HyperTheft: Thieving Model Weights from TEE-Shielded Neural Networks via Ciphertext Side Channels*

Yuanyuan Yuan
The Hong Kong University of Science and Technology
Hong Kong, China
yyuanaq@cse.ust.hk

Zhibo Liu
The Hong Kong University of Science and Technology
Hong Kong, China
zliudc@cse.ust.hk

Sen Deng
The Hong Kong University of Science and Technology
Hong Kong, China
sdengan@cse.ust.hk

Yanzuo Chen
The Hong Kong University of Science and Technology
Hong Kong, China
ychenjo@cse.ust.hk

Shuai Wang
The Hong Kong University of Science and Technology
Hong Kong, China
shuaiw@cse.ust.hk

Yinqian Zhang
Southern University of Science and Technology
Shenzhen, China
yinqianz@acm.org

Zhendong Su
ETH Zurich
Zurich, Switzerland
zhendong.su@inf.ethz.ch

ABSTRACT

Trusted execution environments (TEEs) are widely employed to protect deep neural networks (DNNs) from untrusted hosts (e.g., hypervisors). By shielding DNNs as fully black-box via encryption, TEEs mitigate model weight leakage and its follow-up white-box attacks. However, this paper uncovers that the confidentiality of TEE-shielded DNNs can be violated due to an emerging threat towards TEEs: ciphertext side channels of TEEs create weight-dependent observations during a DNN's execution. Despite the potential of inferring DNN weights from ciphertext side channels, existing techniques are inapplicable due to their over-strong requirements and the high precision required by DNN weights. A DNN can have millions of weight elements, and even a few incorrectly recovered weight elements may make the DNN non-functional.

We propose a novel viewpoint that focuses on the functionality of DNN weights, rather than each weight element's exact value. Accordingly, we design HyperTheft to directly generate weights that are functionality-equivalent to the victim DNN using ciphertext side channels. HyperTheft is established for highly practical settings; it exhibits the weakest requirement compared to prior methods. When only knowing a victim DNN's input type and task type (which are public and denote the minimal information required to use a DNN), HyperTheft can recover its weight using ciphertext side channels logged during the victim DNN's *one* execution. The whole procedure does *not* require attackers to 1) query the victim DNN, 2) have valid data that the DNN accepts, or 3) know the victim DNN's structure. Our evaluations generate more than 8K DNN weights which constantly achieve 77%∼97% test accuracy in different DNN runtimes, including various versions of PyTorch and DNN executables. Our recovered weights can subsequently enable training data leakage and severe bit-flip attacks.

*The extended version of a CCS 2024 paper [78].

1 INTRODUCTION

Deep neural networks (DNNs) have been exponentially deployed on various platforms (e.g., cloud servers, edge devices) given their high intelligence in solving real-life tasks. DNNs' intelligence is encoded in their weights which are trained over humongous datasets, with extensive human expertise and computing resources required. Nevertheless, as DNNs are white-box accessible on the host machine, a malicious host can directly copy their weights to steal DNN intelligence and launch white-box attacks [66, 80], posing a major security and privacy threat to modern DNNs [36].

To address the trust concern, Trusted Execution Environment (TEE), such as AMD SEV [35], Intel SGX [31, 33], etc., is proposed to protect DNNs [29, 41, 46]. With memory encryption, TEE provides isolated execution to shield a DNN as fully black-box on the host machine. Although recent studies launched successful side-channel attacks on TEE-shielded programs based on their secret-dependent information flows [24, 53, 69], DNN weights are believed secure under TEE protection: modern DNNs implement a constant-time computation paradigm, where the computations are achieved as a sequence of matrix operations with constant[1] data/control flows, eliminating mainstream micro-architecture side channels [27, 80].

Despite the promise, this paper uncovers severe DNN weight leakage due to the ciphertext side channel [42, 43] recently disclosed in AMD SEV.[2] Essentially, modern TEEs adopt *deterministic* encryption to support efficient random memory access and large memory encryption [17, 42], and the ciphertext of each memory write value only depends on the plaintext and the written address. Hence, if attackers observe that the ciphertexts of two consecutive

[1]Recent works have proposed the multi-exit DNN [47] whose execution may terminate early for some inputs. However, multi-exit DNN's control and data flows only depend on the final predictions and do not leak DNN weights.
[2]Practical attacks have been demonstrated on AMD SEV and reported to the vendor; yet, the attacks are feasible to any deterministic-encryption-based TEEs via hardware attacks such as memory bus snooping [40], cold boot attack [25], etc.

memory writes at the same address do not change (i.e., ciphertext collision), they can infer the equality relation between two plaintext written values [42, 43]. Intuitively, as matrix computations inside a DNN involve nested loops with DNN weights, which often repeatedly access the same memory address, ciphertext collisions should correlate with DNN weights to a large extent.

Yet, recovering DNN weights from ciphertext collisions is inherently challenging due to the following reasons.

① **A More Challenging Threat Model.** Unlike cryptographic software (i.e., the attack target of prior ciphertext side-channel attacks [42, 43]) whose implementation details are known to attackers (e.g., the RSA algorithm is public), the computation graphs of DNNs are often *private*. Thus, attackers only have a ciphertext side channel trace consisting of all ciphertext collisions logged from the victim DNN's whole execution. They *cannot* investigate how each weight element is involved in DNN computations and is consequently leaked via each ciphertext collision.

② **Partial Leakage of DNN Weights.** The ciphertext side-channel leakage in cryptographic software is lossless, as it is due to memory writes *directly* determined by each private key bit [42, 43]. However, memory writes of a DNN's execution rely on *intermediate computing results* derived from the weight, ciphertext collisions therefore do not leak all weight elements. Since a DNN often has millions of weight elements which are *highly correlated*, failing to recover a few weight elements can result in non-functional DNNs whose predictions are purely random [62, 75].

③ **Over-Strong Requirements of Query Attacks.** One may conduct query-based attacks, which use a student model to duplicate confidence scores of a DNN's predictions over query inputs [12, 32, 58, 68], to imitate the victim DNN's behaviors. However, TEE-shielded DNNs do not return confidence scores, greatly increasing the cost of query-based attacks [80]. While recent hardware attacks can be adopted to reduce the cost by recovering partial DNN weights, they require knowing the DNN's structure (which can be private) and are limited to quantized DNNs [61]. Importantly, the query inputs must cover all classes of the victim DNN's training data, which is impractical if the DNN is trained on private datasets like medical images. Noteworthy, training a student model to generate identical ciphertext collisions as the victim DNN is also infeasible, as the generation of ciphertext is non-differentiable.

Solution: A New Perspective of DNN Weights. This paper takes a holistic view on DNN weights by considering the victim DNN's *functionality* of solving its intended task. Instead of recovering exact weight elements (i.e., conventional ciphertext side-channel attacks) or duplicating a DNN's predictions for specific inputs (i.e., prior query-based attacks), we present a novel and highly effective technique to represent and extract functionalities from ciphertext side channels of *unknown* DNNs performing *unknown* tasks. Specifically, we design a hyper-network, HyperTheft, that takes ciphertext collisions logged from the victim DNN's execution (i.e., a ciphertext side-channel trace) as inputs and *directly outputs functional weights* for a surrogate model. With weights generated from HyperTheft, the surrogate model is able to solve the victim DNN's task. The surrogate model may have the same or different structure as the victim DNN, depending on the attacker's knowledge.

HyperTheft delivers highly stealthy and generic attacks. It considers both regression and classification, two fundamental tasks of all modern DNN applications such as image recognition, disease diagnosis, financial forecast, etc. To attack a DNN performing regression or binary classification, HyperTheft only requires ciphertext collisions from its one execution. For k-class classification ($k > 2$) — in case k is unknown — HyperTheft decouples it as k different binary classifications (i.e., belonging to the k-th class or not), and generates weights for k surrogate models (with each one for a binary classification) by observing the victim DNN's (minimal) k executions. Further, inspired by stochastic training algorithms of DNNs (e.g., SGD [2]), we introduce stochasticity into HyperTheft's weight generation, such that *multiple* functionality-equality weights can be generated using only one side-channel trace; the corresponding surrogate models (for the same task) can form a majority voting to further improve their accuracy.

Practicality: The Weakest Knowledge. Distinct from all prior weight stealing techniques, HyperTheft does not interact with the victim DNN; it only passively observes ciphertext side channels *without* querying the victim DNN. Thanks to our well-designed training algorithm for hyper-network (see Sec. 6), HyperTheft does *not* require valid data accepted by the victim DNN (compared with query-based attacks). Additionally, empowered by our functionality-centric view, HyperTheft does not rely on the victim DNN's structure and is applicable to general DNNs (compared with prior hardware attacks [61, 76]). In Sec. 8.2, we employ HyperTheft to generate more than 8K weights under this *weakest-knowledge* setup, and these weights constantly achieve 77%~97% test accuracy. To comprehensively assess the real-world threats, Sec. 8.3 evaluates how those stronger assumptions in prior works, e.g., knowing the DNN structure or querying the DNN (which may hold in certain scenarios), can further boost HyperTheft.

