Functional Encryption in Secure Neural Network Training: Data Leakage and Practical Mitigations

1st Alexandru Ioniță
Faculty of Computer Science
Alexandru Ioan Cuza University of Iași
Iași, Romania
0000-0002-9876-6121

2nd Andreea Ioniță
Faculty of Computer Science
Alexandru Ioan Cuza University of Iași
Iași, Romania
0009-0009-0007-3051

***Abstract*—With the increased interest in artificial intelligence, Machine Learning as a Service provides the infrastructure in the Cloud for easy training, testing, and deploying models. However, these systems have a major privacy issue: uploading sensitive data to the Cloud, especially during training. Therefore, achieving secure Neural Network training has been on many researchers' minds lately. More and more solutions for this problem are built around a main pillar: Functional Encryption (FE). Although these approaches are very interesting and offer a new perspective on ML training over encrypted data, some vulnerabilities do not seem to be taken into consideration. In our paper, we present an attack on neural networks that uses FE for secure training over encrypted data. Our approach uses linear programming to reconstruct the original input, unveiling the previous security promises. To address the attack, we propose two solutions for secure training and inference that involve the client during the computation phase. One approach ensures security without relying on encryption, while the other uses function-hiding inner-product techniques.**

***Index Terms*—Privacy-preserving Machine Learning, Deep Learning, Functional Encryption**

I. Introduction

With the rapid growth of interest in machine learning and artificial intelligence, more and more services are outsourced to cloud systems to leverage higher computational power. In this context, Machine Learning as a Service (MLaaS) emerges as a practical solution, offering users access to ready-made tools and scalable resources for developing, training, and deploying models directly in the cloud. A major security concern arises during the training phase, where the cloud provider may gain access to the full training dataset. This is a considerable impediment for machine learning models that operate on sensitive data. An obvious example could be medical information that should not be leaked to third parties. This is usually achieved through anonymization but it does not guarantee protection against inference attacks. Other use cases involve company-specific information or pieces of intelligence. In both cases, a method for encrypting the data and training the model on the encrypted data would be of great interest.

Modern cryptography offers many options for encryption expressivity and granular access control, starting with identity-based [1] and attribute-based encryption [2], which provide granular access on encrypted data. On the other side, *functional encryption* (FE) has become very popular in recent years. This encryption primitive allows only partial decryption of the ciphertext: a decryption key has a function attached to it, and it is allowed only to decrypt the result over a function applied to the initial plaintext.

FE has been used lately in a neural network environment to provide the ability to train over encrypted data [3], [4]. Their approach provides a general framework to transform almost any neural network into one that could train over encrypted data. The computation overhead they provide, alongside the accuracy of the modified network compared to the initial one, indicate that these systems could be used in practice. However, an aspect that has not been taken into account is the information leakage which occurs in the training phase. These articles state that *inference* attacks, such as [5], [6], where a large amount of data is collected to retrieve information about training data, are outside their scope. However, they do not address attacks that aim to directly reconstruct input features from intermediate values exposed during training.

Contribution

We address the security issue of using FE to achieve secure training on encrypted data. We show that it is not safe to use the current proposed neural network constructions from FE, since they recover the plaintext immediately before computing the activation function for the first hidden layer.

We stress that our attack presents a novel approach compared to previous ones on machine learning using FE. In the *honest-but-curious* security model in which the previous schemes were proposed, we impersonate an ill-intentioned Cloud Service Provider (CSP) which tries to learn as much as possible from the training process. We make use of only the information leaked in the intermediate values in the training process without needing to collect information from more than one sample, as most other attacks do.

We provide practical and efficient results of our attack being applied to two neural networks trained on the CIFAR-10 [7] dataset.

Finally, we propose two solutions to mitigate the attack: MITIG1 which utilizes function-hiding inner product encryption (FHIPE), while the other (MITIG2) avoids cryptographic methods entirely but relies on the client's participation in the computation process. We provide an implementation for

MITIG2 over the MNIST [8] dataset, and compare it to previous secure training solutions.

II. Preliminaries

A. Functional Encryption

FE is a form of public-key encryption that enables the computation of a specific function over encrypted data, without revealing the data itself. More formally, a FE scheme is composed of four algorithms, as follows:

setup(λ) : The setup algorithm receives a security parameter λ and returns the public and private keys, mpk and msk.

encrypt(X**,** mpk) : The encryption algorithm receives an input X, and encrypts it under the public key mpk, returning a ciphertext ct.

keygen(f**,** msk) : The key generation algorithm receives a function f and the secret key msk, and returns a decryption key sk_f.

decrypt(ct**,** sk_f) : The decryption algorithm recovers $f(X)$ from the decryption key sk_f and the ciphertext ct (which is obtained by encrypting the original message X)

The neural network training models we will study use different flavors of *Inner Product* FE. In short, the main functionality needed is to compute the inner product $\langle x, y\rangle$ between an encrypted vector x and a vector y associated with a decryption key. This operation is essential in the context of neural networks, where the computation of inner products between input data and weight vectors is a fundamental step repeated at each layer during the forward pass.

Let $W \in \mathbb{R}^{m\times n}$ be a real-valued matrix, and let $X \in \mathbb{R}^n$ be a real-valued column vector. The inner product Z_1 of $\langle W, X\rangle \in \mathbb{R}^m$ is defined as:

$$\begin{pmatrix} x_1 \cdot w_{11} + x_2 \cdot w_{12} + \cdots + x_n \cdot w_{1n} \\ x_1 \cdot w_{21} + x_2 \cdot w_{22} + \cdots + x_n \cdot w_{2n} \\ x_1 \cdot w_{31} + x_2 \cdot w_{32} + \cdots + x_n \cdot w_{3n} \\ \vdots \\ x_1 \cdot w_{m1} + x_2 \cdot w_{m2} + \cdots + x_n \cdot w_{mn} \end{pmatrix} = \begin{pmatrix} z_{11} \\ z_{12} \\ z_{13} \\ \vdots \\ z_{1m} \end{pmatrix}$$

A first FE scheme with this functionality called FEIP (FE for Inner Product) was proposed in [9]. [10] proposed a FE scheme for inner product called **FHIPE** with function hiding capabilities, meaning that the vector y remains hidden to the holder of the decryption key. However, this comes with a considerable overhead in encryption and decryption due to the use of bilinear pairings in algebraic groups.

To successfully decrypt any of the aforementioned schemes, it is necessary to compute the discrete logarithm, a requirement that introduces substantial computational overhead to the entire process. Although other instantiations of FE could be made, for example by using Lattices under the Learning with Errors assumption [11], they also imply a considerable computational overhead.

As our attack is agnostic to the underlying mathematical construction of the FE schemes, we omit the full technical details here, but they can be consulted in the Appendix VI-A.

B. Adversarial Model

As we discuss a security problem, it is important to review the type of adversary that governs the behavior of our attack.
Semi-honest model. This adversary follows the protocol correctly but collects information so that he can learn as much as possible from it. It may apply any polynomial-time algorithm to the data it observes in order to gain more than their due share of knowledge. This type of model was firstly introduced by Goldreich et al. in [12] under the name of ”passive adversary”, as before that only the malicious model existed.
Security in the semi-honest model. A secure protocol in a semi-honest setting has the property that the view obtained in the execution by an adversary is the same as the view obtained in the ideal model. In other words, the attacker should not be able to gain any additional information beyond their authorized output by applying any polynomial-time algorithm to the data available to them.
In our model, the CSP acts as a semi-honest adversary: it follows the protocol as specified but aims to infer additional information from any data it can observe during the computation.

