MLCapsule: Guarded Offline Deployment of Machine Learning as a Service

Lucjan Hanzlik[1], Yang Zhang[1], Kathrin Grosse[2],
Ahmed Salem[1], Maximilian Augustin[3], Michael Backes[1], and Mario Fritz[1]
[1] CISPA Helmholtz Center for Information Security
{hanzlik, zhang, ahmed.salem, backes, fritz}@cispa.de
[2] University of Cagliari
kathrin.grosse@unica.it
[3] University of Tuebingen
maximilian.augustin@uni-tuebingen.de

Abstract

Machine Learning as a Service (MLaaS) is a popular and convenient way to access a trained machine learning (ML) model trough an API. However, if the user's input is sensitive, sending it to the server is not an option. Equally, the service provider does not want to share the model by sending it to the client for protecting its intellectual property and pay-per-query business model. As a solution, we propose MLCapsule, a guarded offline deployment of MLaaS. MLCapsule executes the machine learning model locally on the user's client and therefore the data never leaves the client. Meanwhile, we show that MLCapsule is able to offer the service provider the same level of control and security of its model as the commonly used server-side execution. Beyond protecting against direct model access, we demonstrate that MLCapsule allows for implementing defenses against advanced attacks on machine learning models such as model stealing, reverse engineering and membership inference.

1. Introduction

Machine learning as a service (MLaaS) has become increasingly popular during the past five years. Leading Internet companies, such as Google,[1] Amazon,[2] and Microsoft[3] deploy their own MLaaS. It offers a convenient way for a service provider to deploy a machine learning (ML) model and equally an instant way for a user/client to make use of the model in various applications.

While MLaaS is convenient for the user, it also comes with several limitations. First, the user has to trust the service provider with the input data. Typically, there are no means of ensuring data privacy and recently proposed encryption mechanisms [6] come at substantial computational overhead especially for state-of-the-art deep learning models. Moreover, MLaaS requires data transfer over the network which constitutes high volume communication and provides a new attack surface [20, 24]. This motivates us to come up with a client-side solution such that perfect data privacy and offline computation can be achieved.

Yet, this (seemingly) comes with a loss of control of the service provider. Running the ML model on the client's machine raises concerns about revealing details of the model, damaging the intellectual property of the service provider. Further, granting unrestricted/unrevocable access to the user breaks the commonly enforced pay-per-query business model. Moreover, there is a plethora of attacks on machine learning models that raise severe concerns about security and privacy [26]. A series of recent papers have shown different attacks on MLaaS that can lead to reverse engineering [35, 22] and training data leakage [12, 11, 29, 38, 28]. Many of these threats are facilitated by repeated probing of the ML model. Therefore, we need a mechanism to enforce that the service provider remains in control of the model access as well as provide ways to deploy detection and defense mechanisms in order to protect the model.

[1] https://cloud.google.com/

[2] https://cloud.google.com/vision/

[3] https://azure.microsoft.com/en-us/services/machine-learning-studio/

1.1. Our Contributions

We propose MLCapsule, a guarded offline deployment of machine learning as a service. MLCapsule follows the popular MLaaS paradigm, but allows for client-side execution where model and computation remain secret. MLCapsule protects intellectual property and maintains the business model for the provider. Meanwhile, the user gains offline execution and thereby perfect privacy, as the data never leaves the client's machine.

MLCapsule uses an Isolated Execution Environment (IEE) to provide a secure enclave to run an ML model. IEE provides means to prove execution of code, and only the enclave can decrypt the received data. This keeps the intellectual property of the service provider secure from other processes running on the client's platform.

We also contribute by a proof-of-concept of our solution. Due to its simplicity and availability, we implemented our prototype on a platform with Intel SGX (yet, the current generation should not be used due to devastating attacks for example [8]). Our solution can be used on any IEE platform. In more details, in our solution we design so called MLCapsule layers, which encapsulate standard ML layers and are executed inside the IEE. Those layers are able to decrypt (unseal) the secret weight provisioned by the service provider and perform the computation in isolation. This modular approach makes it easy to combine layers and form large networks. We further evaluate the overhead of entire networks (VGG-16 [31] and MobileNet [14]) and individual (dense and convolutional) layers.

MLCapsule is also able to integrate advanced defense mechanisms for attacks against machine learning models. For demonstration, we propose two defense mechanisms against reverse engineering [22] and membership inference [29, 27] respectively, and utilize a recent proposed defense [17] for model stealing attacks [35]. We show that these mechanisms can be seamlessly incorporated into MLCapsule, with a negligible computation overhead.

2. Requirements and Threat Model

In this section, we introduce the requirements we want to achieve in MLCapsule.

User Side. MLCapsule deploys MLaaS locally, providing a strong privacy guarantee to the user, as her data never leaves her devices. Local execution also avoids the Internet communication between the user and the MLaaS provider and eliminates possible attack surfaces due to network communication [20, 24].

Server Side. Deploying a machine learning model on the client side naively, i.e., providing the trained model to the user as a white box, harms the service provider in the following two perspectives.

- *Intellectual Property:* Training an effective machine learning model is challenging, the MLaaS provider needs to get suitable training data and spend many resources on training and tuning hyper-parameters [36]. All these certainly belong to the intellectual property of the service provider.

