SoK: Machine Learning with Confidential Computing

Fan Mo*, Zahra Tarkhani†‡, Hamed Haddadi*

*Imperial College London, †University of Cambridge, ‡Microsoft Research

Abstract—Privacy and security challenges in Machine Learning (ML) have become a critical topic to address, along with ML’s pervasive development and the recent demonstration of large attack surfaces. As a mature system-oriented approach, confidential computing has been increasingly utilized in both academia and industry to improve privacy and security in various ML scenarios. In this paper, we systematize the findings on confidential computing-assisted ML security and privacy techniques for providing i) confidentiality guarantees and ii) integrity assurances. We further identify key challenges and provide dedicated analyses of the limitations in existing Trusted Execution Environment (TEE) systems for ML use cases. We discuss prospective works, including grounded privacy definitions, partitioned ML executions, dedicated TEE designs for ML, TEE-aware ML, and ML full pipeline guarantee. These potential solutions can help achieve a much strong TEE-enabled ML for privacy guarantees without introducing computation and system costs.

Index Terms—Machine Learning, Privacy, Integrity, Confidential Computing, Trusted Execution Environment.

I. INTRODUCTION

Machine learning (ML) has been established as the most promising approach to learning patterns from data, and its applications exist across data processing of online surfing, financial data, health care, autonomous cars, and almost every data-driven field around us. The wide applications have also lead ML models being run on a diverse set of devices – from low-end IoT/mobile devices to high-performance cloud/data centers to provide both training and inference services. However, wide applications also open a large attack surface on ML’s security and privacy. Recent research has massively explored the way to attack on ML e.g., model stealing [1, 2], model inversion [3, 4], model/data poisoning [5, 6], data reconstruction [7], data construction [8, 9], membership/attribute inference [10, 11], etc. Vulnerabilities shown by these attacks have increasingly perturbed ML developers and users due to the negative consequences they can cause.

ML faces security and privacy issues mainly due to, firstly, the complexity of pipelines that involve many system/software stacks to provide modern features like acceleration. A full ML pipeline covers the collecting of raw data, the complete training and inference phases, the later use of the trained ML model for prediction, and potential re-train and re-use of the ML model. Because data owners, the host of ML computation, the model owners, and result receivers are most likely to be different entities, the pipeline can be segmented across mutually mistrusting individuals and is left with a broad attack surface. Second, the weak robustness and low interpretability of ML algorithms deteriorate the security and privacy problems. The modern ML model is a large-scale nonlinear system that has been progressively adjusted using stochastic gradient descent during training. Until now, we still lack an accurate representation/description of the dynamics of this training progress. Consequently, ML models often have relatively weak robustness, which has been shown in adversarial examples [15] or poisoning attacks [8], i.e., a small change to the training process could cause large negative impacts that are difficult, or even impossible, to detect.

Confidential Computing has emerged as one promising approach to achieving trustworthy ML to guarantee privacy. Confidential Computing, defined in the Linux Foundation project Confidential Computing Consortium’s whitepapers [16], [17], is the protection of data in use, in addition to data protection in storage and transmission, by utilizing hardware-based Trusted Execution Environments (TEEs). TEE is one of the emerging techniques in enabling isolated and verifiable code execution inside protected memory (sometimes called enclaves or secure world), separated from the host privileged system stacks such as operating system or hypervisor. Both hardware and software changes are made to achieve such isolation on modern processors (i.e., CPUs). Compared to the conventional external secure processor, like the Trusted Platform Module that uses separated hardware from the rest of the board/SoC, TEEs still remain and run on existing processors which then provide much more computational resources. TEEs have become nearly an essential component of modern processors, and various types of TEEs are implemented by e.g., ARM, Intel, AMD, and NVIDIA to achieve Confidential Computing.

The rapid growth of Confidential Computing pushes a line of research work that leverage it for ML [18], [19], [20], [21], [22] as well as commercial products (e.g., Confidential Computing service from Microsoft Azure [23], Amazon Web Services [24], Google Cloud [25], etc.). Although many attempts have been made and several limitations (e.g., still limited TEE resources for ML use) are summarized [26], there is still no systematization of Confidential Computing-assisted mechanisms, challenges, and especially potential solutions. It is still unclear what the critical scientific and engineering obstacles are and how to overcome them for achieving trustworthy ML. In this paper, we present a systematic analysis of Confidential Computing-assisted solutions to alleviate the security and privacy problems in ML.

Organization. The next sections of the paper are organized as: § II describes the pipeline and paradigms of ML and existing
attack surfaces; §III outlines the threat model and key components of Confidential Computing; §IV summarizes challenges and existing solutions of using Confidential Computing for ML; § V further presents the challenges and solutions of guaranteeing ML integrity with Confidential Computing; § VII outlines the existing TEE limits in Confidential Computing that hinder its usage in ML; § VII concludes the current research stage and the further work.

II. MACHINE LEARNING

Machine learning (ML) automates the pattern learning from large datasets in the sense that analysts do not need to manually explore data’s latent features and their correlations [27], [28]. In this section, we look into ML pipelines, common paradigms, and attack surfaces that exist among them.

A. Pipeline

The ML pipeline codifies and automates the workflow to produce and apply ML algorithms or models. It generally covers most cases when using ML, which consist of multiple sequential steps forming the lifecycle of using ML.

Data preparation. Data are the basis for ML evidently because ML requires tons of inputs to attempt “trial and error” and learn the patterns. In supervised learning [28], all data are labeled, which usually needs extensive efforts of human annotation. In semi-/unsupervised learning [29], [30], [31], parts/all of the data are unlabeled due to the difficulty of labeling. In reinforcement learning [32], data are in the form of sequences of action, observations, and rewards, which are produced during the learning process. No matter in any form, ML is always data-hungry. Data augmentation [33], [34] is necessary to increase the amount of data by adding slightly modified copies or synthetic copies of existing data.

Model training. Most model training aims to acquire a function \( f_\theta(x) \), where model parameters \( \theta \in \Theta \) the parameter space, capable of mapping an input \( x \) to a predicted decision \( \hat{y} = f_\theta(x) \), a value near to the true decision \( y \). The form of \( f_\theta(\cdot) \) excluding \( \theta \) is called model hyperparameters, which are determined by chosen model architectures e.g., the type of layers, the number of neurons, etc. Searching of the best \( \theta \) set usually is achieved by minimizing the loss \( \ell(y, f_\theta(x)) \) using Stochastic Gradient Descent (SGD) [35], [36] on training sets, i.e., stepped descent by \( \theta \leftarrow \theta - \eta \nabla_\theta f_\theta(x) \), where \( \eta \) is the step size multiplier known as the learning rate. The loss is backward passed through the model, which is called backward propagation. After reaching a good performance on the validation set (or the other subset apart from the training set in cross-validation [37], [38]), the model training stops. \( f_\theta(\cdot) \)'s final model utility/accuracy is reported on the test set.

Model deployment/Inference. By feeding one unseen input data \( x_{\text{new}} \) into the ML function, one can get the prediction \( \hat{y} = f_\theta(x_{\text{new}}) \). This is called the inference stage. Inference could happen at both the model provider side or the data owner side depending on the trust and collaboration form between the providers and the users. In typical cases of ML inference, the model needs to be deployed i) on a server-side device to support centralized service e.g., ML inference as a service, or ii) on the users’ devices for local inference distributively.

Retraining/Updates. ML models require retraining/updating their parameters to adapt to data with unseen features over time. This usually is achieved by fine-tuning the model with newly collected data. Especially, the fine-tuning could focus on the last layers of the model, which is one way to achieve transfer learning [39], [40], as the first layers already generalize well on one type of data (e.g., images in general) [39], [41], [42]. Two examples of such a technique are i) back-propagation on the last layers only [40] and ii) low-shot learning with weights imprinting on the last layer [43]. However, this would depend on whether the pre-trained models have good representatives of the new data. The updated model can be further uploaded to other users or the central server for knowledge sharing [44], [45], [46].

