Enabling Execution Assurance of Federated Learning at Untrusted Participants

Xiaoli Zhang*†, Fengting Li*†, Zeyu Zhang*†, Qi Li*†, Cong Wang‡, and Jianping Wu*†
*Institute for Network Sciences and Cyberspace & Department of Computer Science and Technology, Tsinghua University
†Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University
‡City University of Hong Kong

***Abstract*—Federated learning (FL), as a privacy-preserving machine learning framework, draws growing attention in both industry and academia. It obtains a jointly accurate model by distributing training tasks into data owners and aggregating their model updates. However, FL faces new security problems, as it losses direct control to training processes. One fundamental demand is to ensure whether participants execute training tasks as intended.**

In this paper, we propose TrustFL, a practical scheme that leverages Trusted Execution Environments (TEEs) to build assurance of participants' training executions with high confidence. Specifically, we use TEE to randomly check a small fraction of all training processes for tunable levels of assurance, while all computations are executed on the co-located faster yet insecure processor (e.g., GPU) for efficiency. To prevent various cheating behaviors like only processing TEE-requested computations or uploading old results, we devise a commitment-based method with specific data selection. We prototype TrustFL using GPU and SGX and evaluate its performance. The results show that TrustFL achieves one/two orders of magnitude speedups compared with naive training with SGX, when assuring correct training with a confidence level of 99%.

I. Introduction

Federated learning (FL) is recently proposed to obtain a high-quality collaborative deep neural network (DNN) model without compromising data privacy [1], [2]. Different from traditional centralized training, it distributes training tasks into individual data owners and aggregates their locally-computed model updates as the global model. Such a framework can be widely applied in multifarious application domains, where one user without data wants to buy an accurate model and data owners are incentivized to contribute data yet have privacy concerns or are restricted by privacy regulations like EU GDPR [3]. Take the diagnosis modeling scenario as an example [4], [5]. The medical researchers who need a precise disease predictive model can initiate a learning task with bounty. The clinics/hospitals with a large number of electronic health records can participate in the task, train the model locally, submit model updates, and reap rewards.

However, FL lacks visibility to participants' local training processes, making it vulnerable to misbehaving participants that do not execute training tasks as intended. The misbehaviors may reduce the model accuracy or impede the model convergence. Especially, in an FL-based marketplace [6], it may cause severe economic loss, as participants can defraud rewards with no/less training efforts. Regarding these issues, there are some defense solutions [7]–[9]. For example, they remove/weaken outliers from model updates of participants, so as to prevent model poisoning attacks [7], [9]–[11] or guarantee the convergence of the global model [8].

Unfortunately, the above defenses only concentrate on eliminating markedly different training results mainly incurred by bad input data. None of them considers whether local training processes are correctly executed, which is of equal importance in the FL system with reasons. First, assuring correct execution at participants can be combined with the above solutions [7]–[9], so as to build a trustworthy FL system resisting against data poisoning attacks as well as some stealthy backdoor attacks [12], [13]. Second, it can promote the widespread application of the FL-based marketplace, since it enables aligned transactions by preventing lazy participants with no/less training efforts. However, to our best knowledge, few efforts have explored this problem.

In this paper, we aim to build training assurance of untrusted participants in the FL framework. One pragmatic solution is to leverage hardware-assisted trusted execution environments (TEEs), which shield data and code in an isolated environment (i.e., an enclave) and attest to the correct execution. Compared with software-based alternatives of verifiable computation [14]–[16], this technique is more efficient with several orders of magnitude less overhead. When using TEE in an FL task, data owners with TEE-enabled platforms participate in the task, review the training code, execute it in an enclave, and output model updates with a signature as an attestation. The server checks the correctness of the attestation, distributes rewards, and aggregates the successfully verified model updates as a new global model. The FL task proceeds over iterative epochs until the model converges.

Unfortunately, directly executing all DNN training processes inside the TEE leads to extreme performance degradation. For example, Intel SGX [17], as a typical implementation of TEE that has been widely integrated in commercial Intel processors, is only supported by CPUs. It is about one or two orders of magnitude lower than current processors running DNN (i.e., GPUs) [18]. Meanwhile, today SGX-enabled CPUs are restricted to 128MB Processor reserved memory (PRM) (about 90MB available to applications) [17]. When requiring memory beyond PRM limits, they induce significant

paging overheads [19]. In addition, while the program inside the enclave cannot directly access OS services like system calls, in-enclave system calls incur enclave/non-enclave mode transitions which also result in performance slowdown [19].

Therefore, one critical demand is to enable local training assurance without sacrificing too much performance at the participant side. We envision to randomly verify a small number of training rounds in the TEE for both security and efficiency, since the entire training computation consists of multiple training rounds and each handles a small batch of data. Specifically, we outsource all rounds of training to the co-located GPU processor, whose correctness is ensured by recomputing and checking the random-selected rounds of training in the TEE.

Though promising, implementing the above idea in a platform fully manipulated by untrusted participants is still a non-trivial task. First, direct exposure of sampling decisions may enable participants to evade detection by only executing TEE-requested training rounds. Second, the freshness of training results cannot be guaranteed, as TEE's scheduling and I/O are controlled by the host participant [20]. For example, in the iterative local training procedure, dishonest participants can leverage a small fraction of correctly training results to forge the results of all training rounds. Also, they can use old results with corresponding correct proofs in the previous FL epoch to claim rewards. In addition, they may launch sybil-based cheating attacks to ask for multiple rewards.

To address the first issue, we adopt a *"commit-and-prove"* design. Before exposing the sampling decisions, the TEE requires to receive a commitment message from the participant. The commitment can indicate that all training rounds have finished. Meanwhile, the results of arbitrary training rounds can be efficiently verified later according to the commitment. As for the issue of the freshness, we devise a *"dynamic-yet-deterministic" data selection* method that leverages dynamic input data to make training results of any two rounds different. Such dynamics can be inferred by the TEE in the verification, consequently, preventing replay attacks with previous training results. At the meantime, we bind the selecting decision with each FL epoch's identifier, enabling the server to check the freshness of final model updates. Moreover, we apply *tamper-free identifiers* to participants against sybil attacks. It enforces the bounty distribution proportional to training efforts.

To summarize, our contributions are as follows:

- We propose a practical scheme TrustFL that combines TEE and GPU to efficiently enforce participants' training executions with high confidence in the FL framework.
- Our scheme is full-fledged and leverages the delayed sampling with commitment and dynamic-yet-deterministic data selection techniques to resist against various pitfalls.
- The theoretical analysis demonstrates that our scheme can achieve high-confidence training assurance with sparse sampling. We prototype the TrustFL and estimate the performance using canonical DNNs (VGG16, VGG19, and Resnet50). The results show that TrustFL achieves one/two orders of magnitude speedups than naively using SGX.

II. Problem Statement

A. Preliminaries

Federated learning. Federated learning is a form of distributed and collaborative machine learning which has many different implementations regarding various data distribution and participant scale [1], [2], [21]. Our work aims at assuring DNN training of participants and is compatible with arbitrary FL implementations. In this paper, we take a concrete implementation introduced by Google as an example to elaborate on our scheme.