C. Notations and symbols used

Symbol	**Description**
X	Input layer
$[[X]]$	Encryption of X
n	Size of the input X
W	Weight matrix of size $m \times n$
Z_1	Result of inner product $\langle W, X\rangle$
σ	Activation function
b	Bias vector
A_1	First hidden layer, $A_1 = \sigma(Z_1 + b)$
m	Size of the first hidden layer
dZ_1	Derivative of Z_1
$sk_f(W)$	Functional decryption key of W on f function
$p_{i,j}$	Pixel value at position $\{i, j\}$

III. Related work

A. NN using FE

With the growing demand for security, an increasing number of cryptographic schemes are being integrated into neural networks. As highlighted in [13], several machine learning models support inference using functional encryption (FE), but only one—CryptoNN [3]—also enables private training over encrypted data. More recently, we have identified a second such model, FeNet [4], which extends the ideas introduced by CryptoNN.

1) CryptoNN: In 2019, [3] proposed the use of FE to train neural networks directly over encrypted data. They make use of the previously presented FE scheme FEIP [9] in order to encrypt only the training data set. The authors calculate the inner product of the encrypted data and obtained the result in plaintext in the first hidden layer. The rest of the network is in plaintext, except for the last layer which is encrypted using FEBO. The complete scheme can be seen in Figure 1.

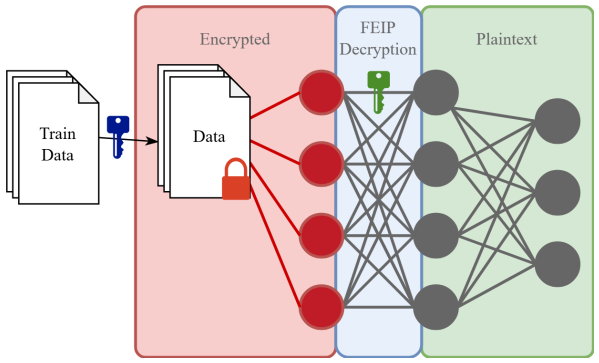


Fig. 1. CryptoNN feed-forward architecture. The client encrypts the input data X using functional encryption (FE) and sends the ciphertext $[[X]]$ to the CSP. In the first hidden layer, the CSP uses the functional decryption key $sk_f(W)$ to compute the inner product $\langle W, X\rangle$ in plaintext. From this point onward, the rest of the neural network operates on plaintext data.

They propose a framework, called CryptoNN, that has accuracy comparable to that of the original neural network model. Essentially, CryptoNN inserts a secure feed-forward step and a secure back-propagation and evaluation steps into the training phase of the normal neural network. A central authority is responsible for generating the cryptographic parameters, including public and private keys, as well as function-specific decryption keys for the server to decrypt the function result.

In order to support matrix computation over encrypted data, they construct secure matrix computation using FE for inner product and for basic operations. Their proposal has three parts: the encryption of input, key generation for the permitted set of functions and finally, the secure computation of function. The first hidden layer takes each encrypted sample X_i and multiplies it with the weights, calculating $\langle WX\rangle$. Since the input is encrypted with the FE scheme, this process becomes:

$$Z_1 = sk_f(W) \cdot [[X]] = \langle W, X\rangle. \tag{1}$$

From that point onward, the standard feed-forward process continues on unencrypted data. It takes the output of the first hidden layer, which is computed in plaintext after the decryption and continues throughout the rest of the network.

It is worth mentioning that even if the first hidden layer is calculated using $[[X]]$, the result is in plaintext because of the properties of FE.

The security of the FE scheme is given by the *Decisional Diffie-Hellman* assumption. However, they address some concerns about possible attacks after the decryption phase. Thus, not related to the security of the encryption scheme. They assume that the server is not an active attacker that will collect a "representative plaintext dataset" for the encrypted training data.

In other words, they prove that an honest-but-curious attacker would, under no circumstances, be able to reconstruct anything with the data that it has. However, in the further sections, we propose an attack on FE neural network schemes, by using FE in order to find the input layer. In our attack, we do not collect a representative plaintext dataset, as we only need one sample in order for the attack to take place.

2) FENet: Although privacy-preserving machine learning (PPML) using FE is still in the early stages, another article that uses FE in order to train neural networks in a secure way is presented by Panzade et al. in [4]. They are the first to use a function hiding inner product encryption in PPML, but also provide a version of encryption of the data in NN using FEIP.

The assumption is that all the computations are made in the MLaaS setting, as in their framework proposal they consider 3 entities: a key management authority, a cloud server, and a client. Similar to CryptoNN, the following security is honest but curious.

The workflow of FENet is actually very similar to CryptoNN: after obtaining master keys from KMA, the client encrypts the input using public keys. However, they use different public keys for each sample. The ciphertexts are then sent to the CS for training, which generates a decryption key based on the weight vector and decrypts the result of the inner product between the input and the weights.

The described process is the first step in the forward propagation, which they refer to as *secure forward propagation*. However, for secure backpropagation, the authors use the categorical cross-entropy loss function and stochastic gradient descent algorithm as an optimizer to adjust the weights. In the inference stage, things are very similar to the secure propagation phase. The sample must be encrypted (with the same encryption algorithm previously used) and after performing secure forward propagation, the classification result is returned to the client.

We stress that the FE is used very similarly in FENet as in CryptoNN in the forward propagation phase. It discloses the plaintext of the product between the encrypted input features and the weights of the first hidden layer, which, as we will show later, leads to information leakage. Although FENet uses FHIPE, the server knows the plaintext values for the weights, thus making it susceptible to our attack, as we can see in Section IV.

B. Attacks on Encrypted ML

An attack against classifiers secured using FE in the training phase was proposed in [6]. The authors examine whether current practical FE schemes leak information about the encrypted input data during the classification process. They show that using this leakage, neural networks can partially reconstruct information that should remain confidential. More precisely, these classifiers involve a first layer with a linear or quadratic function realized using a FE scheme. The output of this layer is then used as training data for the neural network, which acts like an inversion model.

A similar approach to recovering information from inner-product FE-based data classification is presented by Ligier et al. in [5]. They present two attacks: one using an artificial neural network and the other making use of principal component analysis. The principal component analysis method is a statistical procedure that transforms a number of correlated variables into a smaller set of uncorrelated variables, still containing most of the information in it. Both attacks are split into k parts, one for every class, so, this is one of the differences compared to our attack, as we do not need any information besides the weights and the result of the inner product. Their approach using neural networks needs a new neural network for each dataset class in order to find the input layer, which makes the method effective but computationally expensive.

Melis et al. [14] explored unintended feature leakage in collaborative learning by showing that shared gradient updates can reveal sensitive attributes about the participants' local data. Their method involves training an auxiliary classifier to infer private properties from gradients without requiring access to the raw inputs. Although effective, their attack primarily targets feature-level leakage rather than full input reconstruction, and its success depends on the amount of leaked gradient information and the correlation between the target attribute and the model's objective.

Zhu et al. [15] introduced *Deep Leakage from Gradients* (DLG), a new attack that demonstrates how gradient information alone can leak sensitive training data in federated learning. Their method reconstructs both the input and its label by optimizing a dummy input such that its computed gradient matches the observed gradient sent by a client. While their approach works well during the training process, where the weights are re-computed after each iteration, in the case of batched training, the attack becomes inefficient due to the aggregation of gradients.

Carlini et al. have another technique in terms of attacks [16]. They discuss a type of leakage provoked by the memorization of sensitive data in the neural network. They propose a way of verifying if the dataset is prone to be the victim of a memorization attack. While not specific to the case of training a model over encrypted data, this attack can occur on any trained model, and, therefore, may leak information about the input data.

