MLCapsule: Guarded Offline Deployment of Machine Learning as a Service

Lucjan Hanzlik*, Yang Zhang*, Kathrin Grosse*, Ahmed Salem*, Max Augustin†,
Michael Backes‡, Mario Fritz‡
*CISPA, Saarland Informatics Campus
†Max Planck Institute for Informatics, Saarland Informatics Campus
‡CISPA Helmholtz Center i.G., Saarland Informatics Campus

***Abstract*—With the widespread use of machine learning (ML) techniques, ML as a service has become increasingly popular. In this setting, an ML model resides on a server and users can query the model with their data via 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.**

In this paper, we propose MLCapsule, a guarded offline deployment of machine learning as a service. MLCapsule executes the machine learning model locally on the user's client and therefore the data never leaves the client. Meanwhile, MLCapsule offers the service provider the same level of control and security of its model as the commonly used server-side execution. In addition, MLCapsule is applicable to offline applications that require local 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.

I. 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] have deployed 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. Such setups range from image analysis over translation to applications in the business domain.

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 [7] come at substantial computational overhead especially for state-of-the-art deep learning models containing millions of parameters. Moreover, MLaaS requires data transfer over the network which constitutes to high volume communication and provides new attack surface [25], [29]. This motivates us to come up with a client-side solution such that perfect data privacy and offline computation can be achieved.

[1] https://cloud.google.com/
[2] https://cloud.google.com/vision/
[3] https://azure.microsoft.com/en-us/services/machine-learning-studio/

As a consequence, this (seemingly) comes with a loss of control of the service provider, as the ML model has to be transfered and executed on the client's machine. This raises concerns about revealing details of the model or granting unrestricted/unrevocable access to the user. The former damages the intellectual property of the service provider, while the latter breaks the commonly enforced pay-per-query business model. Moreover, there is a broad range of attack vectors on machine learning models that raise severe concerns about security and privacy [31]. A series of recent papers have shown different attacks on MLaaS that can lead to reverse engineering [41], [27] and training data leakage [14], [13], [35], [44], [34]. Many of these threats are facilitated by repeated probing of the ML model that the service provider might want to protect against. 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.

A. Our Contributions

In this paper, 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 while model and computation remain secret. With `MLCapsule`, the service provider controls its ML model which allows for intellectual property protection and business model maintenance. Meanwhile, the user gains perfect data privacy and offline execution, as the data never leaves the client and the protocol is transparent

We assume that the client's platform has access to an Isolated Execution Environment (IEE). `MLCapsule` uses this to provide a secure enclave to run an ML model, or more specifically, classification inference. Moreover, since IEE provides means to prove execution of code, the service provider is assured that the secrets that it sends in encrypted form can only be decrypted by the enclave. This also keeps this data secure from other processes running on the client's platform.

To support security arguments about `MLCapsule`, we propose the first formal model for reasoning about the security of local ML model deployment. The leading idea of our model is a property called *ML model secrecy*. The simulator defined in this property ensures that the client can simulate `MLCapsule` using only a server-side API. In consequence, this means that

if the client is able to perform an attack against MLCapsule, the same attack can be performed on the server-side API.

We also contribute by implementing our solution on a platform with Intel SGX. 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, e.g. we implemented and evaluated the VGG-16 [37] and MobileNet [17] neural networks. In addition, we provide an evaluation of convolution and dense layer and compare the execution time inside the IEE to a standard implementation.

The isolated code execution on the client's platform renders MLCapsule ability to integrate advanced defense mechanism for attacks against machine learning models. For demonstration, we propose two defense mechanisms against reverse engineering [27] and membership inference [35], [34], and utilize a recent proposed defense [21] for model stealing attack [41]. We show that these mechanisms can be seamlessly incorporated into MLCapsule, with a negligible computation overhead, which further demonstrates the efficacy of our system.

B. Organization

The rest of the paper is organized as the following. In Section II, we discuss the requirements of MLCapsule. Section III provides the necessary technical background and Section IV summarizes the related work in the field. In Section V, we present MLCapsule in detail and formally prove its security in Section VI. Section VII discusses the implementation and evaluation of MLCapsule. We show how to incorporate advanced defense mechanisms in Section VIII. Section IX provides a discussion, and the paper is concluded in Section X.

II. Requirements and Threat Model

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

A. Guaranteed Model Secrecy and Data Privacy

User Side. MLCapsule deploys MLaaS locally. This provides strong privacy guarantee to a user, as her data never leaves her devices. Meanwhile, executing machine learning prediction locally avoids the Internet communication between the user and the MLaaS provider. Therefore, the user can use the ML model offline. Possible attack surfaces due to network communication [25], [29] are automatically eliminated.

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 a large amount of efforts for training the model and tuning various hyperparameters [42]. All these certainly belong to the intellectual property of the service provider. Providing the trained model to the client as a white box will result in the service provider completely revealing its intellectual property.
- *Pay-per-query:* Almost all MLaaS providers implement the pay-per-query business model. For instance, Google's vision API charges 1.5 USD per 1,000 queries.[4] Deploying a machine learning model at the client side naturally grants a user unlimited number of queries, which naturally breaks the pay-per-query business model.

To mitigate all these potential damages to the service provider, MLCapsule needs to provide the following guarantees:

- Protecting intellectual property
- Enable the pay-per-query business model

In a more general way, we aim for a client-side deployment being indistinguishable from the current server-side deployment.

B. Protection against Advanced Attacks

Several recent works show that an adversary can perform multiple attacks against MLaaS by solely querying its API (black-box access). Attacks of such kind include model stealing [41], [42], reverse engineering [27], and membership inference [35], [34]. These attacks are however orthogonal to the damages discussed in Section II-A, as they only need black-box access to the ML model instead of white-box access. More importantly, researchers have shown that the current MLaaS cannot prevent against these attacks neither [41], [35], [27], [42].

We consider mitigating these above threats as the requirements of MLCapsule as well. Therefore, we show defense mechanisms against these advanced attacks can be seamlessly integrated into MLCapsule.

III. Background

In this section, we focus on the properties of Intel's Software Guard Extensions (SGX), the major building block of MLCapsule, and recall a formal definition of Attested Execution proposed by Fisch et al. [12]. We also formalize a public key encryption scheme, which we will use for the concrete instantiation of our system.

A. 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 if the host system is malicious and the system's memory is untrusted.

Properties. There are three main properties of Intel SGX:

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

- *Isolation:* Code and data inside the enclave's protected memory cannot be read or modified by any external process. Enclaves are stored in a hardware guarded memory called Enclave Page Cache (EPC), which is currently limited to 128 MB with only 90 MB for the application. Untrusted applications can execute code inside the enclave using entry points called Enclave Interface Functions `ECALL`s, i.e. untrusted applications can use enclaves as external libraries that are defined by those call functions.
- *Sealing:* Data stored in the host system is encrypted and authenticated using a hardware-resident key. Every SGX-enabled processor has a special key called Root Seal Key that can be used to derive a so called Seal Key that is specific to the enclave identity. This key can then be used to encrypt/decrypt data, which can be stored in untrusted memory. One important feature is that the same enclave can always recover the Seal Key if instantiated on the same platform, however it cannot be derived by other enclaves.
- *Attestation:* Attestation provides an unforgeable report attesting to code, static data and meta data of an enclave, as well as the output of the performed computation. Attestation can be local and remote. In the first case, one enclave can derive a shared Report Key using the Root Seal Key and create a report consisting of a Message Authentication Code (MAC) over the input data. This report can be verified by a different enclave inside the same platform, since it can also derive the shared Report Key. In case of remote attestation, the actual report for the third party is generated by a so called Quoting Enclave that uses an anonymous group signature scheme (Intel Enhanced Privacy ID [9]) to sign the data.

Side-channel Attacks. Due to its design, 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. On the other hand, several software attacks have been demonstrated in numerous papers [22], [43], [8]. Those kinds of attacks usually target flawed implementations and 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. [11].

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 using 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 [39] and distributed software-based strategies [24] that can be used to prevent rollback attacks.

B. Definition for SGX-like Hardware

There are many papers that discuss hardware security models in a formalized way. The general consensus is that those abstractions are useful to formally argue about the security of the system.

Barbosa et al. [5] define a generalized ideal interface to represent SGX-like systems that perform attested computation. A similar model was proposed by Fisch et al. [12] but was designed specifically to abstract Intel's SGX and support local and remote attestation. Pass, Shi, and Tramèr [32] proposed an abstraction of attested execution in the universal composability (UC) model. In this paper we will focus on the formal hardware model by Fisch et al. [12]. We decided to use this particular model because it was specifically defined to abstract the features that are supported by SGX, which is the hardware also used by our implementation. However, since we will only use remote attestation in our instantion, we omit the local attestation part and refer the reader to the original paper for a full definition.