Assume that there are participants $\mathcal{P}$ with sensitive data D, and one server $\mathcal{S}$ that is responsible for coordinating distributed learning tasks. To obtain a converged global model θ, the server performs multi-iteration training. At each epoch, e.g., t, the server selects a random subset of participants $\mathcal{P}^t \subseteq \mathcal{P}$ and sends them the latest global model θ^t. Each participant p generates the local model θ_p^{t+1} over local training data D_p^t, and lupoads the local model updates, denoted as $\mathcal{L}_p^t = \theta_p^{t+1} - \theta^t$. With all received updates, the server derives the new global model by performing a weighted average as:

$$\theta^{t+1} = \theta^t + \eta \Sigma_{p \in \mathcal{P}^t} \frac{n_p}{n} \mathcal{L}_p^t, \tag{1}$$

where η is the global learning rate that instructs how much the model updated at every round, n is the total data size and equals to $\Sigma_{p \in \mathcal{P}^t} |D_p^t|$. Note that, the number of training rounds is determined by the time when the model converges, e.g., achieving a target test-set accuracy [21].

SGX. Intel's Software Guard Extension (SGX) [22], [23] only requires trust int the processors and provides integrity and confidentiality guarantee for the execution of user-level code. Specifically, SGX is a set of new instructions supported by Intel Skylake CPUs that creates a protected physical memory known as an enclave. The enclave forbids any other process to access enclave memory (called Enclave Page Cache, i.e., EPC). Yet, it can read and write untrusted memory outside the EPC.

SGX supports remote attestation to prove that the software is running correctly in the enclave. When an enclave is launched with the program code, the processor generates a *measurement* which is a hash digest of the loaded binary. Then, the program in the enclave produces a report including the measurement and supplementary data (e.g., output), and signs it as a *quote* with the private attestation key fused in the processor. The quote as an attestation can be verified by anyone possessing a trustworthy measurement of the binary and accessing Intel's Attestation Service (IAS) [24]. For the sake of simplicity, we follow the formal model of SGX presented in [25] where each enclave instance produces an attestation key pair $(\mathsf{sk}_{\mathsf{TEE}}, \mathsf{pk}_{\mathsf{TEE}})$ and Σ is the existentially unforgeable signature scheme. An enclave can generate an attestation by calculating a digital signature $\sigma = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathsf{TEE}}, \Phi, \mathsf{out})$, as a proof that the output out is produced by the program Φ running in the enclave. The attestation can be further verified by checking whether $\Sigma.\mathsf{Veri}(\mathsf{pk}_{\mathsf{TEE}}, \sigma, \Phi, \mathsf{out})$ outputs 1.

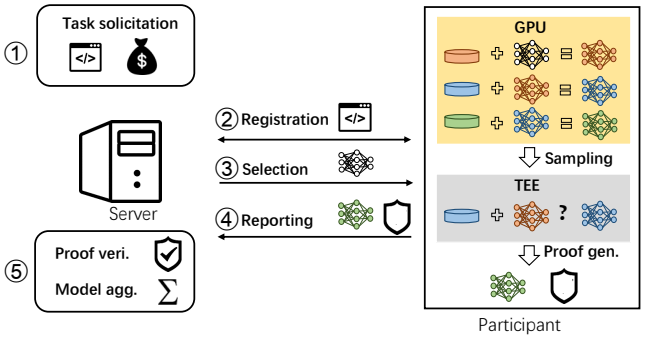

Fig. 1. High-level workflow of TrustFL. Participants leverage TEE and GPU to fast prove the correctness of local training execution. The task owner verifies the proofs before aggregating their results and distributing rewards.

B. Threat Model

We consider a scenario where a task owner desires to obtain a high-quality DNN model yet without substantial training data. He publishes his requirements in an open data marketplace platform, e.g., building upon blockchain [6], [26], and initiates a crowd-sourced federated learning task with bounty. Participants with appropriate data can join the task to realize the economic value of local data and computation resources while preserving data privacy.

• Task owner is curious-but-honest. He faithfully coordinates the overall FL procedure without any deliberate disruption, as he wants to obtain the aggregated DNN model trained by participants. Yet, he is curious about sensitive data of individuals, e.g., for the purpose of abusing data for profits. Here, we adopt the federated learning framework which inherently eliminates the threat of breaching data privacy. Other membership inference attacks [27], [28] that indirectly steal raw training data from models are out of the scope of this paper. They can be well addressed by differential privacy [29], [30] and secure multi-party computation based aggregation techniques [31], which are complementary to our work.

• Task participants are dishonest and want to defraud profits with as less as possible efforts. They may 1) return arbitrary model updates without local training enforcement; 2) perform less training computation, for example, by replacing the original DNN algorithm with a simple one, modifying the hyperparameters like learning epochs, or using less data; 3) launch sybil-based cheating, i.e., use one training efforts to claim multiple rewards.

Our security objective is to build the assurance of training execution at the participant side and increase the detection probability of the above cheating behaviors. We mitigate sybil attacks by diminishing the gains of adversaries with increased computing efforts. We do not prevent untrusted participants from disrupting DNN training results intentionally, e.g., slightly modifying several intermediate values. Nevertheless, we argue that participants have no incentive to make the deviation, and our work can detect such misbehaviors once they are sampled.

III. Overview

We desire to build training assurance at participants in the FL-based marketplace. In particular, we employ the trusted computing (i.e., Intel SGX) and GPU to fast ensure the faithful training execution. Figure 1 shows the high-level workflow of TrustFL with the following five stages:

1) *Solicitation:* The task owner who wants to obtain an accurate model initiates a crowd-sourced learning task on an open platform, calling for contributions from the public. It specifies the training algorithm, rewards, and other instructions like setup tutorials. In this paper, we assume the task owner maintains a central server $\mathcal{S}$ to coordinate the overall FL procedure. Notably, other decentralized alternatives like using blockchain [6], can adopt our scheme. In addition, similar to [32], we assume the training algorithm can be reviewed and checked by participants, so as to convince them that the training code would not steal their data.

2) *Registration:* The interested participants $\mathcal{P}$ with appropriate data and SGX-enabled platform can register themselves to the task with necessary information like data description and identifier. Then, $\mathcal{P}$ reviews and downloads the program, launches an enclave with the program, and generates an attestation report to demonstrate the correct program setup. $\mathcal{S}$ verifies the correctness of the attestation by invoking $\Sigma.\mathsf{Veri}()$. At the end of this stage, there are n participants in the task without any registration permitted. Here, we assume $\mathcal{S}$ and $\mathcal{P}$ can build bidirectional connections to orchestrate successive processes.

3) *Selection:* After the registration, an iterative learning procedure proceeds. At each epoch (e.g., epoch t), $\mathcal{S}$ selects a subset of participants to perform training tasks. In our system, we adhere to the strategy of original FL [29], [33], where participants are randomly selected with a sampling probability. For unselected participants, $\mathcal{S}$ would instruct them when they can reconnect for participation [33].

