Phantom: Privacy-Preserving Deep Neural Network Model Obfuscation in Heterogeneous TEE and GPU System

Juyang Bai, *Johns Hopkins University;* Md Hafizul Islam Chowdhuryy, *University of Central Florida;* Jingtao Li, *Sony AI;* Fan Yao, *University of Central Florida;* Chaitali Chakrabarti and Deliang Fan, *Arizona State University*

https://www.usenix.org/conference/usenixsecurity25/presentation/bai-juyang

This paper is included in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Phantom: Privacy-Preserving Deep Neural Network Model Obfuscation in Heterogeneous TEE and GPU System

Juyang Bai
Johns Hopkins University

Md Hafizul Islam Chowdhuryy
University of Central Florida

Jingtao Li
Sony AI

Fan Yao
University of Central Florida

Chaitali Chakrabarti
Arizona State University

Deliang Fan
Arizona State University

Abstract

In this work, we present *Phantom*, a novel privacy-preserving framework for obfuscating deep neural network (DNN) model deployed in heterogeneous TEE/GPU systems. Phantom employs reinforcement learning to add lightweight obfuscation layers, degrading model performance for adversaries while maintaining functionality for authorized user. To reduce the off-chip data communication between TEE and GPU, we propose a Top-K layer-wise obfuscation sensitivity analysis method. Extensive experiments demonstrate Phantom's superiority over state-of-the-art (SoTA) defense methods against model stealing and fine-tuning attacks across various architectures and datasets. It reduces unauthorized accuracy to near-random guessing (e.g., 10% for CIFAR-10 tasks, 1% for CIFAR-100 tasks) and achieves a 6.99% average attack success rate for model stealing, significantly outperforming SoTA competing methods. System implementation on Intel SGX2 and NVIDIA GPU heterogeneous system achieves 35% end-to-end latency reduction compared with most recent SoTA work.

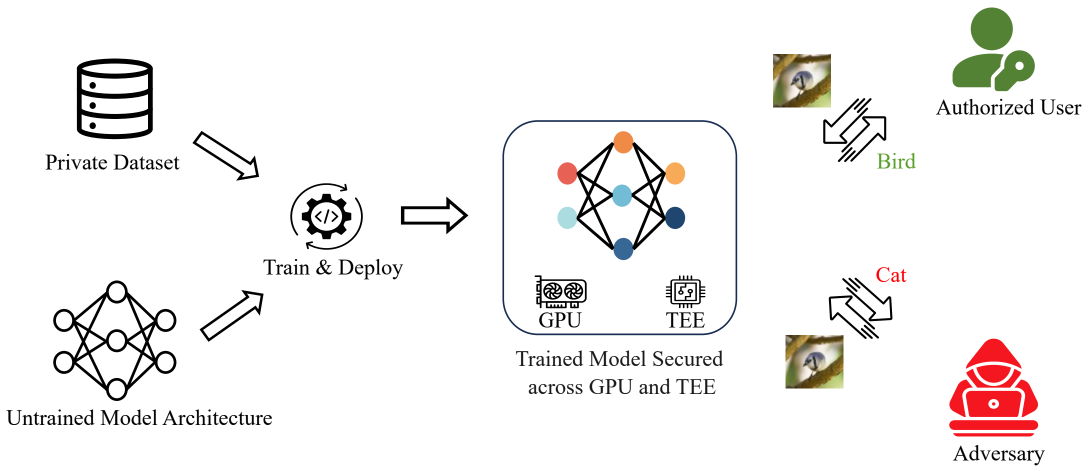
1 Introduction

Machine learning (ML) has revolutionized numerous fields, including computer vision, natural language processing to healthcare and autonomous systems [1–5]. Deploying production machine learning system is a costly and time-consuming process, often requiring a massive amount of training data, computational resources, and expert knowledge. As a result, model owners invest tremendous resources to build SoTA DNNs in order to provide competitiveness. This considerable investment has led to growing concerns about the security of the intellectual property embodied in these models. In particular, the risk of model theft or unauthorized replication by adversaries has emerged as a critical issue.

One of the most threatening attacks that compromise ML confidentiality is model stealing [6]. Model stealing attacks aim to reconstruct a substitute model that has similar inference behavior to the victim model [1] (e.g., accuracy and fidelity). This process typically involves an adversary sending carefully crafted inputs to the target model and using the returned predictions to train their own "knockoff" model [7]. These attacks can be launched against ML models that only expose their functionality to adversaries through inference APIs [8]. More recent attacks leverage a combination of side-channel attacks and reverse engineering techniques to enhance model extraction [9, 10]. For instance, DeepSniffer [11] extracts model architecture information by monitoring the inference-time memory access patterns via GPU memory-bus snooping. DeepSteal [12] exploits the rowhammer-based side channel to exfiltrate partial weight bits, which are then utilized to build an accurate local model. Carlini et al. [13] introduce a *training data extraction attack* that extracts verbatim text sequences from language models. With the additional knowledge of the model via the aforementioned attack vectors, an adversary can then perform more powerful model extraction, called the *fine-tuning attack*, which trains a local model initialized from the partially-known state the victim model (e.g., model weights and architecture) using limited training dataset. Model stealing with fine-tuning attacks leads to direct intellectual property theft and unauthorized model replication, significantly undermining competitive advantage or research efforts from model owners [8, 12].

One approach to protect the confidentiality of DNN models during inference is to use cryptographic primitives. Specifically, homomorphic encryption (HE) [14] enables computations on encrypted data, allowing model inference without decrypting sensitive information [15] (e.g., user inputs). Secure multi-party computation (MPC) [16] divides model parameters and computations across multiple parties to prevent any single entity from accessing the complete model or data [17]. Despite the good privacy guarantees, these cryptographic

[1]In this paper, the victim model refers to a private DNN model trained using proprietary datasets and computational resources. The obfuscated model denotes the model resulting from the application of various protection methods to the victim model. A pre-trained model refers to a publicly available model.



Private Dataset
Train & Deploy
Untrained Model Architecture
GPU
TEE
Trained Model Secured across GPU and TEE
Authorized User
Bird
Cat
Adversary

Figure 1: Overview of Secure Model Deployment. The model owner trains and deploys a DNN model in a heterogeneous TEE and GPU computing system, with sensitive knowledge stored in the TEE. Authorized users receive correct responses, while adversaries are given incorrect answers, protecting the model's private knowledge.

methods dramatically increase computational requirements and the model inference time, making them impractical for real-time applications or resource-constrained edge devices [18].

An alternative strategy leverages trusted execution environments (TEE) supported by mainstream chip vendors such as Intel Software Guard Extensions (SGX) [19] and Arm TrustZone [20]. SoTA TEE (i.e., SGX) establishes isolated secure enclaves that offer hardware-supported data confidentiality and integrity protection against privileged software and advanced hardware exploits [19]. While being more efficient than cryptographic techniques, TEE applications have inherent constraints, such as the limited size of secure memory and non-trivial metadata management overhead [19]. Recent studies on secure ML inference utilize TEE to serve as the root of trust, and outsource computation to external untrusted accelerators (e.g., GPUs) [18, 21, 22]. These works aim to optimize ML system performance by moving the majority of the inference computation externally while relying on operations inside TEE to protect data outside of enclaves (e.g., for user inputs [8]). Unfortunately, since certain portions of the model (e.g., the weight parameters) outsourced are visible to attackers, recent studies demonstrate that adversaries can still leverage the observed partial model information to perform successful model stealing attacks [22]. As a countermeasure, the work in [22] proposes to fine-tune a *public* pre-trained model (PTM) by only updating weights on certain inserted layers, which are kept in the TEE. The layers that belong to the *original* PTM are offloaded externally. Since all the model parameters *influenced* by model owner's dataset are secured inside TEE, model confidentiality is believed to be guaranteed. It is worth noting that such mechanism is limited to scenarios where a secretive model is built from pre-trained models, and does not apply to more general cases where model is initialized (e.g., model architecture) and trained from scratch.

In this paper, we propose *Phantom*, an end-to-end TEE-enabled DNN model privacy-preserving framework in heterogeneous TEE/GPU systems, targeting to defend against model stealing. Phantom employs a reinforcement learning-based architecture search method to add a small portion of redundant lightweight convolution layers together with the corresponding keys, which obfuscates both DNN architecture and weight parameters, creating a transformed version of the original model that is running in the untrusted computing environment (i.e., GPU). In TEE, a simple and lightweight 'MUX' operation with the keys is able to recover the real computing path from obfuscated paths caused by the learned redundant parameters. Then, the authorized user could achieve the complete full function of the protected DNN model. While, the adversary could only access the obfuscated model in the untrusted environment without keys, where the model is trained to provide false output, as shown in Figure 1. In the Phantom framework, we leverage the GPU's computational power by offloading all linear layers (including both benign parameters and learned obfuscated parameters), such as convolutional layers, to the GPU. This allows the model to benefit from hardware acceleration for the most computationally intensive operations. Meanwhile, all non-linear layers (e.g. ReLU) and MUX operations are executed within the TEE, protecting critical components of the DNN model. The contributions of this paper are as follows:

- We propose a novel privacy-preserving DNN model obfuscation framework, *Phantom*, to obfuscate both model architecture and weights through a reinforcement learning based searching algorithm equipped with novel multi-objective reward function for optimizing obfuscation efficacy and overhead. Our obfuscated model effectively degrades the model performance for adversaries to the level of random guessing (e.g., approximately 10% accuracy for CIFAR-10 dataset).

- The obfuscated DNN model is partitioned to execute in TEE/GPU systems, where only the authorized user with keys could achieve full model functionality and accuracy. Extensive experiments on fine-tuning attacks and model stealing attacks demonstrate the superior performance of our methods over SoTA works [22, 23]. Interestingly, we demonstrate our defensive method could successfully defend against powerful fine-tuning attack under various learning rates, assuming adversary could access partial model parameters in an untrusted environment and 10% training data. While, the prior SoTA works [22, 23] both failed in such strong attacks.

- Our system implementation of obfuscated models from Phantom into heterogeneous Intel SGX2 and GPU system reveals an important finding that unlike previous implementation on SGX1, where the bottleneck is computation within TEE, the primary bottleneck becomes the frequent data communication between SGX2 and GPU. This challenge has not been fully investigated and analyzed. To overcome such new challenge, we propose layer-wise obfuscation sensitivity analysis to constraint obfuscation only on the most sensitive layers, thus leaving other layers untouched, to significantly reduce the overall system overhead. Compared with prior work [22], our approach reduces the overall system latency by 35%.