Findings: Broad Attack Surface. HyperTheft can successfully steal weights from popular DNNs (e.g., Transformer, ResNet, etc.) performing various classification and regression tasks over representative datasets (e.g., ImageNet, Chest X-ray, etc.). We consider different runtimes of DNNs: the most popular deep learning (DL) framework PyTorch and the recent DL compiler, Glow [64], that compiles DNN models into executables. We also systematically evaluate various versions of PyTorch and consider different usages of TEEs (i.e., shielding full DNNs or DNN slices). Our recovered weights constantly achieve the objectives of stealing DNN intelligence and enabling white-box attacks against the victim DNN. Although the recovered weights are never trained using the victim DNN's training data, they largely enhance membership inference attack [8, 66] to leak the training data. The recovered weights also bring bit-flip attack [62, 75], which can globally decimate DNN intelligence for nearly all (benign) inputs. In sum, this paper makes the following contributions:

- For the first time, we unveil the high risk of leaking DNN weights via ciphertext side channels of TEE-shielded DNNs, despite that weights in vanilla DNNs (unprotected by TEEs) are free of mainstream micro-architecture side channels. We demonstrate that such weight leakage subsequently enables stealing DNN intelligence and launching white-box DNN attacks.

- To overcome technical hurdles of recovering DNN weights, we propose to directly generate functionality-equivalent weights from ciphertext side channels. We design HyperTheft, which can recover DNN weights passively with only negligible and the weakest knowledge of the victim DNN. HyperTheft applies to general DNNs and is capable of recovering DNN weights by observing only a few executions of the victim DNN.
- We comprehensively evaluate diverse and representative DNNs, datasets, DNN runtimes, and TEE usages, where HyperTheft can constantly recover DNN weights from ciphertext side channels. We also systematically assess how public knowledge in various scenarios can boost HyperTheft's capability, and conduct membership inference and bit-flip attacks based on HyperTheft's recovered weights.

Artifact. The code and data of this paper are provided at https://github.com/Yuanyuan-Yuan/HyperTheft [1].

2 PRELIMINARIES AND BACKGROUND

2.1 DNNs and Terminologies

Since many terms (e.g., parameter vs. weight) of DNNs are not used consistently in existing literature, to avoid ambiguity, we first briefly introduce DNNs and give concrete definitions for terms related to this paper.

A DNN $F = \dots f_{i+1} \circ f_i \circ f_{i-1} \dots$ consists of multiple connected layers and each layer is a function $f(x) = \sigma(\theta x + b)$ where σ is the non-linear activation function. The computation graph is often constant in modern DNNs and does not change in different executions. Each DNN is designed to solve an intended task by assigning the prediction y to an input x. Depending on whether y is discrete or continuous, the task is categorized as classification or regression, respectively. A DNN's capability of solving its task is formed during the training stage, which updates $[\theta, b]$ using the training data. The trained DNN can run with various runtimes, depending on its deployed platform.

Definitions. We define the following terms for this paper.

• *DNN Weight*: θ and b are often dubbed as weight and bias of f. To ease the presentation, we refer to both θ and b as a single singular term "weight" in the rest of this paper. In particular, *the term "DNN weight" in this paper denotes* $[\theta, b]$ *of all layers in a DNN.* Since $[\theta, b]$ constitute a matrix, we refer to elements of matrix $[\theta, b]$ as "weight elements". We use the uppercase W to denote DNN weight, and the lowercase w_i to indicate weight of the i-th layer. Similarly, F denotes a DNN and f_i indicates its i-th layer.

• *Functionality & Behaviors*: A well-trained weight W can enable a DNN's intelligence, which is reflected in two aspects: 1) the overall functionality of solving the DNN's intended task (e.g., classifying digits); and 2) the behavior of predicting y^* for a specific input x^*.

• *DNN Structure & Parameters*: Following prior literature [22, 74], structure denotes the computation graph of a DNN, which includes 1) the number of layers, 2) how each layer is implemented, and 3) how different layers are connected. For instance, LeNet and ResNet are two different structures. Parameters indicate the hyperparameters in DNN structures, e.g., kernel sizes in convolutional layers.

• *Task Domain*: DNNs are designed to provide predictions from a fixed set.[3] For example, a DNN classifying cat vs. dog only outputs `cat` or `dog` even given a horse image. Therefore, to have meaningful predictions, DNNs also require valid inputs. The validity of inputs is determined by the DNN's task, for example, if a DNN classifies cat vs. dog, its valid inputs must be cat or dog images. Here, `cat` & `dog` images form the "task domain" of the DNN.[4]

• *Input Type*: To distinguish the subtle difference between public and private information of DNN's valid inputs, we further define "input type" over the task domain. Consider two medical diagnosis DNNs that accept chest X-ray images. Suppose the two DNN developers have training images of *non-overlapping* diseases, these two DNNs will support diagnosing different diseases, leading to different task domains. However, they share the same input type of "`chest X-ray`" image. Note that the `cat` & `dog` images mentioned above are from a different input type of "`natural`" image [4, 6]; see more cases in Sec. 8. In practice, having data from the victim DNN's task domain may be impractical, as these data are often private (e.g., medical images of certain diseases). However, obtaining data sharing the same input type with the victim DNN is often feasible (e.g., medical images of benign cases). Previous query-based [12, 32, 68] and hardware attacks [61, 76] require data covering the victim DNN's full task domain, while this paper loosens the requirement to only input type, delivering a more practical attack.

Hyper-Network. A hyper-network is a special DNN that generates weights for a target DNN [13, 26, 63]. The machine learning community has designed various hyper-networks to study the statistics of DNN weights for a specific task [63, 81]. Nevertheless, conventional hyper-networks require data from the same task domain of the target DNN, and only generate weights of identical functionality, impeding their application in stealing DNN weights. By carefully designing the training algorithm of hyper-networks (see Sec. 6.2), this paper presents a *task-wise* generalizable hyper-network, which can generate weights for the target DNN (i.e., the attacked TEE-shielded DNN) without using data from its task domain. By leveraging ciphertext side channels from the target DNN, our generated weights can exhibit different *unseen* functionalities.

2.2 TEE Protection and Mitigated Attacks

Model Stealing. Besides preventing attackers from copying DNN weights, TEE also mitigates query-based model stealing [12, 32, 58, 68] (a.k.a., knowledge distilling). To understand how the mitigation works, we first elaborate on motivations behind this attack. Typically, attackers first query the victim DNN using their own data and then train a student model to duplicate the victim DNN's prediction confidences (i.e., an input's probabilities of belonging to *all* possible predictions) on the queried data. Because DNN training is data-intensive and costly, query-based attacks aim to obtain a functionality-equivalent student model using fewer training data (i.e., the queried data) guided by the victim DNN's confidence scores. TEE-shielded DNNs mitigate such attacks by only returning the prediction *without* confidence scores [80]. Thus, despite that attackers can still query a TEE-shielded DNN, they only label their queried data and the attack's cost becomes comparable to training a new

[3]Text DNN's outputs are concatenated using words from a fixed vocabulary.
[4]The task domain is also referred to as "problem domain" in existing literature [56].

DNN from scratch [80]. Note that the query data must cover the TEE-shielded DNN's full task domain; if the query data are from a subset of the task domain, the student model only learns the victim DNN's partial functionality w.r.t. this subset [12, 32, 68, 80].

The black-box view of TEE-shielded DNNs also mitigates the following popular DNN attacks.

Data Privacy: Membership Inference. Membership inference attack (MIA) [8, 66] aims to infer if an input is included in the victim DNN's training data — a successful MIA indicates a severe training data leakage. Existing attacks primarily leverage DNN's prediction confidence, intermediate results, and/or gradients to infer membership of a given input. Since TEE-shielded DNNs are purely black-box[5] and only return final predictions, no available information can be leveraged to infer an input's membership.

DNN Integrity: Intelligence Depletion. Unlike adversarial attacks that only fool a DNN to mis-classify the crafted adversarial examples, bit-flip attack (BFA) can *globally* deplete DNN intelligence such that the victim DNN randomly guesses predictions for almost all (non-adversarial) inputs [44, 62, 75]. To launch BFA, attackers first require localizing weight elements that are critical to the DNN's intelligence and then leverage rowhammer attacks [37] to flip bits of these weight elements. The localization process is conducted by computing gradients over different weight elements, which is prohibited by the black-box view of TEE-shielded DNNs. As a result, attackers have to randomly flip bits in a DNN's weight, which is impractical due to the massive search space and the high cost of rowhammer attacks [62, 75].