4) *Reporting:* $\mathcal{P}$ gets the latest global model θ_0^t, performs local training with the program $\mathsf{DNN}()$ and data D using GPU, generates model update $\mathcal{L}_{\mathcal{P}}^t$, and invokes the program inside the TEE to generate a proof $\sigma_{\mathcal{P}}$. More precisely, the TEE runs the following algorithm: $\mathsf{Prove}(\mathsf{sk}_{\mathsf{TEE}}, \mathsf{DNN}(), D, \theta_0^t, \mathcal{L}_{\mathcal{P}}^t) \rightarrow \sigma_{\mathcal{P}}$, which verifies the outside training execution by checking random-selected training rounds.

5) *Verification and Aggregation:* As model updates are sent from participants, $\mathcal{S}$ performs the function: $\mathsf{Verify}(\mathsf{pk}_{\mathsf{TEE}}, \mathsf{DNN}(), \theta_0^t, \mathcal{L}_{\mathcal{P}}^t, \sigma_{\mathcal{P}}) \rightarrow 1/0$, so as to confirm whether the participant $\mathcal{P}$ honestly conducts the intended training with the high-probability assurance. If $\mathsf{Verify}()$ returns 1, $\mathcal{S}$ distributes rewards to corresponding participants and aggregates these updates for the new global model as shown in Equation (1). If the global model converges, $\mathcal{S}$ ends the learning task, otherwise, go back to the third stage.

Challenges. While proofs generated by TEE can be easily verified using the SGX attestation scheme, how to fast generate proofs using TEE while resisting against various attacks is challenging:

TABLE I
NOTATIONS

(I, d, H)	(index, data content, HMAC) for one data
D_i	a batch of training data for round i
λ	security parameter
θ_0^t	global model at the training epoch t
θ_i^t, h_i^t	model parameter and its hash value at training round i (for brevity, we omit the superscript t)
R	the number of training rounds at each epoch
l	the number of sampled training rounds
v_i	random number in [1,R]
B	the number of training data in a batch
N	the number of the total training data at one participant
T	random nonce of enclave as the participant identifier
ζ	the commitment message
Φ	a program running in the enclave
hash()	one-way cryptographic hash function
SNG()	SGX random number generator
PRF()	pseudo-random function
DNN()	DNN training algorithm
Σ	existentially unforgeable signature scheme

• Straightforward sampling enables the participant to cheat by merely executing the TEE-requested rounds of training and omitting others. To fix this problem, we adopt a "commit-and-prove" design. It guarantees that all training rounds are completed before sampling (see Section IV-A).

• TEE is not a ready-made panacea, since its scheduling and I/O are controlled by the host participant. These flaws make our scheme hard to guarantee the freshness of training results in the iterative FL procedure. Regarding these issues, we devise a comprehensive countermeasure with the key technique of "dynamic-yet-deterministic" data selection (see Section IV-B).

IV. DESIGN DETAILS

In this section, we present how to efficiently verify whether participants honestly perform local training. We outsource all training rounds outside the enclave, and ensure the training correctness by checking the random-selected rounds inside the enclave. Below, we introduce our baseline design with delayed sampling and commitment, and present how to defend against pitfalls caused by unreliable TEE scheduling and I/O. We denote the programs running outside the enclave and inside the enclave as Prog_o and Prog_i, respectively. All other notations used in the paper are listed in Table I.

A. Baseline Design

As described in Section III, we follow the *"commit-and-prove"* paradigm. The program outside the enclave Prog_o stores the hash digests of all model parameters θs at the end of each training round, and sends a commit message to the TEE when all training rounds finish. Notably, the reason for keeping hash digests instead of raw model parameters is that the size of model parameters is large. Storing all of them may introduce forbidden storage overhead. For instance, the size of VGG16 model parameters [34] is about 57MB, if executing 1000 training rounds could consume 57GB for model parameters, contrastively 32MB for model hashes generated by SHA256. All model parameters of training rounds can be recomputed starting from the global model of the current epoch. In addition, to reduce such the computation overhead, Prog_o can record several intermediate parameters as checkpoints at the cost of suitable storage overhead.

When Prog_i requests to check one random training round, Prog_o returns the input model parameters, the hash value of the output parameters, and the corresponding training data. Then, Prog_i checks whether these parameters are consistent with the commitment, executes the training with the corresponding data and input parameters, and compares the hash values of their output model parameters with those transferred from the outside. The successful verification of all randomly selected training rounds ensures that the participant have honestly performed all training rounds with a high confidence. Finally, Prog_i requests the model parameters of the last round, checks with the commitment message, computes the model updates (i.e., a vector of the changes in all parameters between this last output model and the global model), generates a proof as the attestation of the local execution and this updates (see line 24-29 in Figure 2). Otherwise, Prog_i returns an error. A valid proof would be further verified by the server.

Design of the commitment scheme. Particularly, the above commitment scheme should meet two requirements: 1) the commit message should include the results of all training rounds and keep the order of these results; 2) the arbitrary result can be verified according to the commitment efficiently.

One straightforward method (named as SH-based method) is to commit a sequent of hash digests of model parameters, i.e., $\zeta = \{h_1, h_2, \cdots, h_R\}$, where R is the number of total training rounds[1]. In verification, Prog_i randomly fetches several pairs of two successive hash values, requests raw input model parameters from the Prog_o, and recomputes their hash values to ensure the coincidence with the commitment. However, the performance of this method is tightly dependent on the number of total training rounds. Concretely, the time of loading all hash values into the SGX memory-constrained environment increases proportionally as R grows. More importantly, randomly fetching several hashes may also trigger the SGX paging mechanism and result in the slowdown, if the size of the commitment message exceeds PRM limits.

To gain constant performance inside the SGX regardless of the number of training rounds, we adopt the Merkle hash tree [35] based commitment method (denoted as MHT-based method). Specifically, the Prog_o builds a Merkle tree, a complete binary tree, where leaf nodes are hash values of model parameters in the order. The internal nodes are the hash values of the concatenation of their two children. The tree root is submitted to Prog_i as the commitment. Later, Prog_i generates several random numbers to identify the training rounds to be verified. Then Prog_i requests the corresponding input parameters, the hash value of the output parameters, and auxiliary information to check the correctness of the commitment. Note that, the auxiliary information consists of

[1] R can be derived from the number of local training data N, the number of training epochs E, and the batch size B, i.e., $R = \frac{EN}{B}$. The latter two are hyperparameters specified by the server.

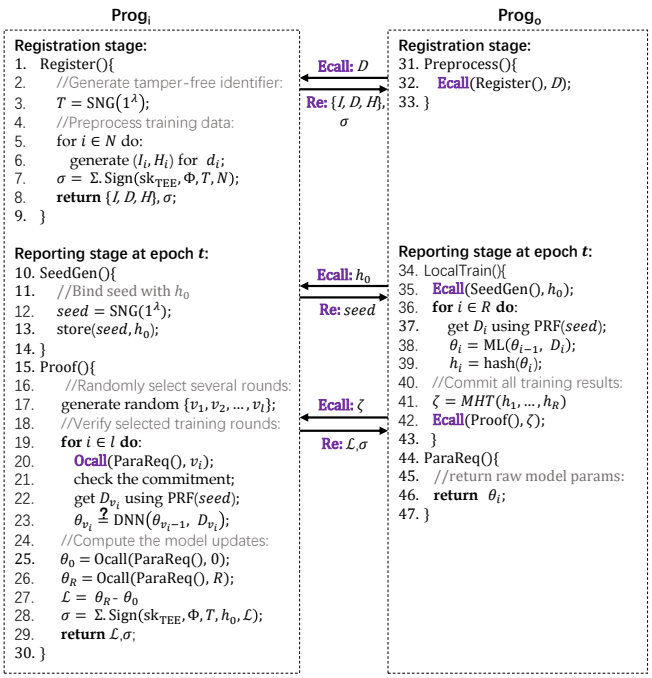

Fig. 2. The algorithm for assuring local training execution with TEE. All training rounds are conducted by the Prog$_o$ at the host participant. Several random-selected rounds are recomputed and checked by the Prog$_i$.

sibling nodes of the nodes in the path from the requested leaf node to the tree root.