2 Background and Related Work

2.1 Attack Methods

Prior works have identified and explored numerous attack vectors that can be potential threats to model privacy and intellectual property [7, 8, 24–29]. Model stealing attacks aim to reconstruct a functionally equivalent model by querying the target model and observing its outputs. Tramèr et al. [8] demonstrated the feasibility of such attacks against black-box machine learning models, highlighting the vulnerability of models exposed through prediction APIs. Their work showed that even with limited query access, attackers could reproduce models with high fidelity, raising concerns about the protection of proprietary model architectures and parameters. Building upon this, Jagielski et al. [24] introduced high-fidelity extraction attacks that could recover models that are functionally close to the victim model and match its architecture. Their approach combined optimization-based extraction with data-free distillation, achieving SoTA extraction fidelity for both neural networks and decision trees. Orekondy et al. [7] proposed the Knockoff Nets framework, which demonstrated that it's possible to steal machine learning models via prediction APIs without any knowledge about the model's architecture or the data it was trained on. Their method used a reinforcement learning-based strategy to adaptively generate queries, significantly improving the efficiency of model-stealing attacks. Recently, Carlini et al. [30] showed that model stealing attacks can be even more powerful than previously thought. They demonstrated that it's possible to extract models with near-perfect fidelity using orders of magnitude fewer queries than prior work, highlighting the increasing sophistication of these attacks.

Fine-tuning attacks represent a more powerful threat to model privacy. In these attacks, adversaries start with an obfuscated model and fine-tune it on a small dataset to recover its full functionality or adapt it to a new, related task. This approach can inadvertently reveal information about the original model architecture or training data. Song and Shmatikov [31] demonstrated how fine-tuning attacks could compromise the privacy of language models. Their work showed that models fine-tuned on sensitive data could leak information about that data, even when the pre-training dataset was public. The effectiveness of these attacks is further amplified in the context of transfer learning, a popular technique in which pre-trained models are adapted for new tasks. Wang et al. [28] explored how transfer learning can be exploited to steal functionality from pre-trained models. They showed that an attacker with access to a fine-tuned model could potentially recover significant information about the base model, compromising its intellectual property. He et al. [32] introduced a novel attack called "Model-Inversion Fine-Tuning" (MIFT), which combines model-inversion techniques with fine-tuning to reconstruct training data from black-box access to fine-tuned models. Their approach showed that even models fine-tuned on a small amount of private data could be vulnerable to privacy leaks.

2.2 Trusted Execution Environment

With the increasing concerns about the security of data on remote computing platforms (e.g., cloud computing), TEEs have garnered significant interest in recent years. TEEs address the risk of data exposure by protecting against attacks from privileged software, such as malicious operating systems or hypervisors, as well as physical threats like memory bus snooping. In a TEE, the CPU serves as the root of trust, offering data security for user-defined regions of an application called *enclaves* through hardware-enforced encryption and attestation [19, 33–35]. Variants of TEEs have been adopted by all major processor vendors, including Intel SGX [19], AMD SEV [36], and ARM TrustZone [20]. Given the strong security guarantees provided by TEEs, many privacy-sensitive workloads have been designed to leverage their capabilities, including those aimed at protecting the privacy of entire models. Recent works have investigated the possibility of preserving model privacy by executing selective layers inside the TEE while offloading the majority of computations to more powerful external accelerators, such as GPU [21, 22, 37]. SoTA works [22, 37] can protect model parameters by only executing non-linear layers inside TEE, while offloading majority of the large linear layers (which can be computed over *masked* model parameters [21]).

Current State of Intel SGX. Intel introduced trusted computing in their consumer processor series with SGX1 (i.e., *SGX-Client*) [19]. A key limitation in this early adaptation is limited available system memory for enclave processes (particularly, <128MB usable enclave memory). Enclaves with larger memory footprints suffer from non-trivial performance degradation due to the expensive memory swapping operations between protected and unprotected memory regions. Intel SGX2 (i.e., *SGX-Server*) [38] uses *total memory encryption* instead of limited enclave memory, superseding the earlier SGX implementation. This extends the available protected memory for enclave processes (i.e., upto 512GB), which significantly reduces the enclave execution overhead for large memory footprint applications. SGX2 offers near-native performance, making it more efficient and capable of handling larger datasets and more complex queries in machine learning applications.

2.3 TEE-Shielded DNN

TEEs have emerged as a promising approach to protect the privacy and integrity of DNN during inference. TEEs provide hardware-isolated secure enclaves that safeguard sensitive code and data from unauthorized access, even in the presence of a compromised operating system.

Several research efforts have explored the use of TEEs for securing DNN inference. These approaches can be categorized into two primary methods: *TEE-only* [39–41] and *TEE-GPU* partition [18, 21, 22, 42] methods. TEE-only method encapsulates the entire DNN model within TEE, providing comprehensive protection. In contrast, TEE-GPU partition method adopts a hybrid approach, protecting privacy-sensitive components of the DNN model in TEE while leveraging GPU acceleration for non-sensitive computations, thereby optimizing both security and performance.

Ohrimenko et al. [39] implement secure machine learning through a dual-enclave architecture leveraging SGX (server-side) and TrustZone (client-side) TEEs. The system aims to protect model privacy through data-oblivious algorithms and core primitives that prevent information leakage via memory access patterns. The framework executes training entirely within client TrustZone enclaves, while server-side SGX handles secure aggregation of encrypted client updates. Memory constraints are alleviated through streaming processing and optimized data structure design, enabling large-model processing in secure chunks. PPFL [40] offers model privacy in federated learning through a dual-TEE architecture utilizing TrustZone (client-side) and SGX (server-side). PPFL employs greedy layer-wise training where each layer trains independently within TEE until convergence. All training occurs within client TrustZone, while server-side SGX handles secure aggregation. This mechanism ensures continuous protection by keeping model layers inside TEEs during both training and aggregation phases, only exposing them after convergence, making PPFL the first framework to achieve full model protection in federated learning. T-Slices [41] is a novel framework for secure deep learning inference on TrustZone-enabled edge devices with limited trusted memory. The key innovation lies in its dynamic slicing technique that enables execution of large DNN models within constrained TEEs by fragmenting each layer into smaller *slices* that fit within available trusted memory, overcoming the limitations of traditional layer-wise partitioning approaches. The system employs an optimized memory management scheme where encrypted slices are sequentially loaded into TEE, processed securely, and aggregated, with trusted memory allocated only for actively processing slices. By maintaining data encryption outside TEE and performing all computations within the secure environment, T-Slices ensures complete model and data confidentiality while efficiently managing memory constraints. However, these TEE-only approaches suffer from significant performance overhead due to limited computational capabilities and memory constraints within secure enclaves, especially for large deep learning models. These limitations have motivated hybrid approaches that combine TEE security with accelerator performance.

One of the pioneering works in TEE-GPU partition methods is Slalom, proposed by Tramèr and Boneh [21]. Slalom leverages Intel SGX to protect DNN privacy and integrity. The key innovation of Slalom is its novel partitioning approach: linear layers are outsourced to a GPU for acceleration, while nonlinear operations are performed inside the TEE. This strategy improves performance compared to fully TEE-based execution while maintaining strong security guarantees. Slalom demonstrated that it's possible to achieve both privacy and efficiency in DNN inference, opening up new possibilities for secure machine learning. Focusing on the model privacy challenges of edge computing, Mo et al. [18] introduced DarkneTZ, a system designed for edge devices using Arm TrustZone. DarkneTZ explores the trade-offs between security and performance by selectively executing sensitive layers of a DNN within the TEE. Their work demonstrated that protecting even a small number of layers can significantly enhance model privacy with minimal performance overhead. This approach is particularly valuable for resource-constrained devices where full model protection might be impractical. Zhang et al. [22] conducted a comprehensive evaluation of existing TEE-shielded DNN partition (TSDP) approaches. He found that current TSDP approaches follow a training before-partition strategy, which may not provide the level of security originally assumed and may have potential vulnerabilities in these systems. TEESlice [22], a partition-before-training strategy to generate a hybrid model, which is a public backbone and private slices. This innovative approach allows for more precise control over which parts of the model are protected, addressing the memory constraints often encountered in mobile TEEs. By carefully designing the partitioning scheme, TEESlices achieves a balance between security and performance that is well-suited to on-device deployment scenarios. Expanding the concept of model partitioning, Lee et al. [42] introduced Occlumency, a system that leverages SGX for privacy-preserving inference on edge devices. Occlumency proposes optimizations to reduce the memory footprint and computational overhead of secure inference, making it feasible to run protected models on devices with limited resources. Their work highlights the importance of considering hardware limitations when designing TEE-based protection schemes.

While TEE-based approaches offer strong security guarantees, they face several challenges. Memory limitations of TEEs, especially on edge devices, can restrict the size of models that can be fully protected. Performance overhead due to context switching and secure memory access can impact inference speed. Additionally, the inability to utilize specialized hardware accelerators (e.g., GPUs) from within the TEE can limit computational efficiency.

3 Threat Model

Following the prior model privacy protection works [37], we consider a powerful adversary could access the untrusted execution environment, including the operating system (OS) and GPU. The adversary can observe and steal the model knowledge and data outside the TEE, but cannot access the ones inside the TEE. In line with industry practices and previous research [18, 22, 43], we assume that the deployed DNN models provide only class labels as output to both adversaries and authorized users. Any intermediate results, such as prediction confidence scores, remain protected within the TEE. We assume the adversary can query the victim model with limited numbers to build a transfer dataset, which can be used to train a surrogate model for model stealing attack [12, 22]. Furthermore, we assume the adversary can obtain at most 10% original training dataset for fine-tuning attack, which is a more powerful assumption based on prior works [12, 23].

4 Phantom

4.1 Overview

In this work, we propose *Phantom*, a novel privacy-preserving deep neural network model obfuscation framework in heterogeneous TEE and GPU system, targeting to defend against model stealing and fine-tuning attacks, while maintaining high computational efficiency. Phantom employs a reinforcement learning-based architecture search method to add small and lightweight learned layers together with the corresponding "keys", creating a structurally different yet functionally equivalent transformed form running in untrusted environment. Note that, such keys are the index indicating the redundant layers computed in the untrusted environment that mislead the final model outputs. The keys are only known to the authorized user stored and processed within TEE. This architectural transformation serves to obscure the model's true structure and functionality from potential adversaries in untrusted environment.

