TEESlice: Slicing DNN Models for Secure and Efficient Deployment

Ziqi Zhang*
Peking University
China
ziqi_zhang@pku.edu.cn

Lucien K. L. Ng†
The Chinese University of Hong Kong
Hong Kong
luciengkl@ie.cuhk.edu.hk

Bingyan Liu*
Peking University
China
lby_cs@pku.edu.cn

Yifeng Cai*
Peking University
China
caiyifeng@pku.edu.cn

Ding Li*
Peking University
China
ding_li@pku.edu.cn

Yao Guo*
Peking University
China
yaoguo@pku.edu.cn

Xiangqun Chen*
Peking University
China
cherry@sei.pku.edu.cn

ABSTRACT

Providing machine learning services is becoming profit business for IT companies. It is estimated that the AI-related business will bring trillions of dollars to the global economy. When selling machine learning services, companies should consider two important aspects: the security of the DNN model and the inference latency. The DNN models are expensive to train and represent precious intellectual property. The inference latency is important because modern DNN models are usually deployed to time-sensitive tasks and the inference latency affects the user's experience. Existing solutions cannot achieve a good balance between these two factors. To solve this problem, we propose TEESlice that provides a strong security guarantee and low service latency at the same time. TEESlice utilizes two kinds of specialized hardware: Trusted Execution Environments (TEE) and existing AI accelerators. When the company wants to deploy a private DNN model on the user's device, TEESlice can be used to extract the private information into model slices. The slices are attached to a public privacy-excluded backbone to form a hybrid model that has similar performance to the original model. When deploying the hybrid model, the lightweight privacy-related slice is secured by the TEE and the public backbone is put on the AI accelerators. The TEE provides a strong security guarantee on the model privacy and the accelerators reduce the computation latency of the heavy model backbone. Experimental results show that TEESlice can achieve more than 10× throughput promotion with the same level of strong security guarantee as putting the whole model inside the TEE. If the model provider wants to further verify the correctness of the accelerator's computation, TEESlice can still achieve 3-4× performance improvement.

CCS CONCEPTS

• **Software and its engineering** → **Software safety**; • **Computing methodologies** → **Neural networks**; • **Security and privacy** → *Trusted computing*.

KEYWORDS

Neural networks, Model protection, TEE

ACM Reference Format:
Ziqi Zhang, Lucien K. L. Ng, Bingyan Liu, Yifeng Cai, Ding Li, Yao Guo, and Xiangqun Chen. 2022. TEESlice: Slicing DNN Models for Secure and Efficient Deployment. In *Proceedings of the 2nd ACM International Workshop on AI and Software Testing/Analysis (AISTA '22), July 18, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3536168.3543299

1 INTRODUCTION

The advance in deep learning techniques has spawned enormous new commercial services and products. The market size and the revenues of AI technologies will sustainably grow. One popular commercial model to provide deep learning services is MLaaS (Machine Learning as a Service). It is adopted by leading commercial companies such as Google, Amazon, and Microsoft.

There are two important aspects of the MLaaS paradigm: the security of the companies' model and the inference latency. Protecting the model security is important because the DNN models are precious intellectual property for the companies. Training DNN models is expensive in terms of time, money, and human effort. It is reported that a model trained by Google costs 61,000 USD per training run [4]. The inference latency is important because

*Key Laboratory of High-Confidence Software Technologies (MOE), School of Computer Science, Peking University
†Department of Information Engineering, The Chinese University of Hong Kong

AISTA '22, July 18, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9387-4/22/07…$15.00
https://doi.org/10.1145/3536168.3543299

more and more DNN applications are deployed on mobile or embedded devices. Such applications are sensitive to the inference time because they are usually used for real-time tasks, such as speech recognition and object detection in autonomous driving.

When providing deep learning services, there are three options for the companies to choose from. However, none of the approaches can achieve a perfect balance between the DNN model protection and low latency. (1) The first approach is that the user uploads the input data and the company's cloud server returns the inference results. This approach suffers from large communication latency and the user may not want to upload the private data. (2) The second approach is that the user and the company collaboratively compute the inference output via some cryptography-based techniques (*e.g.* homomorphic encryption (HE) and multi-party computation (MPC)). The drawback is that cryptography algorithms introduce significant overhead and are orders of magnitude slower than state-of-the-art other approaches [11]. (3) The third approach is that the companies deploy the model on the users' devices [7]. As illustrated in Figure 1, with modern AI accelerators (*e.g.* GPU, NPU, and TPU), this approach provides offline low-latency DNN inference service. Given that inference time is an important factor for many modern DNN applications, in this paper, we will focus on this approach.

However, sending the model to the user device may leak the model information. For the companies, the DNN model is an expensive property and should be carefully protected. Besides, if the model weights are leaked, an adversary may reverse-engineer the training data and tampers with the data privacy. To solve this problem, Trusted Execution Environments (TEEs) can be used to deploy the models [3]. TEEs use both hardware and software to guarantee the confidentiality and security of the protected code. Popular TEEs include Intel SGX and ARM TrustZone. But according to the prior literature [11], TEEs are at least an order of magnitude less efficient than the best available untrusted accelerators.

The reason for the high latency for the TEE-based solution is that it puts the whole DNN model inside TEEs and treats different model parts equally. In this paper, we argue that *different model parts should be treated differently.* To reduce the inference latency, only the important private model layers should be protected by the inefficient TEEs. Shadownet [10] tried to only put the non-linear layers inside the TEE. But Shadownet trades model security for inference efficiency and the security level is weak. As we will illustrate in Section 2.3, their defense can be cracked by carefully designed attacks. In this paper, we propose TEESlice that only protects the private model slices with TEEs to reduce inference latency.