IV. Our Attack

In this chapter, we present an attack that can be done by a CSP while supervising a training process of a NN. The attack is applicable to the two models presented in the previous section, which use FE to preserve the privacy of training data.

Both CryptoNN [3] and FeNet [4] encrypt only the input layer and decrypt the result of the product between the weights and the input, in the first hidden layer, using FE decryption key. Therefore, the CSP has access to the weights of the first hidden layer and their product with the input. We show that we can use this information to partially or fully recover the training data. This attack point is visually depicted in Figure 2.

We emphasize that the type of FE scheme used is irrelevant to our attack, as it only relies on the plaintext revealed after legitimate decryption, staying within the boundaries of

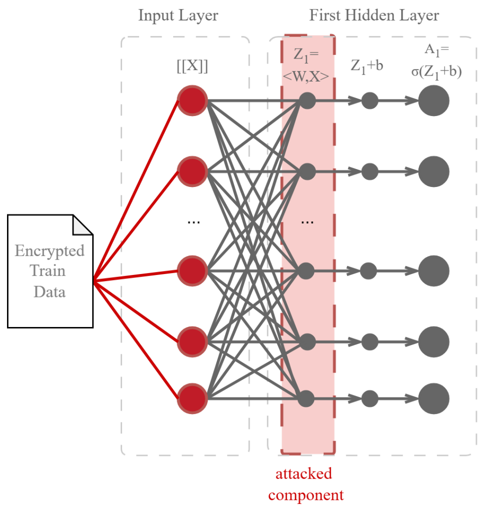
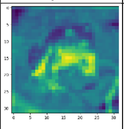
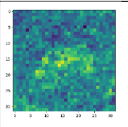
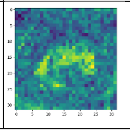
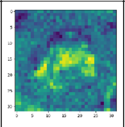
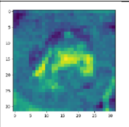


Fig. 2. The figure illustrates the attack point in both training and inference phase: although the input $[[X]]$ is encrypted, the first hidden layer is in plaintext. Using the functional decryption key $sk_f(W)$, the inner product $Z_1 = \langle W, X \rangle$ is revealed, which is then added to the bias b and passed through the activation function to compute $A_1 = \sigma(Z_1 + b)$.

TABLE I
SIMPLE LINEAR PROGRAMMING ATTACK

Layer 1	500	750	900	1000
				

a *honest-but-curious* attacker. The main idea of the attack is to find $X = [X_1, \ldots X_n]$ by making use of the first Layer, which is the inner product between the unknown X and the known weights W. All this information is available to a Cloud administrator, which provides the hardware support for training the network.

We now describe multiple scenarios in which the attack can be applied, demonstrating its flexibility. All attacks variants are constructed using a single sample in the training phase, to emphasize that a large dataset is not needed to reconstruct the input, in contrast with the attacks described at Section III-B.

A. Vulnerabilities when input layer is smaller than the first hidden layer

A straightforward attack scenario arises when the number of neurons in the input layer is smaller than that of the first hidden layer. This use case can be described as follows:

1) n is the number of neurons from the input layer X
2) m is the number of neurons before activation from the first hidden layer $Z_1 = [z_{11}, z_{12}, z_{13}, \ldots z_{1m}]$
3) $n \leq m$
4) $Z_1 = sk_f(W) \cdot [[X]]$ with the aid of *FE*

By decrypting the weight matrix W using the secret key corresponding to the inner product computation with the encrypted input $[[X]]$, we obtain the following plaintext result for Z_1:

$$Z_1 = \begin{pmatrix} x_1 \cdot w_{11} + x_2 \cdot w_{12} + x_3 \cdot w_{13} + \cdots + x_n \cdot w_{1n} \\ x_1 \cdot w_{21} + x_2 \cdot w_{22} + x_3 \cdot w_{23} + \cdots + x_n \cdot w_{2n} \\ x_1 \cdot w_{31} + x_2 \cdot w_{32} + x_3 \cdot w_{33} + \cdots + x_n \cdot w_{3n} \\ \vdots \\ x_1 \cdot w_{m1} + x_2 \cdot w_{m2} + x_3 \cdot w_{m3} + \cdots + x_n \cdot w_{mn} \end{pmatrix}$$

In this matrix, we do not know X but we can find it by solving a system of equations with Gaussian elimination:

$$\begin{pmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1n} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2n} \\ w_{31} & w_{32} & w_{33} & \dots & w_{3n} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ w_{m1} & w_{m2} & w_{m3} & \dots & w_{mn} \end{pmatrix} \cdot X = \begin{pmatrix} z_{11} \\ z_{12} \\ z_{13} \\ \vdots \\ z_{1m} \end{pmatrix}$$

This will lead to a full recovery of the hidden sample. Note that in order for this attack to take place we only need a single entry in the test dataset, and we can process each entry in the dataset individually.

B. Attacking networks with small first hidden layer

When the number of neurons in the first hidden layer is smaller than the input size, we cannot directly apply Gaussian elimination. In this case, the solution space of this system expands exponentially. Finding an arbitrary solution does not help to find one close to the original input, which can be exceptionally challenging, as there are numerous ways to set the variables.

Our attack relies on a *Linear programming* (LP) solver to address a system of linear equations, handling constraints such as equations, inequalities and variable bounds in which the solution is searched. The only inconvenience is that, typically, a linear programming algorithm will aim to optimize the result of some linear objective function, and this is not applicable in our attack. Therefore, we have used the LP solver by setting the coefficients of the objective function to zero.

Although Constraint Satisfaction Programming solvers align with our specific need for a method that does not inherently require an optimization objective, they were ultimately not the chosen approach. Despite offering broad generality for various constraint types, their not very frequent integration into existing scientific computing libraries presented a practical limitation.

We have gradually developed three stages of our attack: simple linear programming with no augmentation and two versions in which we tried to artificially augment the equations, and then the inequalities. Each of these stages is described below:

1) Simple Linear Programming Attack: The first set of equations comes from the inner product $\langle W, X \rangle = Z_1$. In this configuration, the only parameter we varied was the number of neurons in the first hidden layer, but at the same time, this number should have been less than the input layer$(32 * 32)$. So we tested with 500 neurons on the first hidden layer, then 750, 900 and 1000. The results can be seen in Table I. In the first column, we can see the original image, followed by the recovered image in every one of the four test cases. As can be seen, the area of interest (the frog) takes shape, but in a very blurry and with a lot of noise in the first case, with 500 neurons. The increasing number of neurons makes the central piece of the image stand out with more and more precision, but the image still has a lot of noise. Only when the number of neurons is very close to the input size (1024), the result becomes accurate.

2) Linear programming attack using augmentation equations: The results were not anywhere near what we expected, reason why we added some other constraints, based on our empirical research. We observed that, in general, two consecutive pixels are of similar colors, which led us to introduce additional equations of the form:

$$p_{i,j} - \frac{p_{i-1,j} + p_{i+1,j} + p_{i,j-1} + p_{i,j+1}}{4} = 0. \tag{2}$$

When it comes to stringent equations, we had two variables to play with: the number of neurons in the first layer and the number of pixels that are calculated with the formula indicated above. However, our goal was to increase the quality of the image by only adding more calculated pixels, not increasing the number of neurons, as we have already done in the previous case.

The augmentation with stringent equations was another failed attempt, as the quality of the image did not get any better when compared to the one with simple linear programming.

3) Linear programming attack using threshold inequalities: As augmentation with stringent equations was not of very much help, we turned our attention to adding some inequalities in order to solve the system, and therefore to increase the quality of the result image.

