VERITRAIN: Validating MLaaS Training Efforts via Anomaly Detection

Xiaokuan Zhang, *Member, IEEE*, Yang Zhang, *Member, IEEE*, and Yinqian Zhang, *Member, IEEE*

***Abstract*—Machine learning as a service (MLaaS) offers users the benefit of training state-of-the-art neural network models on fast hardware with low costs. However, it also brings security concerns since the user does not fully trust the cloud. To prove to the user that the ML training results are legitimate, existing approaches mainly adopt cryptographic techniques such as secure multi-party computation, which incur large overheads. In this paper, we model the problem of verifying ML training efforts as an anomaly detection problem. We design a verification system, dubbed VERITRAIN, which combines unsupervised anomaly detection approaches and hypothesis testing techniques to verify the legitimacy of training efforts on the MLaaS cloud. VERITRAIN is run inside trusted execution environments (TEEs) on the same cloud machine to ensure the integrity of its execution. We consider a threat model where the cloud model trainer is a lazy attacker and tries to fool VERITRAIN with minimum training effort. We perform extensive evaluations on multiple neural network models and datasets, which shows that VERITRAIN performs well in detecting parameter updates crafted by the attacker. We also implement VERITRAIN with Intel SGX and show that it only incurs moderate overheads.**

***Index Terms*—AutoEncoder, machine learning, trusted execution environment.**

I. INTRODUCTION

NOWADAYS, state-of-the-art neural network models [9] have been deployed in many real-world applications ranging from image recognition [52], [85] to machine translation [14]. Despite their superior performance, advanced neural network models are often very time-consuming to train as the training process requires a huge amount of computing resources. For instance, GPT-3 [14] has over 175 billion parameters and requires 3.14E23 FLOPS of computing for training; this means 355 GPU-years for a Tesla V100, one of the fastest GPUs on the market [55]. To bridge the computing resource gap, leading IT companies, including Google, Amazon, and Microsoft, provide machine learning as a service (MLaaS) in the cloud. In this setting, a user uploads his/her dataset to a cloud server and chooses a certain model type, then pays the cloud to train the model. The market value of MLaaS is growing rapidly, and it is projected to reach 8.48 billion USD by 2025 [70].

While MLaaS brings a lot of benefits, it also raises issues of trust, since the user does not fully trust the cloud. When getting the trained model from the cloud, the user cannot determine whether the cloud indeed performs a faithful *training process* as promised. For example, the cloud may spend less effort to maximize its benefits, such as training with fewer epochs to save its GPU hours. The training process is a blackbox for the users; there is no way for the user to perform verification. Therefore, the users have no choice but to trust the MLaaS providers.

To ensure the user that machine learning results returned by a third party are legitimate, researchers have proposed several approaches for verification. There are a number of works that use cryptographic primitives [11], [15], [22], [24], [25], [36], [38], [47], [54], [57], [60], [66], [68], [79], [81] and secure hardware [44], [46], [49], [90] to verify the ML *inference* results. Also, researchers have proposed methods to secure the *training* process, however, these approaches either rely on customized secure protocols with cryptographic approaches [3], [67], [68], [96] that incur large performance overheads, or are specifically tailored to the Federated Learning (FL) scenario [101], [108] and not applicable to the MLaaS scenario.

Recently, there is a new trend in the Cloud Computing area, dubbed *Confidential Computing* [43], [73], where the user's code is run inside hardware trusted execution environments (TEEs) such as Intel SGX and AMD SEV to project the privacy of the user from Cloud service providers (CSPs). Inspired by the recent trend, in this paper, we take the first step towards ensuring the trustworthiness of the ML *training* process in the context of MLaaS using TEEs. We consider a scenario that involves three parties, a user, an MLaaS model trainer, and a verifier running inside a TEE on the same cloud machine. First, the user asks the cloud to perform the training; After training, the model trainer sends the parameter updates to the co-located verifier; Then, the verifier checks the parameter updates to determine whether the training process is faithful, and sends the verification results back to the user. The integrity of the verification process is protected by the TEE.

Here, the verifier is necessary for two reasons. First, as mentioned before, the user does not trust the cloud. Without the support of the verifier running inside TEE which can be attested, the user has to verify the training results by training

Manuscript received 8 February 2022; revised 6 August 2022; accepted 23 March 2023. Date of publication 12 April 2023; date of current version 16 May 2024. *(Corresponding author: Yinqian Zhang.)*

Xiaokuan Zhang is with the George Mason University, Fairfax, VA 22030 USA, and also with the Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: xiaokuan@gmu.edu).

Yang Zhang is with the CISPA Helmholtz Center for Information Security, 66123 Saarbrücken, Germany (e-mail: zhang@cispa.de).

Yinqian Zhang is with the Southern University of Science and Technology, Shenzhen, Guangdong 518055, China, and also with the Research Institute of Trustworthy Autonomous Systems, Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, Guangdong 518055, China (e-mail: yinqianz@acm.org).

Digital Object Identifier 10.1109/TDSC.2023.3266427

a model on his/her own, which loses the purpose of using the MLaaS service. Second, the cloud does not trust the user, either. To demonstrate its faithful training process, the cloud may send all its intermediate parameter updates after each training epoch to the user for verification; however, this places the burden of verification to the user side. Moreover, the user may learn the hyper-parameters used by the cloud for training a model [97], such as the batch size, which directly jeopardizes the intellectual property of the MLaaS. By introducing the verifier, the user can make sure that the parameters returned by the cloud can be trusted, while the cloud does not have to worry about potential information leakage.

Threat Model: We consider a threat model where the MLaaS model trainer is a *lazy* attacker, who wants to save computation power by cheating on the training process. The attacker uses various techniques, including directly copying trained parameters from an existing model and simulating the parameters instead of training, in order to bypass the verification with *minimum* training efforts.

VERITRAIN*:*We design and implement VERITRAIN, a verification system used by the verifier to validate the ML *training efforts* on the MLaaS cloud. VERITRAIN models the problem of determining the legitimacy of parameter updates as *anomaly detection*, and consists of two components: AutoEncoder (AE) and Stationarity Examiner (SE), which are designed to defend against different adversaries. AE is used to detect large reconstruction errors when the adversary generates abnormal training parameters, while SE is designed to detect simulated parameters by capturing the stochasticity of the training process. To ensure that the result reported by VERITRAIN is trusted, VERITRAIN is run inside a co-located TEE, such as Intel SGX [65]. The user can use techniques such as remote attestation [6] to make sure that VERITRAIN runs as expected. Unlike the previous methods that formally verify each step of the computation, VERITRAIN examines whether the cloud has performed the training faithfully via machine learning and hypothesis testing techniques. We do not use cryptographic mechanisms since it is computationally infeasible to verify each training step. As shown in prior works [3], [96], verifying ML training incurs large performance overheads. For example, with secure three-party protocols, a 4-layer CNN takes about 10 hours to train on the MNIST dataset for 5 epochs [96].

We concentrate on verifying the number of training epochs, i.e., the number of times the *full* training dataset is used to train the model, in this paper. The training efforts (costs) grow linearly when the number of training epochs increases, i.e., more epochs leads to more GPU hours, thus it is the major factor related to training efforts. Note that VERITRAIN can be easily adapted to verify other aspects of training efforts (e.g., batch size).

To validate our design, we perform extensive evaluations on popular ML datasets (e.g., CIFAR-10) and state-of-the-art DNN models (e.g., VGG16). Our results show that VERITRAIN can reliably detect the fabricated parameter updates produced by various kinds of attackers while allowing legitimate parameter updates to pass the tests. We also evaluate the robustness of VERITRAIN against adversarial samples and show that they are not able to fool VERITRAIN entirely. We implement VERITRAIN with Intel SGX enclaves using the Graphene-SGX framework [93] to evaluate the real-world performance overhead of VERITRAIN. Our evaluation results suggest that VERITRAIN, when integrated with SGX, only incurs a moderate overhead compared to the training time required. To the best of our knowledge, we are the first to tackle the problem of validating ML training efforts on the MLaaS cloud.

Contributions: This paper makes the following contributions:

- We propose a new system VERITRAIN running inside TEEs which utilizes AutoEncoder and Stationarity Examiner to verify the ML training efforts on the cloud (Section V);
- We perform evaluations on state-of-the-art deep learning models over multiple datasets, and the results show that VERITRAIN is effective in detecting the fabricated training parameters (Section VI). We also evaluate the robustness of VERITRAIN against adversarial samples (Section VII);
- We implement VERITRAIN with Intel SGX and measure the performance overhead. Our evaluation results show that the overhead is moderate compared to the training efforts needed (Section VIII).

II. BACKGROUND

Unsupervised Anomaly Detection: The unsupervised anomaly detection [18], [39], [102] is a task of identifying anomalies, given only the benign data samples. In unsupervised anomaly detection, usually, the defender first learns a general profile of the normal data points, then marks samples that do not fit into the profile as anomalies. Traditionally, researchers have proposed 1) one-class classification methods, such as one-class SVM [4], [82], 2) clustering methods, such as Gaussian Mixture Models (GMM) [100], [104], 3) reconstruction-based methods, such as Robust PCA [16] to perform unsupervised anomaly detection. Recently, researchers have developed various kinds of AutoEncoders [41], [102], [107], [111] for anomaly detection.

Hypothesis Testing: In statistics, hypothesis testing [58], [71] is an approach used to determine the probability that a proposed hypothesis is true, given the observed data. It usually contains 4 steps: 1) State the null hypothesis H_0 and the alternative hypothesis H_1; 2) Identify a test statistic to assess whether the null hypothesis holds; 3) Compute the *pvalue*; 4) Compare the *pvalue* with a preset threshold α, and accept or reject the null hypothesis. In hypothesis testing, a *pvalue* (*pvalue* $\in [0, 1]$) is used in order to quantify the statistical significance of evidence, which can help decide whether to accept or reject the null hypothesis (H_0): The smaller the *pvalue* is, the stronger the evidence that the null hypothesis should be rejected. If the computed *pvalue* is smaller than the preset threshold α (usually 0.05 [72]), H_0 is rejected, and accept H_1; otherwise H_0 is accepted.

Unit Root Test: In statistics, unit root tests are tests for stationarity in a time series [10], [76]. A time series has stationarity if the shape of the distribution is not influenced by a shift in time. As a result, the properties of such a time series, such as mean and variance, also do not change over time. One of the most popular unit root tests is the *Augmented Dickey-Fuller (ADF) test* [30], [35], which tests the null hypothesis (H_0) that a unit root is present (i.e., non-stationarity) in a time series. MacKinnon [61],

[62] further showed how to approximate the *pvalue* for the ADF test, which is widely adopted in popular statistics libraries, such as the `statsmodels` module in Python. In the ADF test, for a given time series, if *pvalue* < 0.05, H_0 is rejected, meaning that the time series has stationarity; otherwise, H_0 is accepted, meaning that it is non-stationary.

Intel SGX: Intel Software Guard Extensions (SGX) [65] is a set of instructions that are built into modern Intel CPUs for security purposes. It provides hardware-based memory encryption for SGX applications. The SGX application can put the code and data in TEEs called *enclaves*, which cannot be accessed by non-enclave entities, including the OS kernel. In this way, SGX application developers can have direct control over the security of their applications to ensure confidentiality and integrity, without relying on the underlying OS which may not be trusted. To examine the integrity of a remote enclave, Intel SGX provides a feature called *remote attestation*. The remote attestation process generates an unforgeable report, which contains information regarding whether the application is running within an enclave on an Intel SGX-enabled platform, and whether the application is intact (has not been tampered with). Remote attestation is important to ensure that an SGX application running on a remote machine performs execution as expected.

III. Overview

A. Scenario

We consider a scenario with three parties involved: a user, a cloud MLaaS service provider (model trainer), and a verifier. The verifier is running inside the TEE on the same MLaaS machine. The user outsources the training process to the cloud, due to the lack of computing resources (e.g., GPU). However, the user does not fully trust the cloud, so he/she asks the cloud to send the parameter updates of each epoch as well as the designated ML training task to the verifier inside a co-located TEE. The verifier verifies the parameter updates and returns the final parameters to the user, together with a report containing information about the parameters and the ML training task performed by the cloud. Since the Verifier runs on the untrusted cloud, the verifier system may be tampered. Here, the user trusts the TEE (through attestation); putting the verifier inside the TEE can make sure that the verification process is intact and trustworthy. By checking the report and attestation result, the user can decide whether to trust the final parameters. The overall process is depicted in Fig. 1. Note that this scenario is similar to the one presented in [90]. The notions used in this paper is shown in Table I

User: The user selects a training dataset $\mathcal{D}$ and a model type $\mathcal{M}$ [1] provided by the cloud service as the target classifier. $\mathcal{D}$ can be a dataset collected by the user, or a public dataset of the user's choice (e.g., UTKFace [109]). The user also specifies the intended ML task $Task$ (e.g., gender prediction) and the required training epochs n. $\mathcal{D}$ is then sent to the cloud by the user, or downloaded from a third party by the cloud; User's choices of $\mathcal{M}$, n and $Task$ are sent to the cloud as well (STEP 1 in Fig. 1). Note that this is the typical scenario in modern MLaaS platforms,

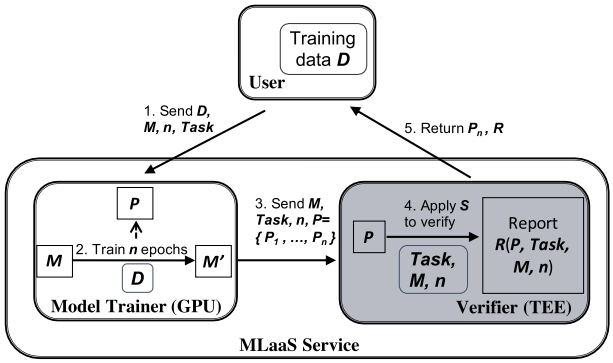


Fig. 1. Overview; Model Trainer and Verifier are co-located.