As Figure 1 shows, TEESlice is a hybrid solution. First, a public DNN backbone is downloaded from the Internet. Instead of deploying the original private model, TEESlice extracts a privacy-related model slice from the private model. The extracted slice is attached to the privacy-excluded DNN backbone and the hybrid DNN model has the same functionality and similar performance as the original private model. The model slice has significantly smaller computation complexity than the original private model. On the user device, the lightweight slices are deployed in the TEEs and the heavy backbone is put on the untrusted accelerators. In this way, the private information in the slices is protected by the TEEs and the heavy computation operations are performed by the accelerator. TEESlice

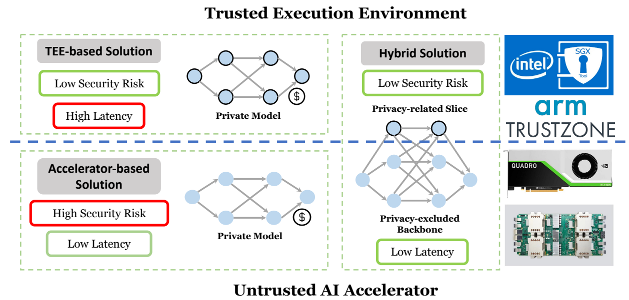

Figure 1: Illustration of security-performance trade-off of different solutions.

enjoys both a low security risk and low inference latency at the same time.

Specifically, TEESlice consists of two stages: private slice extraction and hybrid model deployment. The first stage converts a given private model to model slices with limited labeled data. A generative network is trained simultaneously to supply more training data. The slices are first densely attached to the model backbone and gradually pruned during the training process. After slice extraction, the hybrid model is deployed on the user device. To reduce the swapping overhead of the TEE memory, TEESlice uses the untrusted OS memory as an external cache. To provide strong slice protection, all the input and output feature of the model slices are encrypted. The model owner can also choose to verify the correctness of the outsourced computation to provide a reliable service.

We evaluated TEESlice with Intel SGX and modern commercial GPUs. Experimental results show that TEESlice is up to 14.52× faster than the baseline if the model owner only wants to protect the model information. If the model owner wants to guarantee the correctness of the inference result, TEESlice still achieves 3-4× performance gain.

2 BACKGROUND

2.1 Trusted Execution Environments

A trusted execution environment ensures the protected coda and data are isolated from all the programs on the same host. This hardware guarantees the confidentiality and integrity of the protected software against strong adversaries such as a malicious OS. However, TEE's security guarantee comes with a significant performance cost. The computation ability of TEEs remains a far cry from untrusted devices. For example, the memory limitation of SGX is 128MB and only about 93MB is available for user applications. When the used memory exceeds the memory limitation, the TEE will suffer from severe paging overheads. In Slalom [11], the author evaluated the performance gap between SGX and a high-end GPU (Nvidia TITAN XP). For MobileNet, the throughput of SGX is 16 images/sec while the value of GPU is 56× higher, up to 900 images/sec.

2.2 Deep Neural Networks

The architecture of DNN is composed of various basic layers each layer performs one operation. The layers can be divided into two categories: linear layers and non-linear layers. Linear layers include convolutional layers, fully connected layers, and batch normalization layers. Non-linear layers include ReLU layers and Sigmoid layers. Traditional encryption techniques such as homomorphic encryption can handle linear layers but can not deal with non-linear layers. Linear layers occupy most of the computational costs due to the complex matrix multiplication operations. According to prior literature [11], 98.5% computation resource is spent on linear layers for a VGG16 model. In software community, researchers tried to analyze the DNN behavior with model-based techniques [6, 12, 13].

2.3 Limitation of Prior Work

Some recent work tried to outsource the heavy computation layers from TEEs to co-located AI accelerators. Slalom [11] outsources the linear layers to the GPU and keeps non-linear layers inside TEEs. But Slalom can only protect input privacy rather than model property. If a similar technology is applied to protect model weights, the plain-text input and output easily leak the encrypted weights. ShadowNet [10] obfuscate model weights via linear transformation and outsource the transformed weights. The obfuscation of this work is not built upon known encryption techniques and lacks a strong security guarantee. The transformation technique is similar to affine cipher which is considered highly insecure with respect to a strong adversary.

As ShadowNet [10] is a recently proposed framework to obfuscate model weights on the user side, we depict two possible attacks to reveal the potential threat. Assume a linear layer with n neurons, the weight is $\mathbf{w} = [\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_n]$. ShadowNet obfuscates $\mathbf{w}$ with linear transformation and permutation. The obfuscated weight $\hat{\mathbf{w}}$ is computed as:

$$\hat{\mathbf{w}} = [\mathbf{w}_1 \cdot \lambda_1 + f_1,\ \mathbf{w}_2 \cdot \lambda_2 + f_2,\ \ldots,\ \mathbf{w}_n \cdot \lambda_n + f_n] \cdot P_\pi, \quad (1)$$

where λ_i and f_i are random variables. π is a random permutation and P_π is the corresponding permutation matrix, *i.e.* $P_\pi(i, j) = 1$ if $\pi(i) = j$ otherwise 0. ShadowNet sends $\hat{\mathbf{w}}$ and $[f_1, f_2, \ldots, f_n]$ to the untrusted accelerators.