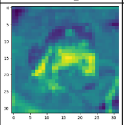
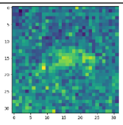
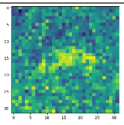
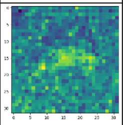
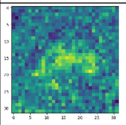
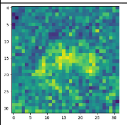
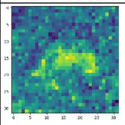
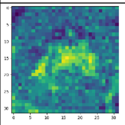
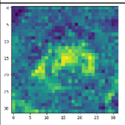
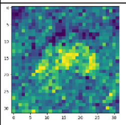
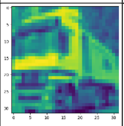
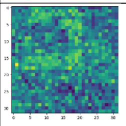
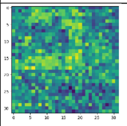
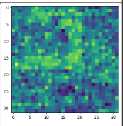
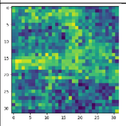
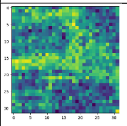
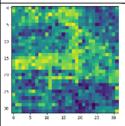
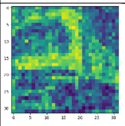
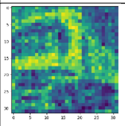
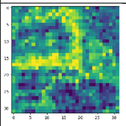
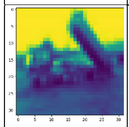
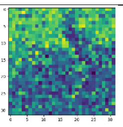
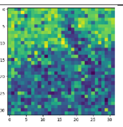
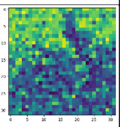
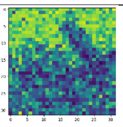
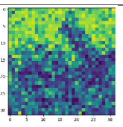
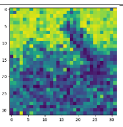
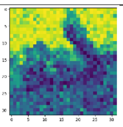
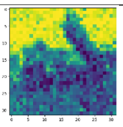
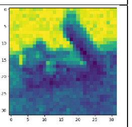
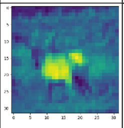
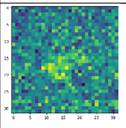
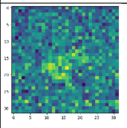
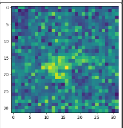
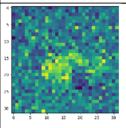
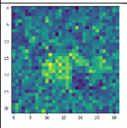
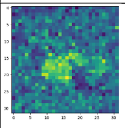
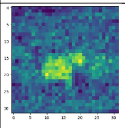
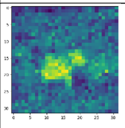
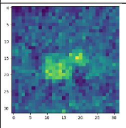
To have more flexibility and remove the imposed limitations in the system, we decided to transform the equation (2) into two inequalities:

$$\left| p_{i,j} - \frac{p_{i-1,j} + p_{i+1,j} + p_{i,j-1} + p_{i,j+1}}{4} \right| \leq t \tag{3}$$

where t is a chosen threshold.

We have made tests with multiple instances of the neural network, with different neuron count on the first hidden layer: 350, 500, 750 and 1000 neurons. Note that the rest of the network, and even the backpropagation phase are irrelevant to this attack. For each instance of the neural network we have run five attacks, with different types of augmentation. First, we have observed that for a neural network with a first dense layer, the best results are obtained when the augmentation inequalities are chosen in a systematic manner, rather than

TABLE II
TEST RESULTS FOR A NEURAL NETWORK WITH A DENSE FIRST LAYER

Layer 1	350			500			750		
Ineq	1/16	1/9	1/4	1/16	1/9	1/4	1/16	1/9	1/4
									
MSE	0.018	0.018	0.016	0.015	0.014	0.011	0.008	0.008	0.010
									
									
									

random. Therefore, we have chosen $1/2$, $1/4$, $1/9$ and $1/16$ pixels, based on the value of $i + j$ modulo 2, 4, 9 and 16.

For some of these tests we have visual and analytical results in Table II. In the first column, we can see the original image, followed by nine tests, separated into three groups, by the number of neurons in the first hidden layer. In each group we have three additional classes, based on the number of inequalities used ($1/16$, $1/9$ and $1/4$ out of the total number of pixels). Under the visual representation, we have the Mean Squared Error (MSE) between the original image and the recovered one.

In Figure 3 we can see how the MSE evolves in each case depending on the number of added inequalities. When the size of the first layer is relatively small, 350 or 500 neurons, the augmenting inequations produce a considerable impact on recovering the input, lowering the MSE from 0.0347 and 0.0265 to 0.0157 and 0.0063 respectively.

Regarding the efficiency of our attack, for a single sample, the runtime varied between 80 and 400 milliseconds, depending on the number of augmenting equations used. This was measured on an average computer. In the case of a CSP having access to the training process, these times could be substantially reduced by the computational power of the Cloud.

We can see that the augmenting equations are a very particular optimization to our first attack. However, similar optimizations can be possible for other data types, such as sound waves, time-series data (e.g. sensor readings) or geodata (e.g. elevation or pollution maps). For categorical data or text input, this optimization cannot be used.

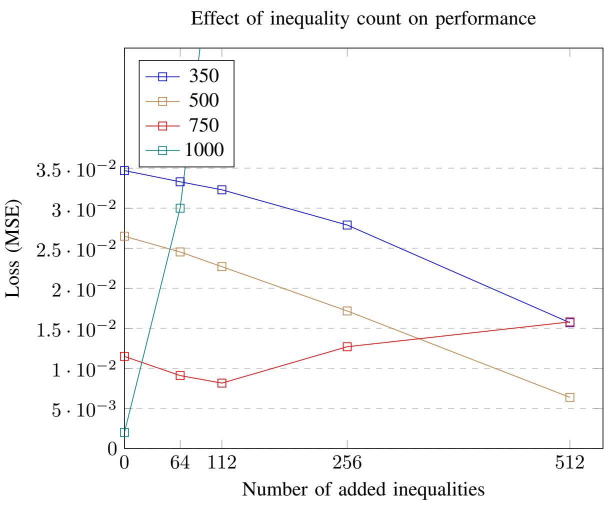


Fig. 3. Increasing inequalities improves results up to a threshold, beyond which it destabilizes the MSE loss, depending on the size of the first hidden layer.

C. *Attack on Convolutional Neural Networks*

We stress that our attack relies on how many equations we can simulate through the FE decryption process. In a Convolutional Neural Network (CNN), the first hidden layer usually is very big. For example, in LeNet5 [17], the first hidden layer consists of 6 feature *layers*, each of them of size 28×28. This results in a number of equations substantially higher than the number of unknown values in the input. Therefore, in these cases we should be able to completely recover the input image, using the approach described in

Section IV-A.

However, we conducted practical tests using a similar approach against CryptoNN applied to LeNet5, by using the linear programming solver used in the previous attack. It should behave the same in the case of a fully defined system of equations. The tests we made were consistent with our expectations, as we were able to fully recover the encrypted images. These tests are also available in our GitHub repository [18].

D. Inference Attack

While the above examples have been presented in the context of training, it is important to emphasize that these vulnerabilities apply also during the inference phase. In this setting, the client encrypts its input and sends it to the server, which uses the functional decryption key to compute the inner product $\langle W, X\rangle$ in plaintext before proceeding with the rest of the computation on unencrypted data. As in the training phase, the attacker has access to both the decrypted inner product $Z_1 = \langle W, X\rangle$ and the weight matrix W, enabling identical reconstruction attacks based on this exposed information.