Unlike prior model partition approaches that place many privacy-related slices in TEE, which introduces high computational overhead inside TEE and frequent data transfer overhead across GPU and TEE in CPU, Phantom outsources computation-intensive layers to GPUs and targets to reduce data transfer frequencies between TEE and GPU. The core of Phantom's protection mechanism lies within the industry-standard TEE, where lightweight MUX operation with the authorized keys is able to recover the real computing path from obfuscated paths caused by the learned redundant parameters, achieving model privacy protection without significant performance penalties.

Phantom's obfuscation process involves three key steps:

1). Analyzing the sensitivity of each layer within the model and identifying the most sensitive ones. This step is essential to help reduce the data communication between GPU and TEE, which is important for reducing performance overhead.

2). Transforming the original model architecture by introducing additional obfuscation layers through a reinforcement learning framework with our proposed multi-objective reward function for maximizing defense performance and minimizing overhead.

3). Training the added obfuscation layers to degrade model performance by maximizing model classification loss and generating the obfuscation layer index as keys.

Importantly, the original model weights remain unchanged during the training phase. This preserves the core functionality and performance of the model for authorized users who possess the necessary keys to recover it. By training the obfuscated weights, Phantom creates a protective shield around the original model without altering its essential components. The obfuscated Phantom model maintains the original model's performance characteristics while significantly increasing its resilience against model stealing and fine-tuning attacks. For authorized users, the obfuscated layers can be bypassed in TEE through lightweight MUX operation, ensuring full model functionality. However, when adversaries query the model without keys, the final prediction incorporates the results of the obfuscated layers, leading to a false final model output and effectively concealing the true architecture and critical weights in an untrusted environment. This makes it extremely challenging for attackers to extract its proprietary information.

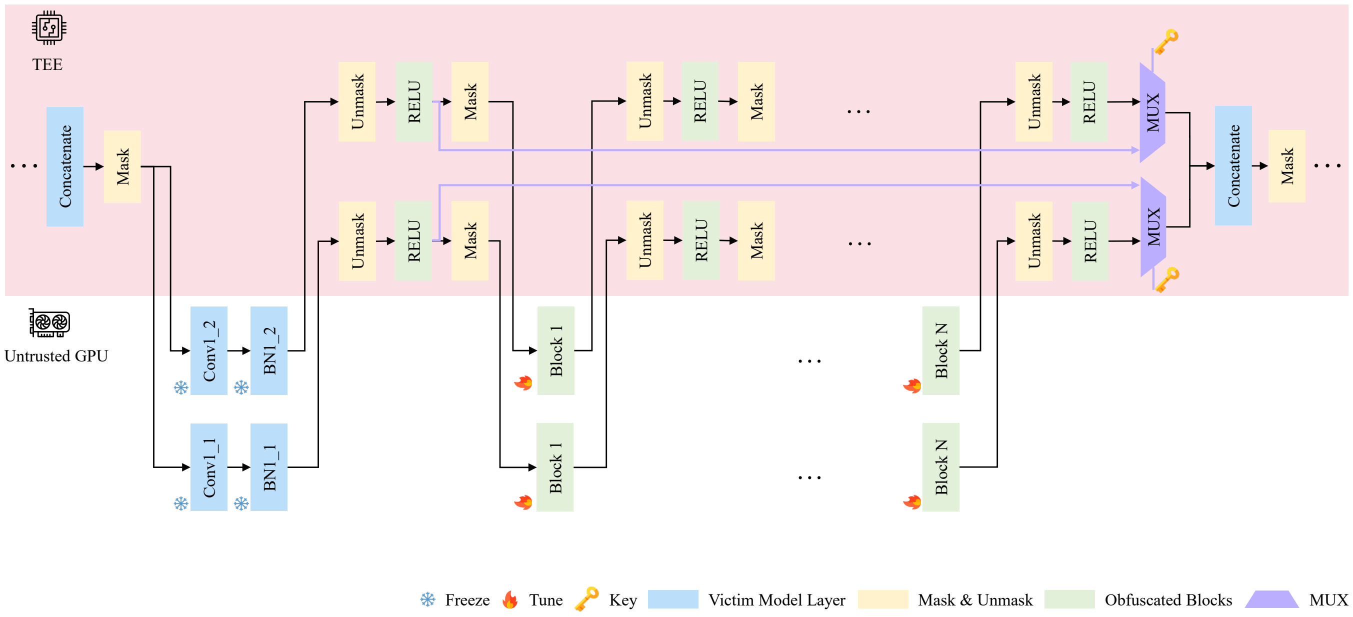


Figure 2: Overview of Phantom. The computation-intensive linear layers, including original convolution layers and redundant convolution blocks, are placed in the untrusted world - GPU. The non-linear layers, like ReLU, are placed in the secure world - TEE. The mask and unmask layers are used to protect the intermediate data transferred between TEE and GPU. The MUX is used to select different computation results depending on the user's keys.

4.2 Design Objective

The design objective of Phantom is to strike an optimal balance between enhancing model privacy and minimizing the computational overhead introduced by privacy-preserving mechanisms. Unlike approaches that fully encrypt models or execute entire or partial models within TEEs, which can incur substantial performance penalities [21, 44], Phantom obfuscates the original model architecture with small searched and trained lightweight layers. By carefully integrating these additional layers, we aim to achieve a level of privacy comparable to more heavyweight solutions while significantly reducing the associated computational and off-chip data communication costs.

Our design objective can be formalized as a min-max optimization problem, balancing the dual goals of maximizing privacy protection and minimizing introduced overhead. Let $\mathcal{M}$ be the original DNN model and $\tilde{\mathcal{M}}(\mathcal{M},\lambda)$ be the obfuscated model with small additional protective layers added according to configuration λ. We aim to find the optimal configuration λ^* that maximizes privacy while simultaneously minimizing the overhead. This can be expressed mathematically as:

$$\lambda^* = \arg\max_{\lambda\in\Lambda} [P(\lambda) - O(\lambda)] \qquad (1)$$

where $P(\lambda)$ quantifies the privacy protection level achieved by configuration λ, $O(\lambda)$ represents the overhead costs, and Λ is the set of all valid configurations. Optimizing this objective function enables simultaneously maximizing privacy and minimizing overhead within a single optimization objective. This formulation encapsulates the inherent trade-off between privacy and performance, seeking a configuration that provides the best possible privacy guarantees while keeping the introduced overhead to a minimum.

4.3 Obfuscated Architecture Search

Our obfuscated architecture search algorithm aims to find the optimal configuration λ^* of lightweight obfuscation layers for a given victim model $\mathcal{M}$. The algorithm seeks to determine the optimal additional obfuscation layers' structure required for effective obfuscation. As shown in Algorithm 1, each obfuscation layer is searched from a defined candidate layer pool, as shown in the Figure 3, which includes no operation (Null) and convolution layers with kernel sizes 1, 3, 5, and 7, a total of 5 candidates for each obfuscation layer. Then, the total search space is defined by the potential obfuscation layer insertion positions (whole layers for now), depth of search (defined as number of obfuscation layers to be added for each insertion position), and candidate size of each layer. Lets define N_v represents the number of layers in the victim model, N_o represents the number of layers in the final obfuscated model, D represents the depth of search. After performing our obfuscated architecture search algorithm, the total number of layers in the obfuscated model, N_o, would be:

$$N_v \leq N_o \leq N_v + D \cdot N_v \qquad (2)$$

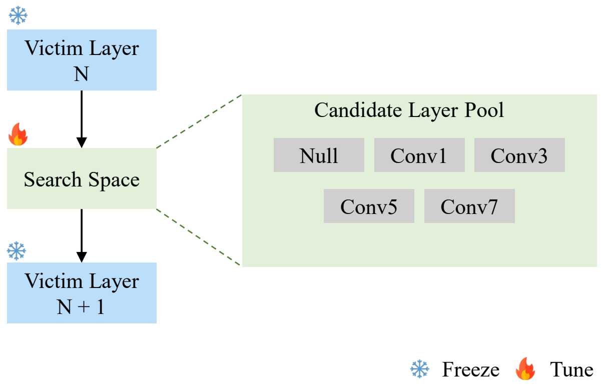


Figure 3: Search space of block. For each block, there are five operation candidates: no operation (Null), convolution layer with kernel size 1 (Conv1), convolution layer with kernel size 3 (Conv3), convolution layer with kernel size 5 (Conv5), and convolution layer with kernel size 7 (Conv7).

We define our obfuscation network as an over-parameterized structure that includes all possible operations for each obfuscation path. The policy gradient-based exploration strategy through the REINFORCE algorithm is used to learn a policy that optimizes operations for each path (Algorithm 1). The policy initially prioritizes exploration when uncertainty is high and gradually transitions toward exploitation as it converges, guided by our defined reward function. The learned policy network produces a probability distribution over the available architectural choices, as shown in Figure 3. Actions are sampled from this distribution, creating a natural balance between exploration and exploitation.

The algorithm evaluates sampled architectures using our designed multi-objective reward function that balances task performance, obfuscation effectiveness, and computational overhead, as below:

$$R = -(-\alpha \cdot \Delta S + \beta \cdot \Delta L) \tag{3}$$

where ΔS denotes the change in a predefined performance score, ΔL represents the change in DNN computing system latency, and α and β are weighting factors for ΔS and ΔL, respectively. For the performance evaluation, instead of using accuracy, which would require training each candidate architecture and thus significantly increase the searching complexity, we utilize the performance evaluation score proposed by Mellor et al. [45]. This score is based on the concept of activation patterns in untrained networks and can be computed quickly without compute-intensive weight training, thereby accelerating the architecture search process while still providing a solid measure of model performance.

Specifically, in our work, the score is calculated as follows:

- For a given untrained network and a minibatch of input data, we compute the binary activation patterns at each layer. These patterns indicate which neurons are active (output > 0) or inactive (output ≤ 0) for each input.
- We then construct a kernel matrix K_H by computing the Hamming distances between these binary activation patterns for all pairs of inputs in the minibatch.
- The final score is computed as $s = \log \|K_H\|$, where $\|\cdot\|$ denotes the matrix norm.