B. Paradigm

Due to the increasing privacy/security concern raised by sharing data, as well as network/computation-resource constraints, the stages of the ML pipeline are typically located to different participating entities for a better trade-off among utility, privacy, and cost. Based on the location of the main ML computation (e.g., training or inference), they can be categorized into: Centralized and Federated machine learning.

Centralized machine learning. In centralized ML, the training or inference computations are conducted at a central place such as a server [47], [48]. For both training and inference, the data have to be collected and stored at/near the center, and then the ML function performs training or inference on the data. In such a case, potential retraining will also happen on the server by collecting new data. Nevertheless, centralized ML raises privacy concerns about others’ data, considering that all data need to be collected. Therefore, centralized training/inference is usually chosen for the cases where collected data are public (or non-private), or the data contributors are collaborators or have any form of trust/contracts with the center.

Distributed/federated machine learning. When data owners do not have the willingness to share data, distributed/federated ML is one solution, where the ML model, instead of the private data, is sharing among participants [49], [44]. In ML inference, the model owner distributes its ML model to end-users, so that users can perform inference locally on their devices. Potentially, they can conduct further fine-tuning on the model for personalization but still for their own usage. In terms of training, the terminology ‘federated learning (FL)’ [44], [45], [46] has been used intensively referring to the paradigm that users send the locally updated model parameters to the server without the need of revealing their data. Unfortunately, recent works have shown that adversaries can still execute attacks to retrieve sensitive information from the model parameters themselves [50], [14], [10].

C. Attack Surface

The attack surface covers many places within the complete pipeline. Here, we categorize existing attack vectors based on
the core underpinning of information security into: i) confidentiality, i.e., privacy of data and intellectual property of models, ii) integrity of the ML process. The attacks on every vulnerability in the ML pipeline can be viewed in light of confidentiality and/or integrity.

Confidentiality-related attacks. An adversary can be curious-but-honest, i.e., interested in exploring (unauthorized) sensitive information of data and the model, but honestly performing ML training/inference without changing the computation results. These attack vectors usually exist in the pipeline stages of model training, model deployment/inference, or model updates, because the data and models are owned by different participating entities. In centralized ML, without advanced protections, the computation host can directly access incoming data.

Similarly, in federated ML, the host who orchestrates all clients’ local training can access their updated models and further infer private information about their local data using these models. Several common attacks are: i) Data Reconstruction Attacks, aiming at reconstructing original input data based on the observed model or its gradients; ii) Attribute Inference Attacks, aiming at inferring the value of users’ private properties in the training data; and iii) Membership Inference Attacks, aiming at learning whether specific data instances are present in the training dataset. On the other side, the confidentiality of models can be targeted. A model that has been trained with great effect (e.g., computation power, data cleaning and collection) is considered the intellectual property of its owner, e.g., machine learning as a service (MLaaS) hosted by Amazon Web Services (AWS). In such cases in inference, one possible attack, called model stealing, is to counterfeit the functionality of this model by exploiting black-box access, e.g., querying a large number of prediction results.

Integrity-related attacks. An attacker could actively exploit training/inference results to the ML integrity. For example, one well-known adversarial attack adds calibrated noises to one image/audio, which leads to wrong prediction results. However, the input with noisy perturbation is invisible to human beings. In another example, attackers could take full control of wearable BCI (Brain computing interface) devices (e.g., wheelchair, robotic arms, or drone) by compromising input signals to on-device ML models. Although such attacks manipulate input data, it breaks the integrity of ML inference. Previous work has also shown the possibility of model accuracy degradation by only compromising thread scheduling on multi-threaded ML pipeline.

Another example is the model poisoning attack in FL that could lead to Byzantine fault. Such fault can easily occur because the server does not control all clients’ local training and cannot verify that their behaviors follow the promised training processes. For example, if one error in values of local updated parameters has been aggregated into the global model, the complete global model will become unusable. Furthermore, one can manipulate the training set e.g., by adding data with calibrated noisy labels. This fools the classifier to have wrong boundaries for some specifically chosen data points. Backdoors can be added in such a way, and the attackers/other users will be able to trigger such backdoors in later use after model deployment.

White-box or black-box. Another wide-used way to categorize ML attack surfaces is based on whether one attack requires access to the internal architecture of a ML model. In black-box attacks, model stealing usually starts from the outside without any prior information and aims to learn the model itself. Membership inference attacks on data privacy are usually black-box because it is already much more efficient than white-box attacks as shown in previous research. White-box access does not significantly increase the attack ‘advantage’ in disclosig membership privacy. The white-box attack includes almost all data reconstruction attacks, some adversarial example attacks, attribute inference attacks, etc. These attacks usually require detailed model parameters/gradients in order to be performed or have reasonable performance. We refer to Papernot et al.’s survey for more details about security and privacy vulnerability in ML.

III. CONFIDENTIAL COMPUTING AND TRUSTED EXECUTION ENVIRONMENTS

Confidential Computing is a quickly emerging technology that provides a level of assurance of privacy and integrity when executing codes on data using Trusted Execution Environments (TEE). Nowadays, most Cloud service vendors have started providing Confidential Computing services, e.g., Google Cloud, Microsoft Azure, etc. They usually leverage the TEE solution supplied by processor manufacturers such as ARM, Intel, AMD, and NVIDIA. In this section, we first discuss threat models considered in Confidential Computing, key components, and then software stacks useful for ML application developers.

A. Threat Models

By considering a stronger adversary who has full access to the host privileged system stacks, Confidential Computing significantly reduces the trusted computing base (TCB). By default, TEEs assume that the attacker controls the host OS/hypervisor as well as the host applications executing in userspace. She could, for example, access and corrupt the host OS kernel and process resources (e.g., memory, threads, network, files, locks). Hence, the adversary could be the service provider, the device owner itself, malicious third-party software installed on the devices, or a malicious or compromised host OS/hypervisor. Under such an environment, they
can perform any possible white-/black-box or confidentiality-/integrity-related attacks.

Confidential Computing enables trust between mutually mistrusting components by utilizing TEEs. TEEs provide an isolated environment running in parallel with the host OS (also known as the rich OS), allowing to securely store data and execute arbitrary code on an untrusted device, with reasonably good performance, through protected and measured memory compartments. Thus, one can demand a remote TEE to perform computation on her (sensitive) code and/or data without revealing them to the TEE’s host. Few TEEs could also extend the level of Hardware-based isolation to peripherals and persistence storage \(^{69}, 70\). The recent commoditization of TEEs both in high-end and low-end mobile devices makes it an ideal candidate to achieve confidentiality and integrity in ML. However, if the code inside an enclave has vulnerabilities, TEEs do not guarantee the protection of the integrity and confidentiality of that computation. Therefore, the attacker could exploit weaknesses in TEEs system \(^{71}\) and their interfaces/APIs \(^{72}, 73\).

Moreover, TEEs are vulnerable to various side- and covert-channel attacks \(^{74}, 75\), physical attacks \(^{76}\), confused deputy attacks \(^{77}\), Denial-of-Service (DoS) attacks, etc \(^{78}, 79\). These attack vectors are hard to defend against and are not considered within the standard TEE threat models.

**B. Key Components of Confidential Computing**

**Root of trust measurement.** Confidential Computing provides hardware-assisted functionalities to establish trust between TEEs and the external computing environment. First, by enabling Root of Trust (RoT) measurement it ensures the integrity of the TEE system \(^{68}, 80\). When the processor measures the chain of all critical system software (including boot loader, firmware, OS, hypervisor, and TEE system) before launching in-enclave code, any integrity violation of the initial state could be detected. A compromised code in the chain cannot escape from being measured and therefore from being detected. The measurement needs hardware-assisted cryptographic keys (e.g., hardware unique keys and endorsement keys), which will be attested later to ensure the integrity of the TEE system \(^{68}, 80\).