TABLE I
List of Notations

Notation	Explanation
$\mathcal{M}$	The target classifier (type) chosen by the user
n	The total number of training epochs
m	The total number of trainable parameters of $\mathcal{M}$
λ	The hyper-parameter settings of training
$\mathcal{D}$	The dataset used to train $\mathcal{M}$
$\mathcal{R}$	The report generated and signed by the verifier
$Task$	The machine learning task of training $\mathcal{M}$ on $\mathcal{D}$
$\mathcal{P}_i$	The parameters of $\mathcal{M}$ after training for i epochs
$\mathcal{P}$	The parameter updates of $\mathcal{M}$; $\mathcal{P}=\{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$
$\Delta\mathcal{P}$	$\Delta\mathcal{P}_i=\mathcal{P}_{i+1}-\mathcal{P}_i$, $\Delta\mathcal{P}=\{\Delta\mathcal{P}_1, \cdots, \Delta\mathcal{P}_{n-1}\}$

such as Amazon Machine Learning Services[2] and Google Cloud AI.[3] After that, the user waits for the training result and report from the verifier.

Model Trainer: Given a training dataset $\mathcal{D}$ provided by the user, the MLaaS model trainer trains $\mathcal{M}$ for n epochs with hyper-parameter settings λ on GPUs (STEP 2). The parameters of $\mathcal{M}$ after the ith epoch is denoted as $\mathcal{P}_i$. Then, the parameter updates for n epochs $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ is sent to the verifier inside TEE all together for verification. Besides $\mathcal{P}$, the model trainer also sends $Task$, $\mathcal{M}$, and n to the verifier (STEP 3).

Verifier: After receiving $Task$ and $\mathcal{P}$, the verifier can collect a dataset ($\mathcal{D}_{Verifier}$) on his/her own or use existing datasets with respect to $Task$, as a preparation for the verification process. $\mathcal{D}_{Verifier}$ can come from the same or different distribution of $\mathcal{D}$. In machine learning, it is common to adopt well-known models (e.g., VGG16 [85]) pretrained on large datasets (e.g., ImageNet [29]) and apply transfer learning [48] to obtain a new model for related sub-tasks (e.g., animal classification). Therefore, we assume that the verifier possesses a set of such pretrained models, and the best-match can be used for verification purposes. Based on $\mathcal{D}_{Verifier}$, the verifier verifies $\mathcal{P}$ using a system $\mathcal{S}$ and generates a report $\mathcal{R}$ containing information of $\mathcal{P}$, $Task$, $\mathcal{M}$ and n (STEP 4), which shows the verification result about whether $\mathcal{P}$ is the result of a faithful training process. $\mathcal{R}$ is signed using the private key of the verifier to ensure its integrity. Then, the final parameters $\mathcal{P}_n$ and the report $\mathcal{R}$ are sent to the user (STEP 5). $\mathcal{S}$ should be run inside a Trusted

1.We use $\mathcal{M}$ to represent both a model and a model type in this paper.

2.https://aws.amazon.com/machine-learning
3.https://cloud.google.com/products/ai

Execution Environment (TEE) to make sure that the verification process can be trusted. For example, Intel SGX [65] with remote attestation [6] is one option. Note that though transient execution attacks can be used to steal the secrets inside SGX enclaves and forge signatures [20], Intel processors are being patched. Therefore, we consider such attacks to be out-of-scope.

Necessity of the Verifier: It might be more straightforward if the cloud sends $\mathcal{P}$ to the user directly, without the participation of the verifier. However, this workflow can cause two problems. First, the user does not trust the cloud. Without the support of the verifier, the user has to verify the training results on his/her own. This means the user needs to train the classifier from scratch and compare, i.e., he/she is doing the exact same computation as the cloud, which loses the purpose of using the cloud. Second, the cloud does not trust the user, either. Sending all parameter updates at each epoch to the user may leak the hyper-parameters during training used by the cloud such as the batch size [74], [97], which may be deemed confidential and valuable. By introducing the verifier, the user can make sure that $\mathcal{P}$ returned by the cloud can be trusted, while the cloud does not have to worry about potential information leakage.

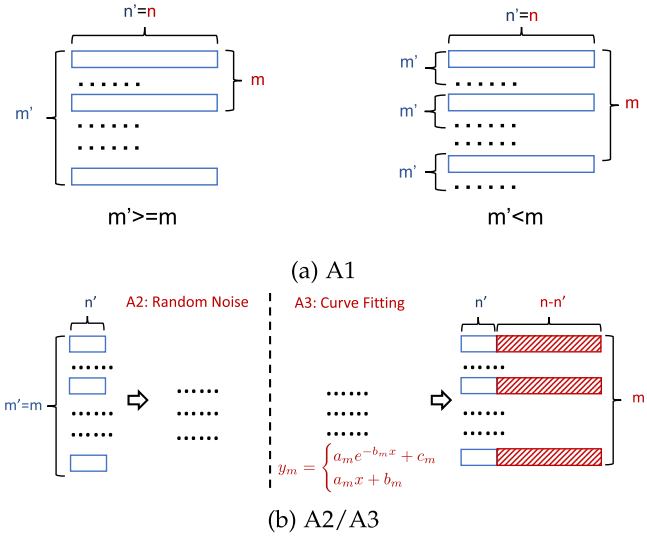


Fig. 2. Illustration of the attacks. In A2/A3, $x = 1, \ldots, n'$. m', n' represent $\mathcal{P}_{Prior}$, while m, n represent $\mathcal{P}_{Attacker}$.

B. Problem Definition

The problem we are trying to solve is to design a system $\mathcal{S}$ on the verifier's side, which can reliably determine whether $\mathcal{P}$ is the legitimate result of training $\mathcal{M}$ targeting $Task$ for n epochs. Formally

$$\mathcal{S}(\mathcal{P}, Task, \mathcal{M}, n) \rightarrow \{True, False\} \tag{1}$$

Here, $True$ means $\mathcal{P}$ is a legitimate training result, while $False$ means otherwise.

Assumptions: We assume that the user chooses a classifier $\mathcal{M}$ that has m trainable parameters, and it is requested to be trained for n epochs. The structure of $\mathcal{M}$ is fixed and known to all parties including the model trainer and the verifier; it can be a well-known one (e.g., VGG16 [85]) or a user-defined one. We denote the intended ML task of the user as $Task_{User}$. We also assume that the verifier possesses a dataset $\mathcal{D}_{Verifier}$ and $\mathcal{P}_{Verifier}$ (parameter updates of $\mathcal{M}$ trained on $\mathcal{D}_{Verifier}$ for n epochs) for verification purposes. In most parts of the paper, we assume $\mathcal{D}_{Verifier}$ either is the same as $\mathcal{D}_{User}$ or follows the same distribution of $\mathcal{D}_{User}$ (See Section VI-B for more details). However, in Section VI-F, we show that this assumption can be further relaxed.

IV. Threat Model

In this paper, the attacker is a *lazy* model trainer, who wants to bypass the verification of the co-located verifier inside TEE with *minimal effort* in the training process, in order to save computation cost.

We assume that the attacker possesses $\mathcal{P}_{Prior}$; $\mathcal{P}_{Prior}$ is a result of training another classifier $\mathcal{M}'$ for n' epochs on $\mathcal{D}_{Prior}$ targeting $Task_{Prior}$, and $\mathcal{M}'$ has m' trainable parameters. Here, $\mathcal{M}'$ shares the same structure as $\mathcal{M}$ since this is one of the attacker's knowledge. The attacker utilizes the prior knowledge of $\mathcal{P}_{Prior}$ to construct $\mathcal{P}_{Attacker}$. To achieve the goal of saving computation cost, we consider the following three types of attacks, where the attacker may re-use existing training results for a different task (A1), or only train a few epochs and then create artificial weights (A2/A3), given the knowledge of the attacker and approaches taken by the attacker (Fig. 2):

Attacker 1 (A1). Direct Copy ($Task_{Prior} \neq Task_{User}$, $m' \neq m$, $n' = n$): $\mathcal{P}_{Prior}$ is the result of training the same $\mathcal{M}$ as the user on another dataset $\mathcal{D}_{Prior}$ targeting a different task $Task_{Prior}$ for n epochs, i.e., $Task_{Prior} \neq Task_{User}$. Note that $\mathcal{D}_{Prior}$ and $\mathcal{D}_{User}$ are completely different datasets; there is no connection between them. For instance, $Task_{User}$ can be a CIFAR-10 classification with Lenet5 and $Task_{Prior}$ can be a UTKFace classification with Lenet5. In this case, $m' \neq m$ (since data points in $\mathcal{D}_{Prior}$ are in different dimension from those in $\mathcal{D}_{User}$) and $n' = n$. The attacker wants to construct $\mathcal{P}_{Attacker}$ based on $\mathcal{P}_{Prior}$ with simple operations, such as truncation and duplication. To reuse $\mathcal{P}_{Prior}$, if $m' > m$, the attacker selects the first m parameters of $\mathcal{P}_{Prior}$ as $\mathcal{P}_{Attacker}$; If $m' < m$, the attacker keeps copying $\mathcal{P}_{Prior}$ until the resulting parameter number is larger than m, then select the first m parameters as $\mathcal{P}_{Attacker}$.

Attacker 2 (A2). Random Noise ($Task_{Prior} = Task_{User}$, $m' = m$, $n' < n$): $\mathcal{P}_{Prior}$ is the result of training $\mathcal{M}$ for n' epochs ($n' < n$) on $\mathcal{D}_{Prior}$ following the same distribution of $\mathcal{D}_{User}$, and $Task_{Prior} = Task_{User}$. This also means $|\mathcal{P}_{Prior}| < |\mathcal{P}_{Attacker}|$. In this case, $m' = m, n' < n$. In another way, the attacker trains fewer epochs to fool the verifier that he/she completes the training. In detail, the attacker tries to generate random values from a certain distribution to mimic the missing parameter updates in $\mathcal{P}_{Prior}$. To fabricate $\mathcal{P}_{Attacker}$, for each parameter in $\mathcal{P}_{Prior}$, the attacker first extracts the n' values representing its value after each training epoch as p_1, and calculate the mean (μ) and standard deviation (σ). After that, the attacker draws $n - n'$ points from the Gaussian distribution $N(\mu, \sigma)$ as p_2. Then the attacker concatenates p_1 and p_2 ($p_1 \| p_2$)

as the parameter updates for this parameter. The process is repeated for all parameters to obtain $\mathcal{P}_{Attacker}$.

Attacker 3 (A3). Curve-Fitting ($Task_{Prior} = Task_{User}$, $m' = m$, $n' < n$): In A3, $\mathcal{P}_{Prior}$ has the same properties as that in A2 ($m' = m, n' < n$); but in A3, the attacker uses curve-fitting techniques to learn the trend of $\mathcal{P}_{Prior}$, and construct $\mathcal{P}_{Attacker}$ based on the curve-fitting results. To perform the curve fitting, the attacker first extracts the n' values for each parameter after each epoch as p_1 (same as in A2), then apply the curve fitting methods on the n' values. In particular, the attacker first applies the $Exponential$ function ($y = a \times exp(-bx) + c$; a, b, c are constants); if the optimal parameters cannot be found after certain number of iterations ($maxfev$), the attacker applies the $LinearRegression$ function ($y = ax + b$; a, b are constants). After finding the corresponding curve C, the attacker records the values of $\{C_{n'+1}, \ldots, C_n\}$ as p_2. Then, the attacker concatenates p_1 and p_2 ($p_1 \| p_2$) as the parameter updates for this parameter. This process is repeated for all the parameters to construct $\mathcal{P}_{Attacker}$.

Other Potential Attackers: Note that there might be other potential attack mechanisms; we defer the discussion on this to Section IX.

Goals and Non-Goals of VERITRAIN *:* According to our threat models, we design a verification system from the verifier's perspective (dubbed VERITRAIN). The goal of VERITRAIN is to detect the three attacks presented in Section IV. Admittedly, the attacker may launch more sophisticated attacks, such as backdoor attacks [8], [45], [99], [105] (see Section IX). However, these attacks require more effort to construct, which contradicts the goal of the attacker: spend as *little* effort as possible to produce legitimate parameters to pass the verification. Also, since VERITRAIN is run inside a TEE, the attacker cannot implant backdoors into the system. Therefore, we consider such attacks to be *out-of-scope*; VERITRAIN is not designed to detect such attacks.

V. VERITRAIN

A. Overview

Overall Design: The input of VERITRAIN is $\Delta\mathcal{P}$, which are the parameter updates (differences) in $\mathcal{P}$ [4]. If the parameters are fabricated, they will exhibit certain characteristics that are different from legitimate ones. To capture such features, we design VERITRAIN with two modules: the AutoEncoder (AE) that computes the reconstruction errors, and the Stationarity Examiner (SE) that performs the hypothesis testing. Intuitively, the fabricated parameters will have large reconstruction errors compared to normal ones in the AE since a well-trained AE is able to recognize anomalies [41], [102], [107], [111], and will have different distributions compared to normal ones in the SE since artificially created parameters will not have the same stochasticity. $\Delta\mathcal{P}$ will be examined by both modules; as long as one of the modules reports that $\Delta\mathcal{P}$ is an anomaly, VERITRAIN rejects it. VERITRAIN is expected to be run inside TEEs, which guarantees the integrity of the verification process. The design of VERITRAIN is depicted in Fig. 3.

4.Here we use $\Delta\mathcal{P}$ instead of $\mathcal{P}$ because $\mathcal{P}$ is more related to the initial settings when starting the training, while $\Delta\mathcal{P}$ is more independent.

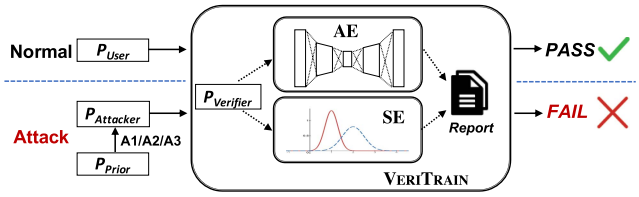


Fig. 3. Overall design of VERITRAIN.