2.3 TEE and Ciphertext Side Channel

TEEs leverage memory encryption to create isolated execution environments for secure DNN computations, where other users and software stacks (e.g., guest kernel, OVMF) cannot access the encrypted memory. The encryption engine encrypts/decrypts memory data on-the-fly and is implemented as a hardware module between the CPU chip and DRAM.

Deterministic Encryption. Two factors must be considered by TEE. First, efficient random memory access that requires independently encrypted memory blocks. Second, encrypting large memory where additional space and latency are needed for counters. To meet these requirements, modern TEEs including AMD SEV [35], ARM CCA [3], Intel TDX [31], and Intel SGX on Ice Lake SP [31, 33], adopt the *deterministic*-mode AES encryption. Specifically, given a memory block, to encrypt its memory value v, the encryption first takes a tweak function T to calculate a mask $m = T(a)$, where a is the address of the block. The encrypted ciphertext is generated as $c = P(v \oplus m) \oplus m$, where P is the encryption function. Therefore, when the same value v is stored at the same address a, the generated ciphertext is always identical (i.e., ciphertext collision).

Leakage Due to Ciphertext Collision. Existing works have leveraged ciphertext collision to infer the plaintext private keys of TEE-shielded cryptographic software [42, 43]. The attack workflow is illustrated in Fig. 1: ⓐ the attacker observes ciphertexts generated from two consecutive memory writes at the same address. If ciphertexts do not change, the two written values should be identical. In contrast, if ciphertexts change, different values are written. ⓑ Based on the cryptographic software's implementation, the attacker manually investigates which instruction induces the ciphertext collisions and consequently infers the plaintext private keys.

Essentially, a similar procedure can be applied to TEE-shielded DNNs. When a TEE-shielded DNN is executing, the attacker observes ciphertext collisions of its memory writes. Nevertheless, as the victim DNN's computation graph is often private, the attacker cannot map ciphertext collisions back to their corresponding DNN computations — only a ciphertext side-channel trace that records ciphertext collisions from all the victim DNN's memory writes is available. Our community still lacks techniques to extract DNN weights from ciphertext side-channel traces.

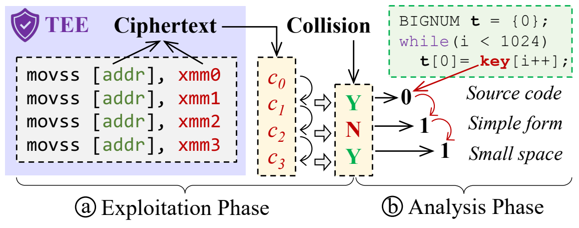


Figure 1: The workflow of ciphertext side-channel attacks.

3 APPLICATION SCOPE

As shown in Fig. 1, a ciphertext side-channel attack consists of two main phases: ⓐ an *exploitation* phase that collects side channels from the victim, and ⓑ an *analysis* phase that recovers secrets from the collected side channels. In previous attacks towards cryptographic software, the analysis phase is conducted by manually analyzing how different execution states of the victim affect ciphertext collisions [42, 43]. This is feasible given the public source code. Cryptographic keys also have a simple form and a small search space; each key only has 1∼2K *independent* binary bits and these bits *directly* determine the ciphertext collisions.

However, manually analyzing how DNN weights affect ciphertext collisions is hardly doable. In practice, the implementation of the victim DNN is often private [80], such that attackers cannot investigate how ciphertext collisions are induced by different computation operators. On the other hand, ciphertext collisions in TEE-shielded DNNs are *not* due to writing weight elements to memory since weights have been preloaded before the execution. Instead, the collisions are induced by a small portion of intermediate computation results of the DNN. As a result, ciphertext side channels only *leak partial and indirect information* of DNN weights, whereas cryptographic keys are fully leaked in prior works.

Moreover, modern DNNs have millions of weight elements, and each weight element is a floating-point number with an unconstrained range under the specified precision, leading to an almost infinite search space. Importantly, DNN weight elements are *highly correlated* — a few incorrectly recovered weight elements (since not all of them are leaked) can result in a non-functional DNN,

[5]Note that previous MIA works assume that "black-box" DNNs return prediction confidence [49], which is different from the black-box in our context.

e.g., 1∼5 incorrect ones out of ∼10M weight elements as shown in previous works [44, 62, 75], denoting a failed attack. This *high integrity requirement* of DNN weights is fundamentally different from cryptographic keys where each bit is independent: even partially recovered bits can be sufficient for practical attacks [54, 71].

Prior works have proposed side-channel analysis techniques for various DNN secrets, e.g., DNN structure [22], DNN input [79], etc. Nevertheless, since DNN weights are free of mainstream side channels, the corresponding analysis approach is rarely studied. Therefore, this paper proposes a DNN weight analysis technique to complete the puzzle of DNN secret research, and bridges it with ciphertext side channels to study the threats of TEE-shielded DNNs. Aligned to previous analysis works [22, 79], we do *not* present a new exploitation tool (ⓐ) because existing tools are relatively mature. Rather, we focus on the analysis phase (ⓑ) and design HyperTheft to automatically generate DNN weights from *already-prepared* ciphertext side channels. We aim to greatly ease the attack requirements and enhance the attack performance. HyperTheft is designed to support any exploitation tools if available (e.g., `CipherLeak` [43], `SEV-Step` [72]).

4 THREAT MODEL AND RELATED WORKS

This section elaborates on the threat model and required knowledge of our work, and compares our technique with prior methods. We omit existing works that hypothesize secret-dependent computations of DNNs (e.g., a DNN prunes its weight for different inputs [30]), assume most DNN weight elements are public [7, 73], or perform brute-force guesswork [5, 19].

TEE-Protection. We assume TEE and its provided protection are functioning properly and faithfully. Specifically, the encryption algorithm of TEE is secure and attackers cannot decrypt ciphertext. Only ciphertext side channels due to deterministic encryption (i.e., a design feature of TEE) are exploitable. Also, all software and hardware involved in TEE are secure; attackers cannot alter their data or control flows to leak secrets. The DNNs deployed inside TEE are conventional DNNs: they are designed and trained normally without any carefully crafted structure or adversarial injections to enable or amplify leakage. The TEE-shielded DNN is fully black-box: attackers cannot view its inputs, (intermediate) outputs, structure, and weight. When querying the TEE-shielded DNN, only the final prediction (without confidence scores) is returned to users.

Attacker. Consistent with the objective of shielding DNNs with TEEs, we assume an untrusted host machine (e.g., a malicious hypervisor, or the host OS) which has full system privileges. Thus, attackers can read the content (i.e., encrypted ciphertext) and address of a memory write via direct software access (as demonstrated on AMD SEV [42, 43]). Besides, attackers are also capable of conducting physical attacks on TEE-shielded DNNs. For instance, attackers can leverage memory bus snooping to read the ciphertext, as exploited on Intel SGX [40]. Having that said, we do not assume a specific ciphertext side-channel exploitation approach; we aim to provide an out-of-the-box solution to automatically generate DNN weights from already-prepared ciphertext side channels.

Attacker's Goals and Incentives. Leaking DNN weights brings the following two threats.

1. Stealing Intellectual Property (IP). The key IP of a DNN is the intelligence encoded in its weight, which produces considerable commercial values. Training DNN weights requires substantial manual efforts to build training data (which are often private) and human expertise to design the training algorithm.

2. Launching White-Box Attacks. As introduced in Sec. 2.2, the white-box access to DNN weights enables severe attack chances, compromising data privacy [8, 66] and breaking DNN integrity [44, 62, 75].

This paper recovers DNN weights from ciphertext side channels by generating functionality-equivalent weights. Despite being different from the victim DNN's weights, our recovered weights fulfill the two attack goals (as evaluated in Sec. 8).

Target DNNs. Unlike existing attacks leveraging characteristics of certain specific DNNs [76] (e.g., binarized DNNs whose weight elements are either 1 or -1), ciphertext side-channel attacks exploit the defects in TEE's design. Therefore, our technique applies to any general DNNs as long as they are "protected" by TEEs.

4.1 Attacker's Knowledge and Actions

Our technique is established for highly practical attack scenarios and assumes attackers having the weakest knowledge of the victim DNN, i.e., only minimal information that specifies the DNN's basic usage — without them, attackers do not even know how to use the stolen DNN and the stealing accordingly becomes meaningless. Specifically, we consider that attackers only know the input type (as defined in Sec. 2.1) and the task type (i.e., classification or regression, which decides whether DNN predictions are discrete or continuous). We clarify that input type and task type are public in TEE-shielded DNNs and are also required by all existing attacks.

On the other hand, our technique does *not* only apply to this weakest-knowledge setting; it also supports incorporating stronger knowledge if available. Table 1 lists the attacker's knowledge required by existing works. Below, we discuss their (in-)availability under various considerations and how our technique can be further enhanced with them.

