SONNI: Secure Oblivious Neural Network Inference

Luke Sperling and Sandeep S. Kulkarni

Department of Computer Science and Engineering
Michigan State University
{*sperli14, sandeep*}*@msu.edu*

Abstract: In the standard privacy-preserving Machine learning as-a-service (MLaaS) model, the client encrypts data using homomorphic encryption and uploads it to a server for computation. The result is then sent back to the client for decryption. It has become more and more common for the computation to be outsourced to third-party servers. In this paper we identify a weakness in this protocol that enables a completely undetectable novel model-stealing attack that we call the Silver Platter attack. This attack works even under multikey encryption that prevents a simple collusion attack to steal model parameters. We also propose a mitigation that protects privacy even in the presence of a malicious server and malicious client or model provider (majority dishonest). When compared to a state-of-the-art but small encrypted model with 32k parameters, we preserve privacy with a failure chance of 1.51×10^{-28} while batching capability is reduced by 0.2%.

Our approach uses a novel results-checking protocol that ensures the computation was performed correctly without violating honest clients' data privacy. Even with collusion between the client and the server, they are unable to steal model parameters. Additionally, the model provider cannot learn any client data if maliciously working with the server.

1 Introduction

Training ChatGPT3(Brown et al., 2020) took 3.14×10^{23} floating point operations or an estimated \$4.6M if replicated using a Tesla V100 cloud instance[1]. QuillBot uses a variety of models for grammar checking, paraphrasing, etc, some of which contain billions of parameters and a took team of engineers to develop[2].

One way for a provider to recuperate the cost of generating machine learning models is to provide machine learning-as-a-service (MLaaS) where the provider permits a client to compare their data with the model. For example, a hospital could pay to have automated screenings for skin cancer. Moreover, millions of users pay for their queries to be answered by ChatGPT or to have their writings improved by QuillBot's premium offerings. This incentivizes the costly training process with financial gain.

The use of MLaaS generates an issue of privacy for the client, as the data it is using is private and the client is concerned about possible leak of this private data to provider. The issue of privacy is especially important to client as the data is often one-of-a-kind and cannot be replaced/regenerated, such as in the case of medical or biometric data. For this reason, for several years, MLaaS systems have been designed with client data privacy in mind.

However, as MLaaS becomes more widespread and the models become more complex, the provider lacks the computational power needed to provide MLaaS on its own. Especially in the context of availability of third-party servers to provide computational capability, it is in the interest of the provider to outsource the MLaaS service to a server that will only provide the computational capability. Even in this context, the client privacy is crucial. Additionally, in this context, the privacy of the provider is critical as well. Specifically, since the development of the model requires significant computational and financial investment, it is critical that the model remains a secret and not revealed to anyone including the client and the server.

Our work focuses on using oblivious neural networks with multikey encryption (Chen et al., 2019b). Here, both the client and model provider each hold a secret key and both are needed for decryption. The

---

[1] https://lambdalabs.com/blog/demystifying-gpt-3
[2] https://quillbot.com/blog/compressing-large-language-generation-models-with-sequence-level-knowledge-distillation/

result is computed by the third-party server and sent back to the client and model provider. The model provider applies their secret key to get a partial decryption and sends it back to the client who is able to fully decrypt and learn the result.

The protocol in (Chen et al., 2019b) provides privacy only for the honest-but-curious and collusion models. In other words, if the server, model provider or client is only going to try to infer from the data it has learned, the privacy property is satisfied. However, if the server is malicious and violates the protocol even slightly, it can leak model parameters without being detected. One way to do this is to encrypt the model parameters as the return value and send them to the client instead of sending the actual result of the model. In this case, the server will not be able to detect this as the data it sees is encrypted with the client's key. It follows that this attack would go completely undetected by the model provider 100% of the time.

A simple solution would be to have the model provider check the result before returning it to the client but this is insufficient. The output of the model applied to the private data is, itself, private data as well. Additionally, it is possible in certain applications to recreate the original data from the output (Mai et al., 2018). Therefore, there is a strong need for a solution that ensures the result was computed correctly without revealing what the output is to the model provider.

In this paper, we identify a novel attack against oblivious neural network inference and propose a secure protocol that we call Secure Oblivious Neural Network Inference, or SONNI, to combat it and ensure security in the presence of a dishonest server and dishonest client/provider (majority dishonest).

The contributions of the paper are as follows:

- We introduce a novel attack, called the Silver Platter attack, to enable model stealing in the oblivious neural network inference setting.

- We propose a novel protocol for oblivious neural network inference that mitigates this novel attack and does not violate the client's data privacy.

- We evaluate our protocol in terms of the level of privacy it provides and the overhead of providing that privacy.

In Section 2 we provide background of the encryption schemes employed in this work. In Section 3 we detail the vulnerability of the existing outsourced neural network inference protocol and the novel attack we construct. We propose a solution and prove its security in Section 4. We provide related work in Section 5 and closing remarks in Section 6.

2 Background

2.1 Fully Homomorphic Encryption

Traditional encryption schemes require decryption before any data processing can occur. Homomorphic encryption (HE) allows arithmetic directly over ciphertexts without requiring access to the secret key needed for decryption, enabling data oblivious protocols. Fully homomorphic encryption (FHE) schemes support an unlimited number of multiple types of operations. The Cheon-Kim-Kim-Song (CKKS) (Cheon et al., 2017) cryptosystem supports additions as well as multiplications of ciphertexts. The message space of CKKS is vectors of complex numbers.