- *Pay-per-query:* Most MLaaS providers implement a pay-per-query business model. For example, Google's vision API charges 1.5 USD per 1,000 queries.[4]

To mitigate all these potential damages to the service provider, MLCapsule needs to provide guarantees that it (1) protects intellectual property and (2) enables the pay-per-query business model. In general, we aim for a client-side deployment being indistinguishable from the current server-side deployment.

Protection against Advanced Attacks. Orthogonal to the previous properties, an adversary can perform multiple attacks against MLaaS by solely querying its API (black-box access). Attacks of such kind include model stealing [35, 36], reverse engineering [22], and membership inference [29, 28]. In particular, MLaaS is vulnerable to these attacks, too [35, 29, 22, 36]. We consider mitigating the above threats as requirements of MLCapsule as well.

3. Background

In this section, we recall the properties of Intel's Software Guard Extensions (SGX). Although our prototype is implemented using SGX, any IEE with similar properties implementing the formal model proposed by Fish et al. [10] can be used as a building block of MLCapsule.

3.1. SGX

SGX is a set of commands included in Intel's x86 processor design that allows to create isolated execution environments called enclaves. According to Intel's threat model, enclaves are designed to trustworthily execute programs and handle secrets even under a malicous host system and an untrusted system's memory.

Properties. There are three main properties of Intel SGX:

- *Isolation:* Code and data inside the enclave's protected memory cannot be read or modified by any

[4] https://cloud.google.com/vision/pricing

external process. Enclaves are stored in a hardware guarded memory called Enclave Page Cache (EPC). Untrusted applications can use enclaves as external libraries that are defined by call functions.

- *Sealing:* Data in the unstrusted host system is encrypted and authenticated by a hardware-resident key. From the processors Root Seal key, the enclave's Seal key is derived (or recovered). Other enclaves can not obtain this key.
- *Attestation:* Attestation provides an unforgeable report attesting to code, static/meta data of an enclave, as well as the output of the computation.

Side-channel Attacks. Intel SGX is prone to side-channel attacks. Intel does not claim that SGX defends against physical attacks (e.g., power analysis), but successful attacks have not yet been demonstrated. However, several software attacks have been demonstrated in numerous papers [18, 37, 7]. The attacks usually target flawed implementations. A knowledgeable programmer can write the code in a data-oblivious way, i.e., the software does not have memory access patterns or control flow branches that depend on secret data. In particular, those attacks are not inherent to SGX-like systems, as shown by Costan et al. [9].

Rollback. The formal model described in the next subsection assumes that the state of the hardware is hidden from the users platform. SGX enclaves store encryptions of the enclave's state in the untrusted part of the platform. Those encryptions are protected with a hardware-generated secret key, yet this data is provided to the enclave by an untrusted application. Therefore, SGX does not provide any guarantees about freshness of the state and is vulnerable to rollback attacks. Fortunately, there exist hardware solutions relying on counters [33] and distributed software-based strategies [19] to prevent rollback attacks.

4. Related Work

In this section, we review related works. We first discuss cryptography and ML, focusing on SGX and homomorphic encryption. We then turn to watermarking.

SGX for ML. Ohrimenko et al. [23] investigated oblivious data access patterns for a range of ML algorithms applied in SGX. Tramèr et al. [34], Hynes et al. [16], and Hunt et al. [15] used SGX for ML – however in the context of training convolutional neural network models. Our work instead focuses on the test-time deployment of ML models. To the best of our knowledge, only Gu et al. [13] consider convolutional neural networks and SGX at test time. The core of their work is to split the network as to prevent the user's input to be reconstructed. In contrast, we focus on protecting the model.

Homomorphic Encryption and ML Models. Homomorphic encryption keeps both input and result private from an executing server [3, 6]. In contrast to our approach, homomorphic encryption lacks efficiency, especially for large deep learning models. Moreover, it does not allow for implementing transparent mechanisms to defend attacks on the ML model: detecting an attack contradicts with the guaranteed privacy in this encryption scheme.

Watermarking serves to claim ownership of an ML model. Watermarking as a passive defense mechanism, where A model is trained to yield a particular output on a set of points [1], or to produce a specified output when meaningful content is added to samples [39]. Instead, `MLCapsule` deployed on the client-side is indistinguishable from the server-side deployment. Further, `MLCapsule` allows us to deploy defense mechanisms to actively mitigate advanced attacks against ML models.

5. `MLCapsule`

In this section we introduce `MLCapsule`. We start with an overview of the participants and then introduce the different execution phases. We then argue that and how `MLCapsule` fulfills our requirements. We focus for the rest of the paper on deep networks, as they have recently drawn attention due to their good performance. Additionally, their size makes both implementation and design a challenge. Yet, `MLCapsule` generalizes to other linear models, as well as decision tress and boosting, as these can be expressed as neural networks, too.

Participants. In `MLCapsule`, we distinguish two participants of the system. On the one hand, we have the service provider (SP) that possesses private training data (e.g. images, genomic data, etc.) that it uses to train an ML model. The service provider's objective is to protect the trained weights and biases, as we assume the design of the network to be public. On the other hand, we have users that want to use the pre-trained model as discussed in Section 2.

