n\times 1}$. In Step 3, parties input the share of the Jacobian matrix with all zeros. In Step 6, for each element of $\langle \boldsymbol{J}[i][j] \rangle$, parties update its value with $\langle s\vec{[i]} \rangle \cdot (\delta_{ij} - \langle s\vec{[j]} \rangle)$, where δ_{ij} denotes as the Kronecker delta function. In Step 9, parties output $\langle \boldsymbol{J} \rangle$.

Protocol 5. Π_{Forward} secretly computes the forward pass of h, an L-layer fully-connected neural network. First, parties hold the linear layer's parameters of $\{\langle \boldsymbol{W}_i \rangle\}_0^{L-1}$ and $\langle \vec{x} \rangle$. In Step 3, parties set $\langle \vec{x} \rangle$ as $\langle \vec{a_0} \rangle$, where $\vec{a_i}$ will represent the intermediate output from the ReLU function of the i-th linear layer. In Step 5, for each linear layer, parties secretly compute the multiplication of the model parameters with $\langle \vec{a_i} \rangle$. In Step 6, parties invoke Π_{ReLU} on $\langle \vec{a}_i \rangle$ and get $\langle \vec{a}_i \rangle, \langle \vec{g}_i \rangle$. In Step 8, parties secretly compute $\langle \vec{a}_{L-1} \rangle$, which is the last linear layer's output. In Step 9, parties output $\{\langle \vec{g}_i \rangle\}_0^{L-2}$ and $\langle \vec{a}_{L-1} \rangle$.

Protocol 5: Secure Forward Pass [37] Π_{Forward}

Input: Parties hold neural network h with its weights $\{\langle \boldsymbol{W}_i \rangle\}_0^{L-1}$ and $\langle \vec{x} \rangle$.
Output: Parties hold intermediate gradients of the ReLU function $\{\langle \vec{g}_i \rangle\}_0^{L-2}$ and $\langle y \rangle$, where $y = h(\vec{x})$.

```
1: $\langle \vec{a}_0 \rangle \leftarrow \langle \vec{x} \rangle$.
2: for $i \in \{0, \cdots, L-2\}$ do
3:     $\langle \vec{a}_i \rangle \leftarrow \langle \boldsymbol{W}_i \rangle \cdot \langle \vec{a}_i \rangle$.
4:     $\langle \vec{a}_i \rangle, \langle \vec{g}_i \rangle \leftarrow \Pi_{\text{ReLU}}(\langle \vec{a}_i \rangle)$.
5: end for
6: $\langle \vec{a}_{L-1} \rangle \leftarrow \langle \boldsymbol{W}_{L-1} \rangle \cdot \langle \vec{a}_{L-1} \rangle$.
7: return $\{\langle \vec{g}_0 \rangle, \cdots, \langle \vec{g}_{L-2} \rangle\}$, $\langle y \rangle \leftarrow \langle \vec{a}_{L-1} \rangle$.
```

Protocol 6: Secure Backward Pass [37] Π_{Backward}

Input: Parties hold neural network h with its weights $\{\langle \boldsymbol{W}_i \rangle\}_0^{L-1}$, the intermediate gradients $\{\langle \vec{g'}_i \rangle\}_0^{L-2}$ from the each ReLU function, and $\langle \vec{y} \rangle$, where $y = h(\vec{x})$.
Output: Parties hold share of $\langle \vec{g} \rangle$, where $\vec{g} = \frac{\partial h(\vec{x})}{\partial \vec{x}}$.

```
1: $\langle \vec{g} \rangle \leftarrow \langle \boldsymbol{W}_{L-1} \rangle$.
2: for $i \in \{L-2, \cdots, 0\}$ do
3:     $\langle \vec{g} \rangle \leftarrow \langle \vec{g'}_i \rangle \circ \langle \vec{g} \rangle$.
4:     $\langle \vec{g} \rangle \leftarrow \langle \boldsymbol{W}_i \rangle \cdot \langle \vec{g} \rangle$.
5: end for
6: return $\langle \vec{g} \rangle$.
```

Protocol 6. Π_{Backward} secretly computes the backward pass of h, an L-layer fully-connected neural network. At the beginning, the linear layer's parameters of $\{\langle \boldsymbol{W}_i \rangle\}_0^{L-1}$ and the ReLU layers' derivatives $\{\langle \vec{g'}_i \rangle\}_0^{L-2}$ that behind each linear layer. As the last layer does not have a linear layer, the number of $\vec{g'}_i$ is $L-1$. In Step 3, parties set the derivative $\langle \vec{g} \rangle$ as the last layer's weights $\langle \boldsymbol{W}_{L-1} \rangle$. In Step 5, for each layer except the last, parties compute $\langle \vec{g'}_i \rangle \circ \langle \vec{g} \rangle$, where $\circ$ represents the element-wise multiplication. In Step 6, parties secret compute $\langle \boldsymbol{W}_i \rangle \cdot \langle \vec{g} \rangle$. In Step 8, parties output $\langle \vec{g} \rangle$, such that $\vec{g} = \frac{\partial h(\vec{x})}{\partial \vec{x}}$.