Informally, this ideal functionality allows a registered party to install a program inside an enclave, which can then be resumed on any given input. An instance of this enclave possesses internal memory that is hidden from the registering party. However, the main property of attested execution is that the enclave creates an attestation of execution. This attestation provides a proof for third parties that the program was executed on a given input yielding a particular output.

Formal Definition. We define a secure hardware as follows.

Definition 1. A secure hardware functionality HW for a class of probabilistic polynomial time programs Q consists of the following interface: $\mathsf{HW.Setup}$, $\mathsf{HW.Load}$, $\mathsf{HW.Run}$, $\mathsf{HW.RunQuote}_{\mathsf{sk}_{\mathsf{quote}}}$, $\mathsf{HW.QuoteVerify}$. HW has also an internal state state that consists of a variable $HW.\mathsf{sk}_{\mathsf{quote}}$ and a table T consisting of enclave state tuples indexed by enclave handles. The variable $HW.\mathsf{sk}_{\mathsf{quote}}$ will be used to store signing keys and table T will be used to manage the state of the loaded enclave.

- $\mathsf{HW.Setup}(1^n)$: given input security parameter 1^n, it generates the secret key $\mathsf{sk}_{\mathsf{quote}}$ and stores it in $HW.\mathsf{sk}_{\mathsf{quote}}$. It also generates and outputs public parameters params.
- $\mathsf{HW.Load}(\mathsf{params}, Q)$: given input global parameters params and program Q it first creates an enclave, loads Q, and then generates a handle hdlthat will be used to identify the enclave running Q. Finally, it sets $\mathsf{T}[\mathsf{hdl}] = \emptyset$ and outputs hdl.
- $\mathsf{HW.Run}(\mathsf{hdl}, \mathsf{in})$: it runs Q at state $\mathsf{T}[\mathsf{hdl}]$ on input in and records the output out. It sets $\mathsf{T}[\mathsf{hdl}]$ to be the updated state of Q and outputs out.
- $\mathsf{HW.RunQuote}_{\mathsf{sk}_{\mathsf{quote}}}(\mathsf{hdl}, \mathsf{in})$: executes a program in an enclave similar to $\mathsf{HW.Run}$ but additionally outputs an attestation that can be publicly verified. The algorithm first executes Q on in to get out, and updates $\mathsf{T}[\mathsf{hdl}]$ accordingly. Finally, it outputs the tuple $\mathsf{quote} = (\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \sigma)$, where $\mathsf{md}_{\mathsf{hdl}}$ is the metadata

associated with the enclave, tag_Q is a program tag for Q and σ is a signature on $(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out})$.
- $\mathsf{HW.QuoteVerify}(\mathsf{params}, \mathsf{quote})$: given the input global parameters params and quote this algorithm outputs 1 it the signature verification of σ succeeds. It outputs 0 otherwise.

Correctness. A HW scheme is correct if the following holds. For all aux, all programs Q, all in in the input domain of Q and all handles hdl we have:

- if there exist random coins r (sampled in run time and used by Q) such that $\mathsf{out} = Q(\mathsf{in})$, and
- $\Pr[\mathsf{HW.QuoteVerify}(\mathsf{params}, \mathsf{quote}) = 0] = \mathsf{negl}(\lambda)$, where $\mathsf{quote} = \mathsf{HW.RunQuote}_{\mathsf{sk}_{\mathsf{quote}}}(\mathsf{hdl}, \mathsf{in})$.

Remote attestation unforgeability is modeled by a game between a challenger C and an adversary $\mathcal{A}$.

1) $\mathcal{A}$ provides an aux.
2) C runs $\mathsf{HW.Setup}(1^n, \mathsf{aux})$ to obtain public parameters params, secret key $\mathsf{sk}_{\mathsf{quote}}$ and an initialization string state. It gives params to $\mathcal{A}$, and keeps $\mathsf{sk}_{\mathsf{quote}}$ and state secret in the secure hardware.
3) C initialized a list $\mathsf{query} = \{\}$.
4) $\mathcal{A}$ can run HW.Load on any input (params, Q) of its choice and get back hdl.
5) $\mathcal{A}$ can also run $\mathsf{HW.RunQuote}_{\mathsf{sk}_{\mathsf{quote}}}$ on input $(\mathsf{hdl}, \mathsf{in})$ of its choice and get $\mathsf{quote} = (\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \sigma)$, where the challenger puts the tuple $(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out})$ into query.
6) $\mathcal{A}$ finally outputs $\mathsf{quote}^* = (\mathsf{md}^*_{\mathsf{hdl}}, \mathsf{tag}^*_Q, \mathsf{in}^*, \mathsf{out}^*, \sigma^*)$.

We define that adversary $\mathcal{A}$ wins the above game if $\mathsf{HW.QuoteVerify}(\mathsf{params}, \mathsf{quote}^*) = 1$ and $(\mathsf{md}^*_{\mathsf{hdl}}, \mathsf{tag}^*_Q, \mathsf{in}^*, \mathsf{out}^*) \notin \mathsf{query}$. The hardware model is remote attestation unforgeable if no adversary can win this game with non-negligible probability.

C. Public Key Encryption

Definition 2. Given a plaintext space $\mathcal{M}$ we define a public key encryption scheme as a tuple of probabilistic polynomial time algorithms:

- $\mathsf{KeyGen}(1^n)$: on input security parameters, this algorithm outputs the secret key sk and public key pk.
- $\mathsf{Enc}(\mathsf{pk}, m)$: on input public key pk and message $m \in \mathcal{M}$, this algorithm outputs a ciphertext c.
- $\mathsf{Dec}(\mathsf{sk}, c)$: on input secret key sk and ciphertext c, this algorithm outputs message m.

Correctness. A public key encryption scheme is correct if for all security parameters 1^n, all messages $m \in \mathcal{M}$ and all key-pairs $(\mathsf{sk}, \mathsf{pk}) = \mathsf{KeyGen}(1^n)$ we have $\mathsf{Dec}(\mathsf{sk}, \mathsf{Enc}(\mathsf{pk}, m)) = m$.

Ciphertext Indistinguishability Against Chosen Plaintext Attacks. We say that the public key encryption scheme is indistinguishable against chosen plaintext attacks if there exists no adversary for which $|\Pr[\mathsf{Exp}_{\mathsf{Enc},\mathcal{A}} = 1] - \frac{1}{2}|$ is non-negligible, where experiment $\mathsf{Exp}_{\mathsf{Enc},\mathcal{A}}$ is defined in Figure 1.

$\mathsf{Exp}_{\mathsf{Enc},\mathcal{A}}(\lambda)$

$(\mathsf{sk}, \mathsf{pk}) = \mathsf{KeyGen}(1^n)$

$(m_0, m_1) \leftarrow \mathcal{A}(\mathsf{pk})$

$b \leftarrow_{\$} \{0, 1\}$

$c \leftarrow \mathsf{Enc}(\mathsf{pk}, m_b) \hat{b} \leftarrow \mathcal{A}(\mathsf{pk}, c)$

else return $\hat{b} = b$

Fig. 1: Indistinguishability against chosen plaintext attacks - security experiment.

IV. Related Work

In this section, we review related works in the literature. We start by discussing cryptography and ML, concentrating on SGX and homomorphic encryption. We then turn to mechanisms using watermarking to passively protect ML models. In the end, we describe several relevant attacks against ML models including model stealing, reverse engineering, and membership inference.

SGX for ML. Ohrimenko et al. [28] investigated oblivious data access patterns for a range of ML algorithms applied in SGX. Their work deals with neural networks, support vector machines and decision trees. Additionally, Tramèr et al. [40], Hynes et al. [20], and Hunt et al. [19] 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. [15] consider convolutional neural networks and SGX at test time. They propose to split the network, where the first layers are executed in an SGX enclave, and the latter part outside the enclave. 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 has been used to keep both input and result private from an executing server [3], [7]. In contrast to our approach, however, homomorphic encryption lacks efficiency, especially for deep learning models containing millions of parameters. Moreover, it does not allows for implementing transparent mechanisms to defend attacks on the ML model: the user cannot be sure which code is executed. Furthermore, detecting an attack might require further steps to prevent harm, which contradict with the guaranteed privacy of inputs and outputs in this encryption scheme.