DNN Knowledge. As in Table 1, all existing works require knowing the victim DNN's task type. Here, the task type only distinguishes regression vs. classification and specifies the DNN's output format (i.e., continuous or discrete). It does *not* indicate the number of or which classes that a classifier can predict (since they are private). Existing query-based attacks require having the victim DNN's confidence scores, which are always unavailable in TEE-shielded DNNs [80]. Previous hardware attacks (those speed up query-based attacks [61, 76]) rely on the victim DNN's structure. However, HyperTheft works without the structure information given our functionality-centric view and the carefully developed hyper-network; see Sec. 6 and Sec. 8.

While DNN structure is protected by TEEs, considering that DNN structure can be leaked via cache side-channel attacks [27, 48, 74] and TEEs are exploitable through cache side channels as well [24, 53, 55, 69], it is reasonable to also evaluate a stronger attacker with the structure knowledge. Hence, to comprehensively assess the threat, Sec. 8.3 further study how the structure information may enlarge the weight leakage.

Table 1: Knowledge required by existing attacks and HyperTheft. ✓ denotes public knowledge. Task Type and Input Type are public and required by all existing works. + and − indicate "required" and "not required" for private knowledge. Confidence of TEE-shielded DNNs is not available in all cases.

		Observation	Target DNN	DNN Knowledge			Data Knowledge		Query
				Task Type	Confidence	Structure	Input Type	Task Domain	
(Algorithmic) Query-Based Attack	[58, 68, 77] [12, 32, 57], etc.	Prediction Confidence	General DNN	✓	+	−	✓	+	+
(Hardware) Side-Channel Attack	DeepEM [76]	Electromagnetic	Binarized DNN	✓	+	+	✓	+	+
	DeepSteal [61]	Rowhammer [37]	Quantized DNN	✓	−	+	✓	+	+
	HyperTheft	Ciphertext Collision [43]	General DNN	✓	−	−	✓	−	−

Data Knowledge. All previous works assume knowing the victim DNN's input type and having data from the victim DNN's task domain (see definitions in Sec. 2.1), because they must train the student model to make it functional. However, this assumption does not always hold. For example, when attacking medical DNNs, data containing certain diseases may not be publicly available. HyperTheft directly generates functional DNN weights (from ciphertext side channels) without training them. Importantly, HyperTheft is task-wise generalizable: it can generate weights of *unseen* functionalities *without* data from the corresponding task domain.

In case data from the victim DNN's task domain are available, HyperTheft's generated weights can be leveraged to initialize the student model for queried-based attacks, significantly reducing the query cost when attacking TEE-shielded DNNs (see Sec. 8.3).

Attacker's Action. As marked in Table 1, all prior attacks assume an active attacker who can frequently query with the victim DNN. In practice, such action is often limited by the query budget (i.e., the number of queries). For instance, querying commercial DNNs incurs economic cost and DNN service providers may limit the number of queries. Our technique, in contrast, enables a passive attack: the attacker does not need to query the victim DNN, but only passively records ciphertext side channels when the victim DNN is executing. HyperTheft is also highly stealthy: it only requires ciphertext side channel traces logged from the victim DNN's a few executions (with each one for a binary classification), and the cost of developing HyperTheft is comparable to training a student model as in prior attacks (as elaborated in Sec. 6).

5 EXPLORATIONS AND INSIGHTS

In this section, we explore key properties of DNN weight and functionality that inspire our technique. We start by visualizing DNN weights w.r.t its performance. Since DNN's expressiveness is supported by its non-linearity, we use the XOR problem, a non-linear task, as a representative example. The XOR task is defined as a binary classification: given an input $x \in \mathbb{R}^2$, the ground truth label is decided as $y = (x[0] > 0) \oplus (x[1] > 0)$. We set a two-layer DNN structure and train 40K different DNNs with this structure to solve the XOR task. These DNNs have varied test accuracy ranging from ~50% (i.e., random guess) to >99.9% (i.e., nearly perfect prediction). This way, we obtain 40K different DNN weights having different performances for the same XOR task.

We then project these weights onto a two-dimensional space via PCA [59] to ease the visualization. As shown in Fig. 2, each dot denotes one DNN weight and its coordinates indicate the values

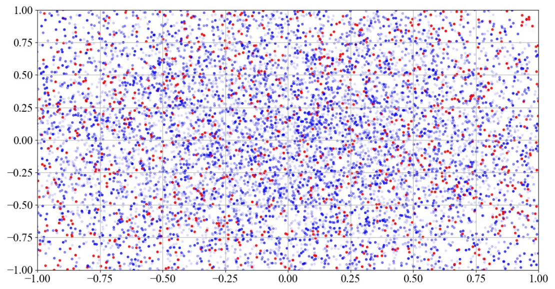
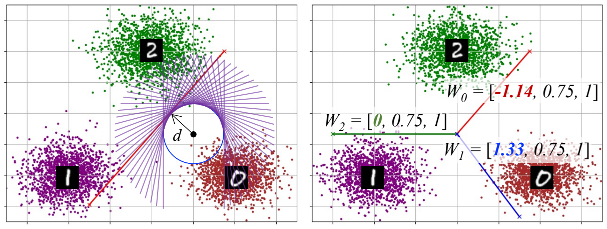


Figure 2: Visualization of different DNN weights w.r.t. their accuracy. Each dot denotes one DNN weight, and its coordinates (which are normalized into $[-1, 1]$) represent the values of weight elements. Weights of $>$ 80% accuracy are marked in red. For blue dots (i.e., weights having $\leq$ 80% accuracy), a more transparent color indicates lower accuracy.

of weight elements. Red dots mark DNN weights having $>$ 80% test accuracy, which can be deemed as functional DNN weights since they enable DNN intelligence. The remaining weights are blue-colored where higher transparency indicates lower accuracy. Fig. 2 reveals that, functional DNN weights (i.e., red dots) sparsely and discontinuously distribute in the whole space. Therefore, we have the following two conclusions.

First, slightly perturbing a few weight elements (i.e., changing a dot's coordinates in Fig. 2) can turn a functional weight (red dot) into a non-functional one (blue dot). Second, DNN weights of distinct elements (i.e., two far-flung red dots) can have equivalent functionality (i.e., solving the XOR task).

Motivation. The first conclusion is aligned to results in prior DNN attacks, which demonstrate that changing a few (out of millions) weight elements can totally deplete DNN intelligence [62, 75]. It also renders the impracticality of per-element recovery of DNN weights: as long as the exact value of a weight element is not recovered under this scheme (e.g., some elements are not leaked), the inferred DNN weight is likely non-functional. The second conclusion can be drawn from DNN training, whose different runs generate distinct but equivalent weights. It is also aligned to existing DNN pruning works, where a DNN's functionality remains same after replacing more than 90% of its weight elements with zeros [82]. This conclusion sheds light on the feasibility of *recovering different but functionality-equivalent DNN weights from partial observations of the victim DNN's weight.*



(a) Decision boundary of digit "0" (b) Decision boundaries of three classes

Figure 3: Decision boundaries for three classes.

DNN Functionality. A DNN's intended task uniquely decides its functionality. Fig. 3 shows an example where a DNN, whose last layer is $y = Sigmoid(\theta x + b)$[6], classifies three clusters of digits "0", "1", and "2". To ease the visualization, we only display the last layer of the DNN. We clarify that our attack is applied to the full DNN.

This task requires the DNN to split the input space (which consists of all valid digits "0", "1", and "2") to separate different digits. The DNN accordingly forms the required functionality by training θ and b. Each row in the concatenated matrix $W = [\theta, b]$ indicates a line drawn by the DNN. Given the trained weight:

$$\theta = \begin{bmatrix}\theta_0\\ \theta_1\\ \theta_2\end{bmatrix} = \begin{bmatrix}-1.14 & 0.75\\ 1.33 & 0.75\\ 0 & 0.75\end{bmatrix}, \quad b = \begin{bmatrix}b_0\\ b_1\\ b_2\end{bmatrix} = \begin{bmatrix}1\\ 1\\ 1\end{bmatrix}, \tag{1}$$

the first row $[\theta_0, b_0] = [-1.14, 0.75, 1]$ is marked as the red line, $\theta_0 \cdot x + b_0 = 0$, in Fig. 3(b). After drawing all three lines as in Fig. 3(b), the DNN's is capable of classifying digits. Overall, DNN structure (i.e., the dimension of W) reflects *how many* lines are drawn, and DNN weight decides *where and how to draw these lines*.

Intuitively, each row of $[\theta, b]$ also characterizes a binary classification. In Fig. 3(a), the red line separates "0" from other digits (i.e., classifying if a digit is "0"). Similarly, the blue and green lines in Fig. 3(b) (derived from the second and third rows of $[\theta, b]$) classifies if a digit belongs to "1" or "2", respectively. As in Eq. 1, despite that the three rows of $[\theta, b]$ only differ in the first element, they represent distinct functionalities for different binary classifications; this is consistent with our first conclusion delivered from Fig. 2.