B. Autoencoder

We design an AutoEncoder (AE) against the direct-copy attacker (A1) and the random-noise attacker (A2), which serves as an anomaly detector. AE consists of two components: Encoder and Decoder. The Encoder projects the input onto the latent space, while the Decoder converts the latent vectors back. The general intuition behind AE is that the parameter updates generated by A1 and A2 are anomalies that have large reconstruction errors compared to those from a faithfully training process.

In the training phase, we use the l_2-norm based mean squared error (MSE) as the loss function to train AE: $L_{MSE} = ||x - x'||_2^2$, where x is the input of AE, and x' is the reconstruction of x. In the testing phase, we also use the MSE between x' and x to measure the reconstruction error, which is further used as the anomaly score (AS) of the anomaly detection process. The input of AE is $\Delta\mathcal{P}$, which are the parameter updates (differences) in $\mathcal{P}$, i.e., $\Delta\mathcal{P}_i = \mathcal{P}_{i+1} - \mathcal{P}_i$. For each $\Delta\mathcal{P}_i$, there is an AS reported by AE; we use the average AS of all $\Delta\mathcal{P}_i$ as the final AS for $\Delta\mathcal{P}$. Intuitively, if $\mathcal{P}$ is a result of a faithful training process, the AS will be low; on the other hand, for $\mathcal{P}_{Attacker}$, the AS will be high. Therefore, when the AS is higher than a preset threshold, it will be considered as an anomaly and be rejected.

AE Structure: We model the Encoder with a multi-layer perceptron (MLP). Suppose that the classifier $\mathcal{M}$ has p trainable parameters, and the latent vector Z has q dimensions. We use a three-layer MLP as the Encoder: the first layer transforms p to a 256-dimension vector; the second layer reduces the dimension to 64; the third layer further converts it to q dimensions. Similar to the Encoder, the Decoder is also a three-layer MLP, which is almost the reversed structure of the Encoder: the first layer transforms the latent vector (q dimensions) to a 64-dimension vector; the second layer converts the dimension to 256; the third layer further extends it to p dimensions, the same as the input of the Encoder.

Grouping Before Training: Normally, the parameter changes are large during the first few epochs, and the changes become smaller during the training. As a result, using only one AE for all n epochs may not be optimal. Therefore, we first separate $\Delta\mathcal{P}$ into g groups (g is set by the verifier), then train one AE for each group; in the testing phase, we average the AS within each group to get the final AS for that group. If the final AS of any of the g groups is too high, $\Delta\mathcal{P}$ is considered as an anomaly. We model the problem of finding the splitting points for grouping as a change point detection (CPD) problem. CPD is the problem

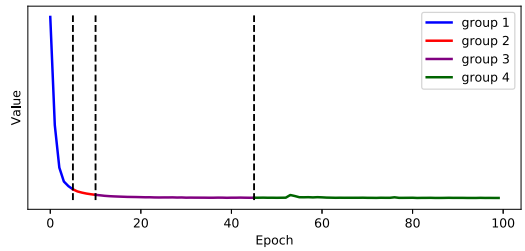
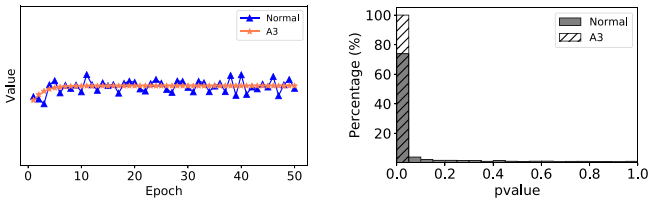


Fig. 4. An example of CPD for $\Delta\mathcal{P}$ when $g = 4$.



(a) Parameter Changes ($\Delta\mathcal{P}$) (b) *pvalue* Distributions

Fig. 5. Comparison between a normal training process and a curve-fitting process (A3).

of finding abrupt changes in a given time series [5], [51]. We use the change points found by existing CPD algorithms as our splitting points for grouping purposes. An example of CPD for $\Delta\mathcal{P}$ when $n = 100$ is shown in Fig. 4. $\mathcal{P}$ has 100 epochs, so $\Delta\mathcal{P}$ has 99 epochs. In this example, we set $g = 4$ to split $\Delta\mathcal{P}$ into 4 groups. As shown in the figure, the binary segmentation method finds the breakpoints to be [5, 10, 45]; therefore, $\Delta\mathcal{P}$ is separated into 4 groups accordingly. Each group of AE handles epochs [1, 5], [6, 10], [11, 45], [46, 99], respectively.

C. Stationarity Examiner

AE can be applied to detect anomalies that have large reconstruction errors; $\mathcal{P}_{Attacker}$ crafted by A1 and A2 can be detected by AE since they generate large noises. However, the curve-fitting attacker (A3) is able to utilize prior knowledge to create $\mathcal{P}_{Attacker}$ that is quite similar to a normal $\mathcal{P}$. The comparison of the $\Delta\mathcal{P}$ fabricated by A3 and the normal $\Delta\mathcal{P}$ is shown in Fig. 5(a). As shown in the figure, the two curves are very close to each other; as a result, such $\mathcal{P}_{Attacker}$ can deceive AE. Therefore, to detect A3, we design our second module, Stationarity Examiner (SE).

The intuition behind SE is that the ML training process is stochastic. In the normal training process, the parameters of the classifier will be fluctuating, due to the stochastic nature of the training process. For $\mathcal{P}_{Attacker}$ that are fabricated by A3, there are inherited trends or patterns. As shown in Fig. 5(a), the $\Delta\mathcal{P}$ of the normal training process is stochastic, i.e., it is less stable than the smooth curve-fitting process. Therefore, by capturing such patterns, we should be able to reliably detect the $\mathcal{P}_{Attacker}$ crafted by A3.

To detect whether a given $\Delta\mathcal{P}$ is fabricated by Attacker 3, we take the following two steps:

Step 1. ADF Test: We first utilize the ADF test (details in Section II) on each parameter to detect whether the parameter update sequence is stationary. If the sequence is stationary, it means that it lacks stochasticity, which further indicates that it is not generated from a legitimate training process.

Step 2. Jensen–Shannon Distance (JSD): Since the verifier possesses $\mathcal{P}_{Verifier}$, to capture the differences of *pvalue* distribution, we calculate the JSD between *pvalue* distribution of $\Delta\mathcal{P}$ and $\Delta\mathcal{P}_{Verifier}$. Jensen–Shannon distance is the square root of the Jensen–Shannon divergence [32], [59], [75], which is a symmetrized and smoothed version of the Kullback-Leibler divergence [53] and is commonly used to measure the similarity between two distributions. Formally, the JSD distance is defined as follows:

$$JSD(u, v) = \sqrt{\frac{1}{2}(KL(u, r) + KL(v, r))}$$

where u, v are two probability distributions; $r = \frac{1}{2}(u + v)$; KL stands for the Kullback-Leibler divergence. When $JSD(Dist_{pvalue}(\Delta\mathcal{P}), Dist_{pvalue}(\Delta\mathcal{P}_{Verifier}))$ is larger than a preset threshold, it means that the *pvalue* distribution of $\Delta\mathcal{P}$ and $\Delta\mathcal{P}_{Verifier}$ are dissimilar, which further indicates that $\Delta\mathcal{P}$ is very likely to be a fabricated result.

Suppose that $\Delta\mathcal{P}$ contains the parameter updates of m parameters and n epochs in total. Therefore, $\Delta\mathcal{P}$ is a $m \times (n-1)$ matrix. To perform the ADF test, for each parameter, we first extract the $n-1$ values associated with it after each training epoch. After that, we run the ADF test on the $n-1$ values to obtain the *pvalue*. This process is repeated for all m parameters, and a list containing m *pvalues* is obtained. After getting the *pvalues* for all the parameters in $\Delta\mathcal{P}$, we compute the distribution of *pvalues* ($Dist_{pvalue}$). The distribution of *pvalues* of $\Delta\mathcal{P}_{Attacker}$ ($Dist_{pvalue}(\Delta\mathcal{P}_{Attacker})$) should be quite different from that of a legitimate $\Delta\mathcal{P}$. A comparison of *pvalue* distributions between a normal training process and a curve-fitting process is shown in Fig. 5(b). It is clear that for the fabricated $\Delta\mathcal{P}$, most of the *pvalues* are close to zero, while those of a benign $\Delta\mathcal{P}$ spread across [0,1]. Therefore, the *pvalue* distribution can be used as evidence to determine whether $\Delta\mathcal{P}$ is fabricated by A3.

D. Training VERITRAIN

Before applying VERITRAIN to test the legitimacy of $\Delta\mathcal{P}$ submitted by the cloud, the verifier needs to have a trained VERITRAIN. Here, the VERITRAIN is pre-trained using a collection of popular models (e.g., VGG16) and datasets (e.g., UTKFace) to obtain the $\mathcal{P}_{Verifier}$. Then, when the verifier receives the $\Delta\mathcal{P}$ from the cloud, the corresponding pre-trained VERITRAIN and $\Delta\mathcal{P}_{Verifier}$ will be used to determine whether it is a legitimate result. In the evaluation (Section VI), we demonstrate the effectiveness of VERITRAIN by testing datasets with the same and different distributions.

E. Implementation

AutoEncoder (AE): AE is implemented using Keras [23] with the Tensorflow backend [1] in Python, and follows the structure mentioned in Section V-B. The change point detection (CPD) is

implemented using the binary segmentation method [34], [84] contained in the `ruptures` module [92]. We train AE using the mean squared error (MSE) as the loss function with the Adam optimizer. The latent dimension is set to 16, and the group number is set to four[5]. We train the AE of each group for 50 epochs with a batch size of 16.

Stationarity Examiner (SE): SE is also implemented in Python. The ADF test is implemented using the `adfuller` function contained in the `statsmodels` module. To calculate the JSD between the given $\Delta\mathcal{P}$ and $\Delta\mathcal{P}_{Verifier}$, we first obtain the *pvalue* lists for $\Delta\mathcal{P}$ and $\Delta\mathcal{P}_{Verifier}$, respectively; then we obtain the *pvalue* distributions ($Dist_{pvalue}$) using the `histogram` function contained in the `numpy` module. After that, we compute the JSD of the two distributions using the `jensenshannon` function in the `scipy` module.

VI. Evaluation

In this section, we perform the evaluation on popular ML datasets and show that VeriTrain is effective in detecting the attacks (Section IV), while allowing the normal parameters obtained through faithful training to pass the tests.

A. Experimental Setup

The experiments presented in this section are conducted on a server equipped with 4 NVIDIA GeForce GTX 1080 Ti GPUs, since we are only evaluating the correctness of VeriTrain. The implementation of VeriTrain with SGX and the performance (overhead) evaluation are presented in Section VIII.

Datasets: In our experiment, we use four datasets: CIFAR-10 [52], UTKFace [109], Insta-LA [7] and Insta-London [7].

- CIFAR-10 is a benchmark dataset, which is frequently used to evaluate image recognition classifiers. It contains 60,000 32×32 color images, which are equally distributed across the 10 classes.
- UTKFace is a dataset of over 20,000 face images; each image is labeled with age, gender, and race. We randomly select 20,000 images as our dataset. The ML task is gender prediction.
- Insta-LA and Insta-London [7] contain samples of Instagram users' location check-in data from Los Angeles and London, respectively. Each check-in sample can be considered as a $(locid, time, catid)$ tuple: $locid$ is a location id, $time$ is the time of the check-in, and $catid$ is the category of the location (e.g., restaurant). There are 8 categories in total. We use the number of check-ins recorded in each hour for each location as the feature vector, and the location's category as the label. Therefore, the classification task is to predict the category (8 classes), given the features (24 features). We also filter out locations that have less than 50 check-ins. Finally, we have 16,028 locations for Insta-LA, and 9,632 locations for Insta-London. Note that these two datasets are used for our different-distribution experiments in Section VI-F.

5.The group number can be set according to the threat model and the requirement of the user. For example, to make the attack harder, the group number can be set to a larger number (e.g., 10).

Models: We train ML models using each dataset and record the parameters, which are later used by VeriTrain for anomaly detection. The models used in our evaluation include Lenet5 [56], VGGFace [78], VGG16 [85] and an MLP model. All classifiers are implemented using Keras with the Tensorflow backend (version 1.15.3). When training the classifiers, we use the Adam optimizer with a learning rate of 0.001 and batch size of 64.

- We use a Lenet5 model for UTKFace and CIFAR-10, respectively. The images are rescaled to 32x32 RGB pixels to be used with Lenet5.
- We use a pre-trained VGGFace model as the feature extractor for UTKFace. We rescale the images into 128x128 RGB pixels to be used with VGGFace. All parameters of the VGGFace model are frozen, meaning that they are not trainable. We add a classification layer on top of the pre-trained VGGFace model for gender prediction.
- We extract block 1 to block 3 of a pre-trained VGG16 model and add a global pooling layer, 2 FC layers followed by a Dropout layer to be used with the CIFAR-10 dataset. We rescale the CIFAR-10 images into 128x128 RGB pixels to be used with VGG16. The parameters of the pre-trained VGG16 model are frozen before the training.
- For Insta-LA and Insta-London, we use a 3-layer MLP model with 40, 30, and 8 neurons, respectively. The evaluation of them are presented in Section VI-F.