Watermarking ML Models. Recently, watermarking has been introduced to claim ownership of a trained ML model. Adi et al. [1] propose to watermark a model by training it to yield a particular output for a number of points, thereby enabling identification. Zhang et al. [45], in contrast, add meaningful content to samples used as watermarks. Watermarking as a passive defense mechanism is opposed to our work in two perspectives. First, `MLCapsule` deployed on the client-side is indistinguishable from the server-side deployment. Second,

MLCapsule allows us to deploy defense mechanisms to actively mitigate advanced attacks against ML models.

Model Stealing and Reverse Engineering. Model stealing has been introduced by Tramèr et al. [41]. Before that, however, Papernot et al. [30] worked on training substitute models. The attacker's goal in both cases is to duplicate a model that is accessible only via an API. Recently, a defense to these attacks called Prada has been proposed by Juuti et al. [21]. We will evaluate Prada's feasibility in MLCapsule.

A different line of work aims to infer specific details of the model such as architecture, training method and type of non-linearities, rather than its behavior. Such attacks are called reverse engineering and were proposed by Oh et al. [27]. We propose a defense against this attack and also show it can be integrated in MLCapsule with little overhead.

Membership Inference. Membership inference attack against machine learning models have been proposed recently. In this setting, the attacker aims to decide whether a data point is in the training set of the target machine learning model. Shokri et al. are among the first to perform effective membership inference against machine learning models [35]. Follow this work, several other attacks have been proposed [44], [16], [34]. In this paper, we propose a defense mechanism to mitigate the membership privacy risks which can be easily implemented in MLCapsule.

V. MLCapsule

In existing MLaaS system, the service provider gives the user access to an API, which she can then use for training and classification. This is a classic client-server scenario, where the service is on the server's side and the user is transferring the data. We focus on the scenario, where the service provider equips the user with an already trained model and classification is done on the client's machine. In this scenario, the trained model is part of the service. This introduces problems, as previously discussed. In this section we will introduce MLCapsule, our approach to tackle those problems. We then argue that and how MLCapsule fulfills the requirements.

A. Overview

We start with an overview of the participants and then introduce MLCapsule with its different execution phases.

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 a ML model. On the other hand, we have users that want to use the pre-trained model as we previously discussed in Section II.

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 make both implementation and design a challenge. Yet, MLCapsule generalizes to other linear models, as these can be expressed as a single layer of a neural network.

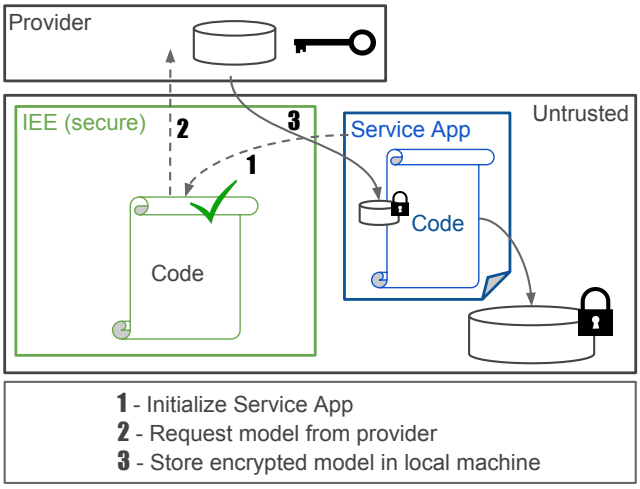


Fig. 2: Our scheme with all steps of initialization.

We consider the design of the applied network to be publicly known. The service provider's objective is to protect the trained weights and biases of all layers.

Approach. We now describe MLCapsule, which fulfills the requirements from Section II. To start, we assume that all users have a platform with an isolated execution environment. Note that this is a viable assumption: Intel introduced SGX with the Skylake processor line. Additionally, the latest Kaby Lake generation of processors supports SGX. It is further reasonable to assume that Intel will provide support for SGX with every new processor generation and over time every PC with an 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* and is depicted in Figure 2. Once the setup is done, the client can use the enclave for the classification task. This step is called the *inference phase* and is depicted in Figure 3. 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.

We now describe MLCapsule in more details.

Setup phase: The user and the service provider interact to setup a working enclave on the users platform. The user has to download the enclave's code and the code of a service application that will setup the enclave on the user's side. The enclave's code is provided by the SP, where the code of the service app can be provided by the SP but it can also be implemented by the user or any third party that she trusts. Note that the user can freely inspect the enclave's code and check what data is exchanged with the SP. This inspection ensures that the user's input to the ML model will be hidden from the SP, as the classification is run locally and the input data never leaves the user's platform.

After the user's platform attests that an enclave instance is running, the SP provides the enclave with secret data

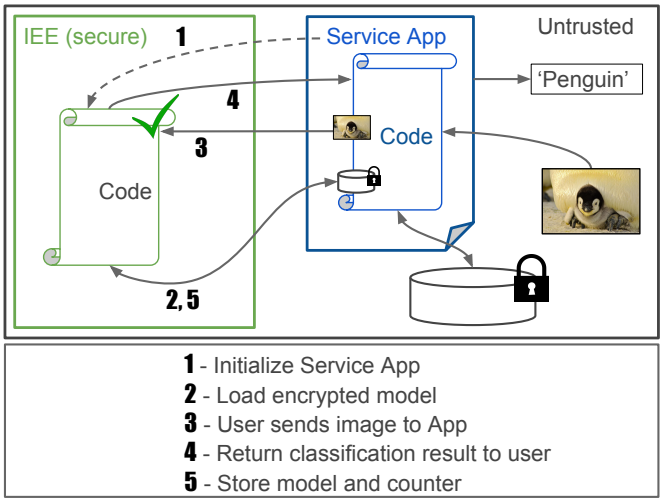


Fig. 3: Our scheme with all steps of offline classification.

composed of amongst others the weights of the network. Finally, the enclave seals all secrets provisioned by the service provider and the service application stores it for further use. The sealing hides this data for other processes on the user's platform. With the end of the setup phase of `MLCapsule`, no further communication with the SP is needed.

Inference phase: To perform classification, the user executes the service app and provides the test data as input. The service app restores the enclave, which can now be used to perform classification. Since the enclave requires the model parameters, the service app loads the sealed data stored during the setup phase. Note that before classification, the enclave can also perform an access control procedure that is based on the user's input data (available to the enclave in plaintext) and the current state of the enclave. Due to some limitations (e.g. limited memory of the IEE), the enclave can be implemented in a way that classification is performed layer wise, i.e. the service app provides sealed data for each layer of the network separately. In the end, the enclave outputs the result to the service app and might as well update its state, which is stored inside the sealed data. This process of classification is depicted in Figure 3.

B. Discussion on Requirements

In this subsection we discuss how `MLCapsule` fulfills the requirements stated in Section II.

User Input Privacy. `MLCapsule` is executed locally by the user. Moreover, the user is allowed to inspect the code executed in the secure hardware. This means that she can check for any communication command with the SP and stop execution of the program. Moreover, off-line local execution ensures that the user's input data is private because there is no communication required with the SP. We conclude that `MLCapsule` perfectly protects the user input.

Pay-per-query. To enforce the pay-per-query paradigm, the enclave will be set up during provision with a threshold. Moreover, the enclave will store a counter that is increased with every classification performed. Before performing classification, it is checked whether the counter exceeds this threshold. In case it does, the enclave returns an error instead of the classification. Otherwise, the enclave works normally.

This solution ensures that the user cannot exceed the threshold, which means that she can only query the model for the number of times she paid. Unfortunately, it does not allow for a fine-grained pay-per-query, where the user can freely chose if she wants more queries at a given time. On the other hand, this is also the model currently used by server-side MLaaS, where a user pays for a fixed number of queries (e.g. 1000 in case of Google's vision API).

Intellectual Property. `MLCapsule` protects the service provider's intellectual property by ensuring that the isolation provided by the hardware simulates in a way the black-box access in the standard server-side model of MLaaS. In other words, the user gains no additional advantage in stealing the intellectual property in comparison to an access to the model through an server-side API.

VI. Security Analysis

In this section, we introduce a formal model for `MLCapsule` and show a concrete instantiation using the abstracted hardware model by Fisch et al. [12] and a standard public key encryption scheme, which we recalled in Section III. The goal is to prove that our construction possesses a property that we call ML model secrecy. We define it as a game that is played between the challenger and the adversary. The challenger chooses a bit which indicates whether the adversary is interacting with the real system or with a simulator. This simulator gets as input the same information as the adversary. In a nutshell, this means that the user can simulate `MLCapsule` using a server-side API ML model. Hence, classification using an IEE should not give the user more information about the model as if she would be using an oracle access to the model (e.g. as it is the case for a server-side API access).