One attack is that the shuffle permutation may be recovered by an attacker because TEEs usually cannot hide the memory access pattern [5]. Then the attacker can match the pair $\mathbf{w}_i \cdot \lambda_i + f_i$ and f_i to recover the weight $\hat{\mathbf{w}}_i$. Another attack is to utilize some public knowledge of the model weights. For example, the attacker knows the architecture of the victim model and can analyze the public model that has the same architecture to extract a certain pattern of the kernel weights. The attacker may exhaust all pairs $(\hat{\mathbf{w}}_i, \hat{\mathbf{w}}_j)$ and compute $\hat{\mathbf{w}}_i - \hat{\mathbf{w}}_j$ to see if the known pattern appears. For example, some kernels of the public model may mainly contain zeros but have a high magnitude on a few weights. The DNN model fine-tuned from these public models or with the same architecture may have similar characteristics. The attacker can find these kernels, analyze the outsourced weights and find the weights that have lots of similar values. These values should approximate f_i because they are generated by $0 * \lambda_i + f_i$.

2.4 Goals and Challenges

We consider a two-party scenario between a model provider P and a model user U. P provides the functionality of a trained DNN model $F(\cdot)$ to the user U but does not want to reveal any more information. The goals of this paper are summarized as follows:

- **Data privacy**: The model provider P learns no information about the user's input data $\mathbf{x}$.
- **Model privacy**: The user U learns no more information about F than what is revealed by $\mathbf{y} = F(\mathbf{x})$.
- t**-Integrity**: For a DNN F and a user input $\mathbf{x}$, the probability that F outputs an incorrect value $\tilde{\mathbf{y}} \neq F(\mathbf{x})$ is less than t.
- **Efficiency**: The inference time of the proposed technique should be faster than the vanilla implementation that put the whole model inside TEE.

To implement TEESlice, there are two major challenges. One challenge is how to select a proper position to attach model slices automatically. Normal DNN models have tens of layers. Achieving satisfactory accuracy requires various model slices attached to different positions. Searching all the combinations of the slice positions and training each combination to find the best model with minimal computational cost inside TEEs are even more complex and time-consuming. TEESlice aims to reduce the search complexity and efficiently find the best slice positions.

Another challenge is how to extract the model slice at the post-training stage in a data-limited scenario. We aim to design a general framework that can protect all the existing DNN models at the post-training stage. It means, given a trained model, we want to convert it to the deployable architecture that a model slice is attached to a public privacy-excluded model backbone. An important challenge is the lack of large-scale training data at the post-training stage. Given the same training dataset, it is easy to extract the model slices. However, in recent years with the rise of privacy concerns and the regulations (such as GDPR), the whole training dataset is not controlled by the model providers. Modern AI companies have turned to new training paradigms, such as federated learning and continuous learning. In these cases, the model owner does not have enough data to extract model slices and needs a data-limited technique.

3 PROBLEM FORMULATION

Assumption. We consider an MLaaS framework where a model provider deploys the model to the client's device. All the computation is performed on the client's device. The private user data does not upload to the provider's server thus the user privacy is protected.

For the client's device, the model owner can control how the model is deployed on the client's device, including which part of the code and data are deployed inside TEEs and untrusted AI accelerators. For TEEs, the secrecy of data and the integrity of the code are guaranteed. We assume the host OS is malicious and the model owner has no control of the data secrecy and program correctness on the host OS. This challenging assumption is the same as prior literature [11]. Note that the assumption of malicious host OS does not hinder the integral deployment of TEE programs because the remote model owner can attest to the code and data of the enclave at the initialization stage [3]. We assume the user is unable to infer any more information about the model than what is intentionally exposed by the model owner.

Defender's Ability. We assume the model owner has a trained model as the teacher $T(\cdot)$ and has white-box access to T (knows the architecture and parameters). The defender can download any public pre-trained model M_{pub} from the Internet. We also assume the defender has a small amount of labeled data D_{label} (*e.g.* validation dataset or 10% of the training data) but does not have large-scale training data.

Defender's Goal. The defender here is the model owner who wants to protect the confidentiality and intellectual property of the model functionality. This functionality is decomposed into the function of the public model backbone and the function of private slices. All the information of private slices should keep secret, including the layer weights and the layer input/output.

4 APPROACH

4.1 Overview

The pipeline of TEESlice is depicted in Figure 2. The proposed framework consists of two stages: private slice extraction and hybrid model deployment.

The private slice extraction stage converts the original private model into model slices with limited labeled data. TEESlice needs to download a public model that contains no private information from the Internet. The model slice is attached to the public model to form a hybrid model. During training, only the privacy-related model slices are updated and the weights of the public model are kept unchanged. To supplement the insufficient training data, we train a generator from the original model. The generated data and the labeled data are combined to train the model slices. In the training process, the hybrid model acts as the student and the original model is the teacher. To further reduce the inference cost of model slices, TEESlice uses an iterative pruning technique that gradually filters less-important slices. At the end of this stage, the hybrid model has a similar performance to the original private model.

Then in the model deployment stage, the hybrid model is sent to the client's device to provide model inference service. The public backbone is deployed on the untrusted AI accelerators and the private slices are deployed inside TEEs. To reduce the paging overhead when dealing with a large matrix, we implement a memory manager that actively switches the data between TEE's memory and the untrusted rich memory. To prevent the leakage from layer inputs/outputs, the feature map exchange between the TEEs and accelerators is also encrypted. Besides, as the adversary may contaminate the client OS, the AI accelerator may produce incorrect results. To guarantee the integrity of the inference results, TEESlice takes the output of all the outsourced layer into TEEs and check the correctness with little computation cost.