Compared to the SH-based method, the storage overhead of MHT-based method in the SGX reduces to $O(1)$ from $O(n)$ which would never induce SGX paging overhead. Yet, it requires more hash operations with the complexity of $O(\log n)$ when checking the commitment. Fortunately, such the computation overhead is negligible, as executing the hash function with the small input is extremely lightweight. Therefore, in the paper, we adopt the MHT-based commitment scheme.

B. Guaranteeing The Freshness of Training Results

Since TEE's scheduling and I/O relies on the untrusted host, our scheme is hard to guarantee the freshness of a single training round and final results. Specifically, the Prog$_o$ may perform a small portion of training rounds (e.g., r rounds) and replicate these to camouflage all training rounds. The detection probability for such the misbehavior is small and equals to $(1 - \binom{R-\lceil(\frac{R}{r})\rceil+1}{l} / \binom{R}{l})$ where l is the number of random-selected rounds. For example, if R is 100, r is 10, and l is 5, the detection probability is about 38.23% which is quite low. Meanwhile, in the overall iterative FL procedure, the Prog$_o$ may not invoke Prog$_i$ to generate proofs, yet return previous training results and the corresponding proof to the task owner to defraud rewards. In addition, the participant may create dummy parties and mount sybil attacks for the purpose of over-claiming rewards.

To ensure the freshness of training results, we leverage the dynamics of input training data, which further makes the results of any two rounds or model updates of two epochs differentiable. In detail, we slightly modify the original data selection method in DNN algorithms ("shuffle then per-batch selection") [2] and use *dynamic-yet-deterministic data selection* in each training round. To realize this, Prog$_i$ should negotiate with Prog$_o$ about data organization in advance. One naive method is to load all training data into the enclave, similar to existing TEE-protected machine learning works [32], [36], [37]. With the same data, Prog$_i$ and Prog$_o$ use same selection function to determine the training data in each round. However, this method may trigger frequent paging when fetching data to train from the entire dataset, and further degrade the performance of subsequent execution inside the SGX due to the shrunk memory space.

Our key observation is that, different from these prior arts [32], [37] protecting both confidentiality and integrity of data, we only require to guarantee the data identifiability and integrity. Consequently, we retain all data on the outside of the enclave and perform efficient data preprocess to meet our requirement. Specifically, Prog$_i$ generates an index and HMAC for each training data which would be restored outside the enclave before training. When conducting local training, Prog$_o$ requests a seed from Prog$_i$ and each training data at each round (e.g., i) is selected as follows:

$$I_{ij} = \mathsf{PRF}(seed \times i + j) \bmod N, \tag{2}$$

where I_{ij} is the index of the jth training data at round i, PRF is a pseudo-random function, and N is the number of total training data. Therefore, training data is different at each round and any replay attack is infeasible. When checking the correctness of selected training rounds, Prog$_i$ derives which data should be used, checks the integrity of the outside data with HMAC, trains the DNN, and verifies whether the output parameters equal to the hash value generated by the outside.

Unfortunately, merely leveraging Prog$_i$ may not be sufficient to attest to the freshness of the final model updates, as it is oblivious to whether the used global model is the latest. Here, we bind the training results with the used global model, and resort to the server to check the freshness of training results. Before generating the $seed$, the Prog$_i$ requires to receive the digest of the current global model (denoted as h_0^t) from the Prog$_o$ (see line 10-15 in Figure 2). h_0^t will be contained in the proof of TEE (see line 28 in Figure 2). Notably, these operations implicitly bind the h_0^t with the seed, and consequently bind with the final model updates. Finally, by checking the signature and the model digest, the server can ensure whether the participant used the latest model and faithfully performed the local training.

As for the sybil-based cheating where adversaries desire to use one training result to claim several-fold bounties at each epoch, we eliminate it by enforcing attackers to consume computing resources proportionally to the bounties. We easily use a unique random nonce T produced by the enclave as the *tamper-free identifier* for the participant (this treatment is similar to [26]), as shown in the line 3 of Figure 2. In this way, even if an adversary registers multiple aliases, he must launch the same number of enclave instances to generate proofs for

local training. If he wants to request multiple rewards using one set of training rounds, he requires that all registered enclave instances produce valid proofs for this training set. That means, all enclave instances have to generate the same $seed$. This probability is extremely small which equals to $\frac{1}{2^{|seed|(m-1)}}$ where m is the number of aliases. The overall procedure is shown in Figure 2.

V. Theoretical Analysis

A. Security Analysis

We prove the correctness and soundness of our scheme.

Theorem 1 (Correctness)**.** *The scheme is said to be correct, if for all key pairs* $(\mathsf{sk}_{\mathsf{TEE}}, \mathsf{pk}_{\mathsf{TEE}})$ *generated by enclaves, all training algorithm* $\mathsf{DNN}()$*, all dataset* D*, all global model* θ_0*, all model updates* $\mathcal{L}_{\mathcal{P}}$ *produced by* $\mathsf{DNN}(\theta_0, D)$*, we obtain* $\mathsf{Verify}(\mathsf{pk}_{\mathsf{TEE}}, \mathsf{DNN}(), \theta_0, \mathcal{L}_{\mathcal{P}}, \sigma_{\mathcal{P}}) \rightarrow 1$ *where* $\sigma_{\mathcal{P}}$ *is produced by* $\mathsf{Prove}(\mathsf{sk}_{\mathsf{TEE}}, \mathsf{DNN}(), D, \theta_0, \mathcal{L}_{\mathcal{P}})$.