We begin the description of the model by binding it with the high level description presented in Section V using three algorithms that constitute the interactive setup phase (Train, Obtain and Provide) and the local inference phase (Classify). An overview of how `MLCapsule` fits into this composition and details about the instantiation are given in Figure 4. Moreover, in the next subsection we describe the security model in more details and then present our formal instantiation of `MLCapsule`.

A. Formalization of `MLCapsule`

We now define in more details the inputs and outputs of the four algorithms that constitute the model for `MLCapsule`. Then we show what it means that `MLCapsule` is correct and show a game based definition of ML model secrecy.

- $\mathsf{Train}(\mathsf{train}_{\mathsf{data}})$: this probabilistic algorithm is executed by the service provider in order to create a machine

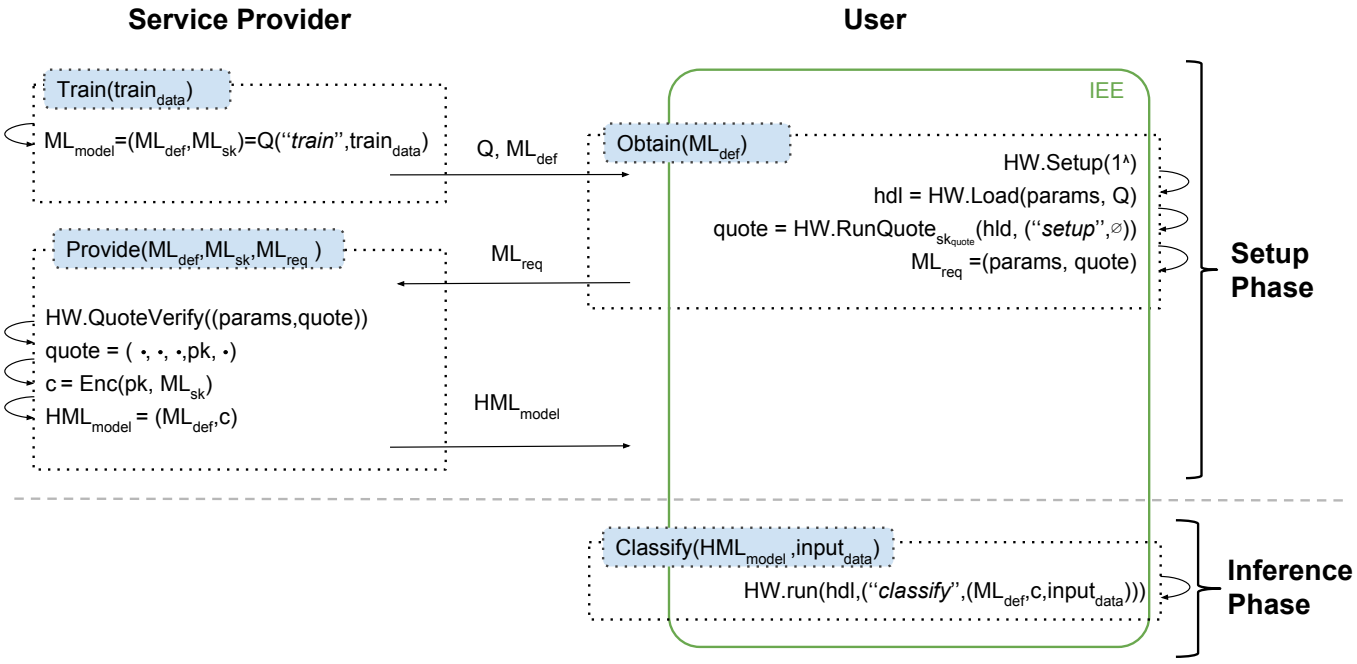


Fig. 4: Formal instantiation of MLCapsule.

learning model $\mathsf{ML}_{\mathsf{model}} = (\mathsf{ML}_{\mathsf{def}}, \mathsf{ML}_{\mathsf{sk}})$ based on training data $\mathsf{train}_{\mathsf{data}}$, where $\mathsf{ML}_{\mathsf{def}}$ is the definition of the ML model and $\mathsf{ML}_{\mathsf{sk}}$ are the secret weights and biases. To obtain the classification $\mathsf{output}_{\mathsf{data}}$ for a given input data $\mathsf{input}_{\mathsf{data}}$ one can execute $\mathsf{output}_{\mathsf{data}} = \mathsf{ML}_{\mathsf{model}}(\mathsf{input}_{\mathsf{data}})$.

- $\mathsf{Obtain}(\mathsf{ML}_{\mathsf{def}})$: this algorithm is executed by the user to create a request $\mathsf{ML}_{\mathsf{req}}$ to use model with definition $\mathsf{ML}_{\mathsf{def}}$. Further, this algorithm is part of the setup phase and shown in steps 1 and 2 in Figure 2.
- $\mathsf{Provide}(\mathsf{ML}_{\mathsf{def}}, \mathsf{ML}_{\mathsf{sk}}, \mathsf{ML}_{\mathsf{req}})$: this probabilistic algorithm is executed by the service provider to create a hidden ML model $\mathsf{HML}_{\mathsf{model}}$ based on the request $\mathsf{ML}_{\mathsf{req}}$. Figure 2 depicts this algorithm in step 3 of the setup phase.
- $\mathsf{Classify}(\mathsf{HML}_{\mathsf{model}}, \mathsf{input}_{\mathsf{data}})$: this algorithm is executed by the user to receive the output $\mathsf{output}_{\mathsf{data}}$ of the classification. Hence, it models the inference phase depicted in Figure 3.

Correctness. We say that MLCapsule is correct if for all training data $\mathsf{train}_{\mathsf{data}}$, all machine learning models $(\mathsf{ML}_{\mathsf{def}}, \mathsf{ML}_{\mathsf{sk}}) = \mathsf{Train}(\mathsf{train}_{\mathsf{data}})$, all input data $\mathsf{input}_{\mathsf{data}}$ and all requests $\mathsf{ML}_{\mathsf{req}} = \mathsf{Obtain}(\mathsf{ML}_{\mathsf{def}})$ we have $\mathsf{ML}_{\mathsf{model}}(\mathsf{input}_{\mathsf{data}}) = \mathsf{Classify}(\mathsf{HML}_{\mathsf{model}}, \mathsf{input}_{\mathsf{data}})$, where $\mathsf{HML}_{\mathsf{model}} = \mathsf{Provide}(\mathsf{ML}_{\mathsf{def}}, \mathsf{ML}_{\mathsf{sk}}, \mathsf{ML}_{\mathsf{req}})$.

ML Model Secrecy. We define model secrecy as a game played between a challenger C and an adversary $\mathcal{A}$. Depending on the bit chosen by the challenger, the adversary interacts with the real system or a simulation. More formally, we say that MLCapsule is ML model secure if there exists a simulator $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$ such that the probability that $|\Pr[\mathsf{Exp}^{\mathsf{secrecy}}_{\mathtt{MLCapsule},\mathcal{A}}\text{-}b(\lambda) = 1] - \frac{1}{2}|$ is negligible for any probabilistic polynomial time adversary $\mathcal{A}$.

$\mathsf{Exp}^{\mathsf{secrecy}}_{\mathtt{MLCapsule},\mathcal{A}}\text{-}b(\lambda)$

$\mathsf{ML}_{\mathsf{model}} \leftarrow \mathsf{Train}(\mathsf{train}_{\mathsf{data}})$
$(\mathsf{ML}_{\mathsf{def}}, \mathsf{ML}_{\mathsf{sk}}) = \mathsf{ML}_{\mathsf{model}}$
$\mathsf{ML}_{\mathsf{req}} \leftarrow \mathcal{A}(\mathsf{ML}_{\mathsf{def}})$
$b \leftarrow_{\$} \{0,1\}$
if $b = 0$ $\mathsf{HML}_{\mathsf{model}} \leftarrow \mathsf{Sim}_1(\mathsf{ML}_{\mathsf{def}}, \mathsf{ML}_{\mathsf{req}})$
else $\mathsf{HML}_{\mathsf{model}} \leftarrow \mathsf{Provide}(\mathsf{ML}_{\mathsf{def}}, \mathsf{ML}_{\mathsf{sk}}, \mathsf{ML}_{\mathsf{req}})$
$\hat{b} \leftarrow \mathcal{A}^{\mathcal{O}(\mathsf{ML}_{\mathsf{model}}, \mathsf{HML}_{\mathsf{model}}, \cdot)}(\mathsf{HML}_{\mathsf{model}})$
else return $\hat{b} = b$

$\mathcal{O}(\mathsf{ML}_{\mathsf{model}}, \mathsf{HML}_{\mathsf{model}}, \mathsf{input}_{\mathsf{data}})$

parse $\mathsf{ML}_{\mathsf{model}} = (\mathsf{ML}_{\mathsf{def}}, \cdot)$
if $b = 0$ **return** $\mathsf{Sim}_2(\mathsf{ML}_{\mathsf{model}}, \mathsf{input}_{\mathsf{data}})$
else return $\mathsf{Classify}(\mathsf{ML}_{\mathsf{def}}, \mathsf{HML}_{\mathsf{model}}, \mathsf{input}_{\mathsf{data}})$

Fig. 5: ML model secrecy experiment.