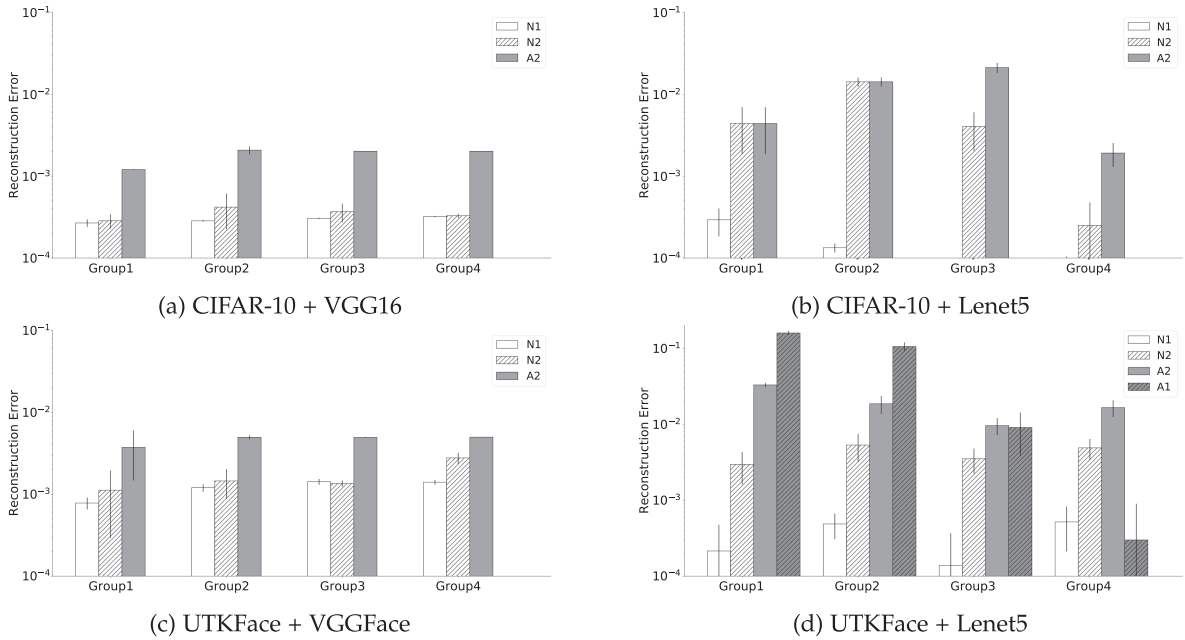
B. Evaluation Scenarios

In the evaluation, we consider two types of scenarios: normal scenarios and attack scenarios. In the normal scenarios, the cloud conducts the training faithfully and submits $\Delta\mathcal{P}_{User}$ to VeriTrain, while in the attack scenarios, it crafts $\Delta\mathcal{P}_{Attacker}$ and submits it. We consider the following scenarios (2 normal + 3 attack) in our evaluation.

Normal Scenarios: We first consider the normal scenarios, where the cloud faithfully conducts the training and submits the legitimate $\Delta\mathcal{P}$ to the verifier.

a) Normal 1 (N1). same dataset: First, we consider the cases where the user and the verifier have exactly the same dataset ($\mathcal{D}_{Verifier} = \mathcal{D}_{User}$). For each testcase, we train the classifiers for 10 times, 100 epoch each, to obtain 10 sets of $\mathcal{P}$. After that, we use $\Delta\mathcal{P}$ to evaluate VeriTrain. Therefore, in this case, $\Delta\mathcal{P}_{Verifier} = \Delta\mathcal{P}_{User} = \Delta\mathcal{P}$.

b) Normal 2 (N2). same distribution: Next, we evaluate the scenario where the dataset used by the verifier ($\mathcal{D}_{Verifier}$) and the dataset used by the user ($\mathcal{D}_{User}$) are not the same, but they follow the same distribution. In this experiment, we split CIFAR-10 and UTKFace into two sub-datasets (A and B), respectively. Each sub-dataset of CIFAR-10 contains 30,000 images, while each of the subsets of UTKFace contains 10,000 face images. The sub-dataset A is used as $\mathcal{D}_{Verifier}$, while the sub-dataset B is used as $\mathcal{D}_{User}$. We train the classifiers with subset A for 10 times, 100 epoch each, and obtain 10 sets of $\mathcal{P}_A$. We repeat this process for subset B, and obtain 10 sets of $\mathcal{P}_B$. Therefore, $\Delta\mathcal{P}_{Verifier} = \Delta\mathcal{P}_A$, and $\Delta\mathcal{P}_{User} = \Delta\mathcal{P}_B$.



(a) CIFAR-10 + VGG16 (b) CIFAR-10 + Lenet5

(c) UTKFace + VGGFace (d) UTKFace + Lenet5

Fig. 6. AutoEncoder evaluation: reconstruction errors (anomaly scores).

TABLE II
MODELS AND DATASETS USED IN THE EVALUATION

Dataset	CIFAR-10	CIFAR-10	UTKFace	UTKFace
Model	VGG16	Lenet5	VGGFace	Lenet5
Trainable Parameters	83,402	81,302	8,193	80,537

TABLE III
SPLITTING POINTS FOR AE GROUPING. $\{g_1, g_2, g_3\}$ MEANS THAT THE 4 GROUPS OF AE HANDLE EPOCHS [1, g_1], [g_1+1, g_2], [g_2+1, g_3] AND [g_3+1, 99] OF $\Delta\mathcal{P}$, RESPECTIVELY

Dataset	CIFAR-10	CIFAR-10	UTKFace	UTKFace
Model	VGG16	Lenet5	VGGFace	Lenet5
Splitting Points	{20, 45, 65}	{5, 10, 60}	{35, 80, 95}	{65, 70, 90}

Attack Scenarios: Next, we consider the attack scenarios presented in Section IV, where the cloud only puts minimal effort in the training process and constructs $\Delta\mathcal{P}_{Attacker}$ in order to fool VERITRAIN.

a) Attack 1 (A1). direct copy: For A1, we use Lenet5 as our classifier $\mathcal{M}$, UTKFace as $\mathcal{D}_{Verifier}$, and CIFAR-10 as $\mathcal{D}_{Prior}$. The attacker constructs $\Delta\mathcal{P}_{Attacker}$ after truncating or duplicating $\Delta\mathcal{P}_{Prior}$ (details in Section IV). We use the trained AE in N1 to get the reconstruction error of AE.

b) Attack 2 (A2). random noise: For A2, we generate the $\Delta\mathcal{P}_{Attacker}$ for all 4 testcases presented in Table II using the approaches presented in Section IV. The first n' parameters are copied from the N2 scenario. Then, we use the trained AE in N1 to get the reconstruction error of AE.

c) Attack 3 (A3). curve-fitting: for A3, we generate $\Delta\mathcal{P}_{Attacker}$ for all 4 testcases presented in Table II using the approaches presented in Section IV. Similar to A2, the first n' parameters are copied from the N2 scenario. We measure the JSD between $\Delta\mathcal{P}_{Verifier}$ and $\Delta\mathcal{P}_{Attacker}$ to evaluate SE.

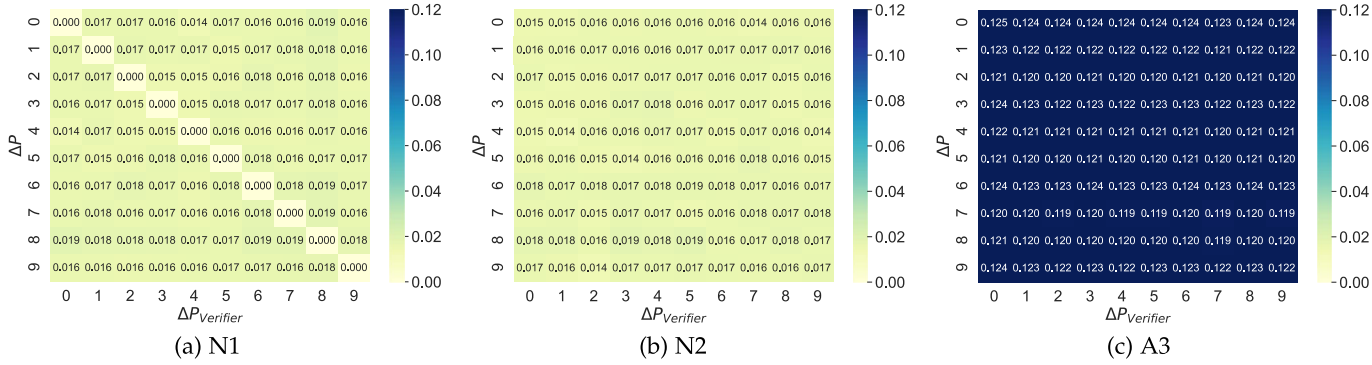
Parameter settings: In our experiment, n is set to 100, so $\mathcal{P}$ has 100 epochs and $\Delta\mathcal{P}$ has 99 epochs. n' is set to 10, so A2 and A3 use legitimate parameters for the first 10 epochs. $maxfev$ is set to 5000 for A3. The values of m depend on the classifiers and datasets. We present the number of trainable parameters of our datasets in Table II.

C. Autoencoder

We first evaluate the performance of the AutoEncoder (AE). For AE, the training set is $\Delta\mathcal{P}_{Verifier}$ generated by training $\mathcal{M}$ on $\mathcal{D}_{Verifier}$, while the test set is the $\Delta\mathcal{P}$ submitted by the cloud. In the normal scenarios, the cloud submits $\Delta\mathcal{P}_{User}$, while in the attack scenarios, it uploads $\Delta\mathcal{P}_{Attacker}$. Ideally, the reconstruction error (anomaly score) of $\Delta\mathcal{P}_{User}$ should be low, while that of $\Delta\mathcal{P}_{Attacker}$ should be high.

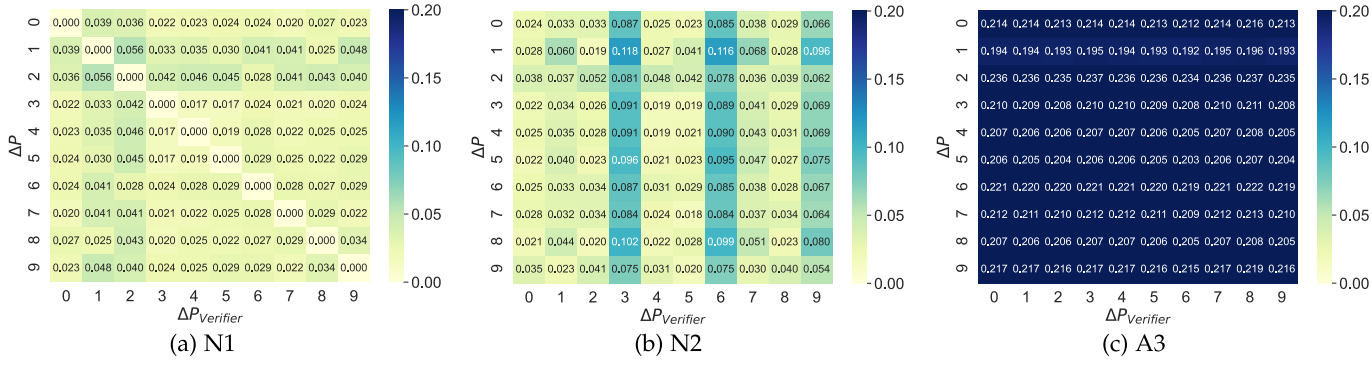
Scenarios: We consider the following four scenarios (2 normal, 2 attack) in our evaluation. As presented in Section VI-B, we have 10 sets of $\Delta\mathcal{P}$ for both normal scenarios and attack scenarios. For *N1: same dataset*, we train AE using $\Delta\mathcal{P}$, and measure the reconstruction error of $\Delta\mathcal{P}$. For *N2: same distribution*, we train the AutoEncoder with $\Delta\mathcal{P}_A$, and measure the reconstruction error using $\Delta\mathcal{P}_B$. For the attack scenarios, we consider *A1: direct copy* and *A2: random noise*. We generate the $\Delta\mathcal{P}_{Attacker}$ using the approaches presented in Section IV.

Results: The reconstruction errors of AE in the 2 normal scenarios and the A1 and A2 scenarios are presented in Fig. 6. As mentioned in Section V-B, we use the CPD algorithm to find the splitting points for separating $\Delta\mathcal{P}$ into groups. The splitting points for grouping are shown in Table III. The reported numbers are the average values and one standard deviation over



(a) N1 (b) N2 (c) A3

Fig. 7. Heatmap of Jensen-Shannon Distance Matrix: CIFAR-10 + VGG16.



(a) N1 (b) N2 (c) A3

Fig. 8. Heatmap of Jensen-Shannon Distance Matrix: CIFAR-10 + Lenet5.

10 rounds of data. For the normal scenarios, as shown in Fig. 6, the reconstruction errors of N1 are smaller than those of N2 in almost all cases.

For A1 (Fig. 6(d)), it is clear that the reconstruction errors of groups 1 and 2 are at least 10 times higher than the numbers of N1 and N2, which means that AE can reliably detect A1. The tests results for A2 are shown in Fig. 6. In Fig. 6(b), the A2 bars are the same as the N2 bars, since the attacker uses the legitimate parameters for the first 10 epochs (Group 1 and 2 for CIFAR10+Lenet5, see Table III), as described in Section VI-B. AE can detect A2 in Group 3 and 4, since the A2 bars are much larger than those of N1 and N2. For other testcases, the A2 bars are always higher than those of N1 and N2 in each group, meaning that parameter updates crafted by A2 cannot deceive AEs.

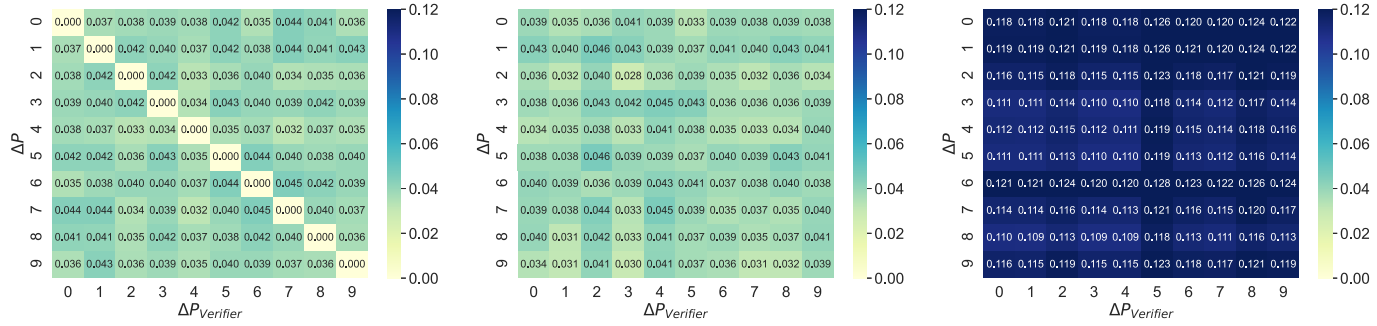
D. Stationarity Examiner

We further evaluate the performance of the Stationarity Examiner (SE). For each testcase, after obtaining the $\Delta\mathcal{P}$, we conduct the ADF test to obtain the *pvalues* and compute the JSD between the $\Delta\mathcal{P}$ and $\Delta\mathcal{P}_{Verifier}$. Ideally, in the normal scenarios, the $\Delta\mathcal{P}_{User}$ should pass the test of the Stationarity Examiner, i.e., has a low JSD with the $\Delta\mathcal{P}_{Verifier}$, while the JSD between $\Delta\mathcal{P}_{Verifier}$ and $\Delta\mathcal{P}_{Attacker}$ should be large.