**Remote trust establishment and attestation** Remote attestation \(^{80}, 81\) allows the user to determine the level of trust/integrity of a remote TEE before transmitting her sensitive data/code. It enables the user to authenticate the hardware, verify the trusted state of the remote TEE, and check whether the intended software is securely running inside the TEE. Attestation could be conducted by a third party besides the user and host \(^{82}\) or directly between the service provider and enclave owner \(^{83}\). For example, the attestation server could be launched by the processor manufacturer/vendor, as in Intel’s SGX attestation service \(^{84}\), which could be different from the host service provider (e.g., a cloud provider).\(^4\)

**Trustworthy code execution and compartmentalization.** The core feature provided by Confidential Computing is the hardware-assisted isolation of enclaves/TEEs from the untrusted environment \(^{68}\). In general, enclave memory regions are protected and managed through hardware and TEE’s system software. Isolation mechanisms for different processor architectures are different (e.g., ARM, Intel, and AMD). For instance, SGX relies on Memory Encryption Engine (MEE) to protect the confidentiality, integrity, and freshness of the CPU-DRAM traffic over enclave memory ranges, while TrustZone relies on separate page tables, hardware privilege layers (e.g., EL3 and Secure EL2/EL1/EL0), and TrustZone address space controller (TZASC).\(^3\) More importantly, they run enclaves in different privilege and execution modes. For instance, SGX enclaves run in userspace mode, while AMD’s SEV and Intel TDX \(^{85}\) run them on hypervisor mode, and TrustZone secure world runs as a virtual separated core from the host core.

Moreover, depending on the TEE/enclave architecture and threat model, it could support additional peripheral compartmentalization and persistent storage sealing. In Table \(I\) we summarize common hardware-assisted TEEs with their providers, supported processors, and features. Among them, TrustZone is developed from a very early period, and it is the most widely-used TEE on mobile/ubiquitous devices. However, it has several constraints including the inflexible protected memory, supporting only one single secure world, and restricted third-party application adaption. To overcome these issues, Recently ARM proposed Confidential Compute Architecture (CCA) which aim to run more flexible enclaves (called realm) in parallel with TrustZone. Therefore, modern TEEs support multi-enclaves and more flexible secure memory sizes with different TCBs. TEE systems which include an OS-like software stack usually have a large TCB size. On the other side, a larger TCB size can usually provide higher usability to developers and users. For instance, when TEEs provide such an OS, they statically link applications into the in-enclave kernel; consequently, unmodified applications could be deployed into enclaves.

**C. Partitioning Frameworks for Developers**

With the low-level hardware and system stacks only, it is still hard for application developers to utilize TEEs. Hence, depending on the use-cases, various software vendors provide deployment and partitioning frameworks for running a diverse set of workloads within enclaves. For example, more recently Amazon Nitro enclaves and Enarx support running the same binary within different types of TEEs through in-enclave WebAssembly (WASM) support, inspired by Haven that ports Drawbridge \(^{100}\) (a Windows library OS) inside an SGX enclave. Similarly, Graphene-SGX \(^{101}\) ports the Linux-based Graphene library OS in an enclave. Scone \(^{102}\) tries

\(^{3}\)Nitro Enclave provided by AWS uses its own remote attestation service (e.g., AWS Key Management Service). In such a case, one AWS Nitro Enclave customer provides assurances to their downstream customer.

\(^{4}\)There are software/language-based TEEs and compartmentalization techniques without the needs of hardware-based guarantees (e.g., sel4), but we do not consider them here as they usually have lower security assurance and are not the mainstream in Confidential Computing.
### TABLE I: Summary of dominant hardware-assisted trusted execution environments

| Main Stream Hardware TEEs | ISA & Year | Supported Processors | Number of Isolation | TCB Size (excluding CPU/SoC package) | Secure Memory Size | Attestation | Application | Application Unchanged | Protections |
|---------------------------|------------|----------------------|---------------------|-------------------------------------|--------------------|-------------|--------------|-----------------------|-------------|
| Software Guard Extensions (SGX) | Intel (2013) | 6th Intel CPU + | multi-enclaves | small, BIOS or firmware | up to 128MB | remote Intel service | desktop-level | ☐ | confidentiality Integrity |
| TrustZone (TZ) | Arm (2005) | ARMv6-A + / Armv8-M + | single secure world | small, firmware, TZ-kernel | typically up to 16 ∼ 64MB | TPM-based | ☐ | mobile-level desktop-level | ☐ | confidentiality integrity |
| Secure Encrypted Virtualization (SEV) | AMD (2016) | AMD EPYC + | multi-VMs | large, VM's OS, firmware | up to available system RAM | secure processor | enterprise-level | ☐ | confidentiality |
| PMP-based TEE | RISC-V (2017) | Xilinx Artix-7, SiFive E31, U54 | multi-enclaves | changeable, Runtime, firmware | changeable | ☐ | ☑ | ☑ | confidentiality integrity |
| Nitro Enclaves | AWS (2020) | Nitro Cards & Security Chip (EC2) | multi-enclaves | changeable, Vms, OS/Hypervisor | up to available instance RAM | remote AWS service | ☑ | ☑ | ☐ confidentiality integrity |
| EdgeLock Secure Enclave | NXP (2021) | i.MX 8ULP-CS, i.MX 9 | single secure world | small, Firmware | changeable typically small | remote Azure Sphere | IoT-level mobile-level | ☐ | confidentiality integrity |
| Confidential Compute Architecture (CCA) | Arm (2022) | Armv9-A | multi-realms | changeable, OS kernel, firmware | up to available system RAM | ☐ | ☑ | ☑ | confidentiality integrity |

α Full TCB size is implementation-specific, and the reported TCB size here is the best case supported by hardware (bare-metal potential TCB).
δ Most difficult for fitting ML based on prior works.
β Strictly speaking, PMP (physical memory protection) is not a TEE but a hardware-based memory isolation feature that enables several frameworks to architect different TEEs on RISC-V.
γ Nitro adds new infrastructure but still builds atop existing CPUs so strictly not an ISA. Application Unchanged: ☐ Major changes needed; ☑ Light changes needed; ● No change needed.

### TABLE II: Open source TEE frameworks

| Open Source TEE Framework | Leading | Supported TEEs | On Top Of | Main Functions | Attestation | Languages | ML Examples |
|---------------------------|---------|----------------|-----------|----------------|-------------|-----------|-------------|
| Intel SGX SDK [87] | Intel | SGX | firmware & driver | API and librarys for SGX app. | ☐ | C/C++ | ● |
| OP-TEE [88] | Linaro | TrustZone | firmware & driver | API and librarys for TZ app. | ☐ | C/C++ | ● |
| MultiZone [89] | HEX-five | RISC-V | firmware & driver | API and librarys for RISC-V Enclave app. | ☐ | C/C++ | ○ |
| Keystone [90] | UC Berkeley | RISC-V | firmware & driver | API and librarys for RISC-V Enclave app. | ☐ | C/C++ | ● |
| Open Enclave SDK [91] | Microsoft | SGX, TrustZone | SGX SDK & OP-TEE | generlized API for TEE app. | ☐ | C/C++ | ● |
| Asylo [92] | Google | SGX | SGX SDK | generlized API for TEE app. | ☐ | C/C++ | ● |
| Teclave [93] | Apache | SGX | SGX SDK | API for Rust programming | ☐ | Rust | ● |
| Gramine [94] | Invisible Things Lab / Intel | SGX | SGX SDK | lightweight library OS to host app. | ☐ | Native executable* | ● |
| Occlum [95] | Tsinghua / Ant Group | SGX | SGX SDK | lightweight library OS to host app. | ☐ | Native executable* | ● |
| SCONE [96] | Scontain | SGX | container & driver | TEE-based container image generation | ☐ | Docker image* | ● |
| Enarx [97] | Red Hat | SGX, SEV, CCA (upcoming) | firmware & driver | WebAssembly sandbox | ● | Wasm binary* | ○ |
| Veracrux [98] | Arm | SGX, TrustZone, Nitro Enclaves, CCA (upcoming) | SGX SDK & OP-TEE | proxy attestaton, policy | ☐ | Wasm binary* | ● |

Attestation: ☐ Not available; ☑ Self-configuration required; ● Work off-the-shelf. * Most modern languages supported by compiling as a target executable/binary. ML examples: ☐ Not exist; ● Exists. However, One non-exist example can be developed using an existing ML framework with corresponding languages.
to reduce TCB by porting musl libc and a portion of Linux Kernel Library (LKL) \[103\]. TrustShadow \[69\] also protects unmodified applications from the host OS by trapping applications exceptions and system calls inside TrustZone, passing the calls to the OS, and verifying the output after parameter marshaling.