Proof. The correctness of our scheme depends on the correctness of SGX. If the participant honestly performs all training rounds, for any random-selected training rounds, $\mathsf{Prog_i}$ in the TEE must produce the output model parameters equaling to those generated by $\mathsf{Prog_o}$. Their hash digests are same as well. With all successfully verified training rounds, TEE would attest to the faithfulness of all training rounds by generating a signature. This signature would be checked by the server following the attestation functionality of SGX. ■

Theorem 2 (Soundness)**.** *The scheme is said to be sound, if for all key pairs* $(\mathsf{sk}_{\mathsf{TEE}}, \mathsf{pk}_{\mathsf{TEE}})$ *generated by enclaves, all training algorithm* $\mathsf{DNN}()$*, all dataset* D*, all global model* θ_0*, any probabilistic polynomial-time adversary* Adv*, the following inequality holds,*

$$\Pr\left(\begin{array}{c}\{\mathcal{L}^*_{\mathcal{P}}, \sigma_{\mathcal{P}}\} \leftarrow \mathsf{Adv}(\mathsf{pk}_{\mathsf{TEE}}, \mathsf{DNN}(), D, \theta_0); \\ 1 \leftarrow \mathsf{Verify}(\mathsf{pk}_{\mathsf{TEE}}, \mathsf{DNN}(), \theta_0, \mathcal{L}^*_{\mathcal{P}}, \sigma_{\mathcal{P}}); \\ \mathcal{L}^*_{\mathcal{P}} \neq \mathcal{L}_{\mathcal{P}}\end{array}\right) \leq \epsilon. \quad (3)$$

where $\mathcal{L}_{\mathcal{P}}$ *is the correct result of* $\mathsf{DNN}(\theta_0, D)$*,* ϵ *is the soundness error and less than* $(\frac{r}{R}+(1-\frac{r}{R})q)^l$*,* r *is the number of honestly training rounds of* Adv*,* l *is the number of random-selected rounds,* q *is the probability that* Adv *can successfully give a qualified input model parameters without training.*

Proof. If an adversary wants to prove its honesty on all training rounds, for each sampled training round, he must give the correct input parameters to $\mathsf{Prog_i}$. It means that he either performs the selected round or successfully guesses the qualified parameters before committing training results to $\mathsf{Prog_i}$. This property is held by the security of one-way hash functions used in MHT construction and commitment verification, as demonstrated in [38].

Randomly sampling l rounds from the total R training rounds, there are $\binom{R}{l}$ choices. Given $(R-r)$ rounds without training, the expectation that $\mathsf{Prog_o}$ can successfully cheat equals to $\sum_{i=0}^{R-r}\binom{r+i}{l}\binom{R-r}{i}q^n(1-q)^{R-r-i}$. Consequently, the cheating probability P_c (i.e., the soundness error ϵ) is:

$$\epsilon = \frac{\sum_{i=\max(l-r,0)}^{R-r}\binom{r+i}{l}\binom{R-r}{i}q^i(1-q)^{R-r-i}}{\binom{R}{l}}. \quad (4)$$

The above P_c is derived under the condition where l samples selected from R rounds are mutually different, thus, it is less than that calculated where l samples can be same in $[1, R]$, as follows:

$$\epsilon < (\frac{r}{R} + (1-\frac{r}{R})q)^l. \quad (5)$$

■

Given a soundness error ϵ, we can derive the lower bound of the number of samples:

$$l \geq \frac{\log \epsilon}{\log(\frac{r}{R} + (1-\frac{r}{R})q)}. \quad (6)$$

As an example, if ϵ is 1%, $\mathsf{Prog_o}$ faithfully executes 100 training rounds from 1000 rounds. Note that while one DNN model usually maintains tens of thousands to millions of parameters, we assume that q is 0.1% which is larger than the actual value. We only need 2 samples. If $\mathsf{Prog_o}$ performs 90% of training rounds, we require 43 samples to guarantee the soundness.

B. Performance Analysis

We theoretically analyze the performance of our scheme and compare with training on GPU, CPU, and SGX. Assume the time of training one batch of data on GPU, CPU, and SGX is t_{GPU}, t_{CPU}, t_{SGX}, respectively. If performing R rounds of training, the total time on GPU, CPU, and SGX is about $T_{\mathsf{GPU}} = R \cdot t_{\mathsf{GPU}}$, $T_{\mathsf{CPU}} = R \cdot t_{\mathsf{CPU}}$, and $T_{\mathsf{SGX}} = R \cdot t_{\mathsf{SGX}}$. In TrustFL, the overall time is $T_{\mathsf{TFL}} = (R \cdot t'_{\mathsf{GPU}} + l(t_{\mathsf{M}} + t_{\mathsf{SGX}} + t_C))$, where t'_{GPU} is the time of our customized training on GPU, t_{M} is the model initializing time in SGX, t_C is the time of verifying the commitment.

We can see that, T_{TFL} includes two parts: T'_{GPU} (i.e., $R \cdot t'_{\mathsf{GPU}}$) and T'_{SGX} (i.e., $l(t_{\mathsf{M}} + t_{\mathsf{SGX}} + t_C)$). Particularly, while l is constant for a given soundness error and honest ratio (i.e., $\frac{r}{R}$) (see Equation (6)), T'_{SGX} is constant for varied R ($R > l$). Therefore, as R grows, T_{TFL} is first dominated by T'_{SGX}, then by T'_{GPU}. It means that, for a large R, TrustFL can obtain comparable performance with that using GPU. If the performance of training on GPU can achieve two orders of magnitude speedups than that with SGX, TrustFL can be about $100\times$ faster than that with SGX. We demonstrate this in Section VI-B.

VI. Implementation and Evaluation

A. Implementation of TrustFL

Task participant side: We prototype TrustFL at participant side using C++ and Python.

$\mathsf{Prog_o}$: We implement the DNN training algorithm outside the enclave with Python based on the TensorFlow framework[2] and secure hash library[3]. We slightly modify the original training algorithms in selecting per batch data using PRF, hashing

[2]https://www.tensorflow.org/

[3]https://docs.python.org/3/library/hashlib.html

over model parameters at the end of each round, and building Merkle tree at the end of all rounds. In practice, directly hashing over the entire model parameters is slow, since the prerequisite operation (i.e., converting parameters of each layer into bytes and concatenating them as one variable) and hashing over the large parameters are time-consuming. For example, in our experiment, computing the hash of VGG16 consumes about 800ms, while purely training with a batch size of 20 only requires 30ms. Inspired by [39], we compact the parameters by calculating the dot product with a vector of random nonces, the result would be the input of the hash function.

Prog_i: As for in-enclave program, we realize all functions depicted in Figure 2 and use original SGX SDK library [17] to calculate HMAC and hash values. There are two key functions:

Data preprocess: There are two methods to perform data preprocess in the enclave. The first (denoted as "copy" method) is to copy data into the enclave, generate a pair of index and HMAC for data, and write them back to the outside of the enclave. The second (denoted as "non-copy" method) is to leverage the characteristic where the enclave can access the entire memory space of the owner process. The Prog_o passes a pointer indicating the data location to Prog_i via ECall, then Prog_i processes each record. These two methods achieve the same security guarantee, since data is provided by the participant without requiring to protect data privacy and both of them can ensure data identifiability and integrity. We compare the performance of these two methods later.

DNN: Porting the whole framework of Tensorflow into SGX is non-trivial due to SGX constraints on system calls. We use Eigen[4] which is a C++ linear-algebra library used in TensorFlow. We implement both feed forward and back propagation of DNNs with support for fully-connected layers, convolutional layers, pooling layers, and activations.

Task owner side: All programs at task owner side are written in C++, including random selection, verification, and weighted averaging based aggregation. In our prototype, we simplify the verification procedure by using the signature scheme built upon Crypto++[5], instead of fully following the SGX remote attestation mechanism. The communication channels between the task owner and participants are built via TCP socket.