A small amount of error is injected into each ciphertext at the time of encryption to guarantee the security of the scheme. This error grows each arithmetic operation performed. Therefore, once a certain threshold of multiplications are performed the error grows too large to decrypt correctly. Although the bootstrap operation reduces this error to allow circuits of unlimited depth, for the purposes of this work no such operations are needed. The following operations are supported:

- Encode: Given a complex vector, return an encoded polynomial.

- Decode: Given a polynomial, return the encoded complex vector.

- Encrypt: Given a plaintext polynomial and public key, return a ciphertext.

- Decrypt: Given a ciphertext and the corresponding secret key, return the underlying polynomial.

- Ciphertext Addition: Given two ciphertexts, or a ciphertext and a plaintext, return a ciphertext containing an approximate element-wise addition of the underlying vectors.

- Ciphertext Multiplication: Given two ciphertexts, or a ciphertext and a plaintext, return a ciphertext containing an approximate element-wise multiplication of the underlying vectors.

- Relinearization: Performed after every multiplication to prevent exponential growth of ciphertext size. This function is assumed to be executed after every homomorphic multiplication in this work for simplicity.

- Ciphertext Rotation: Given a ciphertext, an integer, and an optionally-generated set of rotation keys, rotate the underlying vector a specified number of times. For instance, $Rotate(Enc([1,2,3,4]),1,k_r)$ returns $Enc([2,3,4,1])$.
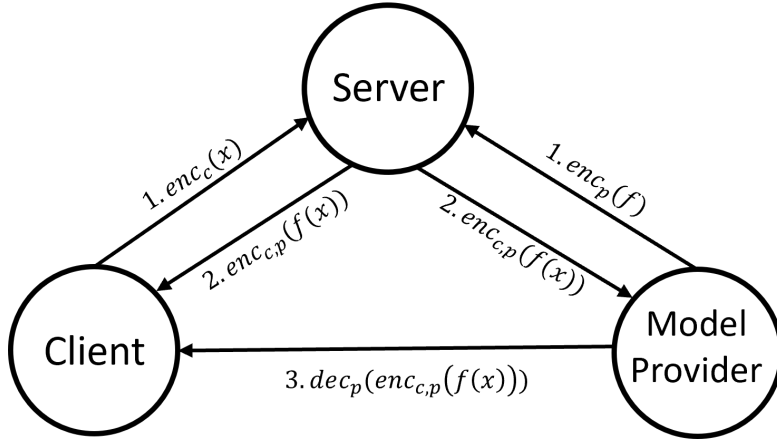
Figure 1: Standard Outsourced Oblivious Neural Network Inference Model

2.2 Multikey Encryption

Chen et al. (Chen et al., 2019b) extended existing FHE schemes to operate on multiple sets of secret key, public key pairs. Consider the case where there are two key pairs owned by two parties (such as the Client and Provider). Each party encrypts a value using their respective public keys. These ciphertexts, denoted by $enc_c(X)$ and $enc_p(Y)$ can be multiplied together homomorphically, resulting in $enc_{c,p}(XY)$. Note that this notation means the ciphertext is now multikey encrypted by the keys of both the Client and the Provider and that $enc_{k_1,k_2}(V) = enc_{k_2,k_1}(V)$. The following additional operations are supported:

- *PartialDec(ciphertext, secret_key)* - given a multikey encrypted ciphertext and one of the secret keys corresponding to a public key encrypting the ciphertext, return a partial decryption of the ciphertext. This partial decryption does not yield any additional information.

- *Combine(partial_decryptions)* - given an array of partial decryptions, return the underlying plaintext value. This combine function only works if there is a partial decryption corresponding to each public key used in the multikey encryption.

3 Proposed Model Stealing Attack

In this section, we demonstrate that the existing approach for using multiparty encryption to provide privacy in outsourced MLaaS systems suffers from a serious violation of privacy for the provider if the server behaves maliciously and tries to leak parameters to the client. We call this attack the Silver Platter attack, as the adversary is delivered the exact model parameters with little effort. Additionally, this attack will go unnoticed by the provider.

The outsourced oblivious neural network inference system based on multi-key FHE (Mukherjee and Wichs, 2016) consists of three parties, the client, the model provider and the server (to which the computation is outsourced).

The client holds some private data x for use as input to a parameterized function f. The model provider, referred to henceforth as the provider, holds a private function f that is comprised of private parameters. The server holds no private data and is instead responsible for computing $f(x)$ given encryptions of f and x.

In this model, there are two privacy requirements: the client wants to ensure that the values of x and $f(x)$ are not released to anyone while the provider wants to ensure that none learns the (parameters of) function f. However, existing client server models typically focus on the privacy requirement of the client and ignore the privacy requirements of the provider or server (Boulemtafes et al., 2020; Yang et al., 2023).

These threat models are insufficient, considering how rampant model stealing attacks have become (Oliynyk et al., 2023). Hence, we assume the byzantine threat model where any of the parties may be byzantine. If the server is byzantine, it may be cooperating with either the client (to steal the model f) or with the provider (to steal x or $f(x)$).

In cases where the server is colluding with the client, it will try to steal some/all model parameters in place of the value for $f(x)$. In cases where the server is cooperating with the provider, it is in the best interest of both of them to return $f(x)$ to the client as failure to do so would raise suspicion on the server and/or provider and it will cause reputational/financial damage to one or both of them. However, the server and provider may exchange additional messages that al-

low them to learn $f(x)$ or x

The communication between client, provider and server is as shown in Figure 1 where we utilize the oblivious neural network inference protocol from (Chen et al., 2019b). The client encrypts their data using fully homomorphic encryption (FHE) and the provider encrypts their function parameters using a different public key. These encryptions (along with the corresponding public keys) are sent to the server. The server then computes $enc_{c,p}(f(x))$ which is returned to both the client and the provider. Partial decryptions are obtained with each party's secret key and are combined by the client to obtain the full decryption of $f(x)$.