E. A Collusion Attack

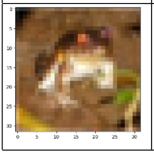
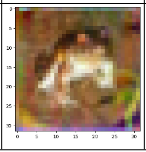
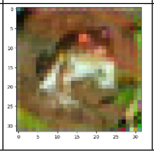
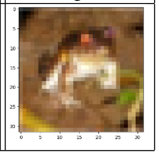
In CryptoNN, the plaintext dataset is encrypted only once, at the pre-processing phase. Then, in the training phase, at each iteration the weights for the neural network change. In a subsequent phase, new decryption keys will be issued for these weights. These new keys can be used to obtain new equations about the encrypted samples. Depending on how many samples are in each iterations and how the blocks on input samples are constructed, it may be possible in CrpytoNN to use decryption keys designated to different blocks with the same sample, in order to obtain more information.

More concrete, the attacker could gather observations of the same input sample across multiple epoch. If the size of the first hidden layer m is $\lceil n/k\rceil$, for $k \in \mathbb{N}$(where n is the size of the input layer) and the training takes place across k epochs, then an attacker could create a system with n equations and n unknowns. This allows him to use Gaussian elimination to fully reconstruct the image, as shown in the first version of our attack, rather than relying on the weaker linear programming attack.

FENet is encrypting every image with a different public key. This removes the collusion of decryption keys for different samples or blocks of samples: we will not be able to use decryption keys designed for two different input samples on the same input, since their decryption keys do not match.

One minimal way to protect against this attack type would be to shuffle the samples between iterations and re-encrypt the dataset. An attacker now needs to reproduce the original order using the information leaked from each iteration, and then combine these pieces of information to recover the full sample. Note that the information leaked from each iteration is actually a system of equations, which can be combined regardless of the encryption and decryption keys used to obtain it.

TABLE III
COMPARING THE THREE ATTACKS

E1	E2	E3	Original
			

Since this attack has some obvious counter-measures, we did not implemented it in our work. Our attack is only based on the information we can retrieve in a single run of the forward propagation phase.

F. Detailed Results and Comparison with Previous Attacks

1) Implementation of the attack: Our attacks are available in our GitHub repository [18]. We have used the **SciPy** library [19] for linear programming solvers. We have obtained the best results using the "interior-point" solver, despite it being deprecated in newer versions of this software. Since the optimization of the objective function is not applicable in our case, we just placed a vector full of zeroes to the objective function. This means that our solution actually ignores the optimization part and searches for any solution.

In addition to the experiments conducted over grayscale CIFAR10 using a first dense hidden layer, we have also run experiments against ResNet56 [20] using unaltered CIFAR10, in order to be able to directly compare our results to DLG [15] and Melis [14].

We have run three experiments, as follows:

- Experiment 1 (E1). From the 16-channel convolutional first layer of ResNet56 we have used only 3 channels in order to fully reproduce the image using our first approach - Gaussian elimination.
- Experiment 2 (E2). In order to also test the linear programming approach, we have used only one channel from the first hidden layer. This experiment has no augmentation inequations.
- Experiment 3 (E3). This is similar to E2, but it also provides around 60 augmenting inequations. Since we have observed that E2 gives good results especially in the central part of the images, but lascks precision for the margins, we have selected these inequations corresponding to pixels from the margins of the image.

A comparison between our results and the ones obtained in [15] are summarized in Figure 4. While Melis has an error of approximate 0.2, the other attack have a considerable lower one, much closer to zero. Note that E2 and E3 are reminded here only for theoretical purposes, since we can fully recover the image using the E1 approach. And even there, we are only using 3 out the 16 channles of the convolutional layer used in ResNet56. Therefore, this approach will be able to maintain it's performance even for smaller networks, while DLG will suffer from reducing the number of gradients. Table

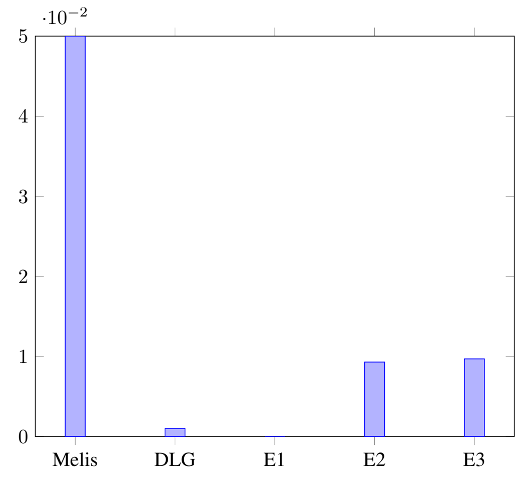


Fig. 4. Comparison between our attacks (E1, E2, E3), DLG [15] and Melis [14] on CIFAR Dataset using MSE loss function. While DLG has an error of approximately 10^{-3}, E1 variant of our experiment fully recovers the image, the MSE being of magnitude $10^{-}29$, a residual error attributed to floating-point precision.

TABLE IV
COMPARISON OF NEURAL NETWORK DATA LEAKAGE ATTACKS IN TERMS OF REQUIRED FEATURES, SUPPORT FOR SINGLE-SAMPLE RECOVERY, AND COMPUTATIONAL DEMANDS.

Attack	Features	Single Sample	Computational power
Carpov [6]	Yes	No	Moderate to High
Ligier et al. in [5]	Yes	No	High
Ours	No	Yes	Low
DLG [15]	No	Yes	High
Melis [14]	No	Yes	Moderate

III presents an example on how these three attacks reconstruct an image from CIFAR-10.

Compared to previous attacks, ours is fundamentally different from any other ones operating on privacy-preserving training of neural networks. Table IV presents a comparative overview of existing neural network data leakage attacks, highlighting the need of features, whether they support single-sample reconstruction, and their associated computational costs.

[6] uses a great amount of computational power in order to retrieve information, by training a new neural network. Compared to all these attacks, ours only uses linear programming, a much more efficient technique with algorithms running in almost quadratic time [21].

While in [15] the attack performs better when the batch size is small, and decreases its accuracy when the batch size grows, in our attack it does not matter.

G. *Applicability of the attack*

We stress that the main issue addressed by private training of NN is to protect sensitive data against the CSP. When outsourcing the training of a neural network in cloud, it is desirable that the CSP learns as little as possible about the training dataset. In this scenario, the service provider is considered a highly capable adversary, operating under a white-box model as it has access not only to the (encrypted) dataset, but also to the full description of the neural network and all the weights and activations available in the training process. Hence, our attack only uses the data already available while observing the training process, without interfering with it (for example, modifying the weights). This makes the attack compliant with the Semi-Honest adversary, which was the model used in CryptoNN and FENet.

V. MITIGATION OF THE ATTACK

Our attack remains effective independently of the specific functional encryption scheme employed, relying solely on the assumption that both the inner product result and the weight matrix are available in plaintext. Hence, a secure training system based on the above-mentioned idea, should not reveal the weights, as the input can be reconstructed. In this section, we present two ways of attack mitigation, one using a more powerful FE scheme and another one that does not use encryption at all but moves a part of the computation on the client's side.

A. *MITIG1 - Solution using FHIPE*

As mentioned above, our attack is possible because the weights are revealed. So, a first idea would be to hide the weights, therefore it came the need of function hiding inner product encryption. However, if the server should now know the weights, it means a trade off in terms of client implication. The client should generate and encrypt the weights and send them to the server along with both the encrypted input and the FHIPE keys for decrypting the inner product. However, the involvement of the client does not stop there, as he has to also calculate the derivative and update the weights locally in order to encrypt them again and send them back to the server.