Approach. We assume that all users have a platform with an isolated execution environment. This is a viable assumption: Intel introduced SGX with the Skylake processor line. The latest Kaby Lake generation of processors also supports SGX. It is reasonable to assume that Intel provides SGX support with every new processor generation and over time every Intel processor will have SGX.

The core idea of MLCapsule is to leverage the properties of the IEE to ensure that the user has to attest that the code is running in isolation before it is provided the secrets by the SP. This step is called the *setup phase*. Afterwards, the client can use the enclave for the classification task. This step is called the *inference phase*. The isolation of the enclave ensures that the user is not able to infer more information about the model than given API access to a server-side model. Figure 1 shows both phases which we now describe in more detail.

Setup phase: The user downloads the code of the enclave provided by the SP and a service application to setup the enclave. This service app can be provided by the SP but also by the user or a trusted third party. After the user's platform attests that an enclave instance is running, the SP provides the enclave with secret data. The enclave's attestation can be part of a larger authentication structure including additional login or SP specific account information allowing only users that paid or are authorized to use the model. Finally, the enclave seals all secrets and the service application stores it for further use. The sealing hides this data form other processes on the user's platform.

Inference phase: To perform classification, the user executes the service app and provides the test data as input. The service app restores the enclave. Since the enclave requires the model parameters, the service app loads the sealed data stored during the setup phase. Before classification, the enclave can also perform an access control procedure based on the sealed data. Due to some limitations (e.g. limited memory of the IEE), the enclave must implement classification layer wise, i.e. the service app provides sealed data for each layer of the network separately. Finally, the enclave outputs the result to the service app.

Requirements. We now briefly discuss how MLCapsule fulfills the requirements.

User Input Privacy. MLCapsule is executed locally, ensuring her privacy and preventing communication with the SP. The user can further inspect the code, in particular checking for any communication with the SP.

Pay-per-query. To enforce the pay-per-query paradigm, the enclave is set up with a threshold. If the threshold is exceeded, and the user exceeds their paid budget, an error is returned. This corresponds to the current, rough grained pay-per query model (batches of 1,000 queries in case of Google's vision API). The key point here is that a given user will only be provisioned once by the SP. Note that during the setup phase the SP will check if the user's platform is running the correct enclave (via attestation) and it also can check the user's account information. A malicious user can create a fresh enclave to reset this counter. However, in such a case the SP will not provision this new enclave with the model's weights and the adversary will not gain any advantage.

Intellectual Property. MLCapsule protects the service provider's intellectual property as the user gains no additional advantage in stealing the intellectual property in comparison to an access to the model through an server-side API.

6. SGX Implementation and Evaluation

In the setup phase, MLCapsule attests the execution of the enclave and decrypts the data sent by the service provider. Both tasks are standard and supported by Intel's crypto library [2]. Thus, in the evaluation we mainly focused on the inference phase and the overhead the IEE introduces.

We used an Intel SGX enabled PC with Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz and Ubuntu 18.04. The implementation uses C++ and is based on Slalom [34], which uses a custom lightweight C++ library for feed-forward networks based on Eigen. Porting well-known ML frameworks, such as TensorFlow, to SGX is not feasible because enclave code cannot access OS-provided features. If not stated otherwise, we used the -O3 compiler optimization and C++11 standard.

In MLCapsule, we wrap standard layers to create MLCapsule layers. Those layers take the standard input of the model layer but the weights are given in sealed form. Inside the enclave, the secret data is unsealed and forwarded to the standard layer function. MLCapsule layers are designed to be executed inside the enclave by providing ECALL's. See Figure 2 for more details. This approach provides means to build MLCapsule secure neural networks in a modular way.

Evaluation.

Since the sealed data is provided from outside the enclave, it has to be first copied to the enclave before unsealing. We measured the execution time of MLCapsule layers as the time it takes to (1) allocate all required local memory, (2) copy the sealed data to the inside of the enclave, (3) unseal the data, (4) perform the actual computation and finally (5) free the allocated memory.

Applications are limited to 90 MB of memory, because there is currently no SGX support for memory swapping. This is visible in our results for a dense layer with weight matrix of size 4096×4096. In this case, we allocate $4 \times 4096 \times 4096 = 64$ MB for the matrix and a temporary array of the same size for the sealed data. Thus, we exceed the 90 MB limit, which leads