The approach in (Chen et al., 2019b) preserves the privacy of the client as the provider and server only sees the encrypted version of the data. It also preserves the privacy of provider if the server is honest-but-curious, as the server only sees the data in encrypted format. However, if server is dishonest, it can collude with client to reveal the model parameters. We discuss such attacks next.

In step 2 of Figure 1, a malicious server sends an encryption of the parameters $enc_{c,p}(f)$ instead of the intended result $enc_{c,p}(f(x))$. $enc_{c,p}(f)$ can be trivially computed from $enc_p(f)$ which is obtained by the server during step 1. This ciphertext is fully decrypted only by the client during step 3, revealing private model parameters. This attacks appears identical to normal operation to the provider. In cases where there are multiple ciphertexts with model parameters, the attack may be repeated to obtain them all.

The reason we call this the Silver Platter attack is that this can be repeated ad infinitum without the provider learning anything about the fact that the model is being compromised. This paper focuses on addressing this attack.

4 Attack Mitigation via Oblivious Results Checking

In this section we propose a protocol that mitigates the Silver Platter attack and we prove its security. First, in Section 4.1 we introduce the problem formally. Next, in Section 4.2 we describe our approach at a high level before going into details in Section 4.3. We then describe the special steps needed to prepare the ciphertext at the beginning of the computation as well as the work needed to check that the result was computed correctly. Lastly, in Section 4.4 we analyze our protocol and prove its security.

4.1 Problem Statement

In this section, we define the problem statement for dealing with Silver Platter attack. Here, a party (client, server or provider) may be honest or dishonest. An honest party follows the prescribed protocol, while a dishonest party could be byzantine and behave arbitrarily. We assume that the dishonest party intends to remain hidden. Therefore, it will send messages of the required type (such as CKKS ciphertexts). However, the data inside may be incorrect.

Recall that in the multiparty communication, the client starts with its own data, x, and desires to obtain a value $f(x)$ where f is a function known to provider. The actual computation of $f(x)$ is done by a server.

In the multiparty communication considered in this paper, we follow a *secure with abort* approach where security is guaranteed at all times, i.e., the provider or server never learns the client data, x or the computed value $f(x)$ but if dishonest behavior is detected, the computation may be aborted. This remains true in all cases, i.e., even if the provider and server send additional messages between them privacy will not be violated. We also want to ensure that the model parameters are never revealed to the client even if the server and client collude and send additional messages between them. However, if everyone behaves honestly then the client should receive the value of $f(x)$. Thus, the privacy requirements are as follows:

Liveness Upon providing $enc_c(x)$ to provider, the client should eventually learn $f(x)$ if the provider and server behave honestly.

Provider privacy The client should not learn the model parameters even if the server is colluding with the client.

Client privacy The provider should not learn x or $f(x)$ even if the server is colluding with the provider.

Oblivious Inference The server should not learn input x, model parameters, or result $f(x)$.

Secure with Abort If any party violates the protocol then the computation is aborted and the client does not learn anything.

As stated, we follow the *secure with abort* model for handling collusion between the server and client, i.e., the provider will abort the computation if it detects that the server or client is trying to steal model parameters. In this case, the client does not receive $f(x)$. It also does not learn any model parameters.

In our protocol, the client does not need to abort as there is no possibility of the provider and server learning x or $f(x)$.
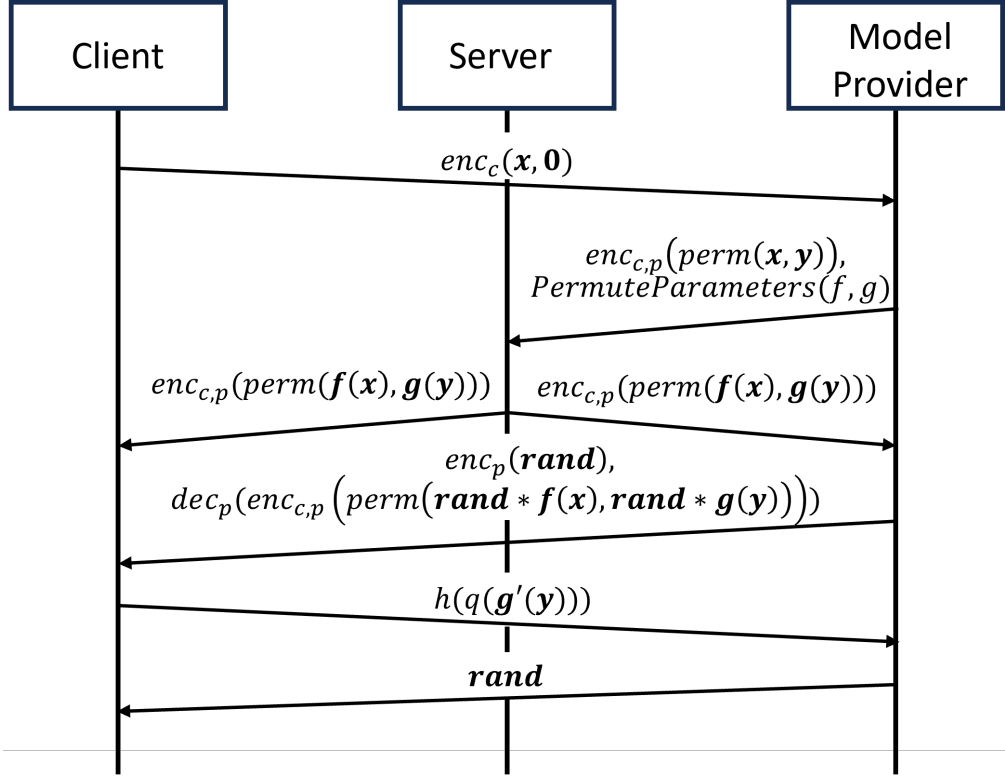
Figure 2: SONNI sequence diagram.