Scenarios: We consider the following three scenarios (2 normal, 1 attack) in our evaluation. For the 2 normal scenarios, we use the same settings as in Section VI-C, and we measure the JSD between $\Delta\mathcal{P}_{Verifier}$ and $\Delta\mathcal{P}_{User}$. For attack scenarios, we consider *A3: curve-fitting*. We generate the $\Delta\mathcal{P}_{Attacker}$ for each testcases using the approaches presented in Section IV.

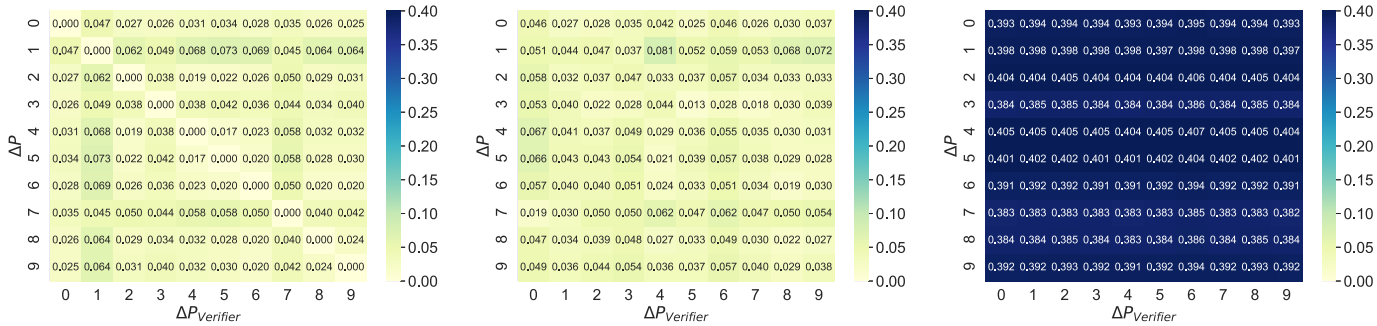
Results: As presented in Section VI-B, we have 10 sets of $\Delta\mathcal{P}$ for both normal scenarios and attack scenarios. To better present the result, we plot the JSD between the $\Delta\mathcal{P}_{Verifier}$ and $\Delta\mathcal{P}$ being tested as heatmaps in Figs. 7–10. X axis means $\Delta\mathcal{P}_{Verifier}$; Y axis means $\Delta\mathcal{P}$. Cell (i, j) means the JSD between the ith set of $\Delta\mathcal{P}_{Verifier}$ and the jth set of $\Delta\mathcal{P}$ being tested. Darker color means larger distance between $\Delta\mathcal{P}$ and $\Delta\mathcal{P}_{Verifier}$, which indicates that $\Delta\mathcal{P}$ is likely to be a fabricated result.

The left figures represent the JSD between $\Delta\mathcal{P}_{Verifier}$ and $\Delta\mathcal{P}_{User}$ in *N1: same dataset* scenario. These heatmaps are symmetric. As shown in the figures, the JSDs are small for all testcases, which is expected. The middle figures represent the JSD between $\Delta\mathcal{P}_{Verifier}$ and $\Delta\mathcal{P}_{User}$ when $\mathcal{D}_{Verifier}$ and $\mathcal{D}_{User}$ follow the same distribution (N2). These heatmaps are asymmetric. We can see that for CIFAR10+Lenet5 (Fig. 8(a)), the JSDs are about twice as high as those in N1 (Fig. 8(b)); for others, the JSDs are about the same compared to those in N1. The colors of all middle figures are light, which means that the $\Delta\mathcal{P}$ in the normal cases can pass the tests. The right figures represent the JSD between $\Delta\mathcal{P}_{Verifier}$ and $\Delta\mathcal{P}_{Attacker}$ crafted by A3 (curve-fitting). It is clear that for all testcases the JSDs of A3 are at least 3 times higher than the



(a) N1 (b) N2 (c) A3

Fig. 9. Heatmap of Jensen-Shannon Distance Matrix: UTKFace + VGGFace.



(a) N1 (b) N2 (c) A3

Fig. 10. Heatmap of Jensen-Shannon Distance Matrix: UTKFace + Lenet5.

normal cases; for CIFAR10+VGG16, it is about 10 times higher. Therefore, the SE can reliably detect the artificial $\Delta\mathcal{P}$ fabricated by A3.

E. Subset of Parameters

To reduce the verification overhead, the user may want to only use a subset of parameters for verification. In this subsection, we study the performance of VERITRAIN when changing the number of parameters used in the verification. In particular, we measure the performance when selecting 5%, 10%, 20%, 50% of the parameters in the CIFAR-10+Lenet5 case.

For AE, we consider three scenarios: N1, N2, and A2. The mean and one standard deviation of reconstruction errors over 10 rounds are shown in Fig. 11. Note that A2 uses legitimate parameters for Group 1 and 2 (first 10 epochs), and crafts parameters for Group 3 and 4. It is clear that in all testcases, the first two bars (N1 and N2) of Group 3 and Group 4 are always smaller than the third (A2); therefore, AE is effective in detecting A2 when only selecting a subset of parameters.

For SE, we consider the same scenarios as in Section VI-D: N1, N2 and A3. Similar to the procedures presented in Section VI-D, for different subsets of parameters, we conduct the ADF test and compute the JSD between the distribution of *pvalues*. For each subset (e.g., 5%), we are able to obtain three 10x10 JSD matrices (similar to Fig. 8). For each of the three matrices, we further compute the average value of the 100 numbers after removing the $zero$ values (diagonal values in N1) and present the average numbers in Table IV. As shown in the table, the JSDs of the first two cases (N1 and N2) are always smaller than 0.078 in the 4 testcases, while the JSDs of A3 are more than 3 times higher ($\geq$0.254). Therefore, SE remains effective even if only a subset of parameters are selected for verification. Moreover, we find that given more parameters, the JSDs of the first two cases will be smaller, indicating that increasing the number of parameters for testing can lower the JSD in the normal cases, thus enhance the detection accuracy of SE.

TABLE IV
JSD WHEN SELECTING A SUBSET OF PARAMETERS

	N1: Same Dataset	N2: Same Dist.	A3: Curve Fitting
5%	0.073	0.078	0.254
10%	0.055	0.068	0.254
20%	0.044	0.061	0.255
50%	0.035	0.055	0.258

According to our evaluation, for the 4 test cases, selecting a subset of parameters does not impact the performance of VERITRAIN much; VERITRAIN can still effectively detect the

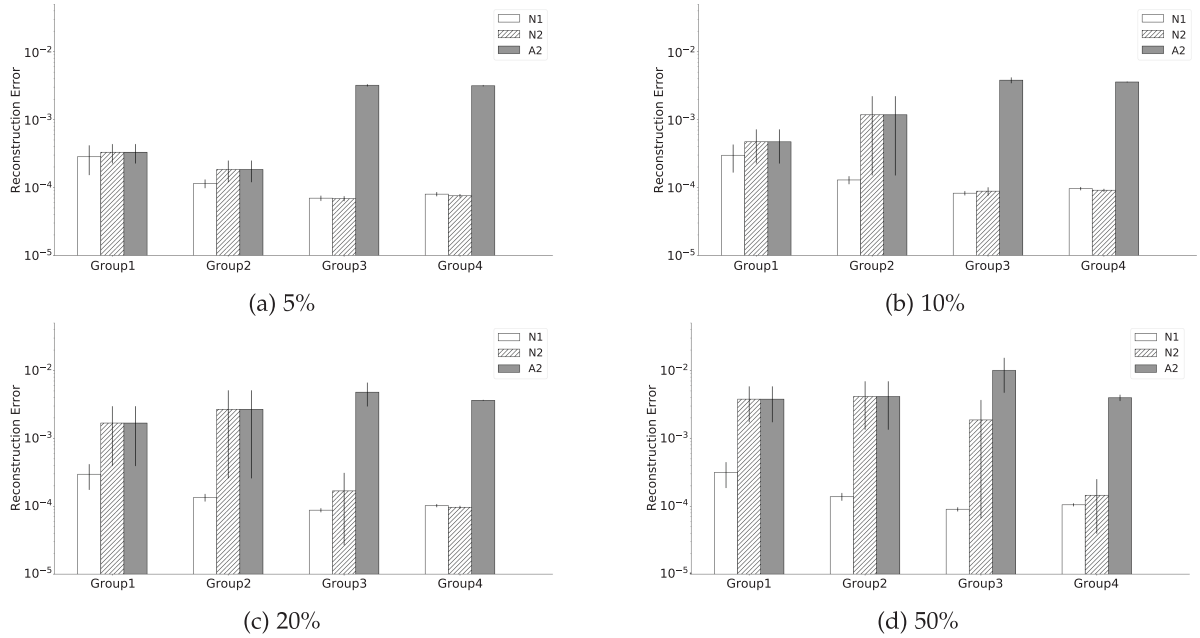


Fig. 11. Performance of AE when selecting a subset of parameters. Note that A2 uses legitimate parameters in Group 1 & 2.

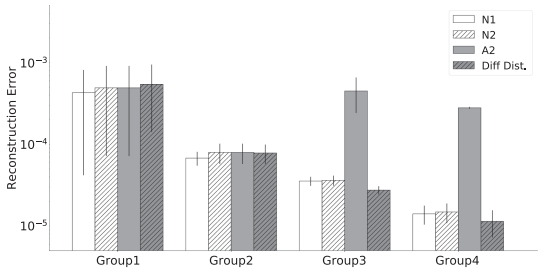


Fig. 12. Different-distribution datasets: AE evaluation.

attacks. Therefore, to reduce the verification overhead, the user may choose to verify a subset of parameters, instead of all, while maintaining legitimacy.

F. Different-Distribution Datasets

So far, we have only evaluated VERITRAIN when $\mathcal{D}_{Verifier} = \mathcal{D}_{User}$ (N1), $\mathcal{D}_{Verifier}$ and $\mathcal{D}_{User}$ follow the same distribution (N2). In this subsection, we further evaluate VERITRAIN when $\mathcal{D}_{Verifier}$ and $\mathcal{D}_{User}$ follow different distributions but target the same ML task.

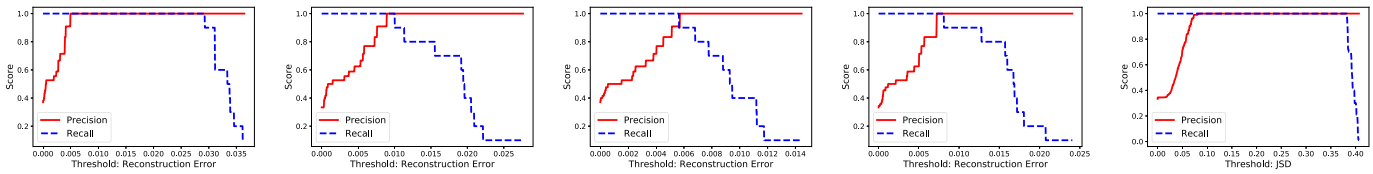
First, we repeat the evaluation of AE for the 3 cases (N1, N2, A2) on the Insta-LA dataset. Then, we reuse the trained AE and use Insta-London as $\mathcal{D}_{User}$ to measure the reconstruction error of AE; in this case, $\mathcal{D}_{Verifier}$ and $\mathcal{D}_{User}$ follow different distributions. The results are shown in Fig. 12. Note that the splitting points for grouping are {5, 10, 45}; Therefore, for Group 1 and 2 (first 10 epochs), A2 uses legitimate parameters. As shown in the figure, in Group 3 and 4, the *A2* bar is always the highest, and it is at least one magnitude larger than others. This result confirms that our AE-based approach also works when $\mathcal{D}_{Verifier}$ and $\mathcal{D}_{User}$ follow different distributions.

We also evaluate the performance of SE when $\mathcal{D}_{Verifier}$ and $\mathcal{D}_{User}$ follow different distributions. We use the same methods as in Section VI-E to calculate the average JSDs; the result are 0.101, 0.102, 0.256 and 0.105 for N1, N2, A3, and different-distribution datasets, respectively. The average JSD of different-distribution datasets is slightly higher than the two normal scenario cases, but it is still a lot lower than that of A3.

Based on our evaluation, we confirm that VERITRAIN remains effective when $\mathcal{D}_{Verifier}$ and $\mathcal{D}_{User}$ follow different distributions but target the same ML task; these legitimate parameters can pass the tests successfully. This indicates that VERITRAIN can perform verification and tolerate differences even if $\mathcal{D}_{Verifier}$ does not follow the same distribution as $\mathcal{D}_{User}$.

G. Threshold Selection

To showcase how a threshold can be selected for VERITRAIN, we use the data generated in Section VI-C and Section VI-D for UTKFace + Lenet5 to plot the Precision-Recall curves. Here, the positive samples are fabricated parameters (A2 for AE, A3 for SE), while the negative samples are the normal parameters (N1, N2). As shown in Fig. 13, different groups of AEs require different threshold settings: for Group 1, it is relatively easy to select a threshold between 0.005 to 0.030 to guarantee perfect Precision and Recall; for other groups, there are clear tradeoffs between Precision and Recall. The verifier can choose to either maximize Precision (minimize *false positives*) or maximize Recall (minimize *false negatives*). For SE, it is clear that choosing a threshold between 0.10 to 0.40 would achieve 100% Precision and Recall.