As another approach, various TEE partitioning frameworks split applications into trusted and untrusted components, e.g., Intel SGX SDK \[104\], Microsoft Open Enclave \[105\], Google’s Asylo \[106\], OP-TEE \[88\], and Keystone \[107\]. There are also language-specific partitioning frameworks such as Civet \[108\] for porting Java classes into SGX enclaves, Trusted Language Runtime \[109\] for running portions of C# applications inside TrustZone, and Glimdring \[110\], a compiler for partitioning applications into SGX enclaves via code annotation. Table I summarises primary commercial TEE frameworks. However, not all of them are feature-rich enough to support ML use cases. Particularly, mobile vendors (e.g., TEE of Qualcomm \[111\], Trustonic \[112\], or Huawei \[113\]) also only allow for limited TEE operations to avoid security risks inside TrustZone secure world \[77\]. Besides, such security concerns link to the less privileged execution of enclaves as well as ease of programming.

IV. CONFIDENTIAL COMPUTING SOLUTION FOR MACHINE LEARNING

A. Overview

In Confidential Computing-assisted ML, data/model owners need to secretly provision their data/model to the untrusted host’s TEE (see Figure 1). Specifically, after preparation of the model and/or data, the owners first perform remote attestation to assure the integrity of the remote TEE, and then, establish secure communication channels to the TEE. Afterward, data/models are provisioned to the TEE, where model training or inference will be performed. The final results will be returned, i.e., a trained model will be transmitted out in training, or the data label will be returned back to the user in inference. Note that in practice the host, model owners, data owners, and result receivers could be distinct entities, or some may not exist depending on use cases. We further classify these cases according to the nature of the host: Server or Client (see Figure 2).

Server-side ML protection. A server-like host aims to provide ML service to its customers. Based on the specific ML functionalities, the host can provide i) Inference as a Service (e.g., \[114\], \[21\]) or ii) Training as a Service (e.g., \[20\], \[115\]). In both cases, schemes should protect data privacy. That is, data have to be secretly provisioned to the untrusted server’s TEE no matter for inference or training. The private information contained in the model or its intellectual property also requires protection if it is owned by another entity (e.g., external model owners/providers), not the server. Then the model will be secretly provisioned into the TEE as well. However, note that the ownership of the ML model can belong to not only external owners but also to the untrusted server itself. The latter case will only exist for ML inference, as performing inference does not require changing the model parameters. In contrast with it, training leads to updates of the model parameters on data which consequently leaks private information about the data (i.e., data privacy).

Client-side ML protection. A client-like host usually acts as the downstream user of a server, and it performs ML based on the ML framework/algorithms and models provided by the server. Common use cases are i) on-device inference (e.g., \[116\]), the client conducts predictions on its own data with another individual’s model, ii) on-device personalization (e.g., \[117\]), the client personalizes the model based on its data for later inference, and iii) federated learning (e.g., \[12\]), the client trains the model to further contribute a global model owned by another party (e.g., server). In all these cases, the client owns the data; thus, this client’s TEE does not need to hide these local data from itself. However, the model can be considered confidential as it could be intellectual property, e.g., the model costs its owner or the server significant effort to

\[\text{Note that we do not use ‘model privacy’ to avoid confusion, as the private information from ML model is basically about the data used to train it, i.e., in a form of data privacy. The effort of building a model from these data is more proper to be called intellectual property.}\]
train. More importantly, e.g., the model itself can leak private information about its previous trainers (server or other clients). The TEE of one client then provides a trusted environment to respect other clients’ data privacy.

B. Key Challenges of Adapting Confidential Computing

The key challenges mostly lie in the feasibility and effectiveness of utilizing TEEs for various ML services due to TEEs’ limitations.

Constrained execution environments. TEEs have constrained resources for reducing the size of TCB. As given in Table I, the commonly used hardware TEE, SGX, provides 128MB secure memory \(^7\) while TrustZone-based TEEs, such as OPTEE or trusty, by default provide up to 16–64MiB secure memory. Depending on the TEEs software stack and architecture, the size and flexibility of protected memory vary. However, such memory sizes are still highly limited compared to the memory consumed by current ML algorithms, which can reach hundreds of MBs or GB. Although techniques using page swapping can increase available memory, it leads to significant overhead (e.g., 100–1000×) when running ML. \(^2\), \(^19\).

In addition, processor capabilities can be limited when running in the TEE mode. For example, privilege instructions are not allowed inside SGX enclaves, which makes supporting widely used functionalities (e.g., filesystem or multi-threading) difficult or even impossible. For instance, the SGX multi-threading model relies on untrusted OS for thread scheduling which could lead to synchronization-based attacks (e.g., AsyncShock \(^20\)). Similarly, making the secure world over-privileged in TrustZone architecture, caused mobile vendors to only provide limited TEE services and much-restricted TEE usage. Indeed, increasing computational resources is always preferable for ML developers but it can reduce the reliability and practicality of TEE-based solutions.

SDK supports. Modern ML framework usually requires numerous libraries and cross-compilation to support high-performant data loading and computations (e.g., matrix multiplication). However, most TEEs provide basic low-level SDKs. Even though open source TEE frameworks have been developed to support sandbox/container (see Table II), it is still hard to port all ML dependencies and necessary libraries into TEEs. For example, WebAssembly supports compiling binaries from many modern languages, e.g., Rust, which could have available ML libraries, such as autograd \(^21\), but they have much-limited functionality compared to Tensorflow or PyTorch \(^22\). OS-based containers can provide richer libraries, but still, the lightweight version OS cannot have a rich environment same as the normal OSs like Linux. Porting a normal OS to TEEs is also infeasible (and overkill in many use cases) because it either exceeds the TEEs’ size limits or leads to a huge TCB which destroys TEEs’ security benefits.

Privacy protection effectiveness. By deploying ML inside TEEs (for training/inference), the untrusted host itself cannot access the ML computations anymore. However, running inside TEEs does not automatically disable all potential privacy leakages. The other parts of the ML pipeline could still leak private information through various attack vectors. For instance, the produced result, e.g., predicted data labels or trained models, usually need to be transmitted out of the TEE through potentially untrusted channels (e.g., network, IPC, or file system) which could be used to exploit private information \(^10\), \(^13\), \(^123\), \(^124\), \(^125\). Thus, one particular strategy could have inconsistent protection effectiveness for other ML scenarios and different types of defined privacy \(^126\), \(^127\). As one example, while hiding model parameters (along with gradient updates and activations) inside TEEs can defend against DRAs, it has very low efficiency for MIAs as most membership information can leak from the model’s outputs, i.e., prediction results transmitted out of the TEE \(^123\), \(^65\), \(^128\). Therefore, for one specific TEE-assisted use case, the target privacy should be defined clearly, and the achieved protection efficiency also requires explicit analyses.