4.2 Mitigation Approach

In this section, we propose our approach for providing privacy to provider as well as the client. Specifically, the approach in (Chen et al., 2019b) already provides privacy for the client data. We ensure that the corresponding privacy is preserved while permitting the provider to have control over the data being transmitted to the client so that it can verify that the server is not trying to leak the provider's private data (namely the model parameters) to the client.

Our approach is based on the observation that machine learning algorithms operate on vectors as opposed to scalars, making FHE well-suited to the task. Ciphertexts in FHE encode and encrypt entire vectors of real values. We take advantage of this fact for our mitigation strategy. We dedicate some slots of the ciphertext (and therefore some indices of the underlying vector) to verify the computation is performed correctly. These slots will not contain $f(x)$ values but rather randomly-generated values for use in a results checking step.

The basic idea of the protocol is to have the server compute $f(x)$ in most slots of the ciphertext and $g(y)$ in other slots, where g is a function that is of the same

form as f (e.g., if f is a linear/quadratic function then g is also a linear/quadratic function with different parameters.). Here, g is a random function generated at the start of the protocol by the provider and y is a random secret input vector also generated by the provider. Upon receiving the answer from the server, the provider will get the decryption of $g(y)$ by requesting it from the client. And, if the value of $g(y)$ matches then the provider will return the result to the client. Since g and y are only known to the provider, if $g(y)$ is computed correctly then it provides a confidence to the provider that the server is likely to have performed honestly. If $g(y)$ is not computed correctly, then the provider would realize that either server or client is being dishonest and can refuse to provide the partial decryption of the encrypted value of $f(x)$ necessary to learn $f(x)$.

We observe that the use of the extra slots for $g(y)$ reduces the number of slots available to the client. Also, the approach for providing privacy is probabilistic, i.e., there is a possibility that a dishonest server could use some of the slots used by f to leak information without being caught. This probability will be low if the number of slots used for g are high. However, in this case, the overhead will be higher as

the number of slots available for f would be lower. We discuss this in Section 4.4 and show that the probability of a successful model stealing attack against our system is negligible.

4.3 Protocol Details

SONNI is built on top of the existing oblivious neural network inference protocol with added steps. The protocol is detailed in Algorithm 1.

First in step 1, the client generates a ciphertext containing the vector x in the first d slots of the ciphertext and sends it to the provider. The provider inserts the y values in slots $d+1..d+m$, where m slots are used for y. Subsequently, provider permutes this vector in step 3. It also permutes function f and g in a similar fashion. As an illustration if the f was $x[1]+2x[2]$ and $(x[1],x[2])$ is permuted to $(x[2],x[1])$ then f would be changed to $2x[2]+x[1]$.

Additionally, to *permute the computation*, the provider can also change f to be $2x[2]+x[1]+0*y[1]$, where $y[1]$ is one of the values used for the computation of $g(y)$. Likewise, the computation of g can use some of the values in the x vector with coefficient 0. It could use any arithmetic operation that uses y (respectively, x) while computing the value of $f(x)$ (respectively $g(y)$) in such a way that the final answer will be independent of the y (respectively x) value. This will prevent the server from performing dataflow techniques to determine the slots used for f and g.

Since the server is not aware of the placement of x and y in the ciphertext and it uses the permuted input, it computes permuted $f(x)$ and $g(y)$ in step 6. This value is sent to both the client and the provider.

In step 10, the client uses the mask provided by the provider to compute $enc_{c,p}(g'(y))$, where $g'(y)$ denotes $g(y)$ masked by a random vector known to the provider (similarly $f'(x)$ denotes $f(x)$ masked by that same random vector). This value and the partial decryption provided by provider is used to compute $g'(y)$. If $g'(y)$ is computed correctly, then the model owner is convinced that the entire vector was computed correctly.

Since the computation of $g'(y)$ by the client may have a small error due to the use of FHE whereas the value computed by the provider is the exact value of $g'(y)$, we use $g'_c(y)$ to denote the value computed by the client and $g'_p(y)$ to be value computed by the provider. Each value is quantized such that $q(g'_c(y))$ will be equal to $q(g'_p(y))$

Then, in steps 11-14, the client and provider perform a privacy-preserving comparison protocol to ensure $q(g'_c(y))$ was computed correctly. After this

passes, the client and provider jointly decrypt the result ciphertext containing $f(x)$ and only the client learns $f(x)$. One of the concerns in this step is a malicious provider that tries to compromise the privacy of the client. Specifically, in this step, if the provider specifies the incorrect indices such that the indices correspond to $f'(x)$ instead of $g'(y)$, then the provider could learn the value of $f(x)$. To prevent this, the proof that the client computes the value of $g'(y)$ correctly is achieved via a zero-knowledge approach discussed later in this section.

The above protocol has two important stages, ciphertext preparation (lines 2-5) and result checking (11-14). Next, we provide additional details of these steps. Ciphertext preparation refers to the manipulation done by the provider before it sends the data to server to compute $f(x)$ and $g(y)$ whereas the result checking refers the the computation between the client and provider so that it can verify that $g(y)$ was computed correctly by the server.