In other words, for a secure training of an NN, a part of the overhead computation has to be done on the client's part. For each feed-forward stage, the client has to generate, encrypt and send the weights to the server. On the other hand, at each backpropagation stage, the client has to calculate the derivative and then update, encrypt and send the new weights. A formal description of the client and server methods can be observed in Algorithms 1 and 2.

The advantage of this system is that the calculation of the inner product is done in the cloud. However, the disadvantages are not easy to overcome. Not only does the client become an integral part of the interaction, but the use of bilinear applications in FHIPE—constructions that perform exceedingly slowly in real-world implementation—further compounds the issue.

Although [4] also uses FHIPE in training process, there is a fundamental difference between their scheme and MITIG1. In [4] FHIPE is used in the same way as regular FE: The server receives the encrypted input $[[X]]$ and the weights W, computes a secret key for the weights, and obtains through decryption the product between input and weights $W \cdot X$. Since

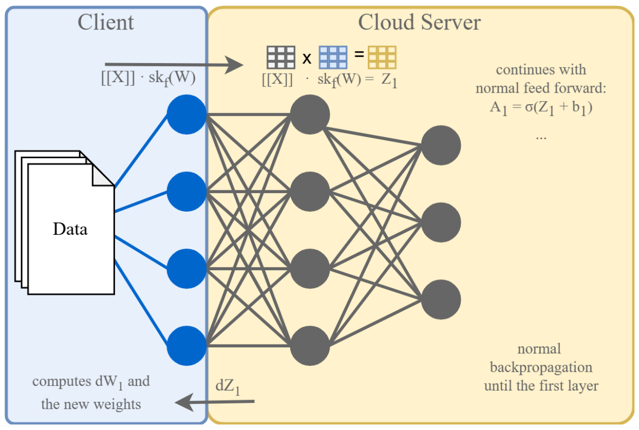

Client
Cloud Server
[[X]] · sk_f(W)
[[X]] · sk_f(W) = Z_1
continues with normal feed forward: A_1 = σ(Z_1 + b_1)
...
Data
normal backpropagation until the first layer
computes dW_1 and the new weights
dZ_1

Fig. 5. MITIG1 feed-forward and backpropagation using FHIPE. The client generates the weights W, encrypts both the input X and weights W using FHIPE, and sends them to the server. Due to the properties of FHIPE, the server computes the inner product $Z_1 = \langle W, X\rangle$ directly in plaintext. During backpropagation, the server sends the derivative of Z_1 to the client, who updates and re-encrypts the weights for continued training.

the server has access to both $W \cdot X$ and W, our attack is easily applicable in their system. In our point of view, the correct usage of FHIPE implies that the weights remain hidden to the server, thus, in order to compute the gradients in the backpropagation phase, needs to communicate with the client.

Since MITIG1 involves a great amount computing overhead in the training process, combined with the fact that it also involves an interraction between the client and server during the training phase, we proposed another variant of this mitigation, more efficient without losing security. Therefore, we will focus our attention in the next part on the MITIG2 scheme. However, we stress that MITIG1 still has a great theoretical importance: although current FE schemes are limited, in the perspective of a new FE scheme with homomorphic decryption key capabilities, this scheme could completely remove the communication overhead, being able to readjust the decryption key to for the new weights according to the gradients, without leaking additional information.

Algorithm 1 MITIG1 Client

```
1:  for $j = 1, 2, \ldots epoch$ do
2:      Generate $W$
3:      for each sample/batch X do
4:          $[[X_i]] = FHIPEEncrypt(X_i)$
5:          $sk_f(W) = FHIPEKeyDerive(W)$
6:          Send $[[X_i]], sk_f(W)$ to server
7:          The server calculates $Z_1 = sk_f(W) \cdot [[X]]$ that continues with feed-forward
8:          On back-propagation, the client receives $dZ_1$
9:          $dW = \frac{1}{m} \cdot dZ_1 \cdot x$
10:         $W = W - learningRate \cdot dW$
11:     end for
12: end for
```

Algorithm 2 MITIG1 Server

```
1: for $j = 1, 2, \ldots epoch$ do
2:     for each sample/batch X do
3:         Receives $[[X_i]], [[W]]$ from the client
4:         Computes $Z_1 = FHIPEDecrypt([[X_i]] \cdot [[W]])$
5:         Feed-forward process
6:         On back-propagation, the server computes $dZ_1$
7:         Sends $dZ_1$ to the client
8:     end for
9: end for
```

B. MITIG2 - Solution without Encryption

Building on top of MITIG1, we considered an alternative approach: if the weights are generated on the client side, we could avoid encryption at all. This perspective led to our second concept for a secure system: computing the inner product locally, in plaintext, and sending only the result to the server as input to the first hidden layer. This leads to a solution very similar to Split Learning (SL) [22] in which the NN is split in the training phase between a client and a server. While usually SL models are focused on distributed and collaborative learning, we show in the following sections that it can also be used in our case to provide a strictly stronger privacy guarantee and better running time compared to previous models using FE.

For each feed-forward step, the client has to recalculate the inner product between the input and the weights. For each backpropagation, it needs to calculate the derivative and update the weights. The server then computes as usual forward- and back-propagation on the rest of the network. Indeed, the overhead of the client is considerably larger in this case, but it is interesting to see if the overall time is increased because, in fact, we do not work anymore on encrypted data, but on plaintext. Considering the time saved by eliminating the need for encryption, the trade-off of having the client execute some resource-intensive operations on their end no longer appears as burdensome.

In order to speed up the solution, by reducing the interaction between the server and the client, we processed inputs in batches. Although the client sends the same amount of information, it will actually receive considerably less, since the gradients of the input of the first hidden layer will only be sent after a batch is processed.

C. Inference in MITIG1 and MITIG2

After the secure training phase in both MITIG1 and MITIG2, the weights W connecting the input layer to the first hidden layer remain known only to the client. To enable secure inference, the client encrypts these weights using a functional inner product encryption (FEIP) scheme and sends the corresponding functional decryption key $sk_f(W)$ to the server. During inference, each client encrypts its input X and transmits the ciphertext $[[X]]$ to the server. Using the functional decryption key, the server computes the inner product $Z_1 = sk_f(W) \cdot [[X]] = \langle W, X\rangle$ between the encrypted input and the encrypted weights, obtaining the result in plaintext.

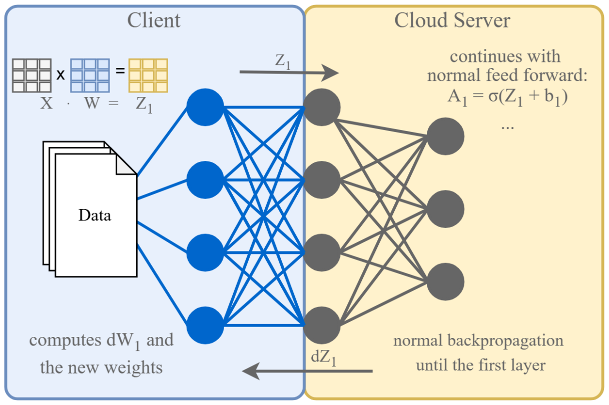


Fig. 6. MITIG2 feed-forward and backpropagation without encryption. The client generates the weights and computes $Z_1 = \langle W, X \rangle$, then sends Z_1 in plaintext to the server, which continues the standard feed-forward computation. During backpropagation, the server computes the derivative of Z_1 and sends it back to the client. The client then updates the weights, recalculates Z_1, and the process continues.