C. Existing Solutions of Confidential ML

Previous research has been dedicated to achieving ML with TEEs, and some aim to overcome the above challenges. We summarize previous research in Table III. Complete ML training/inference inside TEEs. The most straightforward approach is to deploy a complete ML training/inference process inside TEEs. In such a case, the maximum capability of the ML task is strictly limited to the TEE’s space and computing constraints. Then, the strategy is to use memory space efficiently – trading off the total number of the model’s layers and the number of neurons – and to maximize the efficiency of every “bit” of TEEs’ secure memory for ML computation. Specifically, we list several practical approaches as follows. i) Conducting inference instead of training. Training consumes much more computational resources because of backward propagation (e.g., memory used to save model gradients and intermediate activations). ii) Choosing a small batch size. A large batch size leads to large memory usage because every sample in this batch produces its own activations for all model layers. iii) Balancing the feature extractor (e.g., convolutional layers) and the classifier (e.g., fully connected layers). A properly designed feature extractor can decrease the feature dimension but still capture key features, enabling a compact classifier to achieve good performance.

Partitioned execution. To avoid exceeding the maximum secure computational resource and memory swapping, one approach is efficient compartmentalization (or partitioned execution) to actively optimize the memory usage in ML. Figure 3 shows two types of partitioned execution: i) Layer-based partition \(^128\), \(^117\), \(^22\), \(^39\), which works for models with layer architecture in general, and ii) Feature map-based partition \(^133\), \(^135\), which specifically aims to convolutional layers because of their high memory cost. Among them, whether to keep the first part or the last part inside TEEs (cases 1 and 2) depends on the security/privacy protection

\(^8\)The SGXv2 \(^118\) extensions allow additional flexibility in runtime management of enclave resources (e.g., adding memory to an enclave after the enclave is built and running) and multithreaded execution within an enclave.
| Work & Year | Confidential Computing | Machine Learning | Advanced Features |
|------------|------------------------|-----------------|-------------------|
| TEE Type   | SDK                    | DL Library      | Train-design  | Locat-ion | Protect Aim | Overhead (execution time) |       |
| Obliv. MP’16  [18] | SGX                    | Intel SGX SDK   | fast CNN    | ●         | S           | D               | 1 − 3% on Native | Data-obliviousness |
| Chiron’18  [20] | SGX                    | Intel SGX SDK   | Theano      | ●         | S           | D               | 4 − 20% on Native | Multi-enclaves, Ryoan VM |
| PRIVADO ’18 [13] | SGX                    | Intel SGX SDK   | ONNX        | ○         | S           | D               | ~ 17.18% on Native | Compiler, data-obliv. |
| Myelin’18  [13] | SGX                    | Intel SGX SDK   | TVM         | ●         | S           | D               | 4.7 − 9.2% on Native | DP for data obliviousness |
| Slalom’18   [12] | SGX                    | Intel SGX SDK   | Eigen       | ○         | S           | D               | SU. 4 − 20× on SGX | Freeva’s algorithm + GPU |
| Graviton’18 [129] | GPU-TEE               | CUDA RT driver  | Caffe        | ●         | S           | D               | 17 − 33% on GPUs | secure CUDA for TEE on GPUs |
| Occlumency’19 [41] | SGX                    | Intel SGX SDK   | Caffe       | ○         | S           | D               | ~ 72% on Native | On-demand loading, Channel partition |
| TensorSCONE’19 [117] | SGX                    | Intel SGX SDK   | TF           | ●         | S           | D               | ~ 3× on Native | Docker supported, Compiler |
| YerbaBuena’19 [138] | SGX                    | Intel SGX SDK   | Darknet     | ○         | S           | D               | up to 7.5% on Native | Layer-wise partition, Privacy measure |
| OMG’20  [116] | TrustZone              | SANCTUARY       | TFLM        | ○         | C           | M               | 2.11× on Native | - |
| DarkeTZ’20  [117] | TrustZone              | OP-TEE          | Darknet     | ●         | C           | M               | 3 − 10% on Native | Layer-wise partition, Privacy measure |
| TrustFL’20  [133] | SGX                    | Intel SGX SDK   | TF           | ●         | C           | TI              | ~ 2× on Native | GPU-outsourcing, Random sampling |
| Telekine’20 [32] | GPU-TEE               | ROCm & CUDA     | MXNet       | ●         | S           | D               | 10 − 41% on GPUs | Timing attacks, data-obliv. |
| MLCapsule’21 [133] | SGX                    | Intel SGX SDK   | Eigen       | ○         | C           | M               | 55 − 116% on Native | - |
| Trusted-NV’21 [134] | TrustZone              | OP-TEE          | -           | ○         | C           | M               | 22.8% on Native | Weights & Feature-map partition |
| Mem-Env’21  [135] | SGX                    | Azure CC (VM)   | Darknet     | ○         | S           | D               | 9 − 100% on Native | Channel & Y-plane partition |
| PPFL’21  [22] | SGX + TZ               | OE + OP-TEE     | Darknet     | ●         | C + S       | D               | ~ 15% on Native | Layerwise training, Privacy measure |
| Goten’21  [136] | SGX                    | Intel SGX SDK   | Eigen       | ○         | S           | D               | SU. 6.84× on SGX | GPU-outsourcing, Non-colluding servers |
| Citadel’21 [112] | SGX                    | SCONE            | TF           | ●         | S           | M + D           | 9 − 73% on Native | Multi enclaves for training |
| Fair. Audr’22 [138] | SGX                    | SGX-LKL-OE      | PyTorch     | ○         | S           | M + D           | ~ 30% on Native | Fairness audit, Modler + regulator |

1The reported overhead is given to show general performance and cannot be cross-compared due to measuring under different experimental settings. SU refers to speed-up, instead of overhead. Training: ● Not available; ○ Available Location: S = Server; C = Client

Protect Aim: D = Data privacy; M = Model intellectual property; TI = Training integrity

| Layer-based Partition |
|-----------------------|
| Keep first part in TEE |
| Keep last part in TEE |
| Rolling, on layers in TEE |

| Feature map-based Partition |
|----------------------------|
| Rolling, on sections in TEE |
| Rolling, on channels in TEE |

Fig. 3: Partition on the ML process (shown with model architectures) in order to protect it or a part of it inside TEEs.

Rolling on layers inside TEEs (case 3) naturally works for ML inference, because one input propagates forward throughout layers and never goes back; that is, it works in the way that the next layers are loaded into a TEE until one layer’s computation executes finished in the TEE. Rolling on feature maps (cases 4 and 5) reduce the realtime memory usage by applying GEMM (i.e., General matrix multiply) and img2col (i.e., Image to column format transform) functions on divided sections and channels of feature maps. However, this disables parallelization features and can increase computation time. Furthermore, since training involves backward propagation where more resources are required, partitioned execution is more necessary than inference. Partitioned execution for training could lead to higher overhead than inference in 1, 2, and 3 (increased from 10% to 20% approximately [117], [22]), because a higher volume of buffer will be transmitted between the TEE and normal OS. Note that in order to enjoy higher flexibility to support generic ML models, the partition should better not change the model architecture (unlike the model partition in the design of AlexNet [140]).

TEE-assisted accelerators. In addition to optimizing the limited TEE-protected resources, one can also extend the computation power and resources to accelerators (e.g., GPUs, NPU, TPUs, and FPGAs) [129], [136], [141], [142], [143]. However, most accelerators are considered untrusted and do not support TEE functionalities. To address this issue, one way is to use CPU-TEE (e.g., SGX) to coordinate the computation delegation and to establish a secret sharing protocol between the CPU-TEE and untrusted accelerators. In such a way, accelerators are able to perform executions on a “nonsensitive version” of the TEE computation, e.g., masked by one-time pads [19], [136]. This can also accelerate the ML’s linear
 computation i.e., matrix multiplication. There are also TEE-assisted record-and-replay based solutions, which are more suitable for resource-constrained and mobile GPUs. As an example, CODY [143] allows the full computation to be done inside GPU and uses TEEs like TrustZone for secure logging and inference monitoring. However, due to the complexity of heterogeneous SoC architectures, software-only secure channels between TEEs and accelerators without proper hardware-based root-of-trust weaken TEE security guarantees.