This score has been shown to correlate well with the network's final trained accuracy, making it a suitable proxy for performance during architecture search. By using this score instead of accuracy, we can evaluate thousands of potential architectures with several orders of magnitude smaller searching time.

During the training phase of the obfuscated layers, we freeze the weights of the original model parameters while exclusively optimizing the newly added obfuscation layers. These obfuscation layers are trained to maximize the cross-entropy loss, effectively degrading the final obfuscated model's accuracy. Specifically, our loss function takes the form:

$$\max_{\theta} \mathcal{L}(\theta) = -\sum_{i=1}^{N} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i) \tag{4}$$

Where θ represents the model parameters, $\mathcal{L}(\theta)$ is the cross-entropy loss function, N is the number of samples, y_i is the true label for the i-th sample, and $\hat{y}_i$ is the predicted probability for the i-th sample. We intentionally degrade the model's performance when exposed to adversaries by training small obfuscation layers to maximize the loss function. Concurrently, we freeze the weights of the original model layers, thereby maintaining the core functionalities and performance for authorized users.

The training process alternates between architecture search, updating policy using REINFORCE, and model update, training the weights of newly added obfuscation layers. This process continues until convergence or a maximum number of iterations is reached. Once the search converges, we construct the final obfuscated model by keeping the frozen victim model layers unchanged and adding the selected obfuscation layers into the search space following these frozen layers. Algorithm 1 shows the full search pipeline.

The final obfuscated model preserves the original model's functionality and performance for authorized users, who can bypass the obfuscation paths through a secure MUX mechanism within the TEE. Simultaneously, it presents a deceptive structure to potential adversaries, substantially enhancing the model's resilience against unauthorized access and model extraction attempts.

Algorithm 1 Obfuscated Model Search

```
Require: Search space parameters $\mathcal{S}$, Original network $\mathcal{N}$, Training batches $B_t$, Search batches $B_s$
Ensure: Optimized controller $\mathcal{C}$, Trained optional knobs $\mathcal{K}$, Obfuscated model $\mathcal{M}_{\text{obf}}$
 1: BEGIN
 2: Define the number of layers to be obfuscated $L_o$
 3: Define the depth of search space $D_s$
 4: Define the candidate operations $\mathcal{K}$
 5: for each batch $b_t$ in $B_t$ do
 6:     Freeze the original network weights $\theta_{\mathcal{N}}$
 7:     for each layer $l$ in original network $\mathcal{N}$ do
 8:         Sample candidate knobs $\mathcal{K}_l$
 9:         Remove the other knobs
10:     end for
11:     for each batch $b_t$ in $B_t$ do
12:         Maximize the loss function $\mathcal{L}(\theta_{\mathcal{K}})$
13:         Update weight parameters $\theta_{\mathcal{K}}$
14:     end for
15:     for each batch $b_s$ in $B_s$ do
16:         Evaluate the obfuscation performance $\mathcal{P}_o$
17:         Evaluate the inference latency $\mathcal{L}_i$
18:         Calculate the reward $\mathcal{R}$
19:     end for
20:     Calculate mean reward function $\overline{\mathcal{R}}$
21:     Maximize the Reward $\overline{\mathcal{R}}$
22:     Update the controller $\mathcal{C}$
23: end for
24: Generate the obfuscated model $\mathcal{M}_{\text{obf}}$ based on the controller $\mathcal{C}$'s searched result, choosing the optimal knobs $\mathcal{K}_l$ for each layer.
25: Return $\mathcal{C}$ and $\mathcal{M}_{\text{obf}}$
26: END
```

4.4 Layer-wise Obfuscation Sensitivity Analysis

With the advent of SGX2 TEE systems, which are increasingly supported by modern hardware, the memory and computation limitations of SGX1 have been largely mitigated. However, our subsequent system evaluation reveals a new challenge: in current widely adopted SGX2 systems, off-chip data transfer has emerged as the primary performance bottleneck in heterogeneous GPU and TEE system. This bottleneck becomes particularly significant when considering comprehensive obfuscation techniques. Even though conducting an architecture search across all layers of a DNN model can yield effective obfuscation and defensive performance, it introduces substantial off-chip data transfer overhead between the secure TEE environment and the untrusted GPU environment due to the largely increased number of layers, which have also be revealed in prior works [22, 37]. Therefore, in this work, we propose *layer-wise obfuscation sensitivity analysis* before the

Algorithm 2 Layer-wise obfuscation sensitivity analysis

```
Require: Victim model $\mathcal{M}$, Kernel sizes $K$, Number of models' layers $N$, Training dataset $\mathcal{D}_t$, Test dataset $\mathcal{D}_t$
Ensure: Rank List $\mathcal{R}$
 1: BEGIN
 2: Add convolution layer with kernel size $K$ after each layer in the original model $\mathcal{M}$ to generate N models
 3: for each model $\mathcal{M}_i$ in $N$ do
 4:     for each epoch $e$ in $E$ do
 5:         for each batch $b$ in $Dt$ do
 6:             Maximize loss function $\mathcal{L}(\theta)$
 7:             Update weight parameters $\theta$
 8:         end for
 9:         Measure accuracy $A_i$ on the test dataset
10:     end for
11:     Store model $\mathcal{M}_i$ and corresponding test accuracy $A_i$
12: end for
13: for each model $\mathcal{M}_i$ in $N$ do
14:     Fine-tune $\mathcal{M}_i$ on 10% $\mathcal{D}_t$ for $E_f$ epochs
15:     Measure fine-tuning attack accuracy $A_{f,i}$ of $\mathcal{M}_i$ on $\mathcal{D}_t$
16: end for
17: Rank the models $\mathcal{M}_i$ based on $1/A_{f,i}$, where $A_{f,i}$ is their fine-tuning attack accuracy
18: Return $\mathcal{R}$
19: END
```

above discussed obfuscated architecture search to reduce the number of obfuscated layers by constraining the searching only applied to the sensitive layers and excluding the insensitive layers.

Layer-wise obfuscation sensitivity analysis is used to identify which layer in a victim model, after being augmented with lightweight obfuscation layers, is more effective against the strong fine-tuning attack. To perform a layer-wise obfuscation sensitivity analysis on the original victim model $\mathcal{M}$ with N layers, we first generate N models where only one lightweight convolution layer with the same kernel size is added after each individual layer. For example, the i^{th} model has only one convolution layer added after the i^{th} layer. Then, all these N generated models are trained, with frozen backbone mode, through only tuning the newly added obfuscation layer to maximize the cross-entropy loss. Then, we conducted fine-tuning attack on these N obfuscated models, achieving N attack accuracies, which could indicate the obfuscation effectiveness of each layer. It is easy to see that the lower of these attack accuracy, the less success of fine-tuning attack, and thus more effective of the added obfuscation layer in a particular postion of network. Therefore, we use the $1/accuracy$ as the sensitivity score of each layer and only pick the top ranked k (top-k) layers, rather than the entire layers, to conduct the searching algorithm 1 discussed above to significantly reduce the number of added onfuscation layers. Here, the value of K is a hyperparameter of Phantom framework. By using our

Top-K obfuscation sensitivity methods, the total number of layers in the obfusacted model, N_o in equation 2, would become: $N_v \leq N_o \leq N_v + D \cdot K$. In our experiment section, we will discuss the perforamnce and overhead for whole layer searching and Top-K (K=3 as an example) layer searching.

4.5 Intermediate Feature Protection

To ensure the confidentiality of data exchanged between the TEE and GPU, following the prior work [21], we employ a custom one-time pad (OTP) encryption method. This approach safeguards intermediate values during computation without compromising efficiency.

The process begins by establishing a shared finite field. We select a prime number p, typically slightly larger than 2^{24}, to serve as the modulus for our operations. This choice allows us to accommodate 8-bit quantized values while remaining within the bounds of standard floating-point precision. For each data h that requires protection, our system follows these steps:

- **Quantization** The TEE converts h to an 8-bit fixed-point format, yielding $\hat{h}$. This step standardizes our input and minimizes precision loss.
- **Mask Generation** Using a secure random number generator, the TEE creates a unique mask r within the range $[0, p-1]$.
- **Encryption** The TEE computes the encrypted value h_e as follows:
$$h_e = (\hat{h} + r) \mod p \tag{5}$$
This operation effectively obscures the true value with the random mask.
- **GPU Computation** The GPU receives h_e and performs the required linear operation, denoted as $g(\cdot)$. The result, $y = g(h_e)$, is then returned to the TEE.
- **Decryption** To recover the original result, the TEE calculates:
$$g(\hat{h}) = (y - g(r)) \mod p \tag{6}$$
This decryption works due to the homomorphic properties of modular addition over linear functions. To optimize performance, $g(r)$ values are pre-computed and securely stored in TEE, reducing real-time computational overhead.

OTP allows the GPU to operate on encrypted data as if it were unencrypted, while maintaining data confidentiality. The security of this system relies on the single-use nature of each mask and the secure management of these masks within the TEE. By implementing this tailored OTP scheme, it achieves a balance between security and efficiency, enabling secure outsourcing of computations to the GPU while preserving the privacy of sensitive intermediate values.

4.6 Model Partition for TEE-GPU Deployment

Our model partitioning strategy leverages a heterogeneous TEE/GPU system to achieve an optimal balance between computational performance and model privacy. The overall workflow is shown in Figure 2. The TEE hosts non-linear layers, mask and unmask operations for intermediate value protection, and MUX for recovering model functionality using "keys". Concurrently, the GPU handles computationally intensive tasks such as linear layers, batch normalization, and pooling operations. This strategic distribution of tasks capitalizes on the secure processing capabilities of the TEE while harnessing the high-performance computing power of the GPU, resulting in a system that effectively combines privacy and efficiency.

Following this predetermined partitioning strategy, our deployment process allocates specific components of the obfuscated model architecture to either TEE or GPU execution environments. We initialize the TEE with non-linear layer parameters and obfuscation keys, while concurrently loading the GPU with obfuscated linear layer parameters. This deployment approach ensures efficient utilization of both the secure TEE environment and the GPU's computational capabilities.