Algorithm 3 MITIG2 Client
```
1: for $j = 1, 2, \ldots epoch$ do
2:     Generate $W$
3:     for each sample/batch X do
4:         Compute $Z_1 = \langle W, X \rangle$
5:         Send $Z_1$ to the server that continues with feed-forward
6:         On back-propagation, the client receives $dZ_1$
7:         $dW = \frac{1}{m} \cdot dZ_1 \cdot X$
8:         $W = W - learningRate \cdot dW$
9:     end for
10: end for
```

From this point onward, the feed-forward process continues entirely in plaintext.

D. Security

We can observe that our solution provides strictly less information to the cloud server used for training compared to both CryptoNN [3] and FENet. The most important part is that it hides the weights between the input and the first hidden layer. This means that in order to recover the input, the CSP will have to solve a system of quadratic equations, where both W and X are unknown. This problem, solving a system of quadratic equations, is proven to be NP-hard [23].

A similar problem, the *Multivariate quadratic* (MQ) systems over finite fields, has been used before as basis for the security of multiple cryptographic schemes [24]–[26]. However, the use case is a bit different, since this problem only requires a solution to be found. In our case, if there are multiple solutions, we need to reconstruct the one corresponding to the encrypted sample x, making our problem considerably harder.

Investigating the computational hardness of this exact recovery task is a valuable direction for future research, as it lies at the intersection of cryptography, machine learning, and computational complexity. While the general MQ problem has been extensively studied, the added constraint of identifying the correct solution among potentially many introduces a new layer of difficulty that is not well understood.

Algorithm 4 MITIG2 Server
```
1: for each epoch do
2:     for each sample/batch X do
3:         Receives $Z_1$ from the client
4:         $Z_1 \leftarrow Z_1 + b_1$
5:         $A_1 \leftarrow \sigma(Z_1)$
6:         Feed-forward process on plaintext
7:         On back-propagation, the server computes $dZ_1$
8:         Sends $dZ_1$ to the client
9:     end for
10: end for
```

E. Comparisons with Other Models

We have also provided an actual implementation of our MITIG2 scheme, in order to also compare the training time and accuracy of our model, beside the security. We have used a neural network of a single hidden layer with a dense layer of 300 neurons, against the MNIST Dataset [8]. Our tests were performed using a AMD Ryzen 7 Processor with 32GB of RAM under a Debian operating system.

TABLE V
COMPARING OUR SCHEME WITH SIMILAR OTHER ONES

Method	Training Time	Epoch	Accuracy
unencrypted baseline NN [4]	6.8 s	2	96.22%
CryptoNN [3]	52 hrs	2	95.49%
FENet (FIPE) [4]	2 hrs	2	94.87%
FENet (FHIPE) [4]	26 hrs	2	94.5%
MITIG2 (unbatched)	6 min	2	92.6%
MITIG2 (batch 10)	2 min	5	95.6%
MITIG2 (batch 10)	5 min	10	96.7%
SecureNN [27]	30 hrs	-	99.15%
Glyph [28]	8 days	-	98.6%

As we can see from Table V, compared to the other FE-powered NN for training over encrypted data - CryptoNN [3] and FENet [4], our model outperforms them on both accuracy and training time.

Models relying on secure multi-party computation and fully homomorphic encryption - SecureNN [27] and Glyph [28] offer a better accuracy, but their training time is considerable higher than ours.

We have also tested our MITIG2 model across various parameters for batch size and epochs, in order to obtain a better feel about how to accuracy and running time varies with these parameters. The results can be seen in Table VI.

F. Other Possible Mitigation Techniques

There are a series of other possible mitigations of this attack, based on various techniques. The first idea would be to use a more powerful FE mechanism in order to encrypt multiple

TABLE VI
TEST RESULTS FOR A NEURAL NETWORK WITH A DENSE FIRST LAYER

Epochs	Batch size	Running time	Accuracy
10	10	5 min	96.69%
10	100	2 min	95.51%
10	250	1 min	94.62 %
50	10	21 min	97.55%
50	100	6 min	96.8%
50	250	5 min	96.36 %
25	10	11 min	97.39%

layers of the neural network. This is impossible given the current state of the art in FE schemes.

There are however other methods of performing calculations on encrypted data, including homomorphic encryption (HE). This technique permits computations to be carried out on encrypted data without decrypting it first, preserving privacy. Mazzone et al. [29] proposed a NN training scheme where HE was used in order to encrypt multiple layers of a NN during the training process. In this setup, if the encryption is used on at least the first two layers, the attack is no longer applicable.

In order to assure Differential Privacy in machine learning models, controlled noise is added at specific steps of the training process. A similar approach could be used in mitigating our attack by adding noise in the input layer. As an observation, the noise added on further layers would not infer with our attack. However, it was shown in [30] that input perturbation considerably degrades the model, compared to other approaches used in DP.

Trusted Execution Environments (TEE) [31] could represent another method of preventing the attack. However, due to the limited computation power enforced by TEE, it is hard to construct and design such systems for training deep learning models [32].

VI. CONCLUSIONS

The current FE approaches have opened a new promising path for secure training of neural networks over encrypted data. However, current models are severely limited in terms of security, leaking a considerable amount of information. We have shown that under the existing architecture, FE provides little to no protection during training. Our basic attack, which requires no input augmentation, can be applied to any type of training data. The attacks that featured augmentation through inequalities related to neighboring pixels in the images can be applied to a limited type of input features, such as images, sound waves, and videos.

Regarding our proposed mitigation methods, we stress that they offer strictly better results compared to the previously mentioned FE-based approaches in terms of security, training time and accuracy. We have seen that split learning can be adapted to enhance security compared to schemes using functional encryption for securely train deep models in cloud.

Despite information leakage, we consider that the use of a FE-powered neural network provide a promising foundation for achieving secure training over encrypted data. We think that further exploration of FE expressiveness and constructing new schemes for more powerful functions, such as multi-input high-degree polynomials, will eventually lead to more secure neural networks.

REFERENCES

[1] D. Boneh, "Identity-based encryption from the weil pairing." Crypto, 2001.

[2] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," in *Proceedings of the 13th ACM conference on Computer and communications security*, 2006, pp. 89–98.

[3] R. Xu, J. B. Joshi, and C. Li, "Cryptonn: Training neural networks over encrypted data," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1199–1209.

[4] P. Panzade and D. Takabi, "Fenet: Privacy-preserving neural network training with functional encryption," in *Proceedings of the 9th ACM International Workshop on Security and Privacy Analytics*, 2023, pp. 33–43.

[5] D. Ligier, S. Carpov, C. Fontaine, and R. Sirdey, "Information leakage analysis of inner-product functional encryption based data classification," in *2017 15th Annual Conference on Privacy, Security and Trust (PST)*. IEEE, 2017, pp. 303–3035.

[6] S. Carpov, C. Fontaine, D. Ligier, and R. Sirdey, "Illuminating the dark or how to recover what should not be seen in fe-based classifiers," *Proceedings on Privacy Enhancing Technologies*, vol. 2020, no. 2, pp. 5–23, 2020.

[7] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[8] "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*.

[9] M. Abdalla, F. Bourse, A. De Caro, and D. Pointcheval, "Simple functional encryption schemes for inner products," in *IACR International Workshop on Public Key Cryptography*. Springer, 2015, pp. 733–751.

[10] S. Kim, K. Lewi, A. Mandal, H. Montgomery, A. Roy, and D. J. Wu, "Function-hiding inner product encryption is practical," in *International Conference on Security and Cryptography for Networks*. Springer, 2018, pp. 544–562.

[11] J. M. B. Mera, A. Karmakar, T. Marc, and A. Soleimanian, "Efficient lattice-based inner-product functional encryption," in *IACR International Conference on Public-Key Cryptography*. Springer, 2022, pp. 163–193.