Another approach is to extend accelerators hardware, systems stack (e.g., the driver), and API support for enabling accelerator TEEs. Previous works [129], [132] achieve GPU-TEE by relatively small hardware modification (unmodified GPU/CPU core but modified peripherals like PCI) and the necessary system software (e.g., runtime, hypervisor, or drivers, etc.) These solutions could add up to 41% overhead on native GPUs. Similarly, SGX-FPGA [142] builds a secure hardware isolation path between CPU and FPGA to protect the sensitive data in both components and in interactions. It extends the capability of the original SGX to a counterpart FPGA enclave by leveraging the FPGA's physical unclonable function (PUF) to achieve the hardware RoT on the heterogeneous SoC. This approach is also integrated into some recent NVIDIA GPUs, such as H100 Tensor Core GPU Architecture [144]. Therefore, despite the current limitations, Confidential Computing would be supported on future accelerators while ML frameworks/solutions are not prepared yet.

**Attack-based privacy measurement.** The TEE protection needs measurements to show its effectiveness. Currently, the main approach for measuring TEEs security is through red-teaming and performing various attack vectors directly. Whether to and which attack to perform mostly depends on the threat model defined for the TEE which could be inadequate for different ML use cases. As one example, regarding the original input protection with TEEs, one performs attacks to reconstruct input data [128] while there is no need to conduct membership inference attacks. By contrast, TEE protection on membership information does not take care of other private information and only be measured with membership inference attacks [117]. However, due to the poor definition of various types of privacy, the attack-based analysis lacks a theoretical foundation on privacy leakage which sequentially makes the TEE-based privacy guarantee less trustworthy.

**V. CONFIDENTIAL COMPUTING HELPING MACHINE LEARNING INTEGRITY**

Intuitively, other than privacy guarantee, deploying ML training/inference process into TEEs could avoid malicious modifications to this process and therefore ensure integrity. However, in practice, as the other parts of the ML pipeline still can be breached, protecting only the training/inference stage does not always guarantee integrity. In this section, we present the key challenges in enabling ML integrity using TEEs and the solutions that exist or can be adapted.

**A. Key Challenges in Enabling ML Integrity**

**Large attack surface.** There is a large attack surface to breach integrity; however, one cannot expect to deploy the complete ML lifecycle inside TEEs to achieve an integrity guarantee due to the TEEs’ resource constraints. These limitations for achieving a proper integrity guarantee are even more severe compared with the issue we met when guaranteeing confidentiality. Specifically, while a curious-but honest (i.e., confidentiality-related) adversary has the goal to explore specific private information, the integrity-related adversary can be more diverse – it aims at not only actively changing the ML process but also even breaking the process by covertly changing one bit in updated weights/gradients. Indeed, to disclose confidential information, adversaries usually explore the target data/model and contained information in the model may also gradually decrease along with more aggregated information in training. However, to breach integrity, one just needs to change one step in the training process (even in one bit) among participants (e.g., Byzantine attacks [7] or synchronization attacks [59]).

**Uncontrolled input/output space.** While deploying ML training/inference inside TEEs avoids unauthorized direct changes in the produced result/models, the upstream pipeline, i.e., data preparation or input space, is uncontrolled. Editing the inputs and their labels can affect the later ML process (e.g., called “dirty” inputs/labels), which is also the practice of performing some attacks, e.g., poisoning attacks or adversarial example attacks. Adversarial examples [15], [57] to add calibrated noises to the input can seriously perturb the integrity of the input space due to the hardness of detecting the noisy perturbation. In addition to that, the downstream pipeline can be edited maliciously. For example, after the ML algorithm inside the TEE makes a prediction and transmits this result out, a server-side adversary fakes the result, which is also hard to detect on one result receiver side.

**Low interpretability of ML.** One major reason for the difficulty of detecting integrity perturbations is the low interpretability of current ML/DL [145]. Specifically, current practitioners train ML models using SGD, i.e., weights of neurons are updated iteratively to find the optimal solution. There is still no comprehensive theoretical framework to fully interpret the internal steps of ML training. Therefore, one cannot determine whether the weight value of a neuron is uncompromised or not without re-running the ML process using the same setting. With such low interpretability, malicious changes such as the adversarial perturbation in inputs and the changes in model training are hard to trace. Algorithm-based approaches have been proposed to alleviate this situation e.g., by removing outliers that are considered as unreasonable bits [7], by further training the model on adversarial inputs [146], or by cryptography protocols [147]. However, utilizing a system-based isolation technique (i.e., TEEs) to deal with the interpretability issue is not straightforward. One may still need to find a way to identify and assure key elements of the ML pipeline are trustworthy using TEEs.
B. Existing/Adoptable Solutions

Assurance mechanisms. To assure the integrity of ML training/inference, one way is to rerun the complete or a part of the process inside TEEs. Foreexample, the heaviest computation in ML is the matrix multiplication which is preferable to be outsourced to distrusted GPUs for acceleration. To verify whether this matrix multiplication is honestly performed, one can use Freivald’s algorithm which utilizes randomization to reduce time complexity from $O(n^3)$ (best-known matrix multiplication algorithm) to $O(kn^2)$ with a probability of failure less than $2^{-k}$ [148], [19]. Besides, based on the nature of iterative training of ML, one can also sample and verify only a fraction of the training process in TEEs to reduce the overhead [131], [49]. This will require backups for every several training iterations to reach a checkpoint faster, where Merkle hash tree-based method [150] can be used to reduce the storage overhead. This achieves less than 1% soundness errors even for the adversaries who have honestly completed 90% of training rounds [131]. There could be more efficient ways of sampling or verification based on watermarking [151]. How such an assurance guarantees the integrity of other stages in the ML pipeline deserves more investigation.

Input/output space control. The integrity of input/output in the ML pipeline needs to be assured considering that the input/output space is far more uncontrollable than the training/inference process. One way that is generally used for data assurance and can also be adopted in ML input control is to require the digital signature for the data generation. This can be done by hashing the generated sensor data (inside TEEs for example) to produce hash values. After that, sensor data are authenticated using these hash values [152], [153], [154]. At a later stage, the digital signature would also work in a similar way for remote attestation when outputting prediction results, considering that the result receiver trusts the TEE’s behavior. Nevertheless, it becomes harder especially for supervised learning because the data generation usually involves human annotations that are out of control of the digital signature.

Even in semi-supervised/unsupervised learning, it is possible to generate adversarial data without breaching sensors/-TEEs, e.g., by changing physical surroundings or doing abnormal behaviors that the ML does not expect to learn. Therefore, the exploration of how to assure the integrity of the input space using TEEs is still a research question. One example can be adopting detection mechanisms (e.g., [155], [156], [157]) to identify fake/dirty inputs inside TEEs, which deserves further investigations. Previous work also introduced system-wide TEE-assisted information flow control, particularly over peripherals and OS services to detect and mitigate such unauthorized IO-based attack vectors [158]. However, these techniques require fundamental changes in system software including commodity OSs.

VI. LIMITATIONS OF EXISTING TEE SYSTEMS

Despite the benefits of TEEs/enclaves, the right abstractions for securely and efficiently utilizing them are still not clear. Particularly, when combining various TEE solutions (e.g., for multi-platform use cases) or integrating with existing systems. Below we further detail the limitations of existing TEE systems that obstruct their use in ML.