B. Instantiation of MLCapsule

The idea behind our instantiation is as follows. The user retrieves a program Q from the SP and executes it inside a secure hardware. This secure hardware outputs a public key and an attestation that the code was correctly executed. This data is then send to the SP, which encrypts the secrets

Algorithm 1: Hardware Program Q

Input: command, data
Output: out

1: **if** command== "train" **then**
2: run the ML training on datareceiving $\mathsf{ML}_{\mathsf{model}} = (\mathsf{ML}_{\mathsf{def}}, \mathsf{ML}_{\mathsf{sk}})$, store $\mathsf{ML}_{\mathsf{model}}$ and set out $= \mathsf{ML}_{\mathsf{model}}$
3: **else if** command== "setup" **then**
4: execute $(\mathsf{sk}, \mathsf{pk}) = \mathsf{KeyGen}(1^n)$, store sk and set out $= \mathsf{pk}$
5: **else if** command== "classify" **then**
6: parse data $= (\mathsf{ML}_{\mathsf{def}}, c, \mathsf{input}_{\mathsf{data}})$ and execute $\mathsf{Dec}(\mathsf{sk}, c) = \mathsf{ML}_{\mathsf{sk}}$, set $\mathsf{ML}_{\mathsf{model}} = (\mathsf{ML}_{\mathsf{def}}, \mathsf{ML}_{\mathsf{sk}})$ and out $= \mathsf{ML}_{\mathsf{model}}(\mathsf{input}_{\mathsf{data}})$
7: **end if**
8: **return** out

corresponding to the ML model and sends it back to the user. This ciphertext is actually the hidden machine learning model, which is decrypted by the hardware and the plaintext is used inside the hardware for classification. ML model secrecy follows from the fact that the user cannot produce forged attestations without running program Q in isolation. This also means that the public key is generated by the hardware and due to indistinguishability of chosen plaintext of the encryption scheme, we can replace the ciphertext with an encryption of 0 and answer hardware calls using the model directly and not the Classify algorithm. Below we presents this idea in more details.

- $\mathsf{Train}(\mathsf{train}_{\mathsf{data}})$: executes $\mathsf{ML}_{\mathsf{model}} = (\mathsf{ML}_{\mathsf{def}}, \mathsf{ML}_{\mathsf{sk}}) = Q(\text{"}train\text{"}, \mathsf{train}_{\mathsf{data}})$. Output $\mathsf{ML}_{\mathsf{def}}$.
- $\mathsf{Obtain}(\mathsf{ML}_{\mathsf{def}})$: given $\mathsf{ML}_{\mathsf{def}}$, setup the hardware parameters params by running $\mathsf{HW.Setup}(1^n)$ and load program Q using $\mathsf{HW.Load}(\mathsf{params}, Q)$, receiving a handle to the enclave hdl. Execute the HW setup command for program Q by running $\mathsf{HW.RunQuote}_{\mathsf{sk}_{\mathsf{quote}}}(\mathsf{hdl}, (\text{"}setup\text{"}, \emptyset))$ a quote quote (that includes the public key pk). Finally, it sets $\mathsf{ML}_{\mathsf{req}} = (\mathsf{params}, \mathsf{quote})$.
- $\mathsf{Provide}(\mathsf{ML}_{\mathsf{def}}, \mathsf{ML}_{\mathsf{sk}}, \mathsf{ML}_{\mathsf{req}})$: Abort if verification of the quote failed: $\mathsf{HW.QuoteVerify}(\mathsf{params}, \mathsf{quote}) = 0$. Parse $\mathsf{quote} = (\cdot, \cdot, \cdot, \mathsf{pk}, \cdot)$, compute ciphertext $c = \mathsf{Enc}(\mathsf{pk}, \mathsf{ML}_{\mathsf{sk}})$ and set $\mathsf{HML}_{\mathsf{model}} = (\mathsf{ML}_{\mathsf{def}}, c)$.
- $\mathsf{Classify}(\mathsf{HML}_{\mathsf{model}}, \mathsf{input}_{\mathsf{data}})$: Parse the hidden machine model $\mathsf{HML}_{\mathsf{model}}$ as $(\mathsf{ML}_{\mathsf{def}}, c)$ and return $\mathsf{HW.Run}(\mathsf{hdl}, (\text{"}classify\text{"}, (\mathsf{ML}_{\mathsf{def}}, c, \mathsf{input}_{\mathsf{data}})))$.

C. Security

Theorem 1. The `MLCapsule` presented in Section VI-B is model secure if in the used public key encryption is indistinguishable under chosen plaintext attacks and the hardware functionality HW is remote attestation unforgeable.

Proof. We prove this theorem using the game based approach, where we make slight changes that are indistinguishable for the adversary. We start with $\mathbf{GAME_0}$, which is the original ML model security experiment with bit $b = 1$, i.e. $\mathsf{Exp}^{\mathsf{secrecy}}_{\mathtt{MLCapsule},\mathcal{A}}-1(\lambda)$. We end the proof with $\mathbf{GAME_4}$, which is actually the experiment $\mathsf{Exp}^{\mathsf{secrecy}}_{\mathtt{MLCapsule},\mathcal{A}}-0(\lambda)$

$\mathbf{GAME_1}$ Similar to $\mathbf{GAME_0}$ but we abort in case the adversary outputs a valid request $\mathsf{ML}_{\mathsf{req}}$ without running program Q.

It is easy that by making this change we only lower the adversary's advantage by a negligible factor. In particular, an adversary for which we abort $\mathbf{GAME_1}$ can be used to break the remote attestation unforgeability of the hardware model. Thus, we conclude that this change lowers the adversary's changes by a negligible factor.

$\mathbf{GAME_2}$ Now we replace the way the oracle $\mathcal{O}$ works. On a given query of the adversary we will always run $\mathsf{Sim}_2(\mathsf{ML}_{\mathsf{model}}, \mathsf{input}_{\mathsf{data}}) = \mathsf{ML}_{\mathsf{model}}(\mathsf{input}_{\mathsf{data}})$.

Note that this does not change the adversary's advantage, since both outputs should by correctness of `MLCapsule` give the same output on the same input.

$\mathbf{GAME_3}$ We now replace the ciphertext given as part of $\mathsf{HML}_{\mathsf{model}} = (\mathsf{ML}_{\mathsf{def}}, c)$ to the adversary, i.e. we replace c with an encryption of 0.

This change only lowers the adversary's advantage by the advantage of breaking the security of the used encryption scheme. Note that by $\mathbf{GAME_1}$ we ensured that the public key pk inside the request $\mathsf{ML}_{\mathsf{req}}$ is chosen by the secure hardware and can be set by the reduction. Moreover, the oracle $\mathcal{O}$ works independently of $\mathsf{HML}_{\mathsf{model}}$.

$\mathbf{GAME_4}$ We now change how $\mathsf{HML}_{\mathsf{model}}$ is computed. Instead of running $\mathsf{Provide}(\mathsf{ML}_{\mathsf{def}}, \mathsf{ML}_{\mathsf{sk}}, \mathsf{ML}_{\mathsf{req}})$, we use the simulator $\mathsf{Sim}_2(\mathsf{ML}_{\mathsf{def}}, \mathsf{ML}_{\mathsf{req}}) = (\mathsf{ML}_{\mathsf{def}}, \mathsf{Enc}(\mathsf{pk}, 0))$, where $\mathsf{ML}_{\mathsf{req}} = (\cdot, (\cdot, \cdot, \cdot, \mathsf{pk}, \cdot))$.

It is easy to see that this game is actually the experiment $\mathsf{Exp}^{\mathsf{secrecy}}_{\mathtt{MLCapsule},\mathcal{A}}-0(\lambda)$ and we can conclude that our instantiation is ML model secure because the difference between experiments $\mathsf{Exp}^{\mathsf{secrecy}}_{\mathtt{MLCapsule},\mathcal{A}}-0(\lambda)$ and $\mathsf{Exp}^{\mathsf{secrecy}}_{\mathtt{MLCapsule},\mathcal{A}}-1(\lambda)$ is negligible.