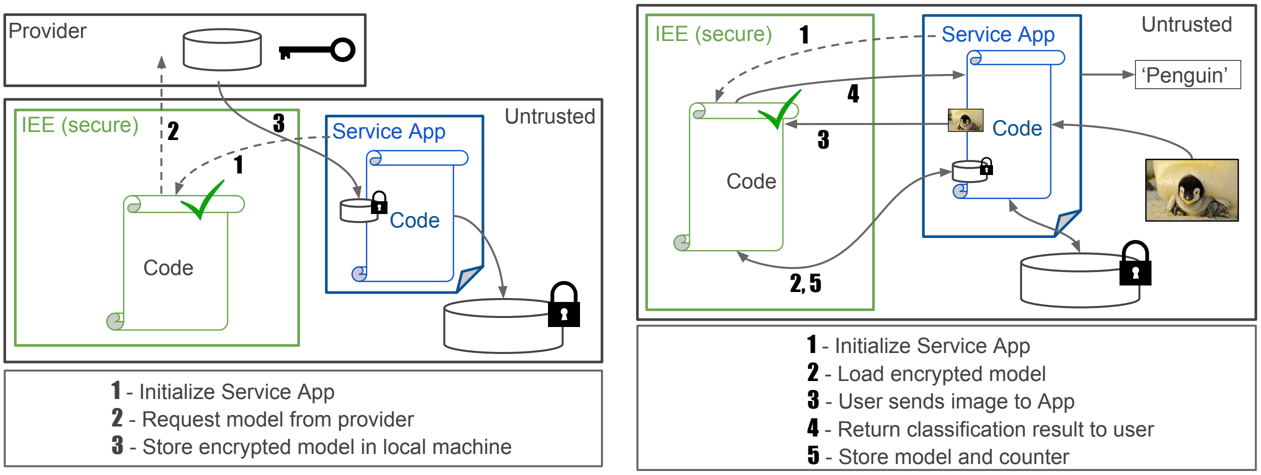
Provider
IEE (secure)
Service App
Untrusted
Code
Code
1
2
3
1 - Initialize Service App
2 - Request model from provider
3 - Store encrypted model in local machine
IEE (secure)
Service App
Untrusted
'Penguin'
Code
Code
1
4
3
2, 5
1 - Initialize Service App
2 - Load encrypted model
3 - User sends image to App
4 - Return classification result to user
5 - Store model and counter

Figure 1: Initialization (left) and inference (right) phase of `MLCapsule`.

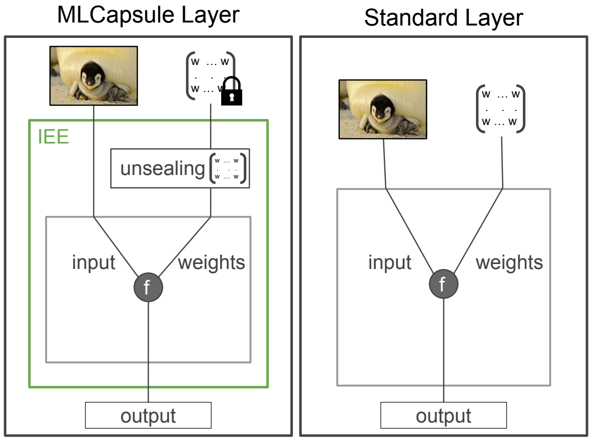

Figure 2: Difference between `MLCapsule` and standard layer.

to a decrease in performance. In particular, the execution of such a layer took 1s and after optimization, the execution time decreased to 0.097ms.

We overcome this problem by encrypting the data in chunks of 2 MB. This way the only large data array allocated inside the `MLCapsule` layer is the memory for the weight matrix. Using 2 MB chunks the `MLCapsule` layers require only around 2 × 2 MB more memory than implementations of standard ML layers. We implemented this optimization only for data that requires more than 48 MB, e.g. in case of a VGG-16 network we used it for the first and second dense layer. Comparisons between `MLCapsule` and standard layers are given in Table 1 and Table 2. From the former, we see that the overhead for convolutional layers averages around 1.2, with a peak to 2.3 in case of inputs of size 512 × 14 × 14. In case of depthwise separable convolution layers, the execution time of `MLCapsule` layers is comparable with a standard layer. In fact, in this case, the difference is almost not noticeable for smaller input sizes. Applying additional activation functions or/and pooling after the convolution layer did not significantly influence the execution time. Dense layers, however, show a larger overhead. For all the kernel dimension the overhead is not larger than 25 times. We also evaluated dense layers without -O3 optimization. The results show that in such a case the overhead of `MLCapsule` is around the factor of 3. We suspect that the compiler is able to more efficiently optimize the source code not using SGX specific library calls. Hence, the increase in performance is due to the optimized compilation.

7. Advanced Defenses

Recently, researchers have proposed multiple attacks against MLaaS: reverse engineering [22], model stealing [35, 36], and membership inference [29, 28]. As mentioned in Section 2, these attacks only require the black-box access (API) to the target ML model, therefore, their attack surface is orthogonal to the one caused by providing the model's white-box access to an adversary. As shown in the literature, real-world MLaaS suffers from these attacks [35, 29, 36, 28].

In this section, we propose two new defense mechanisms against reverse engineering and membership inference. Both mechanisms, together with a proposed defense for model stealing, can be seamlessly integrated into `MLCapsule`.

Table 1: Average dense layer overhead for 100 executions. This comparison includes two ways of compiling the code, i.e. with (marked by *) and without the g++ optimization parameter -O3.

Dimension	MLCapsule *	MLCapsule	Standard *	Standard
256×256	0.401ms	0.234ms	0.164ms	0.020ms
512×512	1.521ms	0.865ms	0.637ms	0.062ms
1024×1024	6.596ms	4.035ms	2.522ms	0.244ms
2048×2048	37.107ms	26.940ms	10.155ms	1.090ms
4096×4096	128.390ms	96.823ms	40.773ms	4.648ms

Table 2: Average convolution and depthwise separable (denoted †) convolution layer overhead for 100 executions and 3 × 3 filters.