4.2 Private Slice Extraction

This section illustrates the pipeline of private slice extraction. TEESlice uses the original private model to train a generative network to supplement the training data. During the training process, the original model is the teacher and the hybrid model acts as the student. The hybrid model is first constructed by adding dense slices to the public model to ensure enough model capacity to learn the teacher's knowledge. Then TEESlice performs an iterative pruning algorithm to filter the less important slices with little accuracy sacrifice. Then we will introduce three modules in different subsections: semi-supervised learning pipeline, densely sliced model training, and iterative slice reduction.

4.2.1 *Semi-supervised Learning Pipeline.*

We denote the labeled dataset as D_{label} and the original private model as the teacher $T(\cdot;\phi)$. The goal of this module is to train two models: a generative network G and a student network S. The generative network G takes a random vector z as input and outputs synthetic samples that are similar to the real training samples in D_{label}. The structure of the student network S will be illustrated in the next subsection. In this subsection, we treat the student $S(\cdot;\theta^S)$ as a black-box model with forward and backward functions. This subsection focus on the data generation and training framework.

The slice extraction framework is motivated by the prior work [1]. The goal of knowledge extraction is to train the student model to have similar outputs as the teacher. The optimization goal can be formulated as

$$H(T,S) = \mathbb{E}_{x\sim D}\ ||T(x;\theta^T) - S(x;\theta^S)||_1, \tag{2}$$

i.e. to minimize the L1 norm between the teacher and the student. In the data-free setting, the input samples are generated by G. Thus the goal becomes

$$H(T,S) = \mathbb{E}_{z\sim p_z}\ ||T(G(z;\phi);\theta^T) - S(G(z;\phi);\theta^S)||_1, \tag{3}$$

where z is the random noise as the image generation seed.

The idea of semi-supervised learning is straightforward. The optimization goal of the student S is to output similar results as the teacher network T given any input. The goal of G is to generate difficult samples. The student's predictions on such samples tend to deviate from those produced from the teacher. The joint goal formulation is as follows:

$$\min_{\theta^S}\max_{\phi}\mathbb{E}_{z\sim p_z}\ ||T(G(z;\phi);\theta^T) - S(G(z;\phi);\theta^S)||_1, \tag{4}$$

To optimize this equation, we adopt a two-stage framework: knowledge extraction and sample generation. The knowledge extraction stage fixes the generator's parameters ϕ and optimizes student model's parameters so that the student has similar behavior as the teacher. The workflow of the sample generation is the opposite. The second stage fixes the student model and optimizes the generator so that the generator produces hard samples that tend to mislead the student. Learning such hard samples improves the efficiency of the knowledge extraction process. The two stages are executed alternately so that the student model and the generator are optimized simultaneously.

4.2.2 *Training of Densely Sliced Model.*

In this subsection, we will introduce how to construct the densely sliced model. Motivated by [8], the general idea of constructing the model is to augment the existing public model with privacy-related slices. Such slices are attached to the public model backbone and are placed parallel with the main blocks. We take CNN as an example to illustrate how to construct the model architecture, but the proposed technique can be applied to other model architectures such as BERT.

Suppose a CNN model implements the function $f(x;\theta)$ and consists of L layers. The function of layer l is B_l and $f(x;\theta)$ can be

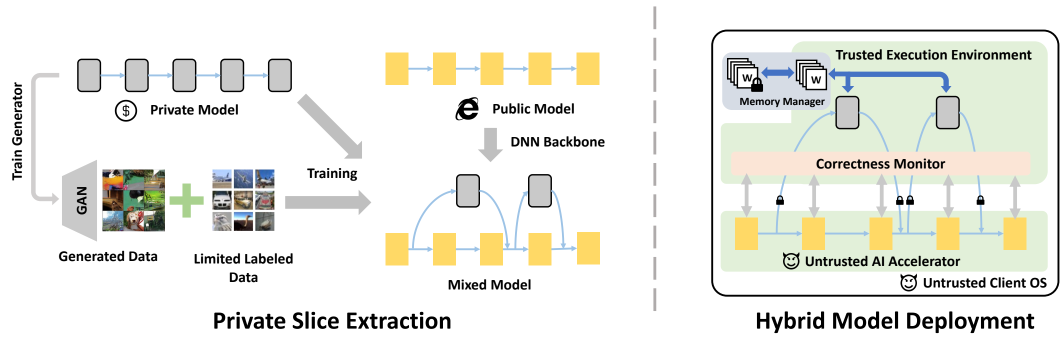

Figure 2: The pipeline of TEESlice.

Algorithm 1: The pipeline of knowledge extraction.

Data: Teacher model T, a little label data D_{label}, an initialized student model S, training epochs *epochs*

Function `KnowledgeExtraction`(T, S, D_{label}):
    for $e \leftarrow 1$ **to** *epochs* **do**
        `// Sample Generation`
        Fix the student network S ;
        Update the generator G to maximize Equation 4 ;
        Fix the generator G ;
        The generator G produces synthetic dataset D_{syn} ;
        `// Knowledge extraction.`
        Merge the dataset $D_{\text{train}} = D_{\text{label}} \cup D_{\text{syn}}$;
        Update the student network S to minimize Equation 4 with D_{train} ;
    end
    return the trained student model S;

---

represented as

$$f(x;\theta) = (B_L \circ B_{L-1} \circ \cdots \circ B_1)(x;\theta). \tag{5}$$

In the assumption of this paper, all the blocks B_l and their parameters are adopted from a public-available pre-trained model. To learn the knowledge of the private task, we augment each layer with several proxy slices $\{A_p^l(\cdot)\}$. The proxy slice A_p^l connects layer B_p with layer B_l. The output of layer B_l are summed by the outputs of $A_p^l(\cdot)$ weighted by different scalar values. Let h_l represent the weighted-summed output after layer B_l, the computation process can be formulated as

$$h_l = B_L(h_{l-1}) + \sum_p^l \alpha_p^l A_p^l(h_p), \tag{6}$$

where $\alpha_p^l \in [0, 1]$ is scalar value to measure the importance of the A_p^l's output.