Intermediate Outputs Reflect Functionality. Given an input x, each layer's output (a.k.a. the intermediate output) describes x's relative position w.r.t. the lines drawn by this layer. Considering Fig. 3(a) where an input is marked as the black dot, suppose the first element of its intermediate output (from the layer we discussed above) is $-d$ ($d > 0$), we know that x is below (since $-d < 0$) the red line $\theta_0 \cdot x + b_0 = 0$ and the distance is $\frac{d}{|\theta_0|}$ (see proof in Appx. A). With this information, we can infer that the first row of $[\theta, b]$ corresponds to a line that locates in the region covered by the purple lines in Fig. 3(a). While in large DNNs, the above case may become more complicated due to layer propagations and the high non-linearity, we can safely conclude that a DNN's intermediate outputs reflect its functionality. Furthermore, given that ciphertext collisions are due to intermediate computations of TEE-shielded DNNs, it is therefore reasonable to believe that *DNN functionality can be reflected from ciphertext side channels*.

[6]The `Sigmoid` activation function is commonly used to output class probabilities.

Generating Functionality-Equivalent Weights. Taking all the insights above, we see the infeasibility of recovering exact weight elements in the context of ciphertext side-channel attack. However, since ciphertext side channel can reflect DNN functionality, we aim to directly generate *functionality-equivalent* weights from the victim DNN's ciphertext side channels. To achieve this, the key obstacles are how to properly represent and extract DNN functionality from ciphertext side channels, and how to limit the required victim's knowledge to only public information. Below, we introduce our solution in Sec. 6.

6 SOLUTION AND TECHNICAL DETAILS

6.1 Overview and Goals

Fig. 4 illustrates the workflow of HyperTheft. Similar to existing automated analysis approaches [22, 79], our technical pipeline also consists of an offline and an online stage. As illustrated in Fig. 4(a), when attacking an unknown TEE-shielded DNN F of unknown weight W in the online stage, we collect one ciphertext side channel trace s from F's execution. We then feed s to HyperTheft to directly generate weight $\hat{\mathcal{W}}$ for a surrogate model $\hat{\mathcal{F}}$ (whose structure can be different from F), so that $\hat{\mathcal{F}}$ is functioning consistently with F.

The offline stage exclusively develops HyperTheft using attacker's own data and DNNs without interacting with the victim DNN; it primarily trains a hyper-network for the weight generation. Overall, the offline stage aims to achieve the following goals:

❶ Handling the partially leaked weight information;
❷ Forming task-wise generalizable weight generation;
❸ Capturing functionalities and their equivalence;
❹ Generating functional weights from a single trace;
❺ Supporting both regression and classification tasks;
❻ Modeling correlations between weight elements;
❼ Maximizing performance with limited observations.

In the following, we first introduce how to build HyperTheft and the training data/pipeline/objective w.r.t. to the seven goals in Sec. 6.2. Then, we introduce our in-depth optimizations in Sec. 6.3.

6.2 Building and Training HyperTheft

Encoder and Decoder (❶). As illustrated in Fig. 4(a), HyperTheft consists of two components: an encoder E that converts ciphertext side channel s into a latent variable z, and a decoder D that generates DNN weights $\hat{\mathcal{W}}$ according to z. Here, we let z have a much lower dimension than both s and $\hat{\mathcal{W}}$ due to the following reasons: 1) considering the frequent memory accesses in DNNs which result in lengthy s (i.e., containing millions of ciphertext collision records), a lower-dimensional z can force E to neglect irrelevant records in s and to focus on functionality-related information (guided by proper training objectives, as will be introduced later). Besides, 2) the dimension expansion process in D (i.e., from z to $\hat{\mathcal{W}}$) encourages D to infer the *unleaked* information (as ciphertext side channels only leak partial weight information), instead of merely transmitting information in s. For efficiency, we implement E and D as multilayer perceptrons (MLPs); empowered by our training algorithms, such simple forms of E and D work sufficiently well in practice.

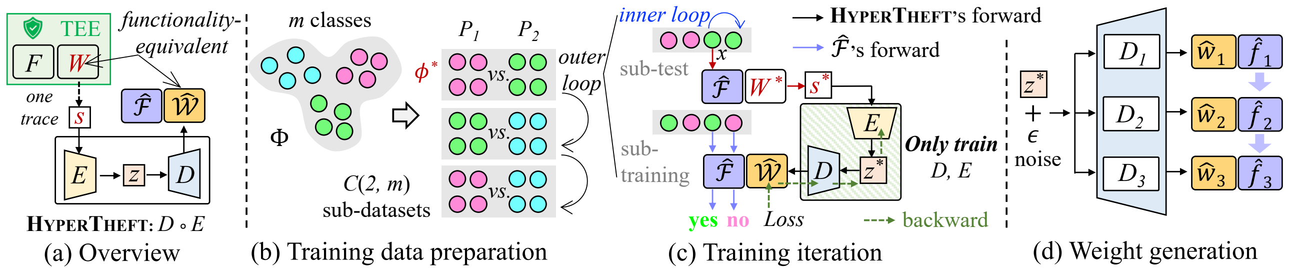


Figure 4: Workflow of HyperTheft. In Fig. 4(a), F and its input, output, and weight W are protected by TEEs. HyperTheft only takes F's one ciphertext side channel trace s as input and generates a different but functionality-equivalent weight $\hat{\mathcal{W}}$ for a surrogate model $\hat{\mathcal{F}}$. $\hat{\mathcal{F}}$ has a different structure from F. Fig. 4(b) illustrates how our training "data" are constructed as different binary classification (or regression) tasks. In Fig. 4(c), we illustrate how each training iteration is performed. Fig. 4(d) shows how we implement stochastic generation via random noise ϵ and separately generate weight $\hat{w}_i$ for layer $\hat{f}_i$.

Training "Data" Construction (❷, ❺). Unlike conventional DNN training data that is formed as a set of input-output pairs, HyperTheft's "training data" is a set Φ of different binary classification or regression *tasks*. This setup helps the weight generation generalize from known (training) tasks to unknown (test) tasks. We first collect some data that have the same input type (e.g., merely natural images) with the victim DNN. Note that their task domain does *not* overlap with the victim DNN's task domain. As in Fig. 4(b), if the victim DNN performs a classification task and suppose the attacker's data have total m classes, we form $C(2, m)$ [7] (i.e., the number of 2-combinations for a set of m elements) sub-datasets with each for a binary classification task. For victim DNNs performing regression tasks, attackers can randomly divide their data into sub-datasets for different regression tasks.

All sub-datasets also have their training and test sets. We denote them as *sub-training* and *sub-test* sets. To prepare "victim" DNNs for the offline training, we also train a DNN $\hat{\mathcal{F}}_{W^*}$ (whose weight is W^* and the structure is the same as our surrogate model) for each task $\phi^* \in \Phi$ using its corresponding sub-training set. We ensure each $\hat{\mathcal{F}}_{W^*}$ is well-trained to a satisfactory accuracy or loss.

Per-Task Granularity and Single-Trace Input (❷, ❹, ❼). As shown in Fig. 4(b)-(c), in each training iteration, we randomly pick a task ϕ^* and its corresponding trained DNN $\hat{\mathcal{F}}_{W^*}$. We then randomly select one input data x from ϕ^*'s sub-test set and feed it to $\hat{\mathcal{F}}_{W^*}$. It's worth noting that using x from the sub-test set does *not* misuse the dataset because our "data" are split as training and test *tasks*; these sub-test sets belong to training tasks. When $\hat{\mathcal{F}}_{W^*}$ is executing with x, we collect a ciphertext side channel s^* as one training input of HyperTheft. Then, HyperTheft takes s^* and outputs $\hat{\mathcal{W}}$. HyperTheft is optimized to generate $\hat{\mathcal{W}}$ that is functionality-equivalent to W^* (i.e., $\hat{\mathcal{F}}_{W^*}$'s weight).

Note that each input of HyperTheft is a ciphertext side-channel trace logged from the victim DNN's only *one* execution. Recall as explored in Sec. 5, a DNN's one execution can reflect rich information of its functionality. The single-trace setup can make the online attack stealthy and minimize the online attack's cost. Sec. 6.3 will introduce our optimizations for this single-trace setup.