4.3.1 Ciphertext Preparation

The security of our proposed protocol relies on the client and server not knowing the indices of the y values. To achieve this, the client appends m zeroes to the end of their data vector x before encryption. This ciphertext is sent to the provider. Then, through a series of masking multiplications, rotations, and additions, the ciphertext is shuffled as shown in Algorithm 2. We note that we do not require arbitrary ciphertext shuffling for the security of our protocol. Our security rests on the server not being able to guess which slots correspond to x values and which slots correspond to y values. The vectors of function parameters are shuffled in the same manner prior to encryption. The provider finally adds random y values to the zero indices in the ciphertext.

4.3.2 Result Checking

If the provider does not verify that the ciphertext they are decrypting does not contain model parameters, the Silver Platter attack will succeed. However, the provider should not learn $f(x)$ so it is not permissible to have the provider decrypt the result, examine it, and return it to the client. Indeed, if this were allowed, the protocol would be open to an attack where the provider works with the server to learn x in a similar manner. Our idea is to give the client access only to the computed $g(y)$. If the client is able to correctly tell the provider the values of $g(y)$, then the provider will help the client decrypt the final result and gain access to $f(x)$.

The provider generates a random vector $rand \in$

Algorithm 1 Secure Oblivious Neural Network Inference

1: C sends to $\mathcal{P}$: $enc_c(x[1],x[2],...x[d],0,...0)$

2: $\mathcal{P}$ computes $enc_{c,p}(x[1],x[2],...x[d],y[1],y[2],y[m])$
3: $\mathcal{P}$ computes $enc_{c,p}(perm(x,y))$
4: $\mathcal{P}$ computes $PermuteParameters(f,g)$ using the same permutation.
5: $\mathcal{P}$ sends to $\mathcal{S}$: $enc_{c,p}(perm(x,y))$, $PermutaParameters(f,g)$

6: $\mathcal{S}$ computes and sends to $\mathcal{P}$ and C: $enc_{c,p}(perm(f(x),g(y)))$
7: $\mathcal{P}$ computes $rand = [-1,1]^{d+m}$
8: $\mathcal{P}$ computes $enc_{c,p}(perm(f'(x),g'(y))) = mult(enc_{c,p}(perm(f(x),g(y))),rand)$
9: $\mathcal{P}$ sends to C: $dec_p(enc_{c,p}(perm(f'(x),g'(y))))$, $enc_p(rand)$
10: C computes $g'(y)$

11: C sends to $\mathcal{P}$: $hash_1 = h(q(g'(y)))$
12: $\mathcal{P}$ computes $g'(y)$
13: $\mathcal{P}$ computes $hash_2 = h(q(g'(y)))$
14: If $hash_1 \neq hash_2$, $\mathcal{P}$ aborts

15: $\mathcal{P}$ sends to C: $rand$
16: C computes $f(x)$

Algorithm 2 Ciphertext shuffle

$indices \leftarrow \emptyset$
for $i = 0$ to m **do**
    $index \leftarrow random(0,d+m)$
    while $index \in indices$ **do**
        $index \leftarrow random(0,d+m)$
    end while
    $indices.add(index)$
end for
$indices.sort()$
$i \leftarrow 0$
for $index \in indices$ **do**
    $maskedvalue \leftarrow mult(ct,e_{index})$ $\triangleright e_i$ denotes a zero vector with a 1 at index i
    $maskedvalue \leftarrow rotate(maskedvalue,d+i-index)$
    $ct \leftarrow mult(ct,\mathbf{1}-e_{index})$
    $ct \leftarrow add(ct,maskedvalue)$
    $i \leftarrow i+1$
end for

$\mathbb{R}^{d+m}$ that contains in each index a random nonzero real value. This vector is sent to the client. Both the provider and the client compute:

$$enc_{c,p}(perm(f'(x),g'(y)))$$
$$= mult(enc_{c,p}(perm(f(x),g(y))),rand)$$

They also compute a partial decryption of this resultant ciphertext. The provider sends their partial decryption to the client that combines the partial decryptions to recover $g'(y) = g(y) \times rand$ (and additionally $f'(y) = f(y) \times rand$).

The provider computes $g'(y)$, as they know g, y, and $rand$ due to having generated each in plaintext at the start of the computation. Now, the provider needs to be convinced that they have the same value of $g'(y)$ as the client without revealing these values to each other. If the client were to send $g'(y)$ to the provider, the provider could take advantage of this by storing $f(x)$ or x in lieu of $g'(y)$ (or an obfuscated version thereof). Similarly, the provider cannot simply send $g'(y)$ to the client because the client could simply lie and state that the values are the same.

A secure two-party computation protocol is needed to prove to the provider that the values of $g'(y)$ are the same without revealing those values to the other. This protocol can work on plaintext values because $g'(y)$ exists in plaintext to both parties at this point in the protocol (as decryption has already occurred). Both parties compute $h(g'(y))$ for some low-collision hash function h. The client sends their result to the provider and the provider verifies that the hashes are identical.

Due to the approximate nature of arithmetic under FHE, the value of $g'(y)$ decrypted by the client is likely to be slightly different than the value computed in plaintext by the provider. To mitigate this, we use a simple uniform quantization scheme with intervals large enough that the noise from homomorphic computations will not affect the performance. Therefore, the client and provider compute $h(q(g'(y)))$ instead of $h(g'(y))$ for a quantization scheme q.

Once the provider is convinced that $g'(y)$ was computed correctly, they share the random vector $rand$ with the client to for use in recovering $f(x)$.