Size	MLCapsule	Standard	Factor	MLCapsule †	Standard †	Factor †
64×224×224	80ms	66ms	1.21	41ms	27ms	1.52
128×112×112	70ms	63ms	1.11	18ms	16ms	1.125
256×56×56	55ms	54ms	1.02	9ms	9ms	1.00
512×28×28	61ms	51ms	1.20	7ms	7ms	1.00
512×14×14	30ms	13ms	2.31	2ms	2ms	1.00

7.1. Detecting Reverse Engineering of Neural Network Models

Oh et al. have shown that a wide variety of model specifics can be inferred using black-box querying [22]. This includes network topology, training procedure as well as type of non-linearities and filter sizes, endangering the safety of intellectual property and increasing the attack surface.

Methodology. No defense has been proposed so far for this type of attack. The most effective method proposed by Oh et al. [22] relies on crafted input patterns that are distinct from benign input. Therefore, we propose to train a classifier that detects such malicious inputs. Once a malicious input is recognized, service can be immediately denied which stops the model from leaking further information, without even communicating with the server.

We focus on the kennen-io method by Oh et al. [22], or the strongest attack. We duplicate the test setup on the MNIST dataset.[5] In order to train a classifier to detect such malicious inputs, we generate 4500 crafted input images with the kennen-io method and train a classifier against 4500 benign MNIST images. The classifier has two convolutional layers with 5 × 5 filters, 10 in the first, 20 in the second layer. Each convolutional layer is followed by a 2 × 2 max pooling layer. A fully connected layer with 50 neurons follows before the final softmax output. In addition, the network uses ReLU non-linearties and drop-out for regularization. We use a cross-entropy loss to train the network with the ADAM optimizer.

Evaluation. We compose a test set of additional 500 malicious input samples and 500 benign MNIST samples that are disjoint from the training set. The accuracy of this classifier is 100%: it detects each attack on the first malicious example. Meanwhile, no benign sample leads to a denied service. This is a very effective protection mechanism that seamlessly integrates into our deployment model and only adds 0.832 ms to the overall computation time. While we are able to show very strong performance on this MNIST setup, it has to be noted that the kennen-io method is not designed to be "stealthy" and future improvements of the attack can be conceived that make detection substantially more difficult.

7.2. Detecting Model Stealing

Model stealing attack aims at obtaining a copy from an MLaaS model [35, 25]. The attacker aims to train a substitute on samples rated by the victim model, resulting in a model with similar behavior and/or accuracy, violating the service provider's intellectual property. To show how seamlessly a defense can be integrated into MLCapsule, we integrate Juuti el al.'s [17] defense against model stealing. This defense, Prada, maintains a growing set of user-submitted queries. Whether a query is appended to this growing set depends on the minimum distance to previous queries and a user set threshold. Benign queries lead to a constant growing set, whereas Juuti et al. show that malicious

[5] http://yann.lecun.com/exdb/mnist/

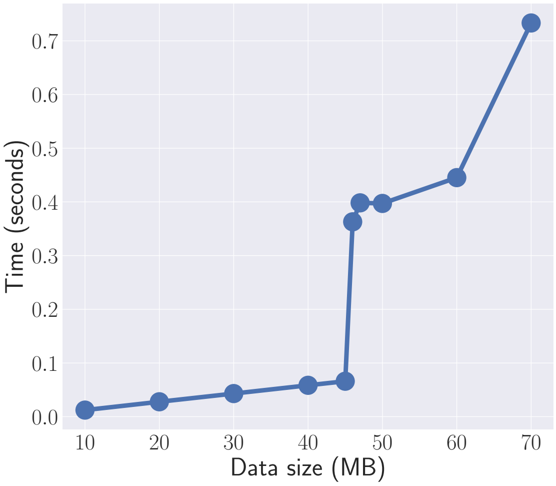

Figure 3: Overhead in seconds to load additional data for a defense mechanism preventing model stealing, namely Prada [17].

samples generally do not increase set size. Hence, an attack can be detected by the difference of the growth of those sets. As detection is independent of the classifier, it can be easily implemented in `MLCapsule`. The resulting computation overhead depends heavily on the user submitted queries [17]. We thus measure the general overhead of first loading the data in the enclave and second of further computations.

Juuti et al. state that the data needed per client is 1-20MB. We plot the overhead with respect to the data size in Figure 3. The overhead for less than 45MB is below 0.1s. Afterwards, there is a sharp increase, as the heap size of the enclave is 90MB: storing more data requires several additional memory operations. An additional time factor is to compute a queries distance to all previous queries in the set. We assume a set size of $3,000$, corresponding to roughly $5,000$ benign queries. Further, a query on the GTSDB dataset[6] is delayed by almost 2s, or a factor of five. For datasets with smaller samples such as CIFAR[7] or MNIST, the delay is around 35ms, where the inference is delayed by a factor of 4 (CIFAR) up to 13.5 (MNIST).

7.3. Membership Inference

Breaching training data is a severe intellectual property leak. Shokri et al. are among the first to demonstrate that a trained model is more confident on a training point. A novel attack by Salem et al. show membership inference attacks can be performed by only relying on the posterior's entropy [28]. We use it as the membership inference attack in our evaluation. However, our defense is general and can be applied to other membership inference attacks as well.