□

VII. SGX Implementation and Evaluation

In the setup phase, `MLCapsule` attests the execution of the enclave and decrypts the data send 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 usage of the IEE introduces over executing the classification without it.

We used an Intel SGX enabled desktop PC with Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz that was running Ubuntu 18.04. The implementation was done using C++ and based on the code of Slalom [40], which uses a custom lightweight C++ library for feed-forward networks based on Eigen. Note that porting well-known ML frameworks, such as TensorFlow, to SGX is not feasible at this point, because enclave code cannot access OS-provided features (e.g. multi-threading, disk, and driver IO). If not stated otherwise, we used the -O3 compiler optimization and C++11 standard.

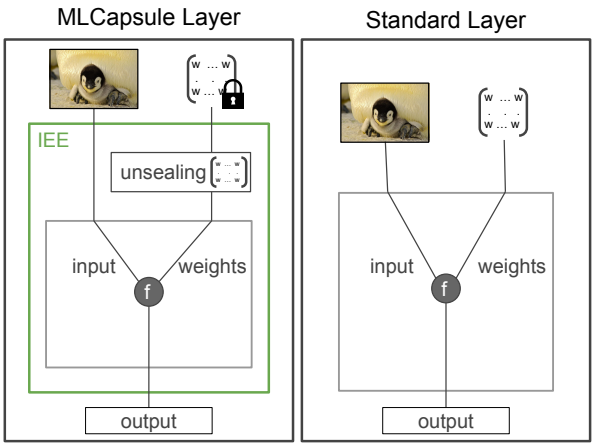


Fig. 6: Difference between `MLCapsule` and standard layer.

In `MLCapsule`, we wrap standard layers to create new `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 6 for more details. This approach provides means to build `MLCapsule` secure neural networks in a modular way.

Since the sealed data is provided from outside the enclave it has to be first copied to the enclave before unsealing. Otherwise, unsealing will fail. 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 standard computation using the unsealed weights and plaintext input
5) free the allocated memory

A. Implementation Issues

Applications are limited to 90 MB of memory, because there is currently no SGX support for memory swapping. Linux provides an OS based memory swap, but the enclave size has to be final and should not expand to prevent page faults and degrade performance. This performance issue is visible in our results for a dense layer with weight matrix of size 4096×4096. In this case, the `MLCapsule` layer allocates $4 \times 4096 \times 4096 = 64$ 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 to a decrease in performance. In particular, the execution of such a layer took 1s and after optimization, the execution time decreased to 0.097s.

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 I, Table II, and Table III. 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 $512 \times 14 \times 14$. In case of dephtwise separable convolutional layers, the execution time of `MLCapsule` layers is comparable with 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. In case of dense layers, we observe 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 3. We suspect that the compiler is able to more efficiently optimize the source code that does not use SGX specific library calls and hence the increase in performance is due to the optimized compilation.

B. Evaluation of Full Classification

In this subsection we combine `MLCapsule` layers to form popular ML networks, i.e. VGG-16 and MobileNet. The first network can be used to classify images and work with the ImageNet dataset (size of images $224 \times 224 \times 3$). Similar, the second network can also be used for the same task. It is easy to see from Table IV that `MLCapsule` has around 2-times overhead in comparison to an execution of the classification using standard layer and without the protection of SGX.

VIII. Advanced Defenses

Recently, researchers have proposed multiple attacks against MLaaS: reverse engineering [27], model stealing [41], [42], and membership inference [35], [34]. As mentioned in Section II, 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 [41], [35], [42], [34].

In this section, we propose two new defense mechanisms against reverse engineering and membership inference (test-time defense). We show that these mechanisms together with a proposed defense for model stealing can be seamlessly integrated into `MLCapsule`.

A. Detecting Reverse Engineering of Neural Network Models

Oh et al. have shown that by only having black box access to neural network model, a wide variety of model specifics can be inferred [27]. Such information include network topology, training procedure as well as type of non-linearities and filter sizes, which thereby turns the model step by step into a white-box model. This equally affects the safety of intellectual property as well as increases attack surface.

Methodology. Up to now, no defense has been proposed to counter this attack. We propose the first defense in this

TABLE I: Average dense layer overhead for 100 executions. This comparison includes two ways of compiling the code, i.e. with and without the g++ optimization parameter -O3.

Matrix Dimension	`MLCapsule` Layer (no -O3)	`MLCapsule` Layer	Standard Layer	Standard Layer (no -O3)
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 II: Average convolution layer overhead for 100 executions and 3×3 filters.

Input/Output Size	`MLCapsule` Layer	Standard Layer	Factor
64×224×224	80ms	66ms	1.21
128×112×112	70ms	63ms	1.11
256×56×56	55ms	54ms	1.02
512×28×28	61ms	51ms	1.20
512×14×14	30ms	13ms	2.31

TABLE III: Average depthwise separable convolution layer overhead for 100 executions and 3×3 Filters.

Input/Output Size	`MLCapsule` Layer	Standard Layer	Factor
64×224×224	41ms	27ms	1.52
128×112×112	18ms	16ms	1.125
256×56×56	9ms	9ms	1.00
512×28×28	7ms	7ms	1.00
512×14×14	2ms	2ms	1.00

TABLE IV: Average neutal network evaluation overhead for 100 executions.

Network	`MLCapsule` Layer	Standard Layer	Factor
VGG-16	1145ms	736ms	1.55
MobileNet	427ms	197ms	2.16

domain and show that it can be implemented seamlessly into `MLCapsule`. We observe that the most effective method proposed by Oh et al. [27] 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. Note that also this detection is running on the client and therefore the decision to deny service can be taken on the client and does not require interactions with a server.

We focus on the kennen-io method by Oh et al. [27] as it leads to the strongest attack. We also 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 image with the kennen-io method and train a classifier against 4500 benign MNIST images. We use the following deep learning architecture:

$$\texttt{Input Image} \rightarrow \texttt{conv2d}(5 \times 5,\ 10)$$
$$\texttt{max}(2 \times 2)$$
$$\texttt{conv2d}(5 \times 5,\ 20)$$
$$\texttt{max}(2 \times 2)$$
$$\texttt{FullyConnected}(50)$$
$$\texttt{FullyConnected}(2)$$
$$\texttt{softmax} \rightarrow \texttt{Output}$$

where `conv2d`$(a \times a,\ b)$ denotes a 2d convolution with a by a filter kernel and b filters, `max`$(c \times c)$ denotes max-pooling with a kernels size of c by c, `FullyConnected`(d) denotes a fully connected layer with d hidden units and `softmax` a softmax layer. In addition, the network uses ReLU non-linearties and drop-out for regularization. We represent the output as 2 units – one for malicious and one for benign. We use a cross-entropy loss to train the network with the ADAM optimizer.

Evaluation. We compose a test set of additional 500 malicious

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

input samples and 500 benign MNIST samples that are disjoint from the training set. The accuracy of this classifier is 100%, thereby detecting each attack on the first malicious example, which in turn can be stopped immediately by denying the service. 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.

B. Detecting Model Stealing

Model stealing attack aims at obtaining a copy from an MLaaS model [41], [30]. Usually, this is done by training a substitute on samples rated by the victim model, resulting in a model with similar behavior and/or accuracy. Hence, successful model stealing leads to the direct violation of the service provider's intellectual property. Very recently, Juuti el al. [21] propose a defense, namely Prada, to mitigate this attack which we implement in `MLCapsule`.

In a nutshell, 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 samples generally do not increase set size. Hence, an attack can be detected by the difference of the growth of those sets.

As the detection is independent of the classifier, it can be easily implemented in `MLCapsule`. The resulting computation overhead depends heavily on the user submitted queries [21]. 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 7. We observe that 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. For each new query, we further compute its distance to all previous queries in the set—a costly operation. We assume a set size of $3,000$, corresponding to roughly $5,000$ benign queries. Table V shows that 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.

C. Membership Inference

Shokri et al. are among the first to demonstrate that ML models are vulnerable to membership inference [35]. The main reason behind their attack's success are common overfitting problems in machine learning: A trained model is more confident facing a data point it was trained on than facing a new one, and this confidence is reflected in the model's posterior. Shokri et al. propose to use a binary classifier to perform membership inference. To derive the data for training the classifier, they rely on shadow models to mimic the behavior of the target model. Even though the attack is effective, it is complicated and expensive to mount. More recently, Salem et al. relax the assumption of the threat model by Shokri et al. and show membership inference attacks can be performed in a much easier way [34]. In particular, Salem et al. show that by only relying on the posterior's entropy, an adversary can achieve a very similar inference as the previous one.[8] This attack is much easier to mount thus cause more severer privacy damages. We use it as the membership inference attack in our evaluation. However, we emphasize that our defense is general and can be applied to other membership inference attacks as well.