Functionality-Centric Training Objective (❶, ❷, ❸, ❻). Our training objective aims to measure the equivalence between the generated weight $\hat{\mathcal{W}}$ and the target weight W^*. As elaborated in Sec. 5, element-wise distance metrics are inapplicable to DNN weights: a weight having similar elements with W^* may exhibit a distinct functionality and is even non-functional. Given that W^*'s functionality was formed when training W^* on ϕ^*'s sub-training set, we can also enable an equivalent functionality for $\hat{\mathcal{W}}$ using this sub-training set. Nevertheless, we should *not* directly train $\hat{\mathcal{W}}$ on the sub-training set, because the victim DNN's training data (or data from the same task domain) is unavailable during online attack.

Intuitively, since the generated weight $\hat{\mathcal{W}} = D \circ E(s^*)$ is decided by E and D[8] which are MLPs, updating E and D's weights can also change $\hat{\mathcal{W}}$. Thus, to make $\hat{\mathcal{W}}$ functionality-equivalent to W^*, we can *indirectly* optimize $\hat{\mathcal{W}}$ by only training E and D with ϕ^*'s sub-training set. To this end, we design the following training objective:

$$\arg\min_{D,E} L(\hat{\mathcal{F}}_{\hat{\mathcal{W}}}, \phi^*), \quad \text{where} \quad \hat{\mathcal{W}} = D \circ E(s^*). \tag{2}$$

L denotes the loss function associated with ϕ^*. For example, L is cross-entropy loss for classification and mean squared error for regression. $L(\hat{\mathcal{F}}_{\hat{\mathcal{W}}}, \phi^*)$ denotes the loss calculated over ϕ^*'s sub-training set when $\hat{\mathcal{F}}$ has the weight $\hat{\mathcal{W}}$ (i.e., the output of $D \circ E$).

As illustrated in Fig. 4(c), similar to conventional DNN training, this objective minimizes the loss L by back-propagating through $\hat{\mathcal{F}}_{\hat{\mathcal{W}}} \rightarrow D \rightarrow E$. However, it does not directly train $\hat{\mathcal{F}}_{\hat{\mathcal{W}}}$, but only trains E and D to optimize their output $\hat{\mathcal{W}}$. As a result, $\hat{\mathcal{W}}$ generated from the subsequent iterations (after training E and D) can reduce the loss $L(\hat{\mathcal{F}}_{\hat{\mathcal{W}}}, \phi^*)$. Finally, the generated $\hat{\mathcal{W}}$ will have an equivalent functionality with W^* when it reduces $L(\hat{\mathcal{F}}_{\hat{\mathcal{W}}}, \phi^*)$ to a satisfactory level.

This training scheme is fundamentally different from conventional DNN training. Since the training optimizes HyperTheft's generation ability across different tasks, it makes the weight generation *task-wise generalizable*: given s^* logged from DNNs solving

[7] In practice, using all $C(2, m)$ sub-datasets is usually unnecessary. Our results in Sec. 8 show that 25-30 sub-datasets are sufficient.

[8] We should *not* modify the logged ciphertext side channel s^*.

different unseen tasks, $D \circ E(s^*)$ generates weights that are capable of solving the corresponding tasks. Importantly, we find that HyperTheft can generate functional weights for DNNs that are much larger than HyperTheft. That is, we do *not* need to build a huge HyperTheft whose weight subsumes all functionality-equivalent weights of the victim DNN. Essentially, the objective in Eq. 2, together with the dimension expansion of D, enable HyperTheft to infer unleaked information in s^* through the functionality perspective. Moreover, representing functionality should require less information than representing the weight itself; this is also supported by existing DNN weight pruning works [20, 51, 82].

6.3 Optimizations for HyperTheft

Orchestrating Training Iterations (❷, ❹). When HyperTheft is taking a ciphertext side channel trace s^* in each training iteration, it is impractical to train HyperTheft using ϕ^*'s whole sub-training set, as it incurs a cost comparable to training total #training iterations DNNs; without sufficient training iterations, the encoder E may be unable to extract useful information from s^*.

Worse, our tentative experiments show that HyperTheft rarely converges under the above setup, because it poses conflicts between different training iterations. Suppose after one iteration, HyperTheft is able to generate (nearly) functional weights for the task ϕ^*. The subsequent iteration, however, requires HyperTheft to generate functional weights for another different task. Since universal weights (that can solve all tasks) do not exist [65], fulfilling the subsequent iteration's requirement may break HyperTheft's generation ability formed in the current iteration.

To reduce the overhead and alleviate conflicts between different iterations, we adopt a sampling strategy. Instead of using ϕ^*'s whole sub-training set, we randomly sample one data instance from the sub-training set to optimize Eq. 2. Different from using the whole sub-training set that *independently and subsequently* trains HyperTheft for each task, this sampling strategy *jointly* trains HyperTheft for all tasks over multiple iterations. It, to some extent, explores the similarity between tasks and helps HyperTheft to form the task-wise generalization. In addition, the sampling only reduces the cost of each training iteration; we still have sufficient iterations (that use different s^* as E's inputs) to train E.

Decoupling Functionality (❹, ❺, ❼). As discussed in Sec. 4, TEE-shielded DNNs only return the final prediction, and we do not assume knowing how many (i.e., reflected from DNN structure) or which (i.e., the task domain) classes the victim DNN can predict. Our current attack pipeline is adequate for binary classification and regression tasks. Below, we discuss how it can be applied to k-class classification ($k > 2$).

As explored in Sec. 5, the full functionality of k-class classification can be decoupled as k (sub-)functionalities of binary classification. Thus, attackers can steal the full functionality via k surrogate models. However, the binary classification decoupled from k-class classification (i.e., whether an input is from a class or not) is slightly different from our training binary classification (i.e., classifies two different classes). To address this, we actively flip the two labels of each sub-training set when training HyperTheft.

As illustrated in Fig. 4(b)-(c), the sub-dataset has two classes P_1 and P_2. If the ciphertext side channel s^* is collected when $\hat{\mathcal{F}}_{W^*}$ is taking an input x from class P_2, we set P_2 in the sub-training set as the label "yes" whereas P_1 as the label "no", and vice versa if x is from class P_1. This way, every time a TEE-shielded DNN executes with an input x of class P_x, HyperTheft can generate a surrogate model which is able to predict whether an input belongs to P_x or not. By collecting ciphertext side channels from the victim DNN's (minimal) k executions, HyperTheft can generate k surrogate models where the i-th surrogate model predicts an input's confidence of belonging to the i-th class, thereby stealing the full functionality. Since HyperTheft's weight generation generalizes across tasks, we only need to train HyperTheft once.

Stochastic and Layer-Wise Generation (❹, ❻, ❼). Inspired by DNN's stochastic training algorithms (e.g., SGD [2], Adam [38]), we also implement a stochastic weight generation by adding a random noise ϵ to z^*, as illustrated in Fig. 4(d). This way, HyperTheft can more extensively utilize one logged side channel to generate different but equivalent weights in different runs. The resulting surrogate models (for the same binary classification or regression task) can form a majority voting (i.e., using the most frequent prediction from these surrogate models as the final prediction) to improve the accuracy [21].

In addition, following advice from [63, 81], HyperTheft uses one encoder E, but n independent decoders $D_1, \ldots, D_n$ to generate weights $\hat{\mathcal{W}} = \{\hat{w}_1, \ldots, \hat{w}_n\}$ for layers $\hat{f}_1, \ldots, \hat{f}_n$, respectively. As shown in Fig. 4(d), each D_i takes an identical input $z^* + \epsilon$ (i.e., the ϵ is fixed after sampled). Beyond our functionality-centric objective that implicitly captures correlations between weight elements, this layer-wise generation can explicitly model the layer propagation in DNNs, while considering the independent execution of each layer.

7 IMPLEMENTATION AND SETUP

We implement HyperTheft in PyTorch (ver. 2.0.0) with about 2.5K LOC. Both the encoder E and decoder D in HyperTheft are implemented as three-layer MLPs with ReLU as the activation function. The latent variable z is set to have 64 dimensions. When generating a DNN weight having v elements, the output of D is a v-dimensional vector; this vector is then reshaped according to the structure of the surrogate model. HyperTheft is trained using Adam optimizer with a learning rate of 0.002 and the training takes 30 epochs. Each epoch takes 15 minutes on one Nvidia GeForce RTX 2080 GPU. Note that training HyperTheft is a one-time effort given its task-wise generalizable weight generation.

Side Channel Preparation. Since a TEE-shielded DNN isolatedly operates in its memory, and given the high privilege of attackers (e.g., hypervisor, OS), ciphertext side channel distinguishes other conventional side channels by its clean and noise-free nature [42, 43]. Also, unlike prior DNN attacks, HyperTheft does not need to interact with or log ciphertext side channels from the victim DNN in the offline stage. Hence, we can speed up the offline data preparation by mimicking an exploitation tool using Intel Pin [50] on attacker's own DNNs. During the online stage, we use these exploitation tools to collect ciphertext side channels from (unknown) TEE-shielded DNNs and steal their weights. To date, two mature exploitation tools, `CipherLeak` [43] and `SEV-Step` [42, 72] have been proposed.