Note that the goal of TEESlice is to minimize the computational cost inside TEEs, thus we need to carefully design the slice algorithm. One important thing is that we constrain the number of proxy slices for each backbone layer to k, *i.e.* $p = l - k$. Otherwise, the number of proxy slices grows rapidly and exceeds the TEE memory. Another elaborated design is that the complexity of the proxy slices is much simpler than that of the backbone layers. Take a ResNet18 model as an example, the backbone layers are mostly convolution layers of kernel size 3×3. Following [8], we choose the convolution kernel size of these proxy slices as 1×1. It means for the same layer input, the computation cost of the proxy slice is $\frac{1}{9}$ of the backbone layers. We limit both the number of proxy slices and the computation complexity of each layer to alleviate the total computational cost inside TEEs.

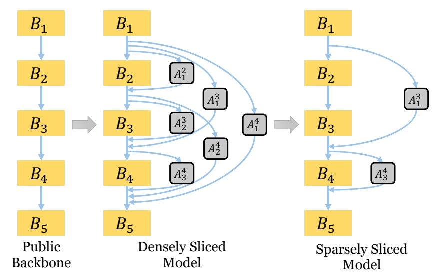

Figure 3: The pipeline of constructing the sliced model.

Figure 3 shows an example of a five-layer backbone model and the number of proxy slices $k = 3$. In the middle of Figure 3, we can see that TEESlice attaches six proxy slices alongside the model backbone. Note that although the number of proxy slices is larger than the number of backbone layers, the TEESlice's computation cost is smaller than putting the backbone whole model inside TEE. It is because the computation complexity of A_p^l is much smaller than B_l. Besides, the number of A_p^l will be reduced by a large margin in the next step.

After constructing the densely sliced model, we use the knowledge extraction pipeline in Algorithm 1 to train the model. The constructed model acts as the student model in the learning pipeline. To ensure that only the proxy slices learn the private knowledge, we update proxy slices and fix the backbone layers. Besides optimizing the parameters of A_p^l, we also optimize the scalar weights α_p^l. In this way, the model can automatically learn which slices are more important. The trained model achieves high performance on the private task and can be deployed across the TEEs and the untrusted AI accelerators on the user's device. However, the densely sliced model still requires plenty of proxy slices to reside inside TEE. To further prune the TEE's computation cost, we perform an iterative slice reduction in the next step.

4.2.3 Iterative Slice Reduction. The goal of this stage is to reduce the number of proxy slices of the hybrid model. This stage consists of two steps: *static slice pruning* and *iterative slice pruning*. In the first step, the densely sliced model prunes several slices that have little contribution to the model output. Then the pruned model is fine-tuned to recover model performance. In the second step, TEESlice iteratively prunes the proxy slices for several rounds. In each round, a few slice layers are pruned and the pruned model is fine-tuned. In this way, the performance loss in each round is under control and can be recovered by the fine-tuning stage. Figure 3 shows the pruned model. After several rounds of pruning-and-fine-tuning, there are only two proxy slices remaining (A_1^3 and A_3^4).

The pruning criterion is the scalar weight α_p^l. During the full model training, the scalar weights α_p^l are optimized with the layer parameters. According to Equation 6, the layers with small α_p^l contribute less to the feature output. The influence of pruning such proxy slices is much smaller than pruning other layers.

The performance loss of pruning is controlled by a pre-defined threshold δ (such as 1%). Let p_{ideal} represent the performance of the original model. The lower bound of the tolerable accuracy is $p_{\text{tol}} = (1-\delta) \cdot p_{\text{ideal}}$ and p_{tol} can be determined before the pruning stage. In each step, the slices are pruned as long as the fine-tuned model performance is above the tolerable limit p_{tol}. If the performance of the pruned model can not recover to the p_{tol}, it means the pruned model needs more training time and TEESlice will not prune new slices.

The pruning pipeline is depicted in Algorithm 2. The static slice pruning needs a pre-defined threshold α_{static} to filter out the proxy slices. The slice layers with $\alpha_p^l < \alpha_{\text{static}}$ are pruned. Heuristically we set α_{static} to be 0.1. The pruned model becomes the student model and is re-trained by the semi-supervised learning framework. The iterative slice pruning needs two hyper-parameters: the number of the pruned layers in each round n and the total number of pruning rounds *rounds*. In each round, we first evaluate the model performance p_r. If the current model satisfies the performance requirement ($p_r > p_{\text{tol}}$), TEESlice selects n proxy slices with the smallest scalar weights α_p^l and trains the pruned model. Otherwise, TEESlice skips the pruning operation and continues to train the insufficiently-performed model.

Algorithm 2: The static and iterative pruning pipeline.