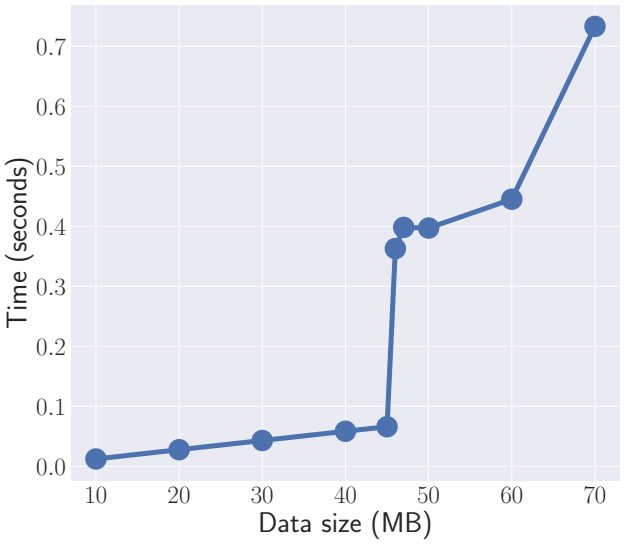


Fig. 7: Overhead in seconds to load additional data for a defense mechanism preventing model stealing, namely Prada [21].

Algorithm 2: 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'

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

[6] http://benchmark.ini.rub.de/
[7] https://www.cs.toronto.edu/~kriz/cifar.html
[8] This is the third adversary model proposed by Salem et al. [34].

TABLE V: Overhead of detecting a model stealing attack, Prada. We assume 3000 samples in the detection set, enough to query 5000 benign samples.

Dataset	Size	SGX	Outside SGX	Factor
MNIST	1×28×28	35ms	2.6ms	13.5
CIFAR	3×32×32	38ms	10.1ms	3.8
GTSDB	3×215×215	2200ms	440ms	5

defined as the following.

$$\eta(P) = -\sum_{P_i \in P} P_i \log P_i$$

Lower entropy implies the ML model is more confident on the corresponding data point. Following Salem et al., the attacker predicts a data point with entropy smaller than a certain threshold as a member of the target model's training set, and vice versa [34].

The principle of our defense is adding more (less) noise to a posterior with low (high) entropy, and publishing the noised posterior. The corresponding algorithm is listed in Algorithm 2. In Step 1, we calculate $\eta(P)$. In Step 2, we derive from $\eta(P)$ the magnitude of the noise, i.e., $\alpha = 1 - \frac{\eta P}{\log |P|}$. 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, according to our experiments, 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. In Step 3, we add noise T to P with $c\alpha$ as the weight. There are multiple ways to initialize T, in this work, we define it as the class distribution of 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 machine learning model.

It is worth noting that 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 [35], or implementing dropout on the neural network model [34]. These defense mechanisms may affect the model's performance as a mature ML model's hyperparameters are normally chosen with a large amount of engineering effort. More importantly, the previous mechanisms (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. For demonstration, we perform experiments on VGG-16 trained on the CIFAR-100 dataset. Our experimental setup follows previous works [34]. Mainly, we divide the dataset into two equal parts for training and testing to train the VGG model.

We mirror the membership inference attack's performance using the AUC score calculated on the entropy of the target model's posteriors. Figure 8a shows the result of our defense with different values of c. As we can see, setting c to 0 -not adding any noise- results in a high AUC score -0.97- which means an attacker can determine the membership state of a point with high certainty. The AUC score starts dropping when increasing the value of c as expected. When the value of c approaches 0.5 the AUC score drops to almost 0.5, which means the best an attacker can do is random guessing the membership state of a point.

We also study the utility of our defense, i.e., how added noise affects the performance of the target model. From Algorithm 2, we see 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 [26], [4]. Formally, $JSD(P, P')$ is defined as:

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

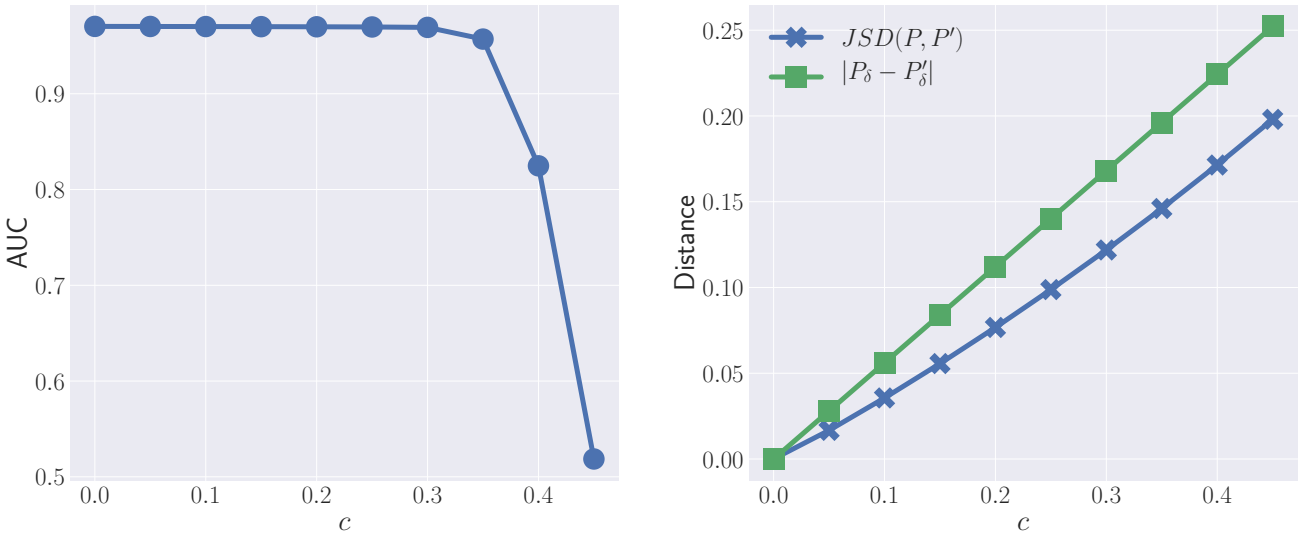
where $M_i = \frac{P_i + P'_i}{2}$. Moreover, we measure the absolute difference between the correct class's original posterior (P_δ) and its noised version (P'_δ), i.e., $|P_\delta - P'_\delta|$, this is also referred to as the expected estimation error in the literature [36], [6], [46]. In Figure 8b, 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, this indicates that our defense mechanism is able to preserve the target model's utility to a large extent.

We measure the overhead of this defense and it only adds 0.026ms to the whole computation. This indicates our defense can be very well integrated into `MLCapsule`.

IX. Discussion

In this section, we address limitations of `MLCapsule`. We start with the limitations of our formal reasoning, and then argue about the defenses we proposed.

Our formal proof shows the our setting is indistinguishable from the access to a MLaaS API. We want to emphasize here that cryptographic proofs do not guarantee security in the case of side channel attacks, such as timing attacks, or the attacks against ML models. This includes model stealing or membership inference, as discussed in this paper.



(a) (b)

Fig. 8: The relation between the hyperparameter controlling the noise magnitude, i.e., c, and (a) [higher is better] membership prediction performance and (b) [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).

Defenses to attacks targeting ML directly are in general an open research question [10]. Yet, `MLCapsule` provides a way to easily integrate any state-of-the-art defense. This extends to other attacks, such as evasion at test time. In evasion attacks, a sample is altered minimal to change the classifiers output. For these malicious test points, state-of-the-art defenses like specific training [23], [33] or verification[18] can be used, in both cases without overhead, as the defense is applied during training. Even more, any defense or mitigation in `MLCapsule` is transparent, as the code can be inspected, and thus `MLCapsule` does not rely on security by obscurity.

Finally, we also consider detection of attacks. Even if found not useful in the area of evasion [10] attacks, such detection might be very useful in `MLCapsule`: in contrast to the standard MLaaS setting, an enclave is tied to a particular person. It is hence possible to identify a user who submitted malicious data. Additionally, setting up a fresh enclave requires some effort. This implies that the service provider is actually able to persecute or expel clients who are found out to run attacks.