The forward pass alternates between GPU and TEE environments, with the GPU performing linear operations and the TEE handling non-linear activations and de/re-obfuscation of intermediate results. This iterative process continues through each layer until reaching the final layer, where the TEE generates the secure prediction. To ensure data integrity and confidentiality, all communication between TEE and GPU is encrypted using the OTP method.

5 Experiment Setup

5.1 Models and Datasets

To evaluate the efficacy of our approach and following similar experiment setup as prior works [22, 23], we utilize three popular DNN model architectures, i.e., AlexNet [1], ResNet-18 [46], VGG-16 [47], and three datasets, i.e., CIFAR-10 [48], CIFAR-100 [48], STL-10 [49] for our experiments.

5.2 Victim Model Training

To create realistic victim models for compare with SoTA defense methods, we used ImageNet pre-trained models as our starting point. We then fine-tuned these models for each combination of architecture and dataset using the following process, mimicking common practices in real-world scenarios: 1) **Final Layer Adaptation:** We replaced the final fully connected layer of each pre-trained model to match the number of classes in our target datasets (10 for CIFAR-10 and STL-10, 100 for CIFAR-100). 2) **Full Model Fine-tuning:** We trained these adapted models on our chosen three datasets, allowing all layers to be updated. This approach enables the models to adjust their learned features to each dataset's specific characteristics. 3) **Data Augmentation:** We applied standard data augmentation techniques, including random cropping and horizontal flipping, to enhance the models' generalization capabilities and mitigate overfitting. 4) **Optimization:** We trained the models using stochastic gradient descent (SGD) with a learning rate of 0.01 and momentum of 0.9. A CosineAnnealingLR learning rate scheduler was employed to gradually adjust the learning rate during training, facilitating more effective fine-tuning. 5) **Performance Monitoring:** We tracked validation accuracy throughout training and implemented early stopping to prevent overfitting, selecting the model checkpoint with the highest validation performance.

This experimental setup, utilizing three model architectures and three datasets, allows us to evaluate our protection approach across diverse scenarios. By fine-tuning ImageNet pre-trained models on our chosen datasets, we closely simulate real-world practices where such models are often the targets of attacks. This approach enables us to assess the robustness and effectiveness of our defense method under various model architectures, providing insights into its performance in protecting fine-tuned models.

5.3 Obfusacted Model Search Implementation

In our model architecture searching algorithm implementation, we define the architecture search space with two layer branches B and a search depth D of three after each layer. Hence, following each frozen victim model layer, there would be six blocks could be added with lightweight obfuscation layers. This configuration provides a balance between the exploration of potential obfuscation strategies and computational feasibility. For REINFORCE learning algorithm training, we employ a Stochastic Gradient Descent (SGD) optimizer with an initial learning rate of 0.01 and a momentum of 0.9. We use a cosine annealing learning rate scheduler and train for 120 epochs, with reinforcement learning updates occurring every 300 steps.

Our experiments are conducted on an NVIDIA A6000 GPU, with training durations ranging from 18 to 38 hours for each model and dataset combination. This variation in training time is primarily due to differences in model architecture complexity and dataset size. Phantom is designed as a one-time effort per model-dataset pair for privacy protection. For example, with our hardware setup, it takes around 26 hours for Phantom to search the optimal obfuscated model for ResNet-18 on CIFAR-10. This obfuscation process combines two essential phases: i) the search phase for identifying optimal positions for lightweight obfuscation layers; ii) the fine-tuning phase for maintaining model functionality while ensuring privacy. Once the obfuscation process is complete, it could be deployed to TEE-GPU heterogeneous system for privacy-preserving inference without further update. This one-time investment is reasonable considering the long-term privacy benefits and protection against model theft, making it a general and practical solution.

5.4 Heterogeneous TEE/GPU System Implementation

We implement Phantom's model obfuscation features on the popular machine learning library PyTorch. There are two key components providing system-level security of Phantom: i) protection of model parameters through TEE-aware *selective offloading* of computation, and ii) protection of model architecture through side channel resilient *model obfuscation*.
Protecting Model Parameters. We utilize homomorphic computation property of linear layers to offload these layers to GPU, while keeping the non-linear layers in TEE only. During execution, the encrypted model is first loaded into the enclave memory and decrypted. On runtime, when a specific layer is offloaded to GPU, we first generate a mask (Section 4.5) and apply it over the layer parameters before sending them to GPU for computation. This ensures the unencrypted copy of model parameters only exists in the TEE, model parameters sent to GPU is always in masked form. Once the masked computation is completed, it is sent back to the TEE for unmasking.
Protecting Model Architecture. In addition to model parameters, Phantom also provides robust model architecture protection. In particular, the model architecture protection scheme in Phantom has three components: i) *key-based model obfuscation* that obfuscates in-memory model architecture with an authorized key (Section 4.1) to derive the real computation path. Phantom only stores the obfuscated model in memory, and utilizes the authorized key to only keep the computation corresponding to real path during runtime; ii) *constant-path computation* over obfuscated model regardless of the authorized key content to prevent leakage of model architecture over data-flow dependent side channel (i.e., memory bus snooping [11], cache template attacks [50–52] or advanced TEE-metadata based attacks [35, 53]). During runtime, Phantom executes all available paths in the obfuscated model regardless of if the specific path is part of real computation dictated by the authorized key. This ensures that the performed computations and memory accesses during inference is not impacted by the authorized key; and iii) *branch-agnostic recovery* of real computation path from the obfuscated model. Phantom utilizes branchless bitwise operations to select specific paths of execution corresponding to specific authorized key value (e.g., $result = (a \cdot \sim key) \,|\, (b \cdot key)$; this operation selects path corresponding to a when $key = 0$ and path corresponding to b when $key = 1$, without introducing any control-flow dependent side channel [54, 55]). This principled approach ensures no control-flow-dependent side channel leakage of model architecture information during runtime.

Overall, Phantom provides complete protection of model

Table 1: Comparison of model performance with and without authorized user keys. In each cell, the left accuracy is with a key, and the right accuracy is without a key.

	AlexNet			ResNet-18			VGG-16		
	CIFAR-10	CIFAR-100	STL-10	CIFAR-10	CIFAR-100	STL-10	CIFAR-10	CIFAR-100	STL-10
No Privacy Left Outside[1]	86.37% / 83.71%	61.96% / 56.46%	80.17% / 76.54%	93.65% / 95.47%	76.79% / 79.94%	86.22% / 87.51%	93.06% / 91.62%	73.11% / 73.03%	89.67% / 89.42%
NNSplitter	84.13% / 59.30%	56.09% / 22.94%	82.37% / 22.99%	93.01% / 9.99%	72.75% / 0.98%	94.79% / 22.28%	93.02% / 10.00%	72.76% / 1.00%	94.80% / 25.11%
Ours (Whole Layer)	**84.13% / 9.90%**	56.09% / 1.00%	**82.37% / 9.97%**	93.01% / 10.00%	**72.75% / 0.94%**	94.79% / 10.53%	93.02% / 10.00%	**72.76% / 0.94%**	94.80% / 10.53%
Ours (Top-3 Layer)	84.13% / 10.03%	**56.09% / 0.98%**	82.37% / 10.86%	**93.01% / 9.90%**	72.75% / 1.19%	**94.79% / 10.04%**	**93.02% / 9.90%**	72.76% / 1.19%	**94.80% / 10.04%**

[1] For 'No Privacy Left Outside' work, the performance with key means the performance of their method's pruned and trained model, while the performance without key is from the pre-trained backbone model.

architecture and parameters. Phantom's novel architecture search enables efficient model obfuscation. In addition, Phantom enables the first side-channel resistant architecture obfuscation by preventing both data- and control-flow-dependent side-channel leakages during runtime.

6 Experiment Results

6.1 Obfuscation Effectiveness

To protect a model's privacy, an effective defense method should significantly degrade the model's performance or even mislead adversaries who lack legal authorization. To evaluate the effectiveness of our obfuscation method, we compare it with two recent SoTA baseline methods [22, 23] across three popular neural network architectures (AlexNet, ResNet-18, and VGG-16) using three datasets (CIFAR-10, CIFAR-100, and STL-10).

Table 1 presents the results, showcasing model performance with and without authorized user keys for the obfuscated model. We evaluate our approach under two different settings: the whole layer and the Top-3 layer. The whole layer setting applies the obfuscation to all layers of the victim model, while the Top-3 setting selects only the three most sensitive layers based on our sensitivity analysis. Our proposed methods under both settings reduced the exposed model performance to nearly random guessing levels for adversaries. For 10-class classification tasks (CIFAR-10 and STL-10), accuracy dropped to approximately 10%, while for the 100-class task (CIFAR-100), it fell to around 1%.

The No Privacy Left Outside method [22] performs worse than our approach and NNSplitter [23] because it doesn't explicitly consider degrading model performance as an objective for privacy protection. Consequently, adversaries can still achieve model performance functionally close to that with authorized access. For adversaries aiming to use or misuse such a model without incurring costs, this defense method may not adequately protect the model's privacy. Our proposed method consistently outperformed both baseline approaches, NNSplitter and No Privacy Left Outside, regarding obfuscation accuracy (model performance without a key) across all model-dataset combinations. This demonstrates the effectiveness of our obfuscation technique in preserving model functionality for authorized users while providing strong privacy protection against unauthorized access.

6.2 Defend Against Fine-Tuning Attack

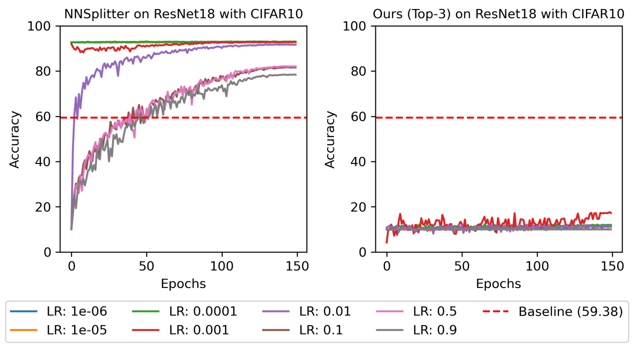


Figure 4: Comparison of defensive performance against fine-tuning attack between NNSplitter and Ours Top-3.