Input: Teacher model T, the public pre-trained model M_{pub}, a little labeled data D_{label}, validate dataset D_{val}, pruning hyper-parameters α_{static}, n, and *rounds*
Output: The sparsely sliced model S_{sparse}
Construct the densely sliced model S_{dense} from M_{pub};
Train the model S_{dense} = `KnowledgeExtraction(`T, S_{dense}, D_{label}`)` ;
`// Static slice pruning`
Prune the model S_{dense} by threshold α_{static} and get S_{static} ;
Train the model S_1 = `KnowledgeExtraction(`T, S_{static}, D_{label}`)` ;
`// Iterative slice pruning`
for $r \leftarrow 1$ **to** *rounds* **do**
  Compute the accuracy p_r of S_r on D_{val} ;
  if $p_r > p_{\text{tol}}$ **then**
    Store the model $S_{\text{sparse}} = S_r$;
    Select n proxy slices with smallest α_p^l ;
    Prune the selected layers of S_r ;
  end
  Train the student model S_{r+1} = `KnowledgeExtraction(`T, S_r, D_{label}`)` ;
end
return The sparsely sliced model S_{sparse}

4.3 Secure Deployment

In this section, we will introduce how the hybrid model is deployed and provide ML service without leaking private information. After the model owner has trained the sparsely sliced model, she can provide a machine learning service by sending the trained model to the client's device. The privacy-related slices are encrypted and the public backbone is stored in plain text. When processing the user's input, the private-related slices are loaded inside TEEs' memory and the backbone is loaded to the untrusted AI accelerators. Note that the privacy-related slices can only be decrypted inside TEEs with the model-owner-provided keys. All the computations related to the proxy slices are conducted inside TEEs. This scheme provides a strong security guarantee on the private information of the slices.

However, the deployment of the sliced model is not easy and there are three major challenges. (1) How to deal with the limited memory size of TEEs and reduce the potential swapping overhead. (2) How to eliminate the information leakage from the input/output of the private proxy slices. (3) How to verify the correctness of the outsourced DNN layers. For the first challenge, we implement a memory manager that uses the untrusted OS's memory as the TEEs' cache. The manager actively switches memory chunks into and out of the TEE memory. The memory chunks that are switched out of TEEs are encrypted. To solve the second problem, we encrypt the input and output features of the private slices. The encrypted input can be fed into linear layers of the public backbone to produce encrypted outputs. The output feature is then decrypted inside the TEEs. As non-linear layers can not operate on the encrypted features, we put such layers inside TEEs for a strong security guarantee. But putting all the non-linear layers into TEEs will cause a severe switch overhead between TEEs and accelerators, we only put the non-linear layers that are located behind the first private slice. To solve the third challenge, we perform Freivalds' algorithm inside TEEs following [11] to verify the correctness of linear operations with little computational cost. In the following subsections, we will introduce each technique in detail.

4.3.1 Memory Management. Managing the secure memory is important when protecting the DNN inference with TEEs. It is because the size of the TEEs' secure memory is very limited but the computation of DNN is memory-intensive. To solve this problem, we

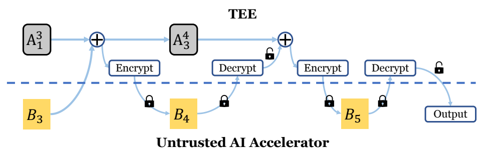

Figure 4: An illustration of feature encryption.

implemented a memory manager that can actively switch memory chunks into and out of TEEs. The memory manager first scans the private slices, computes the total needed memory, and prepares the needed memory chunks. When TEESlice needs more memory than the memory limit, it encrypts the memory chunks and stores them in the untrusted memory. The operations that need the same size of memory share the same memory chunks. When TEESlice needs the data in the untrusted memory, the encrypted chunks are loaded into TEEs for decryption. In this way, the untrusted memory acts as a cache of the TEEs' memory. The overhead of memory switch between TEEs' memory and untrusted memory is less than the overhead between TEEs' memory and external disks. TEESlice can achieve a better balance between the memory support and switch overhead.

4.3.2 Encrypted Feature Computation. As the output of the protected slices may leak information about the slice weights, TEESlice needs to encrypt the slice output before it is evicted to the untrusted environments. It means all the internal features that are computed based on any slice output should be encrypted. In other words, the internal features that lie after the first proxy slice must be encrypted when they are transferred from TEEs to the untrusted AI accelerators. The linear layers on the untrusted accelerators can operate on the encrypted data and produce the encrypted output. The output can be decrypted inside TEEs as if these layers directly compute the plain-text data.

Figure 4 shows how the internal features are encrypted for the sparsely sliced DNN in Figure 3. To make it easy to understand, we assume all the layers are linear layers. The features prior to the layer A_1^3 and B_3 are not encrypted because they are not computed from any proxy slice outputs. After layer A_1^3 all the features should be encrypted when transferred from TEEs to untrusted AI accelerators. The input of layer B_4 is encrypted because the feature is computed by summing the outputs of A_1^3 and B_3. The output of layer B_4 is then decrypted inside the TEE. TEESlice does not encrypt the input of A_3^4 because this layer is inside TEEs. Similarly, the input and output of B_5 are encrypted. The final model output is decrypted by the TEE and transferred to the outside environment.