B. Performance Evaluation

Setup. Both the server and the participant sides run on an Intel Core i5-8500 Skylake 3.0GHz CPU with 16GB RAM. In particular, the participant side supports SGX and sets up an NVIDIA GeForce RTX 2080 Ti GPU to execute the outside training program. All enclave programs are developed using Intel SGX SDK 2.6 and all C++ algorithms are compiled in the same optimization level (-O2). In our experiments, we use CIFAR-10 [40] of 160MB with 32×32 images and ImageNet [41] of 160GB with 224×224 images. As the convolutional neural network, we implement several popular models including VGG16 with 57MB parameters, VGG19 with 548MB parameters, and Resnet50 with 97MB parameters. The latter two perform ImageNet classification.

[4]http://eigen.tuxfamily.org/

[5]https://www.cryptopp.com/

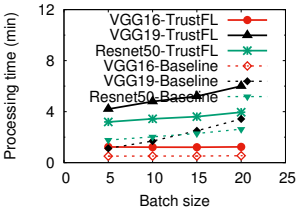

Fig. 3. The processing time of baseline DNN and TrustFL DNN programs with varied batch size.

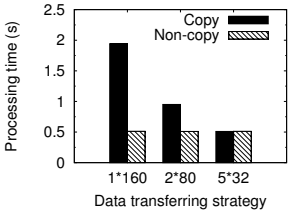

Fig. 4. The processing time of "copy" and "non-copy" methods with varied data transferring strategies.

Here, we focus on the performance evaluation of the participant side, including the performance of the outside training algorithm, in-enclave data preprocess, in-enclave training, and overall gains of TrustFL. We omit the estimation of the server side. Because, compared with the original FL system [33], our design only adds the verification of signatures from selected participants at each epoch. The computation and communication overhead is small and proportional to the number of selected participants.

Outside training performance. While we outsource all training rounds at the co-located server with GPU, we measure how our design affects the performance of the original training on GPU. Figure 3 reflects that the processing time of original DNN (which is the baseline) and DNN programs in the TrustFL training 1000 rounds with different batch size. As expected, our design introduces additional processing delay. Nonetheless, we find that the inflation is constant and only depends on the size of models. More precisely, the extra delay of VGG16, VGG19, and Resnet50 for one training round are about 41ms, 154ms, and 103ms, respectively. As the batch size grows, such the inflation can be amortized to each data. Furthermore, we measure the time of constructing a Merkle tree which is marginal to the training time. For example, building a Merkle tree with 10^2, 10^3, and 10^4 leaf nodes require 4.1×10^{-4}s, 3.78×10^{-3}s, and 3.71×10^{-2}s, respectively.

In-enclave data preprocess efficiency. We compare the processing time of the "copy" and "non-copy" methods with varied data transferring strategies. As shown in Figure 4, we split the entire CIFAR-10 dataset into 1, 2, and 5 parts. We can see that the processing time of the "non-copy" method is constant for various data transferring strategies, as the overhead of invoking ECall to pass file pointers is negligible compared with that of processing data. Contrastively, the "copy" method may suffer from obvious performance degradation as the size of per copied file grows. The reason behind is that the latter may trigger the heavy SGX paging mechanism, if the size of the file copied into the enclave approaches/exceeds the SGX PRM limitation. To sum up, the "non-copy" method outperforms the "copy" method without sacrificing security guarantee.

In-enclave training performance. To understand the performance of executing DNN program inside the enclave, we measure the performance of model initialization and DNN

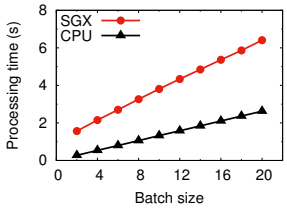

Fig. 5. The training time of VGG16 inside/outside the enclave with varied batch size.

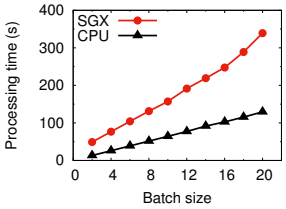

Fig. 6. The training time of VGG19 inside/outside the enclave with varied batch size.

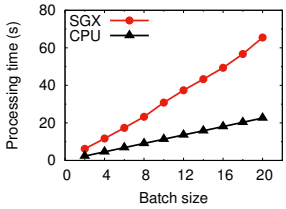

Fig. 7. The training time of Resnet50 inside/outside the enclave with varied batch size.

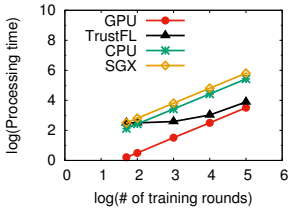

Fig. 8. Processing time of training VGG16 using GPU, TrustFL, CPU, and SGX.

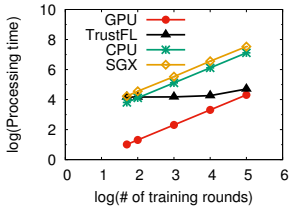

Fig. 9. Processing time of training VGG19 using GPU, TrustFL, CPU, and SGX.

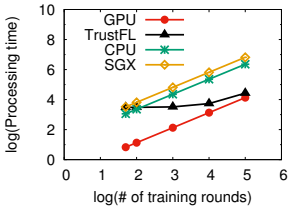

Fig. 10. Processing time of training Resnet50 using GPU, TrustFL, CPU, and SGX.

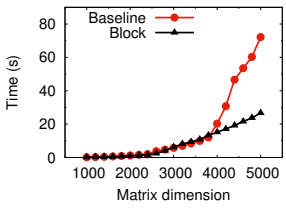

Fig. 11. The performance of matrix multiplication in the SGX with the baseline and blocked optimization.

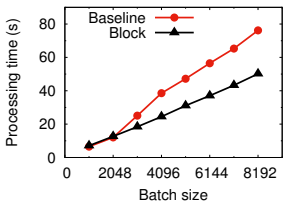

Fig. 12. The performance of the toy model in the SGX with the baseline and blocked optimization.

training for various models. For fairness, we compare the time with that on standard CPU rather than GPU.

We estimate the model initialization time of various DNN models, which is 0.7s, 3.7s, and 4.4s for VGG16, VGG19, and Resnet50, respectively. This time grows with the number of layers of DNN models yet is marginal compared with the training time inside the SGX. Figure 5, Figure 6, and Figure 7 show that the training time increases with the batch size. Also, we can see that training inside the enclave consumes more time than outside the enclave for all batch sizes. This is incurred by SGX paging mechanism, since even if the batch size equals to 2, the peak memory of the training VGG16 can reach to about 275MB exceeding SGX PRM limits. Interestingly, the figures also reflect that the gap of training time inside and outside the enclave magnifies as the batch size grows. This is because, when training, the DNN program produces intermediate results at each layer that would be used in backpropagation. The size of these results enlarges proportionally with the batch size and further incurs more paging overhead.

Overall Gains of TrustFL. We compare the performance of our scheme with three baselines: training on GPU, CPU, and SGX. We measure their processing time for VGG16, VGG19, and Resnet50 with a batch size of 20 under varied training rounds, i.e., $R = 50$, 100, 1k, 10k, and 100k. We set the number of samples as 43, as it is sufficient to guarantee that the soundness error is less than 1% even for the adversaries who have completed 90% of training rounds.