4.4 Security Analysis

In this subsection, we evaluate the theoretical probabilities of attacks successfully working against the proposed system. We refer to the probability of breaking the homomorphic encryption scheme as ε_1 and the probability of breaking the incorporated hash function as ε_2. These values are negligible, given proper selection of encryption security parameters and hash function.

Theorem 1. *Let d be the dimensionality of the client's secret data x and m be the number of ciphertext slots dedicated to containing y values. The probability of successfully performing the Silver Platter attack against SONNI to steal k scalar model parameters is upper bounded by $\left(\frac{d}{d+m}\right)^k + \varepsilon_1 + \varepsilon_2$.*

Proof. The probability of successfully performing the attack depends on the server passing the result checking step of the protocol. The provider performs the ciphertext shuffle locally and, due to ciphertext indistinguishability, the server is unable to differentiate any permutations. Therefore, the server must guess which slots of the ciphertext will not be checked during the result checking step. To hide a single scalar value in the result ciphertext without being found out results in one of the d slots dedicated to the results being selected instead of one of the m slots, or a probability of $\frac{d}{d+m}$. Stealing a second value in this ciphertext is therefore a probability of $\frac{d-1}{d+m-1}$ which is less than the original probability, so the optimal strategy is to independently steal one parameter per result ciphertext for a total probability of $\left(\frac{d}{d+m}\right)^k$ for k model parameters. The only other ways to successfully perform the attack are to break the hash function or the homomorphic encryption scheme, thus the theorem is proved. We note that no modifications may be made to the ciphertext by the client, lest the decryption fail (from the client and provider attempting to partially decrypt different ciphertexts). $\square$

Theorem 2. *The probability of the provider working with the server to successfully steal any amount of client data x or f(x) is upper bounded by $\varepsilon_1 + \varepsilon_2$.*

Proof. The provider and server together only learn encryptions and hashes during the protocol. If the provider is able to break the hash, a new attack is enabled. The server can manipulate the results ciphertext such that the client returns $h(q(f'(x)))$ in place of $h(q(g'(x)))$ to the provider who can then learn $q(f'(x))$ via breaking the hash. Therefore, being able to break either a hash or an encryption is the only way to learn new information about x or $f(x)$, proving the theorem. $\square$

Table 1: Probability of successful Silver Platter attack against SONNI. Vector size of 1024 is used.

m	Parameters Stolen	P(Success)
4	10	0.952
	128	0.513
	256	0.237
	512	0.0311
32	10	0.720
	128	0.011
	256	6.34×10^{-5}
	512	7.07×10^{-11}
512	10	9.25×10^{-4}
	128	2.88×10^{-43}
	256	1.02×10^{-96}
	512	1.23×10^{-307}

In Figure 3, we analyze Theorem 1 quantitatively. We assume in this analysis that the attack will be performed in one round, rather than stealing one parameter per transaction (as that becomes incredibly costly from a financial perspective when the number of parameters reaches thousands or millions). In FHE applications it is a common practice to use 128-1024 size vectors (Jindal et al., 2020). We consider the case where the ciphertext has 1024 slots, some of which (d) are allocated to the client and some (m) are allocated to the provider. When $m = 1$, i.e., only one slot is allocated to the provider for verification of $g(y)$, the probability of a successful Silver Platter is quite high. However, even with a few slots for the provider, the probability of a successful Silver Platter decreases sharply. Table 1 demonstrates this. With $m = 32$, i.e., 32 slots allocated to provider, the probability of stealing 10 parameters is 0.720, the probability of stealing 128 parameters is 0.011, and so on. The probability of stealing 512 parameters is 7.07×10^{-11}. We note that modern machine learning models often include thousands or millions of parameters. The simple projection matrix model featured in (Sperling et al., 2022) includes 32768 parameters. Even if the adversary maximizes their chance of successfully performing the attack by stealing a single model parameter at a time over 32768 transactions and if only 2 ciphertext slots dedicated to verification of $g(y)$, the probability of stealing all these parameters is 1.51×10^{-28}. Stealing just half of the parameters has a chance of 1.23×10^{-14} of success. Batching capability is reduced by only 0.2% while privacy is maintained with only a negligible chance of being violated.

In the case where even a small number of model parameters leaking is undesirable, using an m value of 512 ensures that Silver Platter attacks are extremely unlikely to be successful. Even stealing 10 in this circumstance has only a 9.25×10^{-4} chance of succeeding.
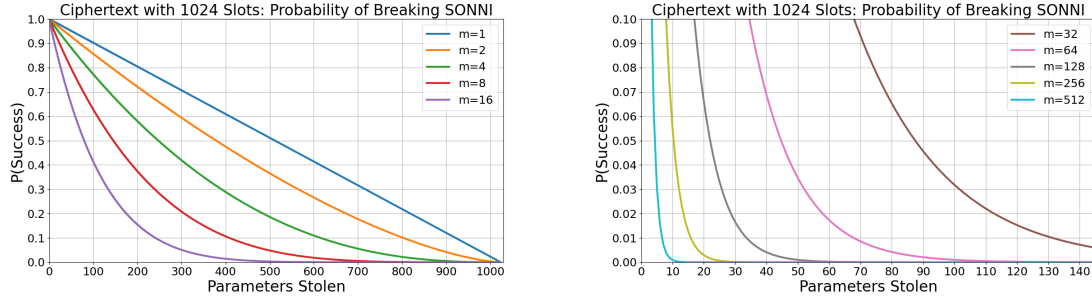
Figure 3: Probability of performing a Silver Platter attack without being detected based on the number of ciphertext slots dedicated to checked values.