A. Widening the Attack Surface

New features lead to new attack vectors. TEEs require rich functionalities and performance features to support running modern ML programs. However, merely adding hardware features without a comprehensive security model and mechanism with existing system software and privilege boundaries (e.g., sharing data or resources) can even increase the system’s complexity and open new attack vectors. For example, ARM TrustZone introduced hardware support for secure and non-secure worlds by adding several privilege levels in each world. However, the architecture limitations, such as insecure sharing mechanism and fixed/one-way security model, combined with insecure software designs lead to new attack vectors, particularly when sharing resources or exchanging data/control between different privilege layers [72], [78]. Figure 4 demonstrates some of these attack vectors in a TrustZone-assisted environment. Previous attacks, such as Boomerang [77] and HPE (Horizontal Privilege Escalation) [159], show how the semantic gap between heterogeneous compartments (e.g., userspace processes and TrustZone TAs (Trusted Apps)) leads to severe privilege escalation threats. Boomerang exploits a confused deputy vulnerability inside a TA via the shared memory between the two worlds and takes advantage of secure world privilege to make the host kernel memory accessible. Similarly, HPE attacks demonstrate that an adversary process could compromise another userspace process through a shared TA or insecure privilege management between two TAs.

Huge TCB caused by in-enclave LibOS. Moreover, in-enclave LibOS approaches for complex applications result in a huge TCB through porting all dependencies inside the enclave. These designs force developers to run large code in a single enclave, resulting in inefficient and over-privileged enclaves. This leads to wide range of new attacks due to exploiting in-enclave vulnerabilities [120], [72], insecure interactions with outside [160], [161], [72], or misusing enclaves to escalate privileges [77], [159], [162]. Currently, there is no systematic way to neither detect nor protect against these threats.

Vulnerable interactions with outside. None of the existing partitioning frameworks fully consider the complex attack surface originating from insecure interactions between the host OS, userspace processes, and enclaves. As a simple example, Vicarte et al. [59] show how asynchronous poisoning attacks on TEE systems lead to changing the accuracy and integrity violation of in-enclave ML models. Civet uses dynamic taint-tracking to control the flow of objects on enclave interfaces, but can not help for proper privilege separation and against more complex attacks (e.g., horizontal privilege escalation (HPE) attacks [159]) and other languages. Moreover, they do not consider in-address space compartmentalization the key issue of handling over-privileged enclaves. Sirius [158] shows
targeting these attacks require fundamental changes to underlying OS and TEE systems. Note that developers sacrifice performance and rich functionality for the security benefits of TEEs; which could be a huge cost without actual benefit considering the insecure designs of existing TEE systems.

**ML inside TEEs but still being attacked.** Existing vulnerabilities of TEEs still leave the deployment of ML under some forms of attacks. For example, by observing the access pattern to TEEs, previous work in Privado [114] can disclose the target class information i.e., classifying encrypted inputs with high accuracy (97% and 71% for MNIST and CIFAR10 respectively). Also, side-channel attacks on TEEs like timing, power, and Electromagnetic analysis could compromise the ML model privacy on some levels, e.g., reconstructing the model architecture and even model parameters [163], [164], [165]. Some of such attacks can be out of scope because our threat model does not include side-channel attacks. However, this still reflects the current limitations of the existing TEE system which deserve further investigations in future design (e.g., efficient oblivious operations for ML [18], [166]).

### B. Programmability & Performance

The quality of ML solutions highly depends on the proper processing of an enormous amount of data. However, almost all TEE systems are not designed considering this essential requirement. Many TEEs lack minimal hardware resources for efficient ML programming on modern CPUs as we explained earlier (Section IV). Currently, resolving these limitations causes significant additional performance overhead. As an example, MPTEE [167] proposes a mechanism for supporting more flexible memory management and dynamic permission enforcement inside enclave memory. It uses MPX’s three bound registers to offer the six common memory permissions (i.e., RWX, RW, RX, R, X, non-permission) with about 35% overhead on average and 57% in some cases. Besides, Oclum [168] proposed an MPX-based SFI technique for supporting in-enclave multi-processing, which adds a 10× slowdown compared to Linux processes. SGXv2 adds special instructions (e.g., EMODFR and EMODPE), to facilitate some of these issues including dynamic memory management. However, the overhead is not clear yet due to the lack of hardware support, and SGXv1 is still the dominating version for a while.

These improvements are not sufficient for most ML use cases. First, current designs can not be integrated with a wide range of accelerators such as GPUs, FPGAs, and TPUs. Despite recent efforts for supporting TEEs in these accelerators as described in Section IV-C their systems software and ML framework designs and evaluations are in the early phases [169], [129], [142], [132]. Depending on the level of protection in different ML stages, the overhead and toolchain supports can be highly diverse. Also, none of the previous work offers scalable solutions with multiple accelerators.

### C. Heterogeneity & Migration

Heterogeneous SoC architectures enable a wide range of functionalities and are now available even for modern IoT/edge platforms [170]. Modern SoCs contain heterogeneous CPUs (e.g., a combination of ARM and RISC-V architectures) and peripherals. As a result, the system stack on such devices includes multiple OSs (e.g., Linux and FreeRTOS), hypervisors, and different TEEs. As a simple example, recent NXP’s i.MX boards, simultaneously support different TrustZone implementations for mixed cortex-A and cortex-M cores (e.g., i.MX 8ULP or 8QuadXPlus). Secure hardware partitioning and sharing hardware resources within such ever-growing complexity is challenging. That is the reason why most systems software (e.g., Android [171]) does not expose heterogeneous TEEs to userspace applications despite hardware availability.

Moreover, heterogeneous TEEs could have different security models and resource constraints which make the migration of TEE-based ML solutions from one platform to another much more challenging. For instance, remote attestation support in TrustZone-based and SGX-based solutions are different. Naively, combining different security protocols does not lead to a secure solution. Also, as summarized in Figure 4, vulnerabilities can easily be introduced and propagated in various
privilege boundaries (within the different or same address spaces). Detecting and protecting against such a complex attack surface is one of the primary obstacles to securely handling TEE migrations. Ad-hoc approaches are neither practical nor scale well. We need a principled approach to deal with heterogeneous TEEs and future security threats. A naive migration could widen or worsen the attack surface.

VII. NEXT GENERATION AND BEYOND

In this section, we present the potential approaches to tackle above discussed issues, towards better privacy-preserving machine learning atop TEEs.

A. Strengthen privacy foundation in Confidential Computing

As one TEE’s primary protection goals, privacy needs to be strictly defined and preferably be defined with theoretical basics (e.g., information theory). While most privacy measures are empirical evaluated by conducting attack\[^1\] defining privacy theoretically is complicated due to the broad existence of attacks in ML.

Among candidates, differential privacy is a well-accepted definition of privacy. DP has been massively explored when applying to ML, but it faces the same generalization problem. As one example, one can apply DP-training \[^{172}\] to show the protection efficiency against DRAs that recovers original training data from gradients computed on it. As expected, this protection is not reasonable because typical DP-training guarantees the “membership” privacy of the trained final model, not (and far from) intermediate gradients. To measure on the pre-gradient level, one needs to adjust the DP’s granularity from sample-level (i.e., sensitivity of samples in the dataset) to feature-level (i.e., sensitivity of features in the sample). This level of DP noises will highly likely destroy the model utility but potentially provides a theoretical-based sense of what is the level of privacy for such protection.

Another line of work is to apply computational constraint-based $\mathcal{V}$ usable information theory \[^{173}\]. $\mathcal{V}$-information allows us to consider an attack family with bound computational power and limited knowledge. With the defined attack family, one can measure the capability of the complete attack family, which may include many attacks, by computing $\mathcal{V}$-information similar to Shannon mutual information. This $\mathcal{V}$-information then reflects the privacy leakage \[^{127}\]. Although such a method provides a theoretical base for measuring privacy, how to define a broad and general attack family still deserves further exploration.

Different types of privacy in ML are fragmental and overlapped, as concluded based on both previous empirical attacks and theoretical analyses. Without a solid understanding of privacy, TEE protection itself is not plausible; protection of one type of privacy could be invalid when the privacy goal changes. Therefore, a rigorous definition of privacy for specific use cases is the cornerstone for TEE protection.

\[^{1}\]Measuring privacy using attack success rates fails to explain and quantify privacy leakages, and then leads to “Arms Race” where current protections needs re-evaluation or re-design when new attacks happen.