Fine-tuning attacks assume a strong adversary could obtain the partial obfuscated model [12, 23], including both its architecture and partial weights, along with a limited portion of private training datasets, typically no more than 10% of the total. This assumption is feasible in practice, as demonstrated by several recent studies. DeepSniffer [11] exploits architectural cues to reconstruct the complete architecture of deep neural networks without prior knowledge of the target model. DeepSteal [12] demonstrates the use of memory side channel attacks to steal the weights of deep neural networks. Additionally, Carlini et al. [13] introduce a training data extraction attack that can extract verbatim text sequences from language models. Their work demonstrates that these models may unintentionally memorize and subsequently reveal portions of their training data. The 10% training dataset assumption is considered robust, as training a model from scratch with randomly initialized weights using only 10% of the dataset often yields functional performance. This is demonstrated in the Baseline row of Table 2, where some models trained on this limited dataset still achieve notable accuracy. If an adversary obtains more than 10% of the training dataset, the necessity of stealing the protected victim model is substantially reduced. In this scenario, they could instead construct a comparable

Table 2: Comparison of defensive accuracy for fine-tuning attack. green indicates success and red indicates failure.

	AlexNet			ResNet-18			VGG-16			Average
	CIFAR-10	CIFAR-100	STL-10	CIFAR-10	CIFAR-100	STL-10	CIFAR-10	CIFAR-100	STL-10	
Baseline (Random)	77.26%	41.87%	52.01%	59.38%	37.33%	50.47%	31.26%	47.91%	64.35%	51.32%
No Privacy Left Outside (LR: 0.01)	🟥 83.98%	🟥 59.21%	🟥 81.13%	🟥 85.09%	🟥 59.27%	🟥 90.71%	🟥 90.79%	🟥 68.97%	🟥 93.97%	🟥 79.25%
No Privacy Left Outside (LR: 0.001)	🟥 80.35%	🟥 54.82%	🟥 79.87%	🟥 78.36%	🟥 50.02%	🟥 86.52%	🟥 85.91%	🟥 60.37%	🟥 93.39%	🟥 74.40%
NNSplitter (LR: 0.01)	🟩 9.99%	🟩 1.00%	🟩 10.00%	🟥 91.79%	🟥 70.39%	🟥 75.89%	🟥 93.19%	🟥 67.91%	🟥 77.99%	🟥 55.35%
NNSplitter (LR: 0.001)	🟥 84.31%	🟥 58.09%	🟥 79.39%	🟥 93.00%	🟥 71.98%	🟥 75.77%	🟥 93.81%	🟥 72.23%	🟥 78.18%	🟥 80.26%
Ours (Whole Layer) (LR: 0.01)	🟩 10.00%	🟩 1.00%	🟩 10.03%	🟩 12.13%	🟩 32.83%	🟩 10.03%	🟩 10.01%	🟩 12.40%	🟩 10.03%	🟩 12.05%
Ours (Whole Layer) (LR: 0.001)	🟩 10.00%	🟩 1.00%	🟩 10.03%	🟩 17.64%	🟩 11.23%	🟩 10.03%	🟩 11.27%	🟩 6.42%	🟩 10.03%	🟩 9.74%
Ours (Top-3) (LR: 0.01)	🟩 10.00%	🟩 1.00%	🟩 10.03%	🟩 10.00%	🟩 1.43%	🟩 10.03%	🟩 10.00%	🟩 13.92%	🟩 10.03%	🟩 8.49%
Ours (Top-3) (LR: 0.001)	🟩 10.00%	🟩 1.00%	🟩 11.18%	🟩 10.00%	🟩 1.00%	🟩 17.22%	🟩 10.00%	🟩 12.67%	🟩 10.03%	🟩 9.23%

model using publicly available standard architectures.

In our experiment, we model the adversary's capabilities based on the realistic attack scenarios described above. We evaluate five settings in our comprehensive fine-tuning attacks assessment: baseline, No Privacy Left Outside [22], NNSplitter [23], ours (Whole Layer), and ours (Top-3). The baseline involves training a model with randomly initialized weights. For No Privacy Left Outside, we use the pre-trained backbone model part of their hybrid model design as a starting point, then train it with 10% of the training dataset. For our approach, Whole Layer means the searching space is whole layer as described in the Algorithm 1. While the Top-3 means the obfuscation layer searching is only applied to the top-3 ranked sensitive layers to reduce the searching space and obfuscation model overhead. For all methods, the training data is the same random 10% of the entire training dataset.

To ensure a fair comparison, we maintain consistent fine-tuning attack settings across all five settings, utilizing 150 training epochs, an SGD optimizer, and a CosineAnnealingLR learning rate scheduler. This scheduler dynamically adjusts the learning rate of the SGD optimizer throughout the fine-tuning process. We consider an obfuscation method successful when the performance of its obfuscated model after being fine-tuned by an adversary is consistently lower than the baseline setting with training from random weights. Otherwise, the fine-tuning attack is successful.

Figure 4 compares the performance of NNSplitter [23] and our (Top-3) obfuscation method on ResNet-18 and CIFAR-10 in defending against fine-tuning attacks. We evaluate the fine-tuning performance across a range of learning rates (0.9, 0.5, 0.1, 0.01, 0.001, 0.0001, 0.00001, 0.000001), covering most commonly used rates for training DNN models. The red dotted line in both left and right figures represents the baseline accuracy. The left plot in Figure 4 shows that NNSplitter exhibits high sensitivity to different learning rates during fine-tuning, with all fine-tuning accuracies higher than the baseline for ResNet-18, indicating unsuccessful defense. In contrast, the right plot in Figure 4 demonstrates our method's defensive performance against fine-tuning attacks under various learning rates. Our approach shows consistent success across all tested learning rates, with accuracy levels remaining between 10% and 20%, which is much lower than the training from random baseline (59.48% accuracy). This uniformity is crucial, as it indicates that an attacker would gain no additional advantage by varying the learning rate. The stability and success of our method's performance suggest that it effectively protects the model's knowledge, making it significantly more challenging for an attacker to infer information even if the obfuscated model is completely leaked in an untrusted environment.

We also conduct a comprehensive comparison of fine-tuning attack accuracy across different methods: No Privacy Left Outside, NNSplitter, ours (Whole Layer), and ours (Top-3). As evidenced by Figure 4, the effectiveness of defensive methods against fine-tuning attacks can be sensitive to learning rate variations, with performance potentially fluctuating across different rates. Considering such, we provide two learning rate settings, 0.01 and 0.001, for each defensive method for fair comparison. In Table 2, we mark the successful defense in green and the unsuccessful defense in red. Our whole layer obfuscation and top-3 obfuscation methods achieve average fine-tuning accuracies ranging from 9.74% to 12.05% and from 8.49% to 9.23%, respectively, both significantly lower than the baseline average of 51.32%. In contrast, No Privacy Left Outside and NNSplitter range from 74.40% to 79.25% and from 55.35% to 80.26%, respectively, both higher than the baseline, indicating the failure of defending against fine-tuning attacks. NNSplitter is successful only for AlexNet with learning rate of 0.01, all other settings failed.

6.3 Defend Against Model Stealing Attack

Model stealing is an attack method where an adversary aims to create a "knockoff" or surrogate model that replicates the functionality of a victim model, using only black-box access to query the victim model. The attack process typically involves

Table 3: Comparison of defensive accuracy for model stealing attack. We **bold** the lowest defensive accuracy. The lower the defensive accuracy for model stealing attacks, the better the performance in defending against model stealing attacks.

	AlexNet			ResNet-18			VGG-16			Average
	CIFAR-10	CIFAR-100	STL-10	CIFAR-10	CIFAR-100	STL-10	CIFAR-10	CIFAR-100	STL-10	
No Privacy Left Outside[1]	19.04%	8.27%	24.15%	31.40%	10.90%	29.19%	30.87%	9.78%	32.92%	21.84%
NNSplitter	**10.00%**	**1.00%**	15.90%	12.50%	1.10%	11.00%	35.60%	14.30%	15.40%	12.89%
Ours (Whole Layer)	**10.00%**	**1.00%**	**10.00%**	**10.00%**	**1.00%**	**10.00%**	**10.00%**	**1.00%**	**10.00%**	7.00%
Ours (Top-3 Layer)	**10.00%**	**1.00%**	**10.00%**	**10.00%**	**1.00%**	**10.00%**	**10.00%**	**1.00%**	**10.00%**	**6.99%**

[1] The accuracy of the model stealing attack 'No Privacy Left Outside' is sourced from their paper.

initializing a surrogate model using a publicly available pre-trained model, sampling images from an out-of-distribution dataset to query the victim model and collect predictions, and then training the surrogate model on these collected image-prediction pairs to mimic the victim model's behavior. The effectiveness of model stealing is evaluated by measuring the surrogate model's accuracy on the victim model's private test dataset, indicating how well the surrogate model performs after being trained on the adversary's constructed dataset.

To assess our approach's resilience against model stealing attacks, we conducted experiments following the protocol established in prior works [7, 22]. We used CIFAR-100 as the out-of-distribution dataset for sampling query data, employing a random sampling strategy [7] to select 30,000 queries [22]. This approach simulates realistic model stealing attacks where attackers strategically choose queries to maximize information gain about the target model.

We evaluated model stealing accuracy across four approaches: No Privacy Left Outside [22], NNSplitter [23], ours (Whole layer) and ours (Top-3). The evaluation covered three different model architectures (AlexNet, ResNet-18, and VGG-16) and datasets (CIFAR-10, CIFAR-100, and STL-10).

As Table 3 shows, our method demonstrated superior protection across all datasets and model architectures. For AlexNet on CIFAR-10 and CIFAR-100, our approach and NNSplitter limited the attack success rate to 10.00% and 1.00% respectively, lower than No Privacy Left Outside at 19.04% and 8.27%. On STL-10, our method achieved the lowest attack success rate of 10.00%, compared to 24.15% for No Privacy Left Outside and 15.90% for NNSplitter. With the more complex ResNet-18 architecture, our approach outperformed existing methods. On CIFAR-10, we reduced the attack success to 10.00%, compared to 12.50% for NNSplitter and 31.40% for No Privacy Left Outside. Similar trends were observed for CIFAR-100 (1.00% vs. 1.10% and 10.90%) and STL-10 (10.00% vs. 11.00% and 29.19%). For VGG-16, our method's efficacy continued to shine. On CIFAR-10, we achieved an attack success rate of 10.00%, outperforming both NNSplitter (35.60%) and No Privacy Left Outside (30.87%). Similar trends were observed for CIFAR-100 and STL-10.