(a) AE: Group 1 (b) AE: Group 2 (c) AE: Group 3 (d) AE: Group 4 (e) SE

Fig. 13. Threshold selection experiment.

VII. Robustness of VeriTrain

So far, we have shown that VeriTrain is effective against the considered attacks. In this section, we investigate the robustness of VeriTrain. In particular, we are interested in whether VeriTrain is robust against adversarial samples[6].

Assumptions: We assume that the attacker knows the structures and parameters of AutoEncoders, which is the *white-box* setting. Unlike traditional adversarial samples on images [17], [42], [63], [87] where the noise added on images need to be imperceptible, in our scenario, the inputs are parameter updates ($\mathcal{P}$); the adversary does not need to worry about the magnitude of noise since there is no bound for the values. Therefore, the attacker can add arbitrary noises to fool VeriTrain.

Generating Adversarial Samples: The attacker is a *lazy* model trainer, who wants to bypass the verification with *minimal* effort. Therefore, the attacker starts from the three attack scenarios presented in Section VI-B to construct $\Delta\mathcal{P}_{Attacker}$. The attacker applies existing methods to generate the adversarial samples $\Delta\mathcal{P}_{Adv}$ from $\Delta\mathcal{P}_{Attacker}$, in order to fool the AutoEncoders. The goal of the attacker is to generate adversarial samples, $\Delta\mathcal{P}_{Adv}$, from $\Delta\mathcal{P}_{Attacker}$ to minimize the reconstruction error on the pre-trained AutoEncoder. Suppose the noise added is δ, and the maximum perturbation is ϵ. We have the following optimization problem

$$\min_{\delta} L_{AE} \quad s.t. \quad ||\delta||_\infty \leq \epsilon \tag{2}$$

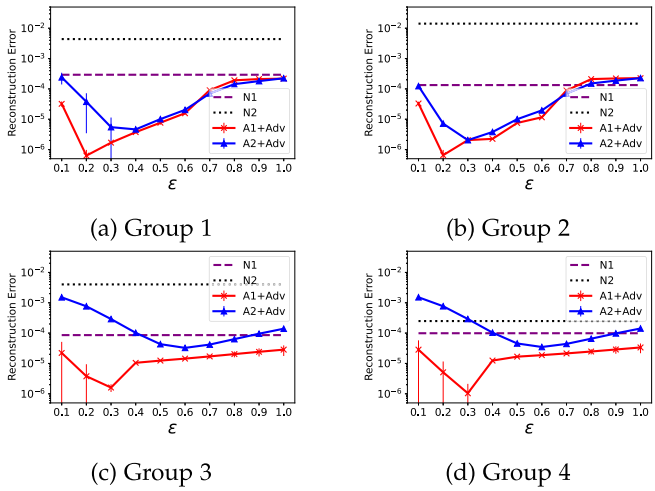
$$L_{AE} = Dec(Enc(\Delta\mathcal{P}_{Adv})) \tag{3}$$

where $\Delta\mathcal{P}_{Adv} = \Delta\mathcal{P}_{Attacker} + \delta$, Enc() and Dec() are encoder/decoder functions of the AutoEncoder, $||\,.\,||_p$ is the l_p-norm. Here we adopt the PGD method [63] to generate adversarial samples. Since PGD tries to maximize the loss function, we set $loss = -L_{AE}$, in order to find the adversarial samples that minimize the reconstruction error for AutoEncoders.

Evaluation: For evaluation, we use the CIFAR-10 + Lenet5 dataset. The attacker starts from the A1 and A2 scenarios, and crafts $\Delta\mathcal{P}_{Adv}$ from $\Delta\mathcal{P}_{Attacker}$ using the PGD method. Specifically, we use $\epsilon \in [0.1, 0.2, \ldots, 1.0]$ and apply the PGD method on the AE of each group to construct the adversarial samples.

First, we measure the reconstruction errors of AEs. The results of AEs are shown in Fig. 14. In the figures, the dashed and dotted lines represent the reconstruction errors of N1 and N2, respectively. As shown in the figures, the adversarial samples

6.Most of the adversarial samples are designed for deep learning; we are not aware of any such techniques for hypothesis testing adopted in this paper. Thus, we focus on adversarial samples against AEs.



(a) Group 1 (b) Group 2 (c) Group 3 (d) Group 4

Fig. 14. Reconstruction errors of AEs with adversarial samples; *A1+Adv/A2+Adv* means results when feeding the adversarial samples constructed from attack scenarios A1 and A2, respectively; *N1/N2* means results when feeding legitimate parameters in normal scenarios N1 and N2.

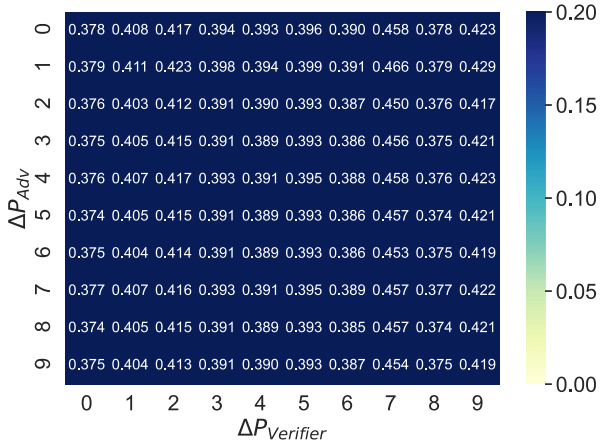
can obtain reconstruction errors that are even lower than the normal scenarios (N1 and N2), thus bypassing the detection of AEs.

We choose the best ϵ that can craft adversarial samples with the lowest reconstruction errors for each group, then we apply the Stationarity Examiner on the adversarial samples. For *A1+Adv*, we choose $\epsilon_1 = 0.3$; for *A2+Adv*, we choose $\epsilon_2 = 0.5$. The heatmaps of the JSD matrix with adversarial samples are shown in Fig. 15. Compared to Fig. 8, the adversarial samples ($\Delta\mathcal{P}_{Adv}$) have larger distances from $\Delta\mathcal{P}_{Verifier}$ than A3 (Fig. 8(c)), thus they are easier to detect by SEs than A3. Therefore, although the adversarial samples can bypass AEs, they cannot evade the detection of SEs.

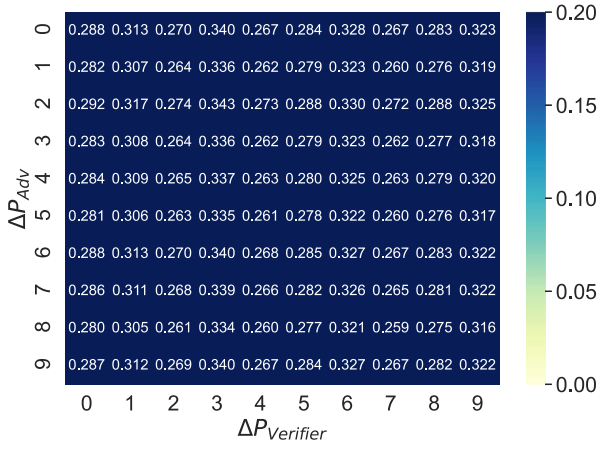
Summary: According to our evaluations, the attacker can utilize existing methods such as PGD to generate adversarial samples to circumvent the AEs effectively. However, the adversarial samples cannot evade the detection of SEs; the adversary needs to work harder to bypass both AEs and SEs. There is a tradeoff between the attacker's effort and gain for fooling VeriTrain; we discuss this in Section IX.

VIII. Implementation With Intel SGX

VeriTrain should be run inside TEEs to ensure the integrity of the verification process. In this section, we present the implementation details of VeriTrain with Intel SGX, as well as the evaluation of the verification cost.



(a) A1+Adv: $\epsilon_1 = 0.3$



(b) A2+Adv: $\epsilon_2 = 0.5$

Fig. 15. Heatmaps of JSD Matrix with adversarial samples.

A. Implementation Details

To evaluate the real-world performance of VERITRAIN with SGX enclaves, we adopt the Graphene-SGX [93] framework since it allows us to run unmodified applications inside Intel SGX. The experiments are conducted on a desktop equipped with 12 Intel(R) Core(TM) i7-8700 CPUs. The desktop runs Ubuntu 18.04 with kernel 5.4.71, and the kernel is patched according to the official setup guide [26] of Graphene-SGX version 1.1. The Intel SGX SDK version is 2.11, and the Intel SGX driver version is 2.11 as well. We implement AE and SE in two different enclaves.

In the previous section, AE is implemented in Keras with Tensorflow; here we re-implement AE in PyTorch since PyTorch is better supported by Graphene-SGX. All the settings are the same as in Section VI-A. The group number of AE is also set to 4. In the training phase, trained AE models are saved locally; in the testing phase, AE models are first loaded into the enclave, then perform inferences. The enclave size is set to 4 G since it is the minimum to make PyTorch work, as suggested by the Graphene-SGX authors [27].

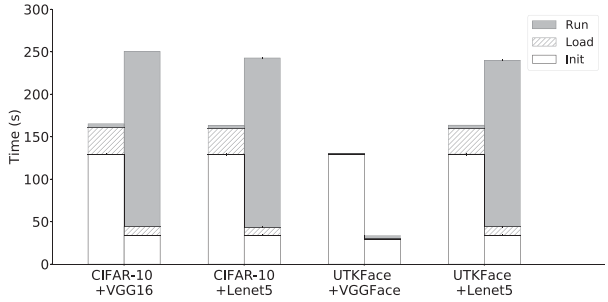


Fig. 16. Verification cost breakdown. In each cluster, left bar is the result of AE, and right bar is the result of SE.

For SE, the `adfuller` Python function we use in the previous section is not supported by Graphene-SGX due to dependency issues. To evaluate the overhead of SE, we re-implement SE using the R language. The enclave size is set to 1 G instead of 4 G since SE does not need to load the PyTorch library and 1 G memory is enough. The ADF test is implemented with the `adf.test` function from the `tseries` package [91]. The Jensen-Shannon Distance (JSD) is computed using the `JSD` method from the `philentropy` package [31].

B. Verification Cost

To evaluate the verification cost (running time) of VERITRAIN, we first study the time breakdown when running VERITRAIN with SGX. After that, we study the cost when choosing a subset of parameters in the CIFAR10+Lenet5 testcase. All the tests were repeated 10 times, and we report the mean and one standard deviation in the figures. We also compare the verification costs with the time needed to train ML models on GPUs and CPUs.

Verification Cost Breakdown: The verification cost can be divided into 3 categories: 1) initializing the enclave, 2) loading the dataset and model, 3) running the code. The cost breakdowns are shown in Fig. 16. For AE (the left bars), the time cost to initialize the enclave is roughly 130 seconds, which is consistent for all 4 testcases. For UTKFace+VGGFace, the total time for AE is roughly 132 seconds, while that of the other three cases is about 165 seconds. The reason is that, for UTKFace+VGGFace, the number of trainable parameters is only 8193, which is about one-tenth of that in other cases (shown in Table II). As a result, the test dataset and trained AEs to be loaded are much smaller, which leads to faster execution. The cost breakdowns of SE are shown in the right bars of Fig. 16. The initialization time is about 30 seconds; it is lower than that of AE, because the enclave size is only 1 G. The total time of UTKFace+VGGFace is also the smallest due to the smaller number of training parameters. While AE spends most of the time on initialization, SE takes a lot of time to perform computation, since the ADF test is computationally intensive.

Verification Cost With Regards to the Number of Training Parameters: We use CIFAR10+Lenet5 to evaluate the cost with regard to the number of training parameters. We measure the

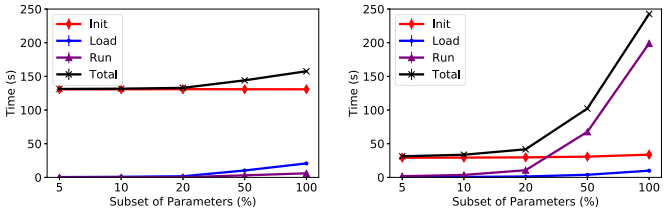


Fig. 17. Verification cost versus number of parameters.

TABLE V
VERIFICATION COST VERSUS TRAINING TIME

Dataset	CIFAR-10	CIFAR-10	UTKFace	UTKFace
Model	VGG16	Lenet5	VGGFace	Lenet5
Training Time: CPU (s)	79,169	1,412	45,462	274
Training Time: GPU (s)	2,153	225	1,113	72
Verification Cost (s)	250	243	34	240

running time when choose 5%, 10%, 20%, 50% of the parameters, similar to Section VI-E. The results of AE are shown in Fig. 17(a). As shown in the figure, the time to load AE, to load data, and to run computation are increasing when selecting more parameters; but the time to initialize the enclave stays at about 130 seconds. For SE (Fig. 17(b)), the initialization time is roughly 32 seconds, about one fourth of that in Fig. (a), due to the different enclave sizes. When selecting more parameters to test, the time to load data only increases by a small margin. The time to run the computation, however, increases drastically: when selecting 10% of the parameters, it takes only about 4 seconds to finish the execution; when 20% and 100% parameters are used, the time cost of finishing computations is 12 seconds, and 194 seconds, respectively. The overhead is growing exponentially due to the nature of the ADF test and the computation of JSD.

Comparison With the Training Time: The total cost of running AE and SE inside SGX enclaves are about 165 seconds and 240 seconds, respectively. Since AE and SE can be run in parallel, we only consider the time cost of SE when calculating the overall cost of VERITRAIN. We report the training time on both GPUs and CPUs for the 4 testcases using the same settings as in Section VI (training for 100 epochs with learning rate 0.001 and batch size 64) and the overall verification cost of VERITRAIN, averaged over 10 rounds in Table V. The training is conducted on a desktop equipped with 12 Intel(R) Core(TM) i7-8700 CPUs and 4 NVIDIA GeForce GTX 1080 Ti GPUs. According to the results, it is clear that using MLaaS (GPU machines) can greatly reduce the time of training. For CIFAR10+VGG16 and UTKFace+VGGFace, with verification cost, the total time is still much smaller (<2.5%) than training directly on CPUs. For smaller datasets and models (e.g., UTKFace+Lenet5), the user may choose to train the model locally on CPUs since the performance speed-up for using MLaaS is small.