Algorithm 1: Noising mechanism to mitigate membership inference attack.

```
Input: Posterior of a data point $P$, Noise posterior T
Output: Noised posterior $P'$
1: Calculate $\eta(P)$ # the entropy of $P$
2: $\alpha = 1 - \frac{\eta(P)}{\log|P|}$ # the magnitude of the noise
3: $P' = (1 - c\alpha)P + c\alpha T$
4: return $P'$
```

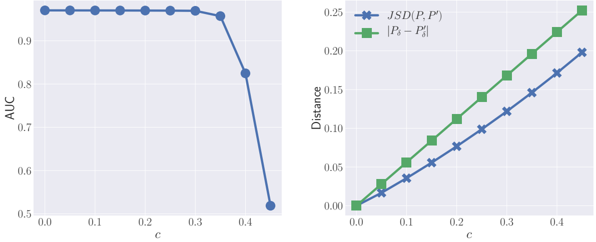

Figure 4: The relation between c, the hyperparameter controlling noise magnitude, and left [higher is better] membership prediction performance and right [lower is better] target model utility. $JSD(P, P')$ denotes the Jensen-Shannon divergence between the original posterior P and the noised P', while $|P_\delta - P'_\delta|$ is the absolute difference between the correct class's posterior (P_δ) and the noised one (P'_δ).

Methodology. We define the posterior of an ML model predicting a certain data point as a vector P, and each class i's posterior is denoted by P_i. The entropy of the posterior is defined as $\eta(P) = -\sum_{P_i \in P} P_i \log P_i$. This entropy allows, using a threshold, to predict membership of a point [28].

The principle of our defense is adding more (less) noise to a posterior with low (high) entropy, and publishing the noised posterior, as formalized in Algorithm 1. We calculate $\eta(P)$ (line 1) and derive from it the magnitude of the noise, i.e., $\alpha = 1 - \frac{\eta P}{\log|P|}$ (line 2). Here, $\frac{\eta P}{\log|P|}$ is the normalized $\eta(P)$ which lies in the range between 0 and 1. Hence, lower entropy implies higher α (i.e., larger noise) which implements the intuition of our defense. However, directly using α generates too much noise to P. Thus, we introduce a hyperparameter, c, to control the magnitude α: c is in the range between 0 and 1, its value is set following cross validation. We thus add noise T to P with $c\alpha$ as the weight (line 3). There are multiple ways to initialize T, in this work, we define it as the class distribution of

[6] http://benchmark.ini.rub.de/
[7] https://www.cs.toronto.edu/~kriz/cifar.html

the training data. Larger $c\alpha$ will cause the final noised P' to be more similar to the prior, which reduces the information provided by the ML model.

Our defense is the first one not modifying the original ML model's structures and hyperparameters, i.e., it is a test-time defense. Previous works either rely on increasing the "temperature" of the softmax function [29], or implementing dropout on the neural network model [28]. These defense mechanisms may affect the model's performance and (implicitly) treat all data points equally, even those that are very unlikely to be in the training data. In contrary, our defense adds different noise based on the entropy of the posterior.

Evaluation. We train VGG-16 on the CIFAR-100 dataset (divided in equal parts for training and testing) and reproduce the experimental setup of previous works [28]. We mirror the attack's performance using the AUC score calculated on the entropy of the target model's posteriors. Figure 4 on the left shows the result of our defense depending on c. Setting c to 0, e.g., no noise, results in 0.97, a high AUC score. Thus an attacker can determine the membership state of a point with high certainty. The AUC score starts dropping when increasing c, as expected. When the value of c approaches 0.5 the AUC score drops to almost 0.5, where the best an attacker can do is randomly guessing the membership status of a point.

We study the utility of our defense, i.e., how added noise affects the performance of the target model. Algorithm 1 shows that our defense mechanism only adjusts the confidence values in a way that the predicted labels stay the same. This means the target model's accuracy, precision, and recall do not change. To perform an in-depth and fair analysis, we report the amount of noise added to the posterior. Concretely, we measure the Jensen-Shannon divergence between the original posterior (P) and the noised one (P'), denoted by $JSD(P, P')$, following previous works [21, 4]. Formally, $JSD(P, P')$ is defined as:

$$JSD(P, P') = \sum_{P_i \in P} P_i \log \frac{P_i}{M_i} + P'_i \log \frac{P'_i}{M_i}$$

where $M_i = \frac{P_i + P'_i}{2}$. Moreover, we measure the absolute difference between the correct class's original posterior (P_δ) and its noised version (P'_δ), i.e., $|P_\delta - P'_\delta|$, this is also referred to as the expected estimation error [30, 5, 40]. In Figure 4 on the right, we see that both $JSD(P, P')$ and $|P_\delta - P'_\delta|$ increase monotonically with the amount of noise being added (reflected by c). However, when c is approaching 0.5, i.e., our defense mechanism can mitigate the membership inference risk completely, $JSD(P, P')$ and $|P_\delta - P'_\delta|$ are still both below 0.25, indicating that our defense mechanism preserve the target model's utility to a large extent.