The results indicate that our obfuscation method effectively prevents the leakage of private knowledge from the victim model to the adversary, thereby hindering the construction of an accurate surrogate model. Our approach demonstrates remarkable resilience against model-stealing attacks across various model architectures and datasets.

6.4 System Overhead Evaluation

We evaluated the runtime overhead of the Phantom scheme by comparing it to a baseline model and a SoTA model parameter protection scheme, No Privacy Left Outside [22]. To evaluate different TEE backends, we conducted experiments on two platforms: *SGX2 platform*, represented by an Intel Xeon Gold 6342 processor with the latest Intel SGX (**SGX2**) implementation, and *SGX1 platform*, represented by an Intel Core i7 9700K processor with an older SGX (**SGX1**) implementation. To enhance computational efficiency, we leveraged GPUs for accelerating lightweight obfuscated convolution layers. Specifically, the SGX2 platform utilizes an NVIDIA A40 GPU, while the SGX1 platform utilizes an NVIDIA GTX 1080Ti GPU.

We evaluated the inference speed of Phantom on our real system protocol, focusing on total execution latency and its breakdown components: GPU latency (the time needed to complete the computation inside the GPU), TEE latency (the time required to finish the computation inside the TEE), and data transfer time (the time needed to transfer data between the GPU and TEE). The inference latency reported is for a single batch with a size of 100. To mitigate cold start effects for the GPU, we conducted a warmup step before measurements. All measurements were repeated 50 times to obtain average inference times.

For the system-level overhead analysis, we chose ResNet-18 as our baseline model. We evaluated a layer-branched version of ResNet-18, where all linear convolution layers were split into two parallel branches, aligning with the experiment setting defined in Section 5.3. The baseline ResNet-18 model serves as our reference point for inference time evaluation. We evaluated the performance of this baseline on three different device settings: GPU-Only, TEE-Only, and heterogeneous

Table 4: Inference Time Overhead Evaluation. We **bold** the lowest total inference time with its breakdown time.

		SGX2				**SGX1**			
		Total Execution Latency (*ms*)	GPU Latency (*ms*)	TEE Latency (*ms*)	Data Transfer Latency (*ms*)	Total Execution Latency (*ms*)	GPU Latency (*ms*)	TEE Latency (*ms*)	Data Transfer Latency (*ms*)
ResNet-18[1]	GPU-only	2.51	2.51 (100%)	-	-	4.65	4.65 (100%)	-	-
	TEE-only	34.27	-	34.27 (100%)	-	492.7	-	492.7 (100%)	-
No Privacy Left Outside	TEE-GPU	17.42	1.72 (10%)	5.96 (34%)	9.74 (56%)	341.95	4.40 (1%)	238.39 (70%)	99.17 (29%)
Ours (Whole Layers)	TEE-GPU	37.11	4.92 (13%)	12.02 (32%)	20.17 (54%)	543.05	2.57 (0%)	388.43 (72%)	152.06 (28%)
Ours (Top-3 Layers)	TEE-GPU	**11.33**	**1.62 (14%)**	**3.34 (29%)**	**6.41 (57%)**	**196.12**	**4.02 (2%)**	**145.04 (74%)**	**47.07 (24%)**

[1] The ResNet-18 evaluated here is a layer-branched version in which all convolutional layers are split into two parallel branches.

TEE-GPU. The GPU-Only setting, where the entire model is placed on the GPU, serves as the upper bound for inference speed. The TEE-Only setting, with the whole model in the TEE, offers the best privacy protection but serves as the lower bound for inference speed. The heterogeneous TEE-GPU setting, the focus of our work, leverages both GPU's computation power and TEE's protection ability.

We evaluated the obfuscation strategies of No Privacy Left Outside and ours under the heterogeneous TEE-GPU platform setting. In terms of total execution latency, our Top-3 obfuscation strategy demonstrated the best performance among the tested approaches. On the SGX2-based platform, our Top-3 obfuscation strategy required 11.33 ms compared to 17.42 ms for No Privacy Left Outside and 37.11 ms for our whole layer obfuscation strategy. Similarly, on the SGX1-based platform, the latencies were 196.12 ms, 341.95 ms, and 543.05 ms, respectively. This superior performance is due to lower computational requirements inside the TEE and reduced data transfer needs between GPU and TEE, as shown in the breakdown times in Table 4.

In particular, as shown in Table 4, data transfer latency constitutes the dominant portion of overall execution latency on the SGX2-based TEE-GPU platforms, accounting for around 50%-60% across all evaluated models. In contrast, the older SGX1-based platform experiences significant performance degradation primarily due to computational bottlenecks within the TEE, as highlighted by prior work [22] and further corroborated by our experiments on SGX1. We observe that more than 70% of the total execution latency in SGX1-based TEE-GPU platform is due to TEE-execution overheads (compared to <35% TEE related overheads in SGX2). Overall, the improved computational performance of SGX2, relative to its predecessor, has effectively shifted the primary bottleneck from TEE computation to data transfer latency for TEE-shielded DNN approaches.

SGX2 introduces two significant enhancements over SGX1 that have a substantial impact on ML workloads. First, SGX2 increases the maximum encrypted memory (EPC) capacity to 512GB, a significant upgrade from SGX1's 128MB limit. This eliminates the need for frequent and expensive memory swapping operations during runtime, which posed a critical bottleneck for large and memory-intensive ML workloads in SGX1. Second, SGX2 enables resource sharing among enclaves within the same process, facilitating efficient multithreading and parallel execution. These advancements collectively reduce SGX2's computational overhead to approximately 10%-15% of native execution, making earlier observations about SGX1's computational bottlenecks less applicable. Our findings reveal that, with SGX2, the primary performance-limiting factor shifts to data communication overhead in future heterogeneous systems.

We encourage the research community to consider this new bottleneck when designing future TEE-Shielded DNN algorithms, as data transfer time has become a critical factor in the overall performance of these systems.

7 Conclusion

This paper introduced Phantom, a novel framework for obfuscating deep neural networks in heterogeneous TEE-GPU systems. Phantom employs reinforcement learning to add lightweight obfuscation layers, effectively protecting model privacy while preserving authorized performance. Our layer-wise sensitivity analysis enables efficient, informed obfuscation. Extensive experiments demonstrate Phantom's superior defense against fine-tuning and model-stealing attacks compared to SoTA methods. Implementation on SGX2 systems revealed data transfer as the primary bottleneck, with our Top-3 obfuscation strategy significantly reducing latency. Phantom represents a significant step towards practical DNN privacy protection, balancing strong security with computational efficiency.

Ethics Considerations

In developing Phantom, we conducted a thorough ethical analysis to ensure responsible innovation while protecting legitimate security interests. Our analysis focused on key stakeholder impacts and technical design decisions that could affect different parties involved in the deployment and use of our system.

From a stakeholder perspective, we primarily considered model owners who face significant risks from model theft that could undermine their competitive advantage. Our research addresses their need for intellectual property protection while maintaining reasonable implementation costs through our Top-K sensitivity optimization. For legitimate users, our system ensures authorized users maintain high model accuracy with minimal performance overhead, where the Top-K layer sensitivity analysis specifically optimizes this balance between security and efficiency. Regarding the broader research community, while our work advances model protection techniques, we acknowledge potential misuse risks. Therefore, we focus on defensive techniques and transparently document their capabilities.

Our technical design choices were guided by ethical considerations at each step. We chose obfuscation over full encryption to balance security with practical deployment constraints, particularly for resource-constrained organizations. Our selective layer protection approach through sensitivity analysis achieves robust security while minimizing system overhead. Throughout development and evaluation, we used only public datasets (CIFAR-10, CIFAR-100, STL-10) and commit to open-sourcing our implementation to ensure transparency and reproducibility.

Open Science

In alignment with USENIX Security's open science policy, we commit to making our research artifacts publicly available. The artifacts include the complete source code for the Phantom framework implementation, along with evaluation scripts for model training and attack simulation.

Our repository contains essential datasets, implementations, configurations, hyperparameters, and performance measurements. We provide comprehensive documentation with detailed setup instructions and deployment guidelines. All artifacts are accessible through our public GitHub repository[2], ensuring complete transparency of our research.

Acknowledgments

This work is supported in part by the National Science Foundation under Grant No.2452657, No.2503906, No. 2019536 and No.2505209.

[2]The Phantom framework implementation and related artifacts are available at: https://github.com/ASU-ESIC-FAN-Lab/PHANTOM_USENIX

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012).

[2] A Vaswani. "Attention is all you need". In: *Advances in Neural Information Processing Systems* (2017).

[3] Riccardo Miotto et al. "Deep learning for healthcare: review, opportunities and challenges". In: *Briefings in bioinformatics* 19.6 (2018), pp. 1236–1246.

[4] OpenAI: Marcin Andrychowicz et al. "Learning dexterous in-hand manipulation". In: *The International Journal of Robotics Research* 39.1 (2020), pp. 3–20.

[5] Ruochen Jiao et al. "Learning representation for anomaly detection of vehicle trajectories". In: *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2023, pp. 9699–9706.

[6] Sanjay Kariyappa, Atul Prakash, and Moinuddin K Qureshi. "Maze: Data-free model stealing attack using zeroth-order gradient estimation". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 13814–13823.

[7] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. "Knockoff nets: Stealing functionality of black-box models". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 4954–4963.

[8] Florian Tramèr et al. "Stealing machine learning models via prediction {APIs}". In: *25th USENIX security symposium (USENIX Security 16)*. 2016, pp. 601–618.

[9] Varun Chandrasekaran et al. "Exploring connections between active learning and model extraction". In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 1309–1326.

[10] Jingtao Li et al. "Neurobfuscator: A full-stack obfuscation tool to mitigate neural architecture stealing". In: *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE. 2021, pp. 248–258.

[11] Xing Hu et al. "Deepsniffer: A dnn model extraction framework based on learning architectural hints". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 385–399.

[12] Adnan Siraj Rakin et al. "Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories". In: *2022 IEEE symposium on security and privacy (SP)*. IEEE. 2022, pp. 1157–1174.

[13] Nicholas Carlini et al. “Extracting training data from large language models”. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 2633–2650.