IX. DISCUSSION

Limitations: VERITRAIN requires that for each (dataset, model) pair, there is a designated set of AE and SE to validate the training efforts. When the dataset and distribution are unknown to the verifier, VERITRAIN may not work well. However, as we show in Section VI-F, VERITRAIN is robust when $\mathcal{D}_{Verifier}$ and $\mathcal{D}_{User}$ follow different distributions. Moreover, transfer learning techniques may be applied to improve the generality of VERITRAIN to deal with unknown datasets. We leave it as our future work.

SGX Attacks: It is known to the research community that Intel SGX is susceptible to a list of attacks; here we discuss the three most well-known attacks and how VERITRAIN mitigates them. 1) Iago attacks [19]. In our implementation, we use the Graphene-SGX framework [93], which effectively mitigates Iago attacks by checking the system call return values. 2) Side-channel attacks [12], [83], [94], [103]. Side-channel attacks can only jeopardize the confidentiality of SGX applications; they cannot change the execution flows of SGX code. Therefore, it cannot affect the verification result of VERITRAIN. 3) Replay/Rollback attacks [28]. For example, in our scenario, the cloud attacker may directly send a report previously signed by the verifier to the user to get a quick reward, instead of performing training faithfully. To defeat this type of replay attack, the user can supply a *nonce* to the cloud, and ask the cloud to send the *nonce* to the verifier. Then the verifier can include this *nonce* in the final report. When receiving the report, the user can check whether the *nonce* is valid, thus thwarts the replay attack.

Potential Improvements: To improve the performance and reduce the overhead, the verifier may choose to distribute the computations of VERITRAIN and run them in parallel. For example, multi-threading mechanisms may be used to split the workload and run them on different threads. Using lower-level languages (e.g., C/C++) to implement VERITRAIN may reduce the overhead as well. Moreover, the verifier may initialize the enclaves only once and use them for multiple rounds, instead of initializing them every time, so that the initialization cost is paid only once. To lower the training efforts of the verifier, state-of-the-art techniques such as Few-shot Learning (FSL) [33] may also be applied so that the training can be done with only a few samples.

Other Applications: VERITRAIN is designed to establish trust between the user and the cloud in MLaaS scenarios with regard to the number of training epochs; however, it can also be adapted to verify other aspects of the training process in the context of MLaaS, such as the model type and the hyper-parameters. We leave a thorough exploration as our future work. Note that VERITRAIN cannot be applied in the federated learning (FL) scenario, since in FL training datasets of participants are confidential; without the knowledge of the datasets, VERITRAIN is not able to perform verification.

Recent Advances of TEEs: Intel SGX2 [50], [64] is a set of instructions that enable Enclave Dynamic Memory Management. A recent news [89] reported that Intel's new generation of Ice Lake Xeon processors supports 512 GB SGX enclaves. With such a larger enclave size, it would greatly reduce the overhead of

VERITRAIN. Recently, GPU-based TEEs are also attracting a lot of attentions. In the academia, recent works such as Telekine [49] and Graviton [95] implement TEEs within GPUs to better utilize the hardware acceleration while perserves the privacy. In the industry, companies such as NVIDIA already releases H100 GPUs [73] that supports TEEs to protect the entire workloads. It would be also interesting to see how VERITRAIN can be integrated with GPU-TEEs. We leave a thorough investigation on these new TEEs as our future work.

Other Potential Attackers: Note that there might exist a realistic lazy attacker for the case where $Task_{Prior} = Task_{User}$, $m' = m$. When $n' < n$, the attacker can start training from the nth epoch; when $n' \geq n$, the attacker can directly send the parameters of the first n epochs. However, in this scenario, as the training parameters are all legitimate, doing so does not jeopardize the integrity of the training process. This is similar to the legitimate use of transfer learning; therefore, we do not consider it as an attack. For A2, there might exist another smarter attacker that calculate μ and σ based on parameters from the latter epochs rather than from all n' epochs, since intuitively the paramater changes are getting smaller during the training process. In this case, we can use larger group numbers for VERITRAIN to catch the attacks. There are counterless such attack models; we cannot enumerate every such attacker. Instead, what we are trying to do is to show the promising results from VERITRAIN, and the potential that it can be improved in the future to deal with other attackers.

Advanced attacks to fool VERITRAIN *:* The attacker may utilize more advanced attacks besides adversarial samples to bypass VERITRAIN. For example, the attacker may carefully craft adversarial samples so that they can bypass the detection of AEs and SEs simultaneously. However, as shown in Section VII, crafting such adversarial samples is not easy even in the *white-box setting*; in the black-box setting where the attacker does not know the internals of the VERITRAIN, it would take a lot of queries to construct the adversarial samples. The attacker does not have enough incentives to do so, since the goal of the attacker is to save computation. There is a tradeoff between the effort needed for the attacker to fool VERITRAIN, and the verifier may also use techniques such as adversarial training [2], [37], [63], [80], [86], [87], [98] to enhance VERITRAIN, which makes the attack harder. We leave a thorough evaluation of such tradeoffs as our future work.

X. RELATED WORK

In this section, we discuss the related work of secure machine learning, which mainly falls in three areas: secure inference and secure training. We focus on protocols of securing neural networks instead of other classifiers (e.g., SVM). We also discuss attacks on DNN models.

Secure Inference: Many existing works focus on verifying the inference of neural networks, i.e., the inference results were generated on the given input and model correctly. These works aim to secure the ML inference, while our work focuses on verifying the training process. The approaches can be divided into two categories: cryptographic primitives and secure hardware.

1) Cryptographic primitives: There are a number of works that make use of cryptopgraphic approaches [11], [15], [22], [24], [25], [36], [38], [47], [54], [57], [60], [66], [68], [79], [81] to secure the ML inference, such as fully homomorphic encryption (FHE), zero-knowledge proof and secure multi-party computation (MPC). CryptoNets [38] is the first work that uses fully homomorphic encryption (FHE) for secure inference. There are many follow-up works [11], [15], [24], [47], [81] trying to optimize CryptoNets. SafetyNets [36] and vCNN [57] were proposed for verifiable DNN execution on untrusted cloud servers using Succinct Non-interactive ARguments of Knowledge (SNARK). There are also many papers utilize MPC techniques to secure ML inference [54], [60], [66], [68], [79]. However, due to the nature of cryptographic approaches, they suffer from large overheads.

2) Secure hardware (TEE): Researchers have also used secure hardware such as Intel SGX to verify the DNN inference while reducing the overheads. Tramèr and Boneh propose Slalom [90], which allows a TEE (e.g., SGX) to outsource the execution of all linear layers in a DNN to a co-located untrusted device. In this way, Slalom can offer verifiable inference while achieving a performance boost. The scenario is quite similar to ours, which uses TEEs to ensure the trustworthiness of the machine learning process. However, we consider the ML training process, instead of inference, and we use unsupervised ML and hypothesis testing to lower the performance overhead, instead of cryptographic approaches. There are other works that use TEE to secure the ML inference [44], [46], [49].

Secure training: Only a few works focus on securing the training process. Existing works mainly utilize cryptographic primitives such as MPC to secure the training of neural networks. SecureML [68] introduces secure MPC protocols for non-linear activation functions for neural network training and prediction. ABY^3 [67], SecureNN [96], and QUOTIENT [3] also devise customized MPC protocols to secure DNN training. VeriML [110] is a framework proposed for integrity assurances and fair payments in MLaaS by applying SNARK on randomly selected individual iterative training procedures. Zande [106] explores the use of zk-SNARK for efficient verification of outsourced ANN training. VerifyNet [101] is a framework for privacy-preserving and verifiable training of neural networks in the federating learning (FL) scenario. VerifyNet utilizes homomorphic hash functions to allow the server in a federated learning scheme to prove to the clients that the server has correctly aggregated the gradients of all online users. Although these methods can verify the ML execution from a theoretical perspective, their empirical overhead is prohibitive.

Closest to our work is TrustFL [108], which was recently proposed to ensure that participants in federated learning perform the training tasks as intended using TEEs. In TrustFL, the training is outsourced to the co-located GPU for performance, while the correctness is maintained by recomputing and checking randomly selected rounds inside TEEs. Compared to VERITRAIN, TrustFL incurs large overhead due to re-training the DNN inside TEEs, while VERITRAIN only incurs small overhead due to AE inferences and ADF tests. Moreover, TrustFL is specifically tailored to the FL scenario and not applicable to MLaaS.

Attacks on DNN Models: There are a lot of works trying to find more effective adversarial samples [13], [17], [21], [42], [63], [69], [77], [87] to fool deep neural networks. Some recent works propose mechanisms to generate adversarial images against Variational AutoEncoders (VAE) [40], [88] to make them generate incorrect images. In response, researchers have also introduced defenses to make AutoEncoders more robust against attacks [2], [37], [80], [86], [98]. Another category of attacks against ML models is the backdoor attack [8], [45], [99], [105], where the attacker manipulates the ML training process to implant triggers, to make the classifier misclassify samples with specific features. In our scenario, the goal of the attacker is to bypass the verification with *minimal* training effort, which is different from the goal of backdoor attacks.

XI. CONCLUSION

In this paper, we design and implement VERITRAIN, a framework running inside TEEs for verifying the *training efforts* of neural networks on the MLaaS cloud, which consists of AutoEncoders (AE) and Stationarity Examiners (SE). We consider a threat model where the *lazy* model trainer crafts artificial parameter updates by directly copying from a trained model or simulating them, in order to *minimize* the training effort. Through extensive evaluations, we show that VERITRAIN is effective in detecting these abnormal parameter updates while allowing legitimate training results to pass the tests. Even when the attacker knows VERITRAIN as a white-box, it is hard to generate adversarial samples to fool AEs and SEs at the same time. We further implement VERITRAIN with Intel SGX and demonstrate that its performance overhead is moderate.

REFERENCES

[1] M. Abadi et al., "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 265–283.

[2] G. Adam, H. Bryan, N. See Kiong, and N. Wee Siong, "Robustness of autoencoders for anomaly detection under adversarial impact," in *Proc. AAAI Conf. Artif. Intell. (Int. Joint Conf. Artif. Intell.*, 2020, pp. 1244–1250.

[3] N. Agrawal, A. Shahin Shamsabadi, M. J. Kusner, and A. Gascón, "QUOTIENT: Two-party secure neural network training and prediction," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 1231–1247.

[4] M. Amer, M. Goldstein, and S. Abdennadher, "Enhancing one-class support vector machines for unsupervised anomaly detection," in *Proc. ACM SIGKDD Workshop Outlier Detection Description*, 2013, pp. 8–15.

[5] S. Aminikhanghahi and D. J. Cook, "A survey of methods for time series change point detection," *Knowl. Inf. Syst.*, vol. 51, pp. 339–367, 2017.

[6] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proc. 2nd Int. Workshop Hardware Architectural Support Secur. Privacy*, 2013.

[7] M. Backes, M. Humbert, J. Pang, and Y. Zhang, "walk2friends: Inferring social links from mobility profiles," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 1943–1957.

[8] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov, "How to backdoor federated learning," in *Proc. Int. Conf. Artif. Intell. Statist.*, PMLR, 2020, pp. 2938–2948.

[9] Y. Bengio et al., "Learning deep architectures for AI," *Found. Trends Mach. Learn.*, vol. 2, no. 1, pp. 1–127, 2009.

[10] A. Bhargava, "On the theory of testing for unit roots in observed time series," *Rev. Econ. Stud.*, vol. 53, no. 3, pp. 369–384, 1986.

[11] F. Bourse, M. Minelli, M. Minihold, and P. Paillier, "Fast homomorphic evaluation of deep discretized neural networks," in *Proc. Annu. Int. Cryptol. Conf.*, 2018, pp. 483–512.

[12] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *Proc. 11th USENIX Workshop Offensive Technol.*, 2017.

[13] W. Brendel, J. Rauber, and M. Bethge, "Decision-based adversarial attacks: Reliable attacks against black-box machine learning models," 2017, *arXiv:1712.04248*.

[14] T. B. Brown et al., "Language models are few-shot learners," 2020, *arXiv:2005.14165*.

[15] A. Brutzkus, R. Gilad-Bachrach, and O. Elisha, "Low latency privacy preserving inference," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 812–821.

[16] E. J. Candès, X. Li, Y. Ma, and J. Wright, "Robust principal component analysis?," *J. ACM*, vol. 58, no. 3, pp. 1–37, 2011.

[17] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *Proc. IEEE Symp. Secur. Privacy*, 2017, pp. 39–57.

[18] Y. Chai, L. Du, J. Qiu, L. Yin, and Z. Tian, "Dynamic prototype network based on sample adaptation for few-shot malware detection," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 5, pp. 4754–4766, May 2023.

[19] S. Checkoway and H. Shacham, "Iago attacks: Why the system call API is a bad untrusted RPC interface," in *Proc. 18th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2013, pp. 253–264.

[20] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre: Stealing intel secrets from SGX enclaves via speculative execution," in *Proc. IEEE Eur. Symp. Secur. Privacy*, 2019, pp. 142–157.

[21] J. Chen, M. I. Jordan, and M. J. Wainwright, "HopSkipJumpAttack: A query-efficient decision-based attack," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 1277–1294.

[22] X. Chen, J. Ji, L. Yu, C. Luo, and P. Li, "SecureNets: Secure inference of deep neural networks on an untrusted cloud," in *Proc. Asian Conf. Mach. Learn.*, 2018, pp. 646–661.