We note that a failed attempt at model stealing in this circumstance is more dire than a failed attempt at breaking an encryption scheme, for instance. It is anticipated that the server and provider will have a financial relationship which would be compromised if the server is found to be dishonest. Also, the server is likely to be a company that provides a server farm. That means that risking this relationship for dishonest behavior that is very likely to be caught is not worthwhile. This may risk legal or financial consequences.

Also, our protocol preserves client privacy. This is especially important for a client as its data, x or $f(x)$, is often impossible to replace, as the data corresponds to medical information, biometric information, etc. By contrast, revealing a very small number of parameters does not significantly affect the provider. As discussed in Section 3, the Silver Platter is undetectable for the provider if one uses the existing approach (Chen et al., 2019b). With the level of security provided to both the client and provider, we can provide privacy guarantees to both of them with a reasonable overhead.

5   Related Work

Oblivious neural network inference aims to evaluate machine learning models without learning anything about the data and has seen much attention in recent years (Mann et al., 2023). XONN achieves this with a formulation similar to garbled circuits (Riazi et al., 2019). There is a privacy concern related to outsourcing these computations. Namely, that the model will be learned by the server. FHE has been used for inference that not only protects sensitive client data, but also prevents learning the provider's private data (i.e. the model) (Chen et al., 2019b). As we discuss thoroughly in this work, this approach is open to the undetectable Silver Platter Attack. Our work protects not only client data, as is expected in the classical obliv-

ious neural network inference formulation, but also that of the provider.

Model stealing attacks aim to learn models from black-box access. Broadly speaking, this either attempts to steal exact model values or general model behavior. Our work examines the former scenario. This falls into three categories: stealing training hyperparameters (Wang and Gong, 2018), stealing model parameters (Lowd and Meek, 2005; Reith et al., 2019), and stealing model architecture (Oh et al., 2019; Yan et al., 2020; Hu et al., 2019), often using side-channel attacks.

Gentry's seminal work constructing the first FHE scheme (Gentry, 2009) paved the way for more privacy-preserving applications on the cloud. Concerns over data privacy led to the development of multi-key encryption featuring multiple pairs of public and private keys. The first use of multi-key FHE was proposed by López et al. (López-Alt et al., 2012). Work followed suit, improving speed as well as supporting more modern schemes (Chen et al., 2019b; Chen et al., 2019a; Ananth et al., 2020). These schemes are developed for secure cloud computation. Specific optimizations have been proposed for federated learning (Ma et al., 2022).

Threshold HE schemes requiring a subset of key holders collaborating to decrypt ciphertexts have been devised as early as 2001 (Cramer et al., 2001; Damgård and Nielsen, 2003). Threshold FHE schemes have also seen popularity over the years both shortly after the advent of FHE (Myers et al., 2011) and in recent years using modern FHE schemes (Jain et al., 2017; Sugizaki et al., 2023; Chowdhury et al., 2022; Jain et al., 2017). As we demonstrate in this paper, these techniques alone are not sufficient to ensure data privacy in the outsourced server setting, as these methods do not verify that the proper value is being computed prior to decryption.

Collusion attacks are often not considered in the context of FHE systems, as they tend to use the client-server model and only account for the privacy of the

client, even in the biometrics setting where privacy is highly desired (Engelsma et al., 2022; Drozdowski et al., 2019; Sperling et al., 2022). Our work shows that assuming the honest-but-curious threat model for the participants in the MLaaS setting opens the way for undetectable and 100% successful attacks.

6 Conclusion

In this paper, we demonstrated that the current approach to oblivious neural network inference (Chen et al., 2019b) suffers from a simple attack, namely the Silver Platter attack. It permits the server to reveal all model parameters to the client in a way that is completely undetected by the provider.

We also presented a protocol to deal with the Silver Platter attack. The protocol relied on the observation that FHE based techniques utilize vector-based computation. We used a few slots in this vector to compute a function $g(y)$ that could be verified by the provider so that it can gain confidence that the server is not trying to steal the model parameters with the help of the client. Our protocol provided a tradeoff between the overhead (the number of slots used for verification) and the probability of a successful attack. We demonstrated that even with a small overhead in terms of the number of slots used for verification, the probability that a server can steal several parameters remains very small. Modern machine learning models often involve billions of parameters. We showed that the probability of stealing even 1% of the parameters is very small.

Future work directions include finding a modified shuffle method that allows the evaluation of functions that incorporate rotations to work on shuffled ciphertexts. Some matrix-vector multiplication methods require rotation, for instance, which is not supported by this work (Halevi and Shoup, 2014). Extensions of this work could also include devising a protocol that does not increase the multiplicative depth of the circuit, leading to a reduced overhead. Our protocol is secure against majority dishonest attacks, but it is not clear if it is the server or the client who is dishonest when the results checking step fails. Future work can tackle the problem of what to do logistically when the check fails. If the server is malicious it should indeed be blacklisted, but this should not open the way to a dishonest client attempting to get an honest server blacklisted. We leave that as an open problem.

REFERENCES

Ananth, P., Jain, A., Jin, Z., and Malavolta, G. (2020). Multi-key fully-homomorphic encryption in the plain model. In *Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16–19, 2020, Proceedings, Part I 18*, pages 28–57. Springer.

Boulemtafes, A., Derhab, A., and Challal, Y. (2020). A review of privacy-preserving techniques for deep learning. *Neurocomputing*, 384:21–45.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Chen, H., Chillotti, I., and Song, Y. (2019a). Multi-key homomorphic encryption from tfhe. In *Advances in Cryptology–ASIACRYPT 2019: 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8–12, 2019, Proceedings, Part II 25*, pages 446–472. Springer.