Then we will illustrate the feature encryption mechanism and why the linear layers can correctly handle such features. Let $\mathbf{x}$ represent the plain-text feature. We first quantize $\mathbf{x}$ to integer field and then encrypt the quantized feature. We choose the 8-bit quantization algorithm. Let ζ and δ be the minimal and the maximal value of $\mathbf{x}$, respectively. A 8-bit integer can represent the value range from $[\zeta, \delta]$ to $[0, 2^8)$. The quantization scale factor is $s = \frac{2^8-1}{\delta-\zeta}$ and the bias factor is $z = -\lceil \zeta \cdot s \rceil$. The quantization function is

$$\hat{\mathbf{x}} = \text{clip}(\lceil s \cdot \mathbf{x} + z \rceil), 0, 2^8 - 1). \tag{7}$$

Then we encrypt the quantized feature with a one-time-pad mask. Let q be a large prime number and $\mathbf{r}$ be a random mask with the same size as $\mathbf{x}$. Each value in $\mathbf{r}$ is a random number selected from $[0, q]$. The encrypted feature is computed by

$$\mathbf{x}_e = (\hat{\mathbf{x}} + \mathbf{r}) \ \% \ p. \tag{8}$$

$\mathbf{x}_e$ is sent to the untrusted accelerators and becomes the input of the linear layers. Let linear$(\cdot)$ be the linear function. Note that here the linear layers include both FC layers and convolutional layers. The layer output on the encrypted data is linear$(\mathbf{x}_e)$. To decrypt the layer output, TEESlice needs to pre-compute linear$(\mathbf{r})$ following the prior work [11]. In the TEE, the layer output linear$(\hat{\mathbf{x}})$ can be recovered by linear$(\mathbf{x}_e)$ − linear$(\mathbf{r})$ because:

$$\begin{aligned} &\text{linear}(\mathbf{x}_e) - \text{linear}(\mathbf{r}) = \text{linear}((\hat{\mathbf{x}} + \mathbf{r}) \ \% \ p) - \text{linear}(\mathbf{r} \ \% \ p) \\ &= \text{linear}((\hat{\mathbf{x}} + \mathbf{r}) \ \% \ p - \mathbf{r} \ \% \ p) = \ \text{linear}((\hat{\mathbf{x}} + \mathbf{r} - \mathbf{r}) \ \% \ p) \\ &= \text{linear}(\hat{\mathbf{x}} \ \% \ p) \ = \ \text{linear}(\hat{\mathbf{x}}) \end{aligned} \tag{9}$$

The last equation holds as long as p is larger than 2^8.

Note that following [11], the computation of linear$(\mathbf{r})$ has two choices. One choice is to compute linear$(\mathbf{r})$ in an off-line phase inside the TEE and store the results. Another choice is that the model owner computes linear$(\mathbf{r})$ and sends the results to the clients along with the private model. In both cases, the TEEs' online computation workload is not increased. The only additional overhead is to store/load the pre-computed results outside/into the TEE. With our memory management mechanism, this overhead can be reduced by a large margin.

4.3.3 Correctness Verification. One security concern of outsourcing DNN computation is that the computation results may be wrong and TEESlice should verify the correctness and raise the alarm as soon as the outsourced results are wrong. This concern is realistic as the user's OS may be compromised by a third-party adversary. The adversary may tamper with the GPU software and induce the GPU to produce wrong results. The computation cost of correctness verification should be much smaller than the cost of performing the original computation inside the TEE.

Following [11], TEESlice adopts Freivald's Algorithm [2] to verify the correctness of linear layers. We will briefly introduce the mechanism to verify FC layers and convolutional layers. Let $\mathbf{y}$ be outsourced result and FC layers can be represented as $\mathbf{y} = fc(\mathbf{x}; W) = \mathbf{x}^\top W$. To verify FC layers, TEESlice first sample a random vector $\mathbf{u}$ that has the same shape as $\mathbf{x}$. Similar to the feature encryption, TEESlice needs to pre-compute an intermediate result $\tilde{\mathbf{u}} = W\mathbf{u}$. In the online phase, TEESlice compares $\mathbf{y}^\top\mathbf{u}$ and $\mathbf{x}^\top\tilde{\mathbf{u}}$. If the two values are equal, the outsourced result should be correct with high probability. Otherwise, TEESlice thinks the outsourced result is tampered and raises alarms. The complexity to verify the results is $|\mathbf{x}| + |\mathbf{y}|$. It is much smaller than the weight multiplication complexity $|\mathbf{x}| \cdot |\mathbf{y}|$.

The algorithm to verify convolutional layers is similar but with minor modification. Let c_{in} and c_{out} be the channel size of the input feature and output feature. The size of $\mathbf{u}$ is c_{out} and $\tilde{\mathbf{u}}$ is computed by multiplying W and $\mathbf{u}$ along the output channel. The two verification vectors $\mathbf{y}^\top\mathbf{u}$ and $\mathbf{x}^\top\tilde{\mathbf{u}}$ should be compared with proper reshape. The verification complexity is $|\mathbf{x}| + |\mathbf{y}|$ as well, compared to the convolutional complexity $|\mathbf{x}| \cdot k^2 \cdot c_{out}$ (k is the convolution kernel).

Table 1: The system throughput for different solutions. The performance promote over the SGX baseline is displayed in brackets.

		ResNet18	ResNet50	ResNet101
SGX		31.79	7.48	4.82
GPU		969.69 (30.5 ×)	300.46 (40.16 ×)	164.52 (34.13 ×)
TEESlice	w/o verify	301.88 (9.49 ×)	108.65 (14.52 ×)	43.74 (9.07 ×)
	w/ verify	125.49 (3.94 ×)	27.11 (3.62 ×)	15.14 (3.14 ×)

5 EXPERIMENTS

5.1 Experiment setting

We implemented TEESlice with Python and C++. The slice extraction part is written with Pytorch 1.5 and the model deployment part is implemented with Intel SGX SDK 2.15. We reused some code from Goten [9]. We reimplemented the basic deep learning operations (such as convolution operations and matrix multiplication) with SGX SDK. It is because the memory size of the TEE is too small for even one layer of the DNN models. The required memory size for one layer of a ResNet18 model may easily exceed the TEE limitation. Existing implementations of deep learning operations do not consider this problem and can not be used in our scenario. To solve this problem, we reimplemented a memory-aware version of the basic DNN operations. In our implementation, the input/output features and layer weights are split into small chunks which are managed by the memory manager. These chunks are loaded into the TEE memory in turn to perform certain computations.