`CipherLeak` is coarse-grained but is more scalable by operating in a page granularity, while `SEV-Step` can precisely track memory write in an instruction granularity but is costly.

We configure `CipherLeak` following the default setup. However, when setting up `SEV-Step`, we note that its current implementation is based on Linux kernel Ver. 5.14, which is too old to be compatible with the latest SEV-SNP firmware (Ver. 1.55) required to launch a guest VM for DNNs. We have contacted the developers for support, and the upgrading is still in process by the time of submission given the considerable manual efforts required. Hence, we simulate `SEV-Step` using Intel Pin. Our preliminary explorations show that ciphertext side channels collected using our Pin-based simulation and `SEV-Step` are identical (on those programs supported by both). As suggested by the developers, a potential (and might be the only) factor that could differ our Pin-based simulation from `SEV-Step` is due to the multiple memory writes occurred during a given APIC timer interval, where some memory writes can be periodically missed by `SEV-Step` [72]. Thus, for completeness, we also benchmark HyperTheft towards this impact in Appx. B. Overall, HyperTheft constantly achieves promising performance even when considerable (e.g., $^{63}/_{64}$) memory writes are missed.

Considering the stealthy and efficiency, we employ `CipherLeak` to exploit large Transformer-Based DNNs (e.g., ViT [18]). Note that it is often impractical to put these extremely large DNNs into TEEs. Existing works divide large DNNs into slices and only shield sensitive slices via TEEs [28, 52, 80], leaving other slices as public. Therefore, we adopt HyperTheft to recover weights of the TEE-shielded DNN slices. For moderate-size DNNs (e.g., ResNet, LSTM) that can be fully shielded by TEEs, we use `SEV-Step` (simulated via Pin due to the compatibility issue) to collect ciphertext side channels and recover the full DNN weights.

Side Channel Representation. We first record ciphertext collisions for different addresses in the victim DNN's (isolated) memory region. If two consecutive writes to the same address have the same content, we record a bit 1 for this address; otherwise, we record a bit 0. This way, we collect a binary collision sequence for each address. Given that DNN intermediate outputs are floating-point numbers, most addresses do not have collisions and can be neglected to reduce the number of target addresses. Then, we rank the remaining collision sequences based on the order of their first writes, and concatenate them as one single sequence. This concatenated sequence denotes one ciphertext side-channel trace.

8 EVALUATION

In this section, we first introduce the evaluation setup in Sec. 8.1. We then evaluate HyperTheft under the weakest knowledge in Sec. 8.2, where victim DNNs are running with the latest version of PyTorch (ver. 2.1.0). Sec. 8.3 evaluates the attack surface by considering DNN executables and different versions of PyTorch, and studies how stronger knowledge (under certain possible scenarios) can enhance the weakest-knowledge attacks. Sec. 8.4 further shows that attacks mitigated by TEEs can be largely enhanced by our recovered weights.

In Appx. C, we present DNN modules that induce the leakage. Overall, the leakages are due to basic computation operators shared by different DNNs.

8.1 Evaluation Setup

Table 2: Evaluated datasets and victim DNNs. ImageNet is evaluated under a *cross-dataset* setting. For ViT, we recover the weights of the multi-head self-attention layers.

Dataset	Input Type	Task Type	Remarks	DNNs	MSE/Acc.
Stock [34]	Stock price	Reg.	Sequence	LSTM	1.46∼1.95
Chest X-ray [70]	Medical image	Classif.	2-class	LeNet	90∼95%
				ResNet	90∼95%
MNIST [16]	Digit	Classif.	2-class	LeNet	94∼98%
				ResNet	94∼98%
				ViT	94∼98%
CIFAR10 [39]	Natural image	Classif.	2-class	LeNet	90∼95%
				ResNet	90∼95%
				ViT	90∼95%
ImageNet [15]	Natural image	Classif.	Multi-class & Cross-dataset	7 DNNs*	90∼95%

* LeNet, ResNet, VGG, SqueezeNet, MobileNet, DenseNet, and ViT.

DNNs and TEE Usage. Our evaluation considers representative DNNs and different usages of TEEs. For classical moderate-size DNNs (DNNs in Table 2 except for ViT), we put the full DNNs into TEE and generate their full weights using HyperTheft. However, given the large size of Vision Transformer (ViT) [18], it is impractical to shield the full DNN via TEEs. Following recent works [28, 52, 80], we consider shielding only sensitive slices of ViT and using HyperTheft to recover weights of the shielded DNN slices. Since ViT's effectiveness is due to the self-attention, we shield ViT's multi-head self-attention layers via TEEs and use HyperTheft to generate the corresponding weights.

"Data" (i.e., Task) Construction. Table 2 lists our adopted datasets. We refer interested readers to Appx. D for their details. Stock dataset is used to predict the stock price for different companies which is a regression task. We divide Stock dataset according to the company to form different regression tasks. We use MNIST, CIFAR10, and Chest X-ray to evaluate HyperTheft for binary classification w.r.t. different input types. For each dataset, we randomly choose two classes to form a binary classification as our test "data". The remaining classes are used to construct $C(2, 8) = 28$ binary classifications as tasks in our training "data", so that task domains of HyperTheft's training and test data/tasks do not overlap.

ImageNet is used to evaluate HyperTheft for multi-class classification. Since both CIFAR10 and ImageNet are natural images, we consider a *cross-dataset* setting to eliminate potential bias within the same dataset: we train HyperTheft using binary classifications constructed via CIFAR10 but evaluate it with k-class classification formed via ImageNet. We set $k \in \{2, 10\}$. Overlapped classes between ImageNet and CIFAR10 are excluded.

For cross-validation, we consider five different combinations of training and test "data" splits in each setting, resulting in ($5 \times$ #datasets $\times$ #DNNs) distinct test tasks with each one corresponds to one unique victim DNN. For each test task, we generate 100 weights from the victim DNN's 100 different executions.

Surrogate Model Structure. For image DNNs, we follow the common practice and design the surrogate model as convolutional layers followed by fully-connected layers; the activation function is ReLU. We implement two surrogate models of different depths. The first one, dubbed as Conv, has 3 convolution + 2 fully-connected layers. The second one, dubbed as $\text{Conv}_{\text{deep}}$, has 5 convolutional + 3 fully-connected layers. For our regression task, since inputs are discrete sequences, our surrogate model is a basic recurrent neural network (RNN) following the common practice. We clarify that our surrogate model is much simpler than the victim DNNs.

Note that ViT (and all transformer-based DNNs) is implemented using only fully-connected (FC) layers. Also, in the slice-based protection, the structure of each TEE-shielded slice can be easily inferred from public DNN slices. Therefore, our surrogate model has the same structure as the ViT's self-attention layer. However, given the dense computations of FCs, training ViT requires careful regulations like dropout to avoid over-fitting [18], which can be private in practice. Thus, for the weakest-knowledge setting in Sec. 8.2, we assume attackers do not know the regulation. Sec. 8.3 then evaluates impacts of the regulation information.

Evaluation Criteria. Following existing works [32, 80], we use two criteria, fidelity and functionality, to evaluate DNN weights recovered by HyperTheft. Fidelity (*Fid*) calculates the percentage of test inputs (from the sub-test set of each test task) where the surrogate model and the victim DNN have identical predictions (including incorrect prediction). For classification, *Fid* can be directly calculated via predicted labels, whereas for regression, because the prediction is continuous numbers, we deem two predictions as identical if their difference is less than 2 (the stock prices vary with a range around 100). Functionality (*Fun*) denotes the task's own evaluation metric. *Fun* is the MSE or accuracy of all test inputs for regression or classification tasks, respectively. Higher accuracy indicates better results whereas lower MSE is better.

8.2 The Weakest-Knowledge Attack

In this section, we generate 100×5 different weights for each victim DNN, leading to more than 8K weights and evaluation results (for 17 different victim DNNs). To better reflect the average and fluctuations, we report the ranges of these results in Table 3.

Regression & Binary Classification. As shown in Table 3, HyperTheft can successfully recover DNN weights for both regression and binary classification using a *single* side-channel trace logged during the victim DNN's one execution. The recovered weights are functional (*Fun*) and also consistent (*Fid*) with the victim DNN's weights. We note that *Fid* is higher than *Fun* on average, because *Fid* also counts incorrect predictions. It also reflects that HyperTheft infers specific behaviors of the victim DNN beyond its overall functionality, despite that HyperTheft never queries the victim DNN. Recall that ciphertext collisions are generated due to the victim DNN's intermediate outputs. As explored in Sec. 5, these intermediate outputs specify both a DNN's functionality (i.e., how the input space is split) and its specific behaviors (i.e., where and how the splitting lines are drawn), rendering the superiority of HyperTheft's weight generation scheme.