[23] F. Chollet, "Keras: The python deep learning API," [Online]. Available: https://keras.io

[24] E. Chou, J. Beal, D. Levy, S. Yeung, A. Haque, and L. Fei-Fei, "Faster cryptonets: Leveraging sparsity for real-world encrypted inference," 2018, *arXiv:1811.09953*.

[25] M. Comiter, S. Teerapittayanon, and H. Kung, "Checknet: Secure inference on untrusted devices," 2019, *arXiv:1906.07148*.

[26] G. Contributors, "Building – graphene documentation - read the docs," [Online]. Available: https://graphene.readthedocs.io/en/latest/building.html

[27] G. Contributors, "Graphene examples - pytorch," [Online]. Available: https://github.com/oscarlab/graphene/blob/master/Examples/pytorch/pytorch.manifest.template

[28] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptol. ePrint Arch.*, vol. 2016, 2016, Art. no. 86.

[29] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.

[30] D. A. Dickey and W. A. Fuller, "Distribution of the estimators for autoregressive time series with a unit root," *J. Amer. Stat. Assoc.*, vol. 74, 1979, Art. no. 366.

[31] H.-G. Drost, "Philentropy: Information theory and distance quantification with R," *J. Open Source Softw.*, vol. 3, no. 26, 2018, Art. no. 765.

[32] D. M. Endres and J. E. Schindelin, "A new metric for probability distributions," *IEEE Trans. Inf. Theory*, vol. 49, no. 7, pp. 1858–1860, Jul. 2003.

[33] L. Fei-Fei, R. Fergus, and P. Perona, "One-shot learning of object categories," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 28, no. 4, pp. 594–611, Apr. 2006.

[34] P. Fryzlewicz et al., "Wild binary segmentation for multiple change-point detection," *Ann. Statist.*, vol. 42, no. 6, pp. 2243–2281, 2014.

[35] W. A. Fuller, *Introduction to statistical time series*. Hoboken, NJ, USA: Wiley, 1976.

[36] Z. Ghodsi, T. Gu, and S. Garg, "SafetyNets: Verifiable execution of deep neural networks on an untrusted cloud," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 4672–4681.

[37] P. Ghosh, A. Losalka, and M. J. Black, "Resisting adversarial attacks using Gaussian mixture variational autoencoders," in *Proc. AAAI Conf. Artif. Intell.*, 2019, pp. 541–548.

[38] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 201–210.

[39] I. Golan and R. El-Yaniv, "Deep anomaly detection using geometric transformations," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 9781–9791.

[40] G. Gondim-Ribeiro, P. Tabacof, and E. Valle, "Adversarial attacks on variational autoencoders," 2018, *arXiv:1806.04646*.

[41] D. Gong et al., "Memorizing normality to detect anomaly: Memory-augmented deep autoencoder for unsupervised anomaly detection," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2019, pp. 1705–1714.

[42] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," 2014, *arXiv:1412.6572*.

[43] Google, "Confidential computing - google cloud," [Online]. Available: https://cloud.google.com/confidential-computing

[44] K. Grover, S. Tople, S. Shinde, R. Bhagwan, and R. Ramjee, "Privado: Practical and secure DNN inference with enclaves," 2018, *arXiv:1810.00602*.

[45] T. Gu, B. Dolan-Gavitt, and S. Garg, "BadNets: Identifying vulnerabilities in the machine learning model supply chain," 2017, *arXiv:1708.06733*.

[46] L. Hanzlik et al., "MLCapsule: Guarded offline deployment of machine learning as a service," 2018, *arXiv:1808.00590*.

[47] E. Hesamifard, H. Takabi, and M. Ghasemi, "CryptoDL: Towards deep learning over encrypted data," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2016.

[48] M. Huh, P. Agrawal, and A. A. Efros, "What makes imagenet good for transfer learning?" 2016, *arXiv:1608.08614*.

[49] T. Hunt et al., "Telekine: Secure computing with cloud GPUs," in *Proc. 17th USENIX Symp. Netw. Syst. Des. Implementation*, 2020, pp. 817–833.

[50] Intel, "Which platforms support intel software guard extensions (intel SGX) SGX2?," [Online]. Available: https://www.intel.com/content/www/us/en/support/articles/000058764/software/intel-security-products.html

[51] Y. Kawahara and M. Sugiyama, "Change-point detection in time-series data by direct density-ratio estimation," in *Proc. SIAM Int. Conf. Data Mining*, SIAM, 2009, pp. 389–400.

[52] A. Krizhevsky, "Learning multiple layers of features from tiny images," Univ. Toronto, Toronto, Tech. Rep. TR-2009, 2009.

[53] S. Kullback and R. A. Leibler, "On information and sufficiency," *Ann. Math. Statist.*, vol. 22, no. 1, pp. 79–86, 1951.

[54] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow: Secure tensorflow inference," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 336–353.

[55] L. Labs, "OpenAI's GPT-3 language model: A technical overview," [Online]. Available: https://lambdalabs.com/blog/demystifying-gpt-3/

[56] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE Proc. IRE*, 1998.

[57] S. Lee, H. Ko, J. Kim, and H. Oh, "vCNN: Verifiable convolutional neural network," *IACR Cryptol. ePrint Arch.*, vol. 2020, 2020, Art. no. 584.

[58] E. L. Lehmann and J. P. Romano, *Testing Statistical Hypotheses*. Berlin, Germany: Springer Science & Business Media, 2006.

[59] J. Lin, "Divergence measures based on the shannon entropy," *IEEE Trans. Inf. Theory*, vol. 37, no. 1, pp. 145–151, Jan. 1991.

[60] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious neural network predictions via minionn transformations," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 619–631.

[61] J. G. MacKinnon, "Approximate asymptotic distribution functions for unit-root and cointegration tests," *J. Bus. Econ. Statist.*, vol. 12, no. 2, pp. 167–176, 1994.

[62] J. G. MacKinnon, "Critical values for cointegration tests" Queen's Economics Department Working Paper, Tech. Rep., 2010.

[63] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *Proc. Int. Conf. Learn. Representations*, 2018.

[64] F. McKeen et al., "Intel software guard extensions (intel SGX) support for dynamic memory management inside an enclave," in *Proc. Hardware Architectural Support Secur. Privacy*, *2016*, pp. 10:1–10:9.

[65] F. McKeen et al., "Innovative instructions and software model for isolated execution," in *Proc. 2nd Int. Workshop Hardware Architectural Support Secur. Privacy*, 2013.

[66] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A cryptographic inference service for neural networks," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 2505–2522.

[67] P. Mohassel and P. Rindal, "ABY: A mixed protocol framework for machine learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018.

[68] P. Mohassel and Y. Zhang, "SecureML: A system for scalable privacy-preserving machine learning," in *Proc. IEEE Symp. Secur. Privacy*, 2017, pp. 19–38.

[69] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "DeepFool: A simple and accurate method to fool deep neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 2574–2582.

[70] MordorIntelligence, "Machine learning as a service market| growth, trends, and forecast (2020–2025)," [Online]. Available: https://www.mordorintelligence.com/industry-reports/global-machine-learning-as-a-service-mlaas-market

[71] J. Neyman and E. S. Pearson, "Contributions to the theory of testing statistical hypotheses," *Stat. Res. Memoirs*, pp. 1–37, 1936.

[72] R. Nuzzo, "Scientific method: Statistical errors," *Nature News*, vol. 506, pp. 150–152, 2014.

[73] Nvidia, "NVIDIA confidential computing," [Online]. Available: https://www.nvidia.com/en-us/data-center/solutions/confidential-computing/

[74] S. J. Oh, M. Augustin, B. Schiele, and M. Fritz, "Towards reverse-engineering black-box neural networks," in *Proc. Int. Conf. Learn. Representations*, 2018.

[75] F. Österreicher and I. Vajda, "A new class of metric divergences on probability spaces and its applicability in statistics," *Ann. Inst. Stat. Math.*, vol. 55, pp. 639–653, 2003.

[76] S. G. Pantula, G. Gonzalez-Farias, and W. A. Fuller, "A comparison of unit-root test criteria," *J. Bus. Econ. Statist.*, vol. 12, no. 4, pp. 449–459, 1994.

[77] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *Proc. IEEE Eur. Symp. Secur. Privacy*, 2016, pp. 372–387.

[78] O. M. Parkhi, A. Vedaldi, and A. Zisserman, "Deep face recognition," 2015.

[79] D. Rathee et al., "CrypTFlow2: Practical 2-party secure inference," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2020, pp. 325–342.

[80] M. Salehi et al., "ARAE: Adversarially robust training of autoencoders improves novelty detection," 2020, *arXiv:2003.05669*.

[81] A. Sanyal, M. Kusner, A. Gascon, and V. Kanade, "TAPAS: Tricks to accelerate (encrypted) prediction as a service," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 4497–4506.

[82] B. Schölkopf et al., *Learning With Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. Cambridge, MA, USA: MIT Press, 2002.

[83] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using SGX to conceal cache attacks," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, Springer, 2017, pp. 3–24.

[84] A. J. Scott and M. Knott, "A cluster analysis method for grouping means in the analysis of variance," *Biometrics*, vol. 30, pp. 507–512, 1974.

[85] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.

[86] C. Sun, S. Chen, and X. Huang, "Double backpropagation for training autoencoders against adversarial attack," 2020, *arXiv:2003.01895*.

[87] C. Szegedy et al., "Intriguing properties of neural networks," 2013, *arXiv:1312.6199*.

[88] P. Tabacof, J. Tavares, and E. Valle, "Adversarial images for variational autoencoders," 2016, *arXiv:1612.00155*.

[89] A. Technica, "Intel's ice lake xeon comes out swinging at AMD's epyc milan," [Online]. Available: https://arstechnica.com/gadgets/2021/04/intels-ice-lake-xeon-comes-out-swinging-at-amds-epyc-milan/

[90] F. Tramèr and D. Boneh, "Slalom: Fast, verifiable and private execution of neural networks in trusted hardware," in *Proc. Int. Conf. Learn. Representations*, 2019.

[91] A. Trapletti, K. Hornik, and B. LeBaron, "Package 'tseries' - cran," [Online]. Available: https://cran.r-project.org/web/packages/tseries/index.html

[92] C. Truong, L. Oudre, and N. Vayatis, "ruptures: Change point detection in python," 2018, *arXiv:1801.00826*.

[93] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library os for unmodified applications on SGX," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 645–658.

[94] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 1041–1056.

[95] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on {GPUs}," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 681–696.
[96] S. Wagh, D. Gupta, and N. Chandran, "Securenn: 3-party secure computation for neural network training," *Proc. Privacy Enhancing Technol.*, vol. 2019, no. 3, pp. 26–49,2019.
[97] B. Wang and N. Z. Gong, "Stealing hyperparameters in machine learning," in *Proc. IEEE Symp. Secur. Privacy*, 2018, pp. 36–52.
[98] M. Willetts, A. Camuto, T. Rainforth, S. Roberts, and C. Holmes, "Improving VAEs' robustness to adversarial attack," 2019, *arXiv:1906.00230*.
[99] C. Xie, K. Huang, P.-Y. Chen, and B. Li, "DBA: Distributed backdoor attacks against federated learning," in *Proc. Int. Conf. Learn. Representations*, 2019.
[100] L. Xiong, B. Póczos, and J. G. Schneider, "Group anomaly detection using flexible genre models," in *Proc. Adv. Neural Inf. Process. Syst.*, 2011, pp. 1071–1079.
[101] G. Xu, H. Li, S. Liu, K. Yang, and X. Lin, "VerifyNet: Secure and verifiable federated learning," *IEEE Trans. Inf. Forensics Secur.*, vol. 15, pp. 911–926, 2020.
[102] H. Xu et al., "Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications," in *Proc. World Wide Web Conf.*, 2018, pp. 187–196.
[103] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Proc. IEEE Symp. Secur. Privacy*, 2015.
[104] X. Yang, K. Huang, and R. Zhang, "Unsupervised dimensionality reduction for Gaussian mixture model," in *Proc. Int. Conf. Neural Inf. Process.*, Springer, 2014, pp. 84–92.
[105] Y. Yao, H. Li, H. Zheng, and B. Y. Zhao, "Latent backdoor attacks on deep neural networks," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 2041–2055.
[106] M. Zande, "Leveraging zero-knowledge succinct arguments of knowledge for efficient verification of outsourced training of artificial neural networks," Master's thesis, University of Twente, 2019.
[107] S. Zhai, Y. Cheng, W. Lu, and Z. Zhang, "Deep structured energy based models for anomaly detection," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 1100–1109.
[108] X. Zhang, F. Li, Z. Zhang, Q. Li, C. Wang, and J. Wu, "Enabling execution assurance of federated learning at untrusted participants," in *Proc. IEEE Conf. Comput. Commun.*, 2020.
[109] Z. Zhang, Y. Song, and H. Qi, "Age progression/regression by conditional adversarial autoencoder," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 4352–4360.
[110] L. Zhao et al., "VeriML: Enabling integrity assurances and fair payments for machine learning as a service," 2019, *arXiv:1909.06961*.
[111] C. Zhou and R. C. Paffenroth, "Anomaly detection with robust deep autoencoders," in *Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2017, pp. 665–674.

Xiaokuan Zhang received the graduate degree from Ohio State University, in 2021. He spent one year as a postdoctoral researcher with the Georgia Institute of Technology. He is an assistant professor with the Department of Computer Science, George Mason University. His research interests mainly lie in system security and privacy, such as side channels, mobile security and blockchain security.

Yang Zhang is a faculty member with CISPA Helmholtz Center for Information Security, Germany. His research concentrates on trustworthy machine learning; he also works on measuring and understanding misinformation and unsafe content like hateful memes on the Internet. He has published multiple papers with top venues, including CCS, NDSS, Oakland, and USENIX Security. His work has received the NDSS 2019 distinguished paper award and the CCS 2022 best paper runner-up.

Yinqian Zhang is a professor with the Southern University of Science and Technology (SUSTech). Prior to joining SUSTech, he was an associate professor with Ohio State University. His research aims to secure computer systems and his recent research interest lies in the application and security of confidential computing.