The overhead of this defense is only 0.026ms to the whole computation. This indicates our defense can be very well integrated into `MLCapsule`.

8. Conclusion

We have presented a novel deployment mechanism for ML models. It provably provides the same level of security of the model and control over the model as conventional server-side MLaaS execution, but at the same time it provides perfect privacy of the user data as it never leaves the client. In addition, we show the extensibility of our approach and how it facilities a range of features from pay-per-view monetization to advanced model protection mechanisms – including the very latest work on model stealing and reverse engineering.

We believe that this is an important step towards the overall vision of data privacy in machine learning [26] as well as secure ML and AI [32]. Beyond the presented work and direct implications on data privacy and model security – this line of research implements another line of defenses that in the future can help to tackle several problems in security related issues of ML that the community has been struggling to make sustainable progress. For instance, a range of attacks from membership inference, reverse engineering to adversarial perturbations rely on repeated queries to a model. Our deployment mechanism provides a scenario that is compatible with a wide spread of ML models with the ability of controlling or mediating access to the model directly (securing the model) or indirectly (advanced protection against inference attacks).

Acknowledgments

This work is partially funded by the Helmholtz Association within the project "Trustworthy Federated Data Analytics (TFDA)" (ZT-I-OO1 4) and partially by the German Federal Ministry of Education and Research (BMBF) through funding for CISPA and the CISPA-Stanford Center for Cybersecurity (FKZ: 16KIS0762).

References

[1] Adi, Y., Baum, C., Cisse, M., Pinkas, B., Keshet, J.: Turning Your Weakness Into a Strength: Watermarking Deep Neural Networks by Backdooring. In: Proceedings of the 2018 USENIX Security Symposium (Security). USENIX (2018) 3

[2] Aumasson, J., Merino, L.: SGX Secure Enclaves in Practice: Security and Crypto Review. In: Pro-

ceedings of the 2016 Black Hat (Black Hat) (2016) 4

[3] Avidan, S., Butman, M.: Blind Vision. In: Proceedings of the 2006 European Conference on Computer Vision (ECCV). pp. 1–13. Springer (2006) 3

[4] Backes, M., Humbert, M., Pang, J., Zhang, Y.: walk2friends: Inferring Social Links from Mobility Profiles. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 1943–1957. ACM (2017) 8

[5] Berrang, P., Humbert, M., Zhang, Y., Lehmann, I., Eils, R., Backes, M.: Dissecting Privacy Risks in Biomedical Data. In: Proceedings of the 2018 IEEE European Symposium on Security and Privacy (Euro S&P). IEEE (2018) 8

[6] Bost, R., Popa, R.A., Tu, S., Goldwasser, S.: Machine Learning Classification over Encrypted Data. In: Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS). Internet Society (2015) 1, 3

[7] Brasser, F., Muller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., Sadeghi, A.R.: Software Grand Exposure: SGX Cache Attacks Are Practical. In: Proceedings of the USENIX Workshop on Offensive Technologies (WOOT) (2017) 3

[8] Bulck, J.V., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R.: Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: Proceedings of the 2018 USENIX Security Symposium (USENIX Security). pp. 991–1008. USENIX (2018) 2

[9] Costan, V., Lebedev, I., Devadas, S.: Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In: Proceedings of the 2016 USENIX Security Symposium (Security). pp. 857–874. USENIX (2016) 3

[10] Fisch, B., Vinayagamurthy, D., Boneh, D., Gorbunov, S.: Iron: Functional Encryption using Intel SGX. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 765–782. ACM (2017) 2

[11] Fredrikson, M., Jha, S., Ristenpart, T.: Model Inversion Attacks that Exploit Confidence Information and Basic Countermeasures. In: CCS. pp. 1322–1333. ACM (2015) 1

[12] Fredrikson, M., Lantz, E., Jha, S., Lin, S., Page, D., Ristenpart, T.: Privacy in Pharmacogenetics: An End-to-End Case Study of Personalized Warfarin Dosing. In: USENIX Security Symposium. pp. 17–32 (2014) 1

[13] Gu, Z., Huang, H., Zhang, J., Su, D., Lamba, A., Pendarakis, D., Molloy, I.: Securing Input Data of Deep Learning Inference Systems via Partitioned Enclave Execution. CoRR abs/1807.00969 (2018) 3

[14] Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H.: MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. CoRR abs/1704.04681 (2017) 2

[15] Hunt, T., Song, C., Shokri, R., Shmatikov, V., Witchel, E.: Chiron: Privacy-preserving Machine Learning as a Service. CoRR abs/1803.05961 (2018) 3

[16] Hynes, N., Cheng, R., Song, D.: Efficient Deep Learning on Multi-Source Private Data. CoRR abs/1807.06689 (2018) 3

[17] Juuti, M., Szyller, S., Marchal, S., Asokan, N.: Prada: protecting against dnn model stealing attacks. In: Euro S&P. pp. 512–527. IEEE (2019) 2, 6, 7

[18] Lee, S., Shih, M.W., Gera, P., Kim, T., Kim, H., Peinado, M.: Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In: USENIX Security Symposium. pp. 557–574. USENIX (2017) 3