We use the popular image classification task to evaluate TEESlice. The dataset is CIFAR10 and the model includes ResNet18, ResNet50, and ResNet101. We conduct the experiments on a desktop computer with a CPU Intel(R) Core(TM) i7-8700 CPU and a GPU GeForce GTX 1080. The evaluation metric is throughput (images/second).

5.2 Inference Efficiency

In this section, we will display the results of the efficiency comparison. We compare TEESlice with two baselines: deploying the model in the SGX (TEE-based solution) and deploying the model on the GPU (accelerator-based solution). The results are displayed in Table 1. The throughput of the SGX baseline is pretty low, ranging from 4.82 to 31.79. It means for the most simple image classification task, the TEE-based solution can not handle large complex models. The throughput of GPU is orders of the throughput of SGX baseline.

For the performance of TEESlice, we evaluated two settings: without correctness verification and with correctness verification. The first setting is reasonable because a correctness guarantee is a less important requirement for the model owner. The first setting evaluates the upper bound of TEESlice under the most basic requirement. When TEESlice does not need to verify the correctness of outsourced computation, TEESlice can achieve about 10× higher throughput than SGX baseline. The most remarkable improvement is ResNet50. TEESlice can process 108.65 images per second and is 14.52× faster than the TEE-based solution. If the model owner has the stronger requirement to verify the computation correctness, TEESlice is still 3-4× faster than the baseline. The throughput improvement demonstrates the effectiveness of TEESlice.

6 CONCLUSION

In this paper, we implemented a novel DNN deployment framework, TEESlice, that utilizes TEEs and GPUs to provide a strong security guarantee on the property of DNN models and minimizing the inference overhead. TEESlice consists of two stages: private slice extraction and hybrid model deployment. The first stage extracts private model slices from the original model in a semi-supervised manner. The slices are attached to a public model to form a hybrid model that has the same functionality as the original model. In the second stage, the private slices are deployed inside the TEE and the public backbone is put on the GPUs. The TEE protects the slices' private information and GPUs conduct the heavy computation of the model backbone. Experiments show that TEESlice can achieve up to 14.52× performance speedup to protect the model secrecy. If the model owner requires to guarantee the correctness of the inference result, TEESlice still achieves 3-4× performance speedup.

ACKNOWLEDGMENTS

We would like to thank the anonymous AISTA reviewers for their valuable feedback. This work was partly supported by the National Natural Science Foundation of China (62141208, 62172009).

REFERENCES

[1] Gongfan Fang, Jie Song, Chengchao Shen, Xinchao Wang, Da Chen, and Mingli Song. 2019. Data-free adversarial distillation. *arXiv preprint arXiv:1912.11006* (2019).

[2] Rusins Freivalds. 1977. Probabilistic Machines Can Use Less Running Time.. In *IFIP congress*, Vol. 839. 842.

[3] Lucjan Hanzlik, Yang Zhang, Kathrin Grosse, Ahmed Salem, Maximilian Augustin, Michael Backes, and Mario Fritz. 2021. Mlcapsule: Guarded offline deployment of machine learning as a service. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 3300–3309.

[4] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. 2020. High accuracy and high fidelity extraction of neural networks. In *29th USENIX Security Symposium (USENIX Security 20)*. 1345–1362.

[5] Ben Lapid and Avishai Wool. 2018. Cache-attacks on the ARM TrustZone implementations of AES-256 and AES-256-GCM via GPU-based analysis. In *International Conference on Selected Areas in Cryptography*. Springer, 235–256.

[6] Yuanchun Li, Ziqi Zhang, Bingyan Liu, Ziyue Yang, and Yunxin Liu. 2021. ModelDiff: testing-based DNN similarity comparison for model reuse detection. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 139–151.

[7] Bingyan Liu, Yao Guo, and Xiangqun Chen. 2021. PFA: Privacy-preserving Federated Adaptation for Effective Model Personalization. In *Proceedings of the Web Conference 2021*. 923–934.

[8] Pedro Morgado and Nuno Vasconcelos. 2019. Nettailor: Tuning the architecture, not just the weights. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 3044–3054.

[9] Lucien KL Ng, Sherman SM Chow, Anna PY Woo, Donald PH Wong, and Yongjun Zhao. 2021. Goten: Gpu-outsourcing trusted execution of neural network training. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 14876–14883.

[10] Zhichuang Sun, Ruimin Sun, Changming Liu, Amrita Roy Chowdhury, Somesh Jha, and Long Lu. 2020. ShadowNet: A secure and efficient system for on-device model inference. *arXiv preprint arXiv:2011.05905* (2020).

[11] Florian Tramer and Dan Boneh. 2018. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287* (2018).

[12] Ziqi Zhang, Yuanchun Li, Yao Guo, Xiangqun Chen, and Yunxin Liu. 2020. Dynamic slicing for deep neural networks. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 838–850.

[13] Ziqi Zhang, Yuanchun Li, Jindong Wang, Bingyan Liu, Ding Li, Yao Guo, Xiangqun Chen, and Yunxin Liu. 2022. ReMoS: Reducing Defect Inheritance in Transfer Learning via Relevant Model Slicing. In *2022 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*.