X. Conclusion

We have presented a novel deployment mechanism for ML models. It 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 [31] as well as secure ML and AI [38]. Beyond the presented work and direct implications on data privacy and model security – this line of research implements another line of defense 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 this compatible with wide spread use of ML models - but yet can control or mediate access to the model directly (securing the model) or indirectly (advanced protection against inference attacks).

References

[1] Y. Adi, C. Baum, M. Cisse, B. Pinkas, and J. Keshet, "Turning Your Weakness Into a Strength: Watermarking Deep Neural Networks by Backdooring," in *Proceedings of the 2018 USENIX Security Symposium (Security)*. USENIX, 2018. 4

[2] J. Aumasson and L. Merino, "SGX Secure Enclaves in Practice: Security and Crypto Review," in *Proceedings of the 2016 Black Hat (Black Hat)*, 2016. 8

[3] S. Avidan and M. Butman, "Blind Vision," in *Proceedings of the 2006 European Conference on Computer Vision (ECCV)*. Springer, 2006, pp. 1–13. 4

[4] M. Backes, M. Humbert, J. Pang, and Y. Zhang, "walk2friends: Inferring Social Links from Mobility Profiles," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017, pp. 1943–1957. 12

[5] M. Barbosa, B. Portela, G. Scerri, and B. Warinschi, "Foundations of Hardware-Based Attested Computation and Application to SGX," in *Proceedings of the 2016 IEEE European Symposium on Security and Privacy (Euro S&P)*. IEEE, 2016, pp. 245–260. 3

[6] P. Berrang, M. Humbert, Y. Zhang, I. Lehmann, R. Eils, and M. Backes, "Dissecting Privacy Risks in Biomedical Data," in *Proceedings of the 2018 IEEE European Symposium on Security and Privacy (Euro S&P)*. IEEE, 2018. 12

[7] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser, "Machine Learning Classification over Encrypted Data," in *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2015. 1, 4

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

[9] E. Brickell and J. Li, "Enhanced Privacy ID from Bilinear Pairing for Hardware Authentication and Attestation," in *Proceedings of the 2010 IEEE International Conference on Social Computing (SocialCom)*. IEEE, 2010, pp. 768–775. 3

[10] N. Carlini and D. Wagner, "Towards Evaluating the Robustness of Neural Networks," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2017, pp. 39–57. 13

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

[12] B. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov, "Iron: Functional Encryption using Intel SGX," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017, pp. 765–782. 2, 3, 6

[13] M. Fredrikson, S. Jha, and T. Ristenpart, "Model Inversion Attacks that Exploit Confidence Information and Basic Countermeasures," in *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2015, pp. 1322–1333. 1

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

[15] Z. Gu, H. Huang, J. Zhang, D. Su, A. Lamba, D. Pendarakis, and I. Molloy, "Securing Input Data of Deep Learning Inference Systems via Partitioned Enclave Execution," *CoRR abs/1807.00969*, 2018. 4

[16] J. Hayes, L. Melis, G. Danezis, and E. D. Cristofaro, "LOGAN: Evaluating Privacy Leakage of Generative Models Using Generative Adversarial Networks," *CoRR abs/1705.07663*, 2017. 5

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

[18] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety Verification of Deep Neural Networks," in *Proceedings of the 2017 International Conference on Computer Aided Verification (CAV)*. Springer, 2017, pp. 3–29. 13

[19] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel, "Chiron: Privacy-preserving Machine Learning as a Service," *CoRR abs/1803.05961*, 2018. 4

[20] N. Hynes, R. Cheng, and D. Song, "Efficient Deep Learning on Multi-Source Private Data," *CoRR abs/1807.06689*, 2018. 4

[21] M. Juuti, S. Szyller, A. Dmitrenko, S. Marchal, and N. Asokan, "PRADA: Protecting against DNN Model Stealing Attacks," *CoRR abs/1805.02628*, 2018. 2, 5, 11

[22] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing," in *Proceedings of the 2017 USENIX Security Symposium (Security)*. USENIX, 2017, pp. 557–574. 3

[23] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards Deep Learning Models Resistant to Adversarial Attacks," *CoRR abs/1706.06083*, 2017. 13

[24] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "ROTE: Rollback Protection for Trusted Execution," in *Proceedings of the 2017 USENIX Security Symposium (Security)*. USENIX, 2017, pp. 1289–1306. 3

[25] L. Melis, C. Song, E. D. Cristofaro, and V. Shmatikov, "Inference Attacks Against Collaborative Learning," *CoRR abs/1805.04049*, 2018. 1, 2

[26] P. Mittal, C. Papamanthou, and D. Song, "Preserving Link Privacy in Social Network Based Systems," in *Proceedings of the 2013 Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2013. 12

[27] S. J. Oh, M. Augustin, B. Schiele, and M. Fritz, "Towards Reverse-Engineering Black-Box Neural Networks," in *Proceedings of the 2018 International Conference on Learning Representations (ICLR)*, 2018. 1, 2, 5, 9, 10

[28] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious Multi-Party Machine Learning on Trusted Processors," in *Proceedings of the 2016 USENIX Security Symposium (Security)*. USENIX, 2016, pp. 619–636. 4

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

[30] N. Papernot, P. McDaniel, and I. Goodfellow, "Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples," *CoRR abs/1605.07277*, 2016. 5, 11

[31] N. Papernot, P. McDaniel, A. Sinha, and M. Wellman, "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, 13

[32] R. Pass, E. Shi, and F. Tramér, "Formal Abstractions for Attested Execution Secure Processors," in *Proceedings of the 2017 Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Springer, 2017, pp. 260–289. 3

[33] A. Raghunathan, J. Steinhardt, and P. Liang, "Certified Defenses against Adversarial Examples," in *Proceedings of the 2018 International Conference on Learning Representations (ICLR)*, 2018. 13

[34] A. Salem, Y. Zhang, M. Humbert, M. Fritz, and M. Backes, "ML-Leaks: Model and Data Independent Membership Inference Attacks and Defenses on Machine Learning Models," *CoRR abs/1806.01246*, 2018. 1, 2, 5, 9, 11, 12

[35] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership Inference Attacks Against Machine Learning Models," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2017, pp. 3–18. 1, 2, 5, 9, 11, 12

[36] R. Shokri, G. Theodorakopoulos, J.-Y. L. Boudec, and J.-P. Hubaux, "Quantifying Location Privacy," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2011, pp. 247–262. 12

[37] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *CoRR abs/1409.1556*, 2014. 2

[38] I. Stoica, D. Song, R. A. Popa, D. PaŁerson, M. W. Mahoney, R. Katz, A. D. Joseph, M. Jordan, J. M. Hellerstein, J. Gonzalez, K. Goldberg, A. Ghodsi, D. Culler, and P. Abbeel, "A Berkeley View of Systems Challenges for AI," *CoRR abs/1712.05855*, 2017. 13

[39] R. Strackx and F. Piessens, "Ariadne: A Minimal Approach to State Continuity," in *Proceedings of the 2016 USENIX Security Symposium (Security)*. USENIX, 2016, pp. 875–892. 3

[40] F. Tramer and D. Boneh, "Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware," *CoRR abs/1806.03287*, 2018. 4, 8

[41] F. Tramér, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing Machine Learning Models via Prediction APIs," in *Proceedings of the 2016 USENIX Security Symposium (Security)*. USENIX, 2016, pp. 601–618. 1, 2, 5, 9, 11

[42] B. Wang and N. Z. Gong, "Stealing Hyperparameters in Machine Learning," in *Proceedings of the 2018 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2018. 2, 9

[43] W. Wang, G. Chen, X. Pan, Y. Zhang, and X. Wang, "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX," in *Proceedings of the 2017 USENIX Security Symposium (Security)*. ACM, 2017, pp. 2421–2434. 3

[44] S. Yeom, I. Giacomelli, M. Fredrikson, and S. Jha, "Privacy Risk in Machine Learning: Analyzing the Connection to Overfitting," in *Proceedings of the 2018 IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 2018. 1, 5

[45] J. Zhang, Z. Gu, J. Jang, H. Wu, M. P. Stoecklin, H. Huang, and I. Molloy, "Protecting Intellectual Property of Deep Neural Networks with Watermarking," in *Proceedings of the 2018 ACM Asia Conference on Computer and Communications Security (ASIACCS)*. ACM, 2018, pp. 159–172. 4

[46] Y. Zhang, M. Humbert, T. Rahman, C.-T. Li, J. Pang, and M. Backes, "Tagvisor: A Privacy Advisor for Sharing Hashtags," in *Proceedings of the 2018 Web Conference (WWW)*. ACM, 2018, pp. 287–296. 12