[12] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game, or a completeness theorem for protocols with honest majority," in *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, 2019, pp. 307–328.

[13] P. Panzade, D. Takabi, and Z. Cai, "Privacy-preserving machine learning using functional encryption: Opportunities and challenges," *IEEE Internet of Things Journal*, vol. 11, no. 5, pp. 7436–7446, 2023.

[14] L. Melis, C. Song, E. De Cristofaro, and V. Shmatikov, "Exploiting unintended feature leakage in collaborative learning," in *2019 IEEE symposium on security and privacy (SP)*. IEEE, 2019, pp. 691–706.

[15] L. Zhu, Z. Liu, and S. Han, "Deep leakage from gradients," *Advances in neural information processing systems*, vol. 32, 2019.

[16] N. Carlini, C. Liu, Ú. Erlingsson, J. Kos, and D. Song, "The secret sharer: Evaluating and testing unintended memorization in neural networks," in *28th USENIX security symposium (USENIX security 19)*, 2019, pp. 267–284.

[17] Y. LeCun, L. Jackel, L. Bottou, A. Brunot, C. Cortes, J. Denker, H. Drucker, I. Guyon, U. Muller, E. Sackinger *et al.*, "Comparison of learning algorithms for handwritten digit recognition," in *International conference on artificial neural networks*, vol. 60, no. 1. Perth, Australia, 1995, pp. 53–60.

[18] A. Ionita and A. Ionita, "Attacks on nn using functional encryption for secure training and efficient mitigation," https://github.com/Juve45/CryptoNN-Attack.

[19] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0:

Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[21] J. van den Brand, "A deterministic linear program solver in current matrix multiplication time," in *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2020, pp. 259–278.

[22] P. Vepakomma, O. Gupta, T. Swedish, and R. Raskar, "Split learning for health: Distributed deep learning without sharing raw patient data," *arXiv preprint arXiv:1812.00564*, 2018.

[23] K. Dvijotham, "Systems of quadratic equations: Efficient solution algorithms and conditions for solvability," in *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2015, pp. 1027–1031.

[24] W. Beullens, "Mayo: practical post-quantum signatures from oil-and-vinegar maps," in *International Conference on Selected Areas in Cryptography*. Springer, 2021, pp. 355–376.

[25] Y.-J. Huang, F.-H. Liu, and B.-Y. Yang, "Public-key cryptography from new multivariate quadratic assumptions," in *Public Key Cryptography–PKC 2012: 15th International Conference on Practice and Theory in Public Key Cryptography, Darmstadt, Germany, May 21-23, 2012. Proceedings 15*. Springer, 2012, pp. 190–205.

[26] K.-A. Shim, C.-M. Park, and N. Koo, "An efficient mq-signature scheme based on sparse polynomials," *IEEE Access*, vol. 8, pp. 26 257–26 265, 2020.

[27] S. Wagh, D. Gupta, and N. Chandran, "Securenn: 3-party secure computation for neural network training," *Proceedings on Privacy Enhancing Technologies*, 2019.

[28] Q. Lou, B. Feng, G. Charles Fox, and L. Jiang, "Glyph: Fast and accurately training deep neural networks on encrypted data," *Advances in neural information processing systems*, vol. 33, pp. 9193–9202, 2020.

[29] F. Mazzone, A. Al Badawi, Y. Polyakov, M. Everts, F. Hahn, and A. Peter, "Encrypt what matters: Selective model encryption for more efficient secure federated learning," in *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 2025, pp. 96–115.

[30] I. Jarin and B. Eshete, "Dp-util: comprehensive utility analysis of differential privacy in machine learning," in *Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy*, 2022, pp. 41–52.

[31] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not," in *2015 IEEE Trustcom/BigDataSE/Ispa*, vol. 1. IEEE, 2015, pp. 57–64.

[32] M. F. Babar and M. Hasan, "Trusted deep neural execution—a survey," *IEEE Access*, vol. 11, pp. 45 736–45 748, 2023.

Appendix

A. Functional Encryption

FE for Basic Operations (FEBO)

[3] et al. presented a FE scheme that supports basic operations.

- **Setup**: generates (G, p, g), secret $s \leftarrow Z_p$ and returns (msk, mpk) where $msk \leftarrow s$ and $mpk \leftarrow (h, g)$, where $h = g^s$.
- **KeyDerive** receives
  1) the keys (mpk, msk)
  2) the commitment cmt
  3) the operation $Delta$
  4) the input of the function y

  and outputs

$$sk_{f_\Delta} = \begin{cases} cmt^s \dot{g}^{-y}, & \text{if } \Delta = + \\ cmt^s \dot{g}^{y}, & \text{if } \Delta = - \\ (cmt^s)^y, & \text{if } \Delta = * \\ (cmt^s)^{y^{-1}}, & \text{if } \Delta = / \end{cases} \quad (4)$$

FE for Inner Product (FEIP)

In order to calculate the inner product of two encrypted vectors, [9] et al. proposed the following FE scheme:

- **Setup**: generates (G, p, g), secret $(s_1, s_2, \dots s_\eta) \leftarrow Z_p^\eta$ and returns (msk, mpk) where $msk \leftarrow s$ and $mpk \leftarrow (h_i, g)_{i \in |\eta|}$, where $h_i = g^{s_i}$.
- **Keyderive** receives the secret key msk, y and returns $sk_f = \langle y, s \rangle$
- **Encrypt** takes the public key, the value x to be encrypted, chooses a random $r \leftarrow \mathbf{Z}_p$ and computes
  1) $ct_0 = g^r$
  2) $ct_i = h_i^r \dot{g}^{x_i}$

  returning the ciphertext
- **Decrypt** takes the ciphertext ct, the public key mpk and functional key sk_f for the vector y and returns the discrete logarithm $g^{\langle x,y \rangle} = \prod_{i \in |\eta|} ct_i^{y_i} / ct_0^{sk_f}$

Function Hiding Inner Product Encryption (FHIPE)

[10] propose a FE scheme for inner product that has the property to hide the y with whom the inner product is calculated. To be more specific, let x be the secret, for which we are calculating the encryption. We use FE to calculate $\langle x, y/rangle$ without revealing neither of x and y.

The construction makes use of bilinear groups and contains four functions:

- **Setup**: generates the public parameters $pp = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, q, e, S)$ and the master secret key $msk = (pp, g_1, g_2, \mathbf{B}, \mathbf{B}^\star)$. Obs: g_1 and g_2 are generators,e is the $\mathbf{B}^\star$ is the inverse of matrix $\mathbf{B}$
- **KeyGen** receives the secret key msk, x (a vector $x \in \mathbb{Z}_\mathrm{g}^\mathrm{n}$) chooses a uniformly random element $\alpha \overset{\mathbb{R}}{\leftarrow} \mathbb{Z}_g$ and returns the pair $sk = (K_1, K_2) = ({g_1}^{\alpha \cdot \det(\mathbf{B})}, {g_1}^{\alpha \cdot x \cdot \mathbf{B}})$
- **Encrypt** takes the secret key $msk$and a vector $y \in \mathbb{Z}_\mathrm{g}^\mathrm{n}$) chooses a uniformly random element $\beta \overset{\mathbb{R}}{\leftarrow} \mathbb{Z}_g$ and outputs the pair $ct = (C_1, C_2) = ({g_2}^\beta, {g_2}^{\beta \cdot y \cdot \mathbf{B}^\star})$
- **Decrypt** takes the ciphertext ct, the secret key sk and the public parameters pp and computes $D_1 = e(K_1, C_1)$ and $D_2 = e(K_2, C_2)$. Then it checks whether there exists $z \in S$ such that $(D_1)^z = D_2$. The result is z.