Chen, H., Dai, W., Kim, M., and Song, Y. (2019b). Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 395–412.

Cheon, J. H., Kim, A., Kim, M., and Song, Y. (2017). Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, pages 409–437. Springer.

Chowdhury, S., Sinha, S., Singh, A., Mishra, S., Chaudhary, C., Patranabis, S., Mukherjee, P., Chatterjee, A., and Mukhopadhyay, D. (2022). Efficient threshold fhe with application to real-time systems. *Cryptology ePrint Archive*.

Cramer, R., Damgård, I., and Nielsen, J. B. (2001). Multiparty computation from threshold homomorphic encryption. In *Advances in Cryptology—EUROCRYPT 2001: International Conference on the Theory and Application of Cryptographic Techniques Innsbruck, Austria, May 6–10, 2001 Proceedings 20*, pages 280–300. Springer.

Damgård, I. and Nielsen, J. B. (2003). Univer-

sally composable efficient multiparty computation from threshold homomorphic encryption. In *Annual international cryptology conference*, pages 247–264. Springer.

Drozdowski, P., Buchmann, N., Rathgeb, C., Margraf, M., and Busch, C. (2019). On the application of homomorphic encryption to face identification. In *2019 international conference of the biometrics special interest group (biosig)*, pages 1–5. IEEE.

Engelsma, J. J., Jain, A. K., and Boddeti, V. N. (2022). Hers: Homomorphically encrypted representation search. *IEEE Transactions on Biometrics, Behavior, and Identity Science*, 4(3):349–360.

Gentry, C. (2009). Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178.

Halevi, S. and Shoup, V. (2014). Algorithms in helib. In *Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I 34*, pages 554–571. Springer.

Hu, X., Liang, L., Deng, L., Li, S., Xie, X., Ji, Y., Ding, Y., Liu, C., Sherwood, T., and Xie, Y. (2019). Neural network model extraction attacks in edge devices by hearing architectural hints. *arXiv preprint arXiv:1903.03916*.

Jain, A., Rasmussen, P. M., and Sahai, A. (2017). Threshold fully homomorphic encryption. *Cryptology ePrint Archive*.

Jindal, A. K., Shaik, I., Vasudha, V., Chalamala, S. R., Ma, R., and Lodha, S. (2020). Secure and privacy preserving method for biometric template protection using fully homomorphic encryption. In *2020 IEEE 19th international conference on trust, security and privacy in computing and communications (TrustCom)*, pages 1127–1134. IEEE.

López-Alt, A., Tromer, E., and Vaikuntanathan, V. (2012). On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 1219–1234.

Lowd, D. and Meek, C. (2005). Adversarial learning. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 641–647.

Ma, J., Naas, S.-A., Sigg, S., and Lyu, X. (2022). Privacy-preserving federated learning based on multi-key homomorphic encryp-

tion. *International Journal of Intelligent Systems*, 37(9):5880–5901.

Mai, G., Cao, K., Yuen, P. C., and Jain, A. K. (2018). On the reconstruction of face images from deep face templates. *IEEE transactions on pattern analysis and machine intelligence*, 41(5):1188–1202.

Mann, Z. Á., Weinert, C., Chabal, D., and Bos, J. W. (2023). Towards practical secure neural network inference: the journey so far and the road ahead. *ACM Computing Surveys*, 56(5):1–37.

Mukherjee, P. and Wichs, D. (2016). Two round multiparty computation via multi-key fhe. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*, pages 735–763. Springer.

Myers, S., Sergi, M., et al. (2011). Threshold fully homomorphic encryption and secure computation. *Cryptology ePrint Archive*.

Oh, S. J., Schiele, B., and Fritz, M. (2019). Towards reverse-engineering black-box neural networks. *Explainable AI: interpreting, explaining and visualizing deep learning*, pages 121–144.

Oliynyk, D., Mayer, R., and Rauber, A. (2023). I know what you trained last summer: A survey on stealing machine learning models and defences. *ACM Computing Surveys*, 55(14s):1–41.

Reith, R. N., Schneider, T., and Tkachenko, O. (2019). Efficiently stealing your machine learning models. In *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society*, pages 198–210.

Riazi, M. S., Samragh, M., Chen, H., Laine, K., Lauter, K., and Koushanfar, F. (2019). {XONN}:{XNOR-based} oblivious deep neural network inference. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1501–1518.

Sperling, L., Ratha, N., Ross, A., and Boddeti, V. N. (2022). Heft: Homomorphically encrypted fusion of biometric templates. In *2022 IEEE International Joint Conference on Biometrics (IJCB)*, pages 1–10. IEEE.

Sugizaki, Y., Tsuchida, H., Hayashi, T., Nuida, K., Nakashima, A., Isshiki, T., and Mori, K. (2023). Threshold fully homomorphic encryption over the torus. In *European Symposium on Research in Computer Security*, pages 45–65. Springer.

Wang, B. and Gong, N. Z. (2018). Stealing hyperparameters in machine learning. In *2018 IEEE*

symposium on security and privacy (SP), pages
36–52. IEEE.

Yan, M., Fletcher, C. W., and Torrellas, J. (2020).
Cache telepathy: Leveraging shared resource at-
tacks to learn {DNN} architectures. In *29th
USENIX Security Symposium (USENIX Security
20)*, pages 2003–2020.

Yang, W., Wang, S., Cui, H., Tang, Z., and Li,
Y. (2023). A review of homomorphic encryp-
tion for privacy-preserving biometrics. *Sensors*,
23(7):3566.