Figure 8, Figure 9, and Figure 10 show that, coincident with theoretical analysis in Section V-B, as R grows, the processing delay of TrustFL is first near to those of using CPU and SGX, then approaches to that of using GPU. Meanwhile, the processing delay of training on GPU, CPU, and SGX increases linearly with the training rounds. When $R = 50/100$, the processing latency of TrustFL is slightly smaller than that with SGX, yet larger than that with CPU. When $R = 100$k, TrustFL can achieve $82\times$, $668\times$, $244\times$ speedups compared with naively using SGX, for VGG16, VGG19, and Resnet50, respectively. Besides, for larger models, TrustFL can bring more speedups than that using SGX. We argue that, while such the huge gap may be caused by the non-optimal C++ based training implementations, TrustFL can approach the performance of training on GPU due to the constant sampling mechanism as analyzed in Section V-B. It is a huge improvement compared with that fully using TEE. In conclusion, TrustFL can obtain comparable performance with that with GPU and about one/two orders of magnitude speedups than that naively using TEE, while achieving a high execution assurance level.

VII. Discussion of DNN Training Optimization in the TEE

As analyzed in Section V-B, if the number of total training rounds R is small, the processing delay of TrustFL is mainly incurred by that of the training inside the enclave. Yet, training one round of DNN inside the enclave faces performance slowdown, compared with that on standard CPU outside the enclave. As shown in Figure 5, Figure 6, and Figure 7, training inside the SGX is about three times slower than that on CPU. The degradation is primarily caused by SGX paging mechanism. In this section, we discuss how to optimize the training in the SGX, so as to improve the performance of TrustFL with small R. Notably, the proposed optimization can be applied to any scenarios where DNNs are executed in TEE.

A neural network consists of multiple layers, like convolutional layer, pooling layer, fully-connected layer, and activation. In one DNN learning process, the network first executes feed-forward algorithm that processes a batch of data layer by layer and compares the output results against labels. The error computed by a loss function is propagated back to generate gradients from the last layer to the first layer. Finally, the model parameters are updated according to these gradients. Based on the details of training DNN, we give refinement proposals with respect to single/multiple layer computation.

Refinement on single layer computation. We inspect the operations in each layer and try to alleviate the paging overhead. We find that the primary operations of each layer is matrix multiplication in both feed forward and back propagation. If matrices are too large ($>$ 90MB PRM), the time of the multiplication increases sharply as shown in Figure 11.

In this regard, it is promising to adopt the classical *matrix blocking optimization* [42] in the DNN training algorithm. The rational of this design is, instead of operating on entire rows/columns of large matrices, using submatrices can avoid that data is frequently swapped in and out. We validate the its effectiveness in Figure 11. According to the basic idea, in the DNN training, for large matrix multiplication, we can cut the input and the kernel of each layer into small blocks fitting in the SGX PRM in an even manner.

To validate this optimization, we implement a toy model with a large fully-connected layer with the kernel of size 5184×5184, since the kernel size of layers in all DNNs described before cannot reach the PRM limits. Figure 12 shows the baseline training time is about $1.5\times$ larger than that with blocking optimization. This means that in the same time period, if the baseline can ensure execution assurance with a confidence level of 99%, the optimized method achieves a confidence level of 99.88%.

Refinement on multiple layer training. Training inside the enclave incurs inevitably paging, since DNN training is memory-demanding. The extra memory consists of the outputs of each layer produced in feed forward (called as intermediate results) and gradients generated in back propagation. The intermediate results is the main component whose size is positively related to the image size and batch size.

To improve the training performance, we need to minimize the memory cost. Inspired by [43], we can build a memory-sharing graph among independent intermediate results. Specially, different from [43], we should guarantee that such the computation can fit in the PRM limit. Otherwise, it may incur paging overheads again. In the future, we will validate the its effectiveness.

VIII. Related Work

Privacy-preserving deep learning. Recent advances proposed various implementations of federated learning [1], [2]. Under the FL framework, some proposals envision to enhance the data privacy via differential privacy [29] and secure multi-parity computation based aggregation [31], accelerate the overall training process with a resource-aware selection strategy [44], and improve communication efficiency between server and participants [45]. Our work is complementary to them, as we focus on eliminating the security concerns of the server by assuring honest training at participant side.

Secure and robust federated learning. Federated learning is fragile to bad participants that may launch model poisoning or backdoor attacks [12], [13]. Auror [7] defends against model poisoning in collaborative learning [1] by clustering participants' model updates and removing outliers. Multi-Krum [8] achieves model convergence by discarding outliers from mean participant gradients in the presence of Byzantine participants. FoolsGold [9] prevents sybil-based poisoning by adjusting the learning rate of strange participants. All of these efforts are a kind of alternative aggregation mechanism at the server side and cannot effectively defend against advanced stealthy attacks, like [12], and lazy participants. In contrast, we enforce local training at the (possibly lazy) participant side. Moreover, our work can be easily combined with the above solutions to defend against malicious participants.

TEE-protected machine learning. Recently, TEE becomes increasingly popular in the area of developing privacy-preserving machine learning. Ohrimenko et al. [32] study centralized ML tasks on an SGX-enabled data center for data privacy, while preventing data leakage exploited by side channels. Our work is different, as we do not need to take side channel attacks into consideration in the data owner platform.

Chiron [46] employs TEE in ML-as-a-service (MLaaS) platforms to protect both data and model confidentiality. Particularly, Chiron enables the data owner to be assured that the model training code would not leak the data, by a confinement mechanism with Ryoan sandbox [47]. Hynes et al. proposed Myelin [37] that perform TEE-protected centralized training for data privacy. They port an optimized numerical libraries generated by TVM stack [48] in the enclave which can achieve similar performance to standard CPU-based training. By comparison, besides different ML settings, we do not execute DNN fully in the TEE for efficiency and not focus on model privacy. In addition, the above optimization can be utilized to enhance the training performance in the TEE.

The most related work to ours is Slalom [36]. It delegates matrix multiplication from TEE to the untrusted co-located GPU for high performance, while preserving DNN inference integrity and/or data privacy. However, their cryptographic blinding solution does not immediately work in DNN training. In contrast, we can fast build the DNN training assurance with high confidence, by harmonizing GPU with TEE.

IX. Conclusion

In this paper, we propose TrustFL, a practical scheme to enhance the FL system with assured training execution at participants. While running the entire training computation inside the TEE leads to a severe drop in performance, we use GPU to execute all rounds of training for efficiency and leverage the TEE to ensure whether outside training is correctly executed with tunable levels of assurance. We conduct both theoretical analysis and extensive experiments. They demonstrate that TrustFL can achieve comparable performance with that on GPU, which is about one/two orders of magnitude faster than that fully on SGX.

Acknowledgments

We thank Lingchen Zhao and Yunzhe Guo for providing valuable input. This work was supported in part by the National Key R&D Program of China under Grants 2017YFB0803202, NSFC under Grants 61572278 and U1736209, Research Grants Council of Hong Kong under Grants CityU 11212717, CityU 11217819, and CityU C1008-16G, and BNR2019TD01004. Qi Li is the corresponding author.