B. Partitioned ML execution for heterogeneous TEEs

For now and in the foreseeable future, most TEEs on modern devices will be heterogeneous, and many will keep small considering the ubiquitousness of devices and the requirement of small TCBS. It is desired to avoid monolithic ML frameworks; so we could partition the solution with a focus on protecting the most critical ML computation or stages.

The partition can be done on different scales. On a low level, most ML’s computation forms with numeral calculations e.g., Multiply–accumulate operations, floating point operations, or matrix multiplication. Running a portion of these computations inside TEE is straightforward, but determining whether computation is not an easy task. First, this determination needs well-defined privacy as above mentioned, e.g., which type of privacy to protect. Second, it requires the knowledge of how these computations have different leakage of the target privacy. Currently, we have a superficial understanding of layer-level privacy differences, i.e., the last layer contains the most membership privacy. However, there is still a need i) to deepen the layer-level understanding of more types of privacy; ii) to investigate efficient layer-wise partitioning approaches; iii) to investigate any more fine-grained privacy understanding (i.e., matrix multiplication and feature maps), as the layer-level partition is still heavy for constrained TEEs. Furthermore, one can also execute some kinds of assurance and verification inside TEEs instead of hiding sensitive parts directly. Such a method has been used for ML integrity checking \[^{131}\] but still deserves further investigation i.e., combining with masking techniques to preserve privacy.

C. Dedicated TEE designs for general ML

Efficient ML execution requires modern computer architectures with parallelization features, e.g., multi-threading on CPUs to load data and on GPUs (or TPU, etc.) to perform matrix multiplication in forward and backward passes. These features are achieved by dedicately designing the hardware and the system of accelerated processing units, which most of current TEEs do not have. One solution is to equip TEEs with such parallelization features to increase the computational capability of ML. However, consequently, new features increase the TCB size. Synchronization bugs cause severe vulnerability of SGX \[^{120}\], and one can also expect a considerable number of bugs that potentially exist in GPU-TEEs. How to avoid such a dilemma is really depending on how we can limit the trust boundary and reduce the TCB when applying parallel processing. One example is to consider one ML accelerated unit (GPU/TPU) to be physically isolated from the rest of the main-board and the processing system in the same way as TPM, and to be accessed only through a secure bus from TEE-enabled CPUs. Although this constrains the trust boundary with assumptions that this dedicated GPU serves only for such private ML and no one can physically breach the GPU, there are more use cases applicable such as MLaaS with proper remote attestation and verification.

Interestingly, another way is to avoid multi-core features but focus on in-memory computation. This potentially closes the
Due to the heterogeneity of devices’ TEEs and ML workload, it is obvious that one solution cannot satisfy all needs. Therefore, designing the ML to be aware of TEEs and the capability of TEEs is critical. To achieve this, ML frameworks will determine the maximum workload to be executed with TEEs which first requires the TEE’s information such as memory size, processor speed, protected storage, and potentially other features like multi-threading and secure communication channels. Then, ML computations from the most sensitive to less are deployed into TEEs. Such a determination involves many variables that influence factors from system cost to ML performance; thus, one can potentially form a multi-objective optimization problem similarly to Neural Architecture Search for resource-constrained environments [177], [178], [179]. Furthermore, one may need techniques such as sparse approach [180], [181] of knowledge distillation [182], [183] to handle the heterogeneity of TEEs and differences in model architectures, especially when updating models in FL.

E. Protecting the full ML pipeline

Protecting the full pipeline is intractable without the participation of multiple TEEs (e.g., [184], [22]). Multiple TEEs are able to provide multiple trusted areas covering more devices and consequently more pieces of the pipeline. To achieve such “multi-party computation” based on TEEs, one needs to enable a verification mechanism for multiple participants so that TEEs from different chip manufacturers can be remotely attested by two or more challengers. For example, potentially one third-party proxy attestation service can help to verify TEEs from both Arm and Intel [98], so that the TEE system could cover multiple locations of the ML pipeline.

In addition to the TEE workflow, determining the most sensitive parts of ML for protection is also not trivial. On the smallest scale, TEE provides a more trustworthy area in the ML pipeline, such as an additional trust base for random number producer or DP noise addition. Besides, investigations on the privacy or integrity of the pipeline’s parts other than training/inference protection, such as data preparation, are vastly missing. Such protection on one specific part of the pipeline will involve again threat (privacy/integrity) definition, sophisticated TEE-based protection design, and performance measurement similar to the training stage protection. After such works on more stages of the pipeline, full ML pipeline protection will be formed concretely and be used to a greater extent and larger scale.

VIII. CONCLUSION

Protecting ML against privacy leakages and integrity breaches by the Confidential Computing techniques is an exciting and challenging new era. Although many studies have been dedicated to running the training and inference processes inside TEEs, they still face the limitation of available trust resources. Since ML requires much more trusted resources; the current protection levels only provides the confidentiality and integrity of that specific stage, training/inference, in the complete ML pipeline. Current defenses against privacy leakage are usually measured through red-teaming and attack-specific metrics, which make the final results less reliable and generalized. Furthermore, large attack surfaces exist at stages, especially at the upstream stage like data preparation, which can cause huge negative impacts on the ML pipeline. Rethinking of how to tackle such a dilemma, i.e., the conflict between the large scale of the complete ML pipeline and the need for small TCB, is of utmost importance when conducting ML with Confidential Computing.

Confidential Computing achieves a hardware-based root-of-trust establishing a more trustworthy execution environment for ML activities, but we should rethink whether “hiding” the training/inference process inside such enclaves is the optimal solution. Future researchers/developers need to better understand the underlying privacy issues in the ML pipeline so that later protections focus on vital parts. The full ML pipeline needs to be evaluated to avoid fruitless labor on training/inference protection only. The current trends indicated that both TEE evolution and ML advances (i.e., partitioning ML execution for TEEs, and designing dedicated TEEs for ML) will lead to more TEE-aware future architectures. These potential solutions can help achieve much stronger privacy and security guarantees without compromising ML computational performance and introducing TEE’s system overhead.

Suggestions to achieve ML with Confidential Computing for the hardware/system/framework designer are:

- Compared to security or integrity, privacy definition is still ambiguous. One needs to define the theoretical-based protection goal, e.g., with differential privacy or V-usable information, to have well-grounded privacy guarantee.
- Protecting the upstream of the ML pipeline such as the data preparation is of utmost importance due to its lack has intractable and negative impacts. It can potentially be realized by integrating TEE-based verification into data signature. Multiple TEEs/Conclaves can further benefit the full ML pipeline protection.
- Developing dedicated TEEs for general ML requires re-thinking and redesigning the hardware acceleration features and corresponding system stack. New paradigms such as in-memory computing that involves a smaller group of hardware and software modules may ease the condition due to the reduction of the TCB size and the attack surface.
- Designing the ML framework to be TEE-aware and partitionable for heterogeneous TEEs requires i) meticulously researching the privacy/integrity weakness of different ML granularity (from layer/feature map to numeral calculations), and ii) managing the TEE system to protect the most sensitive ML components efficiently (e.g., with masking) with a high priority.
[182] E. Diao, J. Ding, and V. Tarokh, “Heterolf: Computation and communication efficient federated learning for heterogeneous clients,” in International Conference on Learning Representations, 2020.

[183] J. Zhang, S. Guo, X. Ma, H. Wang, W. Xu, and F. Wu, “Parameterized knowledge transfer for personalized federated learning,” Advances in Neural Information Processing Systems, vol. 34, pp. 10092–10104, 2021.

[184] N. Volgushev, M. Schwarzkopf, B. Getchell, M. Varia, A. Lapets, and A. Bestavros, “Conclave: secure multi-party computation on big data,” in Proceedings of the Fourteenth EuroSys Conference 2019, 2019, pp. 1–18.