[19] Matetic, S., Ahmed, M., Kostiainen, K., Dhar, A., Sommer, D., Gervais, A., Juels, A., Capkun, S.: ROTE: Rollback Protection for Trusted Execution. In: USENIX Security Symposium. pp. 1289–1306. USENIX (2017) 3

[20] Melis, L., Song, C., Cristofaro, E.D., Shmatikov, V.: Inference Attacks Against Collaborative Learning. CoRR abs/1805.04049 (2018) 1, 2

[21] Mittal, P., Papamanthou, C., Song, D.: Preserving Link Privacy in Social Network Based Systems. In: Proceedings of the 2013 Network and Distributed System Security Symposium (NDSS). Internet Society (2013) 8

[22] Oh, S.J., Augustin, M., Schiele, B., Fritz, M.: Towards Reverse-Engineering Black-Box Neural Networks. In: Proceedings of the 2018 International

Conference on Learning Representations (ICLR) (2018) 1, 2, 5, 6

[23] Ohrimenko, O., Schuster, F., Fournet, C., Mehta, A., Nowozin, S., Vaswani, K., Costa, M.: Oblivious Multi-Party Machine Learning on Trusted Processors. In: USENIX Security Symposium. pp. 619–636. USENIX (2016) 3

[24] Orekondy, T., Oh, S.J., Schiele, B., Fritz, M.: Understanding and Controlling User Linkability in Decentralized Learning. CoRR abs/1805.05838 (2018) 1, 2

[25] Papernot, N., McDaniel, P., Goodfellow, I.: Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples. CoRR abs/1605.07277 (2016) 6

[26] Papernot, N., McDaniel, P., Sinha, A., Wellman, M.: SoK: Towards the Science of Security and Privacy in Machine Learning. In: Proceedings of the 2018 IEEE European Symposium on Security and Privacy (Euro S&P). IEEE (2018) 1, 8

[27] Salem, A., Zhang, Y., Humbert, M., Berrang, P., Fritz, M., Backes, M.: ML-Leaks: Model and Data Independent Membership Inference Attacks and Defenses on Machine Learning Models. In: Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS). Internet Society (2019) 2

[28] Salem, A., Zhang, Y., Humbert, M., Fritz, M., Backes, M.: ML-Leaks: Model and Data Independent Membership Inference Attacks and Defenses on Machine Learning Models. In: NDSS (2019) 1, 2, 5, 7, 8

[29] Shokri, R., Stronati, M., Song, C., Shmatikov, V.: Membership Inference Attacks Against Machine Learning Models. In: Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P). pp. 3–18. IEEE (2017) 1, 2, 5, 8

[30] Shokri, R., Theodorakopoulos, G., Boudec, J.Y.L., Hubaux, J.P.: Quantifying Location Privacy. In: Proceedings of the 2011 IEEE Symposium on Security and Privacy (S&P). pp. 247–262. IEEE (2011) 8

[31] Simonyan, K., Zisserman, A.: Very Deep Convolutional Networks for Large-Scale Image Recognition. CoRR abs/1409.1556 (2014) 2

[32] Stoica, I., Song, D., Popa, R.A., Patterson, D., Mahoney, M.W., Katz, R., Joseph, A.D., Jordan, M., Hellerstein, J.M., Gonzalez, J., Goldberg, K., Ghodsi, A., Culler, D., Abbeel, P.: A Berkeley View of Systems Challenges for AI. CoRR abs/1712.05855 (2017) 8

[33] Strackx, R., Piessens, F.: Ariadne: A Minimal Approach to State Continuity. In: USENIX Security Symposium. pp. 875–892. USENIX (2016) 3

[34] Tramer, F., Boneh, D.: Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware (2019) 3, 4

[35] Tramér, F., Zhang, F., Juels, A., Reiter, M.K., Ristenpart, T.: Stealing Machine Learning Models via Prediction APIs. In: Proceedings of the 2016 USENIX Security Symposium (Security). pp. 601–618. USENIX (2016) 1, 2, 5, 6

[36] Wang, B., Gong, N.Z.: Stealing Hyperparameters in Machine Learning. In: IEEE S&P. IEEE (2018) 2, 5

[37] Wang, W., Chen, G., Pan, X., Zhang, Y., Wang, X., Bindschaedler, V., Tang, H., Gunter, C.A.: Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In: CCS. pp. 2421–2434. ACM (2017) 3

[38] Yeom, S., Giacomelli, I., Fredrikson, M., Jha, S.: Privacy Risk in Machine Learning: Analyzing the Connection to Overfitting. In: Proceedings of the 2018 IEEE Computer Security Foundations Symposium (CSF). IEEE (2018) 1

[39] Zhang, J., Gu, Z., Jang, J., Wu, H., Stoecklin, M.P., Huang, H., Molloy, I.: Protecting Intellectual Property of Deep Neural Networks with Watermarking. In: Proceedings of the 2018 ACM Asia Conference on Computer and Communications Security (ASIACCS). pp. 159–172. ACM (2018) 3

[40] Zhang, Y., Humbert, M., Rahman, T., Li, C.T., Pang, J., Backes, M.: Tagvisor: A Privacy Advisor for Sharing Hashtags. In: Proceedings of the 2018 Web Conference (WWW). pp. 287–296. ACM (2018) 8