[14] Craig Gentry. “Fully homomorphic encryption using ideal lattices”. In: *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 2009, pp. 169–178.

[15] Ran Gilad-Bachrach et al. “Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy”. In: *International conference on machine learning*. PMLR. 2016, pp. 201–210.

[16] Yehuda Lindell. “Secure multiparty computation”. In: *Communications of the ACM* 64.1 (2020), pp. 86–96.

[17] Payman Mohassel and Yupeng Zhang. “Secureml: A system for scalable privacy-preserving machine learning”. In: *2017 IEEE symposium on security and privacy (SP)*. IEEE. 2017, pp. 19–38.

[18] Fan Mo et al. “Darknetz: towards model privacy at the edge using trusted execution environments”. In: *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*. 2020, pp. 161–174.

[19] Victor Costan and Srinivas Devadas. “Intel SGX Explained.” In: *IACR Cryptol. ePrint Arch.* (2016).

[20] Sandro Pinto and Nuno Santos. “Demystifying arm trustzone: A comprehensive survey”. In: *ACM CSUR* (2019).

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

[22] Z. Zhang et al. “No Privacy Left Outside: On the (In-)Security of TEE-Shielded DNN Partition for On-Device ML”. In: *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2024, p. 55. DOI: 10.1109/SP54263.2024.00052. URL: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00052.

[23] Tong Zhou et al. “NNSplitter: an active defense solution for DNN model via automated weight obfuscation”. In: *International Conference on Machine Learning*. PMLR. 2023, pp. 42614–42624.

[24] Matthew Jagielski et al. “High accuracy and high fidelity extraction of neural networks”. In: *29th USENIX security symposium (USENIX Security 20)*. 2020, pp. 1345–1362.

[25] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. “Deephammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips”. In: *USENIX Security Symposium*. 2020, pp. 1463–1480.

[26] Kunbei Cai et al. “DeepVenom: Persistent DNN Backdoors Exploiting Transient Weight Perturbations in Memories”. In: *IEEE Symposium on Security and Privacy*. IEEE. 2024.

[27] Kunbei Cai et al. “Seeds of SEED: NMT-Stroke: Diverting Neural Machine Translation through Hardware-based Faults”. In: *IEEE International Symposium on Secure and Private Execution Environment Design*. 2021.

[28] Binghui Wang and Neil Zhenqiang Gong. “Stealing hyperparameters in machine learning”. In: *2018 IEEE symposium on security and privacy (SP)*. IEEE. 2018, pp. 36–52.

[29] Kunbei Cai et al. “WBP: Training-Time Backdoor Attacks Through Hardware-Based Weight Bit Poisoning”. In: *European Conference on Computer Vision*. Springer. 2024, pp. 179–197.

[30] Nicholas Carlini, Matthew Jagielski, and Ilya Mironov. “Cryptanalytic extraction of neural network models”. In: *Annual international cryptology conference*. Springer. 2020, pp. 189–218.

[31] Congzheng Song and Vitaly Shmatikov. “Overlearning reveals sensitive attributes”. In: *arXiv preprint arXiv:1905.11742* (2019).

[32] Zecheng He, Tianwei Zhang, and Ruby B Lee. “Model inversion attacks against collaborative inference”. In: *Proceedings of the 35th Annual Computer Security Applications Conference*. 2019, pp. 148–162.

[33] Chenyu Yan et al. “Improving cost, performance, and security of memory encryption and authentication”. In: *ACM CAN* (2006).

[34] Md Hafizul Islam Chowdhuryy et al. “D-Shield: Enabling Processor-side Encryption and Integrity Verification for Secure NVMe Drives”. In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2023, pp. 908–921.

[35] Md Hafizul Islam Chowdhuryy, Hao Zheng, and Fan Yao. “MetaLeak: Uncovering Side Channels in Secure Memory Architectures Exploiting Metadata”. In: *IEEE ISCA*. 2024.

[36] David Kaplan, Jeremy Powell, and Tom Woller. “AMD memory encryption”. In: *AMD* (2016).

[37] Zhichuang Sun et al. “Shadownet: A secure and efficient on-device model inference system for convolutional neural networks”. In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2023, pp. 1596–1612.

[38] Intel Corporation. *Intel Software Guard Extensions (Intel SGX) - Key Management on the 3rd Generation Intel Xeon Scalable Processor*. Tech. rep. Technical Report. Intel Corporation, 2021. URL: https://www.intel.com/content/www/us/en/content-details/706019/intel-software-guard-extensions-intel-sgx-key-management-on-the-3rd-generation-intel-xeon-scalable-processor.html.

[39] Olga Ohrimenko et al. "Oblivious {Multi-Party} machine learning on trusted processors". In: *25th USENIX Security Symposium (USENIX Security 16)*. 2016, pp. 619–636.

[40] Fan Mo et al. "PPFL: Privacy-preserving federated learning with trusted execution environments". In: *Proceedings of the 19th annual international conference on mobile systems, applications, and services*. 2021, pp. 94–108.

[41] Md Shihabul Islam et al. "Confidential execution of deep learning inference at the untrusted edge with arm trustzone". In: *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy*. 2023, pp. 153–164.

[42] Taegyeong Lee et al. "Occlumency: Privacy-preserving remote deep-learning inference using SGX". In: *The 25th Annual International Conference on Mobile Computing and Networking*. 2019, pp. 1–17.

[43] Tianxiang Shen et al. "{SOTER}: Guarding Black-box Inference for General Neural Networks at the Edge". In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 2022, pp. 723–738.

[44] Tyler Hunt et al. "Chiron: Privacy-preserving machine learning as a service". In: *arXiv preprint arXiv:1803.05961* (2018).

[45] Joe Mellor et al. "Neural architecture search without training". In: *International conference on machine learning*. PMLR. 2021, pp. 7588–7598.

[46] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[47] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[48] Alex Krizhevsky. *Learning multiple layers of features from tiny images*. Tech. rep. University of Toronto, 2009.

[49] Adam Coates, Andrew Ng, and Honglak Lee. "An analysis of single-layer networks in unsupervised feature learning". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2011, pp. 215–223.

[50] Fan Yao, Guru Venkataramani, and Miloš Doroslovački. "Covert timing channels exploiting non-uniform memory access based architectures". In: *Great Lakes Symposium on VLSI*. 2017, pp. 155–160.

[51] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache template attacks: Automating attacks on inclusive {Last-Level} caches". In: *USENIX Security*. 2015.

[52] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. "Cache telepathy: Leveraging shared resource attacks to learn {DNN} architectures". In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 2003–2020.

[53] Md Hafizul Islam Chowdhuryy and Fan Yao. "IvLeague: Side Channel-Resistant Secure Architectures Using Isolated Domains of Dynamic Integrity Trees". In: *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2024, pp. 1153–1168.

[54] Md Hafizul Islam Chowdhuryy and Fan Yao. "Leaking Secrets through Modern Branch Predictor in the Speculative World". In: *IEEE TC* (2021).

[55] Md Hafizul Islam Chowdhuryy, Hang Liu, and Fan Yao. "BranchSpec: Information leakage attacks exploiting speculative branch instruction executions". In: *IEEE 38th International Conference on Computer Design*. 2020, pp. 529–536.

Appendix

Security Analysis Against Brute-Force Attacks

Suppose we have an original DNN model with $\boldsymbol{L}$ layers. An unknown number $\boldsymbol{N}$ of lightweight layers have been added to the original DNN model for obfuscation. Also, those lightweight layers can be inserted at any position in the original model except before the first layer. In addition, the added lightweight layers are indistinguishable from the original model layers. Our goal is to identify and skip those added lightweight layers to recover the original model.

The number of options to insert $\boldsymbol{N}$ of lightweight layers into the DNN model with $\boldsymbol{L}$ **-** $\boldsymbol{1}$ layers is a classic *stars and bars* problem in combinatorics, which is given by:

$$C(N,L-1) = \binom{N+(L-1)-1}{(L-1)-1} = \binom{N+L-2}{L-2} \quad (7)$$

Alternatively, since $\binom{a}{b} = \binom{a}{a-b}$, we can write:

$$C(N, L-1) = \binom{N+L-2}{N} \tag{8}$$

The total number of configurations $\boldsymbol{T}$ is:

$$T = \sum_{N=1}^{N} \binom{N+L-2}{N} \tag{9}$$

The binomial coefficient $\binom{N+L-2}{N}$ grows polynomially with respect to N for a fixed L. However, as N increases, the values become significantly large due to the combinatorial nature. **Therefore, a brute-force search through all possible configurations is computationally infeasible.**

To illustrate the computational infeasibility of a brute-force attack, let us consider a practical scenario with the following settings:

- Following the same setting of the paper, we consider the number of branches to be 2, the K of Top-K design to be 3, and the depth of each added block to be 3.
- $L = 36$ be the number of layers in the original DNN model (We consider the ResNet-18 model, which has 18 layers. Following the original branching setting with 2, the total number of layers is 18×2).
- $N = 18$ be the unknown number of lightweight layers added to the model. (Following the number of branches, Top-K, and depth of block setting, the number of lightweight layers is 2 * 3 * 3)

$$\begin{aligned} C(18, 35) &= \binom{18+35-1}{18} \\ &= \binom{52}{18} \\ &= \frac{52!}{18! \times (52-18)!} \\ &= \frac{52!}{18! \times 34!} \end{aligned}$$

Calculating the exact value of $\binom{52}{18}$ yields:

$$C(18, 35) = 42{,}671{,}977{,}361{,}650 \tag{10}$$

So, the total number of configurations is approximately:

$$C(18, 35) \approx 4.2 \times 10^{13} \tag{11}$$

With over 10^{13} possible configurations, a brute-force search would require checking each configuration individually, which is computationally infeasible given current technology. To obtain a reliable measurement that aligns with our experimental setup, we conducted 50 rounds of testing on an NVIDIA A6000 GPU with proper warm-up periods. The results showed that testing a single configuration takes an average of 129.14 ms. Therefore, to test all configurations, it will take

$$\begin{aligned} T &= \frac{42{,}671{,}977{,}361{,}650 \text{ configurations}}{7.74 \text{ configurations/second}} \\ &= 5.5131754 \times 10^{12} \text{seconds} \\ &\approx 174{,}688.7 \text{ years} \end{aligned}$$