REFERENCES

[1] R. Shokri and V. Shmatikov, "Privacy-preserving deep learning," in *Proc. of ACM CCS*, 2015.

[2] B. McMahan and D. Ramage, "Federated learning: Collaborative machine learning without centralized training data," Online at: https://ai.googleblog.com/2017/04/federated-learning-collaborative.html, 2017.

[3] E. GDPR, "The eu general data protection regulation (gdpr) is the most important change in data privacy regulation in 20 years." Online at: https://eugdpr.org/.

[4] A. Rajkomar, E. Oren, K. Chen, A. M. Dai, N. Hajaj, M. Hardt, P. J. Liu, X. Liu, J. Marcus, M. Sun *et al.*, "Scalable and accurate deep learning with electronic health records," *NPJ Digital Medicine*, vol. 1, no. 1, p. 18, 2018.

[5] V. Gulshan, L. Peng, M. Coram, M. C. Stumpe, D. Wu, A. Narayanaswamy, S. Venugopalan, K. Widner, T. Madams, J. Cuadros *et al.*, "Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs," *Jama*, vol. 316, no. 22, pp. 2402–2410, 2016.

[6] N. Hynes, D. Dao, D. Yan, R. Cheng, and D. Song, "A demonstration of sterling: a privacy-preserving data marketplace," in *Proc. of VLDB Endowment*. VLDB Endowment, 2018.

[7] S. Shen, S. Tople, and P. Saxena, "A uror: defending against poisoning attacks in collaborative deep learning systems," in *Proc. of ACM ACSAC*. ACM, 2016.

[8] P. Blanchard, R. Guerraoui, J. Stainer *et al.*, "Machine learning with adversaries: Byzantine tolerant gradient descent," in *Advances in Neural Information Processing Systems*, 2017, pp. 119–129.

[9] C. Fung, C. J. Yoon, and I. Beschastnikh, "Mitigating sybils in federated learning poisoning," *arXiv preprint arXiv:1808.04866*, 2018.

[10] L. Zhao, S. Hu, Q. Wang, J. Jiang, C. Shen, and X. Luo, "Shielding collaborative learning: Mitigating poisoning attacks through client-side detection," *arXiv preprint arXiv:1910.13111*, 2019.

[11] L. Zhao, Q. Wang, Q. Zou, Y. Zhang, and Y. Chen, "Privacy-preserving collaborative deep learning with unreliable participants," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 1486–1500, 2019.

[12] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov, "How to backdoor federated learning," *arXiv preprint arXiv:1807.00459*, 2018.

[13] A. N. Bhagoji, S. Chakraborty, P. Mittal, and S. Calo, "Analyzing federated learning through an adversarial lens," *arXiv preprint arXiv:1811.12470*, 2018.

[14] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "Snarks for c: Verifying program executions succinctly and in zero knowledge," in *Proc. of Springer CRYPTO 2013*, 2013.

[15] Z. Ghodsi, T. Gu, and S. Garg, "Safetynets: Verifiable execution of deep neural networks on an untrusted cloud," in *Advances in Neural Information Processing Systems*, 2017.

[16] L. Zhao, Q. Wang, C. Wang, Q. Li, C. Shen, X. Lin, S. Hu, and M. Du, "Veriml: Enabling integrity assurances and fair payments for machine learning as a service," *arXiv preprint arXiv:1909.06961*, 2019.

[17] I. Corporation, "Intel® software guard extensions (intel® sgx) sdk for linux* os," Tech. Rep., 2019.

[18] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *International conference on machine learning*, 2013.

[19] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: Exitless os services for sgx enclaves," in *Proc. of ACM EuroSys*, 2017.

[20] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contract execution," *arXiv preprint arXiv:1804.05141*, 2018.

[21] H. B. McMahan, E. Moore, D. Ramage, S. Hampson *et al.*, "Communication-efficient learning of deep networks from decentralized data," *arXiv preprint arXiv:1602.05629*, 2016.

[22] I. Corporation, "Intel® software guard extensions programming reference," Online at: https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2014.

[23] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions." *HASP ISCA*, vol. 11, 2013.

[24] I. Corporation, "Intel® software guard extensions: Intel attestation service api," Online at: https://software.intel.com/sites/default/files/managed/7e/3b/ias-api-spec.pdf, 2016.

[25] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *Proc. of ACM CCS*, 2016.

[26] F. Tramer, F. Zhang, H. Lin, J.-P. Hubaux, A. Juels, and E. Shi, "Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge," in *Proc. of IEEE EuroS&P*, 2017.

[27] C. Song, T. Ristenpart, and V. Shmatikov, "Machine learning models that remember too much," in *Proc. of ACM CCS*, 2017.

[28] N. Carlini, C. Liu, J. Kos, Ú. Erlingsson, and D. Song, "The secret sharer: Measuring unintended neural network memorization & extracting secrets," *arXiv preprint arXiv:1802.08232*, 2018.

[29] H. B. McMahan, D. Ramage, K. Talwar, and L. Zhang, "Learning differentially private recurrent language models," 2018.

[30] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, "Deep learning with differential privacy," in *Proc. of ACM CCS*, 2016.

[31] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," in *Proc. of ACM CCS*, 2017.

[32] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious multi-party machine learning on trusted processors." in *Proc of USENIX Security*, 2016.

[33] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konecny, S. Mazzocchi, H. B. McMahan *et al.*, "Towards federated learning at scale: System design," *arXiv preprint arXiv:1902.01046*, 2019.

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

[35] R. C. Merkle, "Protocols for public key cryptosystems," in *IEEE Symposium on Security and Privacy*, 1980.

[36] F. Tramer and D. Boneh, "Slalom: Fast, verifiable and private execution of neural networks in trusted hardware," *arXiv preprint arXiv:1806.03287*, 2018.

[37] N. Hynes, R. Cheng, and D. Song, "Efficient deep learning on multi-source private data," *arXiv preprint arXiv:1807.06689*, 2018.

[38] W. Du, J. Jia, M. Mangal, and M. Murugesan, "Uncheatable grid computing," in *Proc. of IEEE ICDCS*, 2004.

[39] H. Shacham and B. Waters, "Compact proofs of retrievability," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2008, pp. 90–107.

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

[41] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Proc. of IEEE CVPR*, 2009.

[42] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *ACM SIGARCH Computer Architecture News*, vol. 19, no. 2. ACM, 1991, pp. 63–74.

[43] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *arXiv preprint arXiv:1604.06174*, 2016.

[44] T. Nishio and R. Yonetani, "Client selection for federated learning with heterogeneous resources in mobile edge," in *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE, 2019, pp. 1–7.

[45] J. Konečnỳ, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," *arXiv preprint arXiv:1610.05492*, 2016.

[46] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel, "Chiron: Privacy-preserving machine learning as a service," *arXiv preprint arXiv:1803.05961*, 2018.

[47] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," *ACM Transactions on Computer Systems (TOCS)*, vol. 35, no. 4, p. 13, 2018.

[48] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: end-to-end optimization stack for deep learning," *arXiv preprint arXiv:1802.04799*, pp. 1–15, 2018.