Multi-Class Classification. As shown in the 14-24 rows in Table 3, HyperTheft is capable of (passively) generating weights for classification of more classes. Although classifying multiple classes is more challenging, HyperTheft alleviates this hurdle by cleverly decoupling the k-class classification as k binary classifications. Nevertheless, we note that Conv (the 14th row) exhibits a relatively lower capability of classifying 10 classes due to its insufficient depth. Nevertheless, when using $\text{Conv}_{\text{deep}}$ (the 15th row) as the surrogate model's structure, the *Fun* and *Fid* are largely improved.

Table 3: Results of the weakest-knowledge attack.

1	Dataset	DNN	Surrogate	#Classes	#Votes	*Fid*	*Fun*
2	Stock	LSTM	RNN	N/A	1	90~93%	1.49~1.97
3	Chest X-ray	LeNet	Conv	2	1	87~91%	83~88%
4		ResNet	Conv	2	1	88~90%	86~89%
5	MNIST	LeNet	Conv	2	1	87~91%	87~89%
6		ResNet	Conv	2	1	86~92%	84~91%
7		ViT	ViT	2	1	91~98%	91~97%
8	CIFAR10	LeNet	Conv	2	1	79~86%	77~83%
9		ResNet	Conv	2	1	78~86%	78~82%
10		ViT	ViT	2	1	80~87%	79~88%
11	ImageNet	LeNet	Conv	2	1	78~87%	79~85%
12		ResNet	Conv	2	1	78~88%	77~86%
13		ViT	ViT	2	1	80~87%	78~87%
14		ResNet	Conv	10	1	70~75%	68~73%
15		ResNet	$\text{Conv}_{\text{deep}}$	10	1	79~84%	76~83%
16		ViT	ViT	10	1	77~86%	78~85%
17		ResNet	Conv	10	5	85~88%	85~87%
18		ResNet	Conv	10	11	89~92%	88~89%
19		ResNet	Conv	10	21	93~95%	91~94%
20		VGG	Conv	10	11	91~94%	90~92%
21		SqueezeNet	Conv	10	11	90~92%	89~92%
22		MobileNet	Conv	10	11	90~93%	88~90%
23		DenseNet	Conv	10	11	92~93%	90~93%
24		ViT	ViT	10	11	91~94%	90~94%

Majority Voting. Table 3's 17-24 rows show that majority voting among multiple surrogate models (for the same task) can further improve *Fid* and *Fun*. As observed, while a single surrogate model based on Conv performs worse when classifying 10 classes, the results can be improved as being comparable to $\text{Conv}_{\text{deep}}$ with multiple Conv-based surrogate models. Note that we introduce stochasticity into the weight generation for majority voting, these different DNN weights are therefore generated using a single side channel trace from the victim DNN's one execution.

Surrogate Model. The above results show that the depth (i.e., the number of layers) of the surrogate model's structure primarily affects HyperTheft's functionality stealing. Without sufficient layers in the surrogate model (i.e., the non-linearity is limited), the generated weights may not be able to capture the functionality of more complex tasks. However, this should not be a major concern, as the insufficient depth problem can be alleviated by majority voting of multiple surrogate models, which is performed fully offline and does not bring extra cost to the victim DNN. In practice, users can set a moderate level of depth for the surrogate model (e.g., ~10). If the recovered DNN weights do not have satisfactory results, users can generate multiple weights (i.e., run HyperTheft multiple times with the same side channel trace) and conduct majority voting.

8.3 Attack Surface in Different Cases

Different Runtimes. To disclose the widespread leakage, we evaluate different versions of PyTorch. We also consider Glow, a popular deep learning compiler that compiles DNNs into executables. Since

Glow does not evolve too much, we only evaluate its latest version. Overall, the same DNN's internal computations are implemented distinctly in PyTorch and Glow-executables.

As in Table 4, for both Glow-executables and different versions of PyTorch, HyperTheft can constantly recover the victim DNN's weights, indicating HyperTheft's superiority and the wide existence of ciphertext side-channel leakage in different runtimes. That is, the leakage is presumably due to issues in DNN's own design (see detailed discussion in Sec. 9), rather than implementation defects in a specific runtime or version.

Table 4: Results of Glow and various PyTorch versions. We evaluate binary classification w/o using majority voting.

Dataset	DNN	Surrogate	Runtime	*Fid*	*Fun*
MNIST	LeNet	Conv	Glow	88~93%	88~92%
	ResNet		Glow	90~94%	88~94%
	LeNet		V1.13	86~91%	87~89%
	ResNet		V1.13	89~90%	85~91%
	LeNet		V1.10	88~92%	87~91%
	ResNet		V1.10	89~93%	86~90%
	LeNet		V1.7	90~92%	86~92%
	ResNet		V1.7	87~94%	88~91%
CIFAR10	LeNet	Conv	Glow	79~80%	77~81%
	ResNet		Glow	78~81%	78~79%
	LeNet		V1.13	78~83%	76~81%
	ResNet		V1.13	79~85%	75~83%
	LeNet		V1.10	80~81%	76~80%
	ResNet		V1.10	77~85%	77~82%
	LeNet		V1.7	79~84%	78~81%
	ResNet		V1.7	78~86%	78~83%

Table 5: Generating weights using ciphertext side channel traces logged from the victim DNN's multiple executions.

Dataset	DNN	Surrogate	#Classes	#Votes	#Traces	*Fid*	*Fun*
ImageNet	ResNet50	Conv	10	1	5	87~89%	86~89%
	ResNet50	Conv	10	1	11	86~91%	86~90%
	ResNet50	Conv	10	1	21	90~94%	92~94%

Multiple Executions. To evaluate how multiple side-channel traces (derived from *different* executions of the victim DNN) can improve HyperTheft's recovered weights, we let HyperTheft generate one weight for each trace and conduct majority voting among these weights. Results are in Table 5. Compared with the 17-19th rows in Table 3, the improvements brought by majority voting among weights generated via, 1) multiple traces vs. 2) HyperTheft's multiple runs using one trace, are comparable, indicating the merit of HyperTheft's stochastic generation.

Knowledge of DNN Structure. We evaluate how structure information boosts the attack by using the victim DNN's structure for the surrogate model. By cross-comparing Table 6 with the 2nd-15th rows in Table 3, we see that the results w/ and w/o structure information are comparable for binary classification and regression. However, when knowing the structure information, the result of 10-class classification is better than using Conv as the surrogate model, but is comparable to the $\text{Conv}_{\text{deep}}$ case in Table 3. This observation is consistent with our conclusion derived from Table 3: the structure information primarily helps attackers to determine an appropriate depth for the surrogate model. However, this can be complemented by majority voting among multiple surrogate models (generated using single trace or multiple traces). For ViT cases, while the regulation mechanism is critical when training a ViT from scratch, it does not notably affect HyperTheft's performance. We infer that those public DNN slices' weights, which were jointly trained with private slices' weights under the same regulation, help HyperTheft to encode the regulation into its generated weight.

Although structures may differ in terms of connectivity or hyperparameters, their implementations share the same vulnerable computing operations. E.g., the `cascade_sum` function (which performs pairwise sum) is frequently called by different layers; see more examples in Appx. C. This further highlights the severity of weight leakage in TEE-shield DNNs. Previous works often require the exact structure information to boost query-based model inference [61], whereas HyperTheft can enhance query-based attacks *without* such information, as evaluated below.

Table 6: Attack using the victim DNN's structure.

Dataset	DNN	Surrogate	#Classes	*Fid*	*Fun*
Stock	LSTM	LSTM	N/A	91~93%	1.44~1.96
Chest X-ray	LeNet	LeNet	2	85~91%	85~89%
	ResNet	ResNet	2	87~89%	86~89%
MNIST	LeNet	LeNet	2	86~94%	85~94%
	ResNet	ResNet	2	90~94%	90~93%
	ViT	ViT	2	91~98%	91~97%
CIFAR10	LeNet	LeNet	2	78~85%	77~83%
	ResNet	ResNet	2	79~85%	77~85%
	ViT	ViT	2	80~88%	79~87%
ImageNet	LeNet	LeNet	2	79~87%	79~86%
	ResNet	ResNet	2	78~85%	78~83%
	ViT	ViT	2	78~86%	78~85%
	ResNet	ResNet	10	79~83%	77~83%

Table 7: Attack with victim DNN's in-task-domain data.

Dataset	DNN	#Classes	*Fun*	Budget
CIFAR10	LeNet (✗)	2	77~83%*	≥ 70%
	LeNet (✗)	2	90~95%	≥ 80%
	LeNet (✓)	2	90~95%	~15%
	ResNet (✗)	2	78~82%*	≥ 70%
	ResNet (✗)	2	90~95%	≥ 80%
	ResNet (✓)	2	90~95%	~15%
ImageNet	ResNet (✗)	10	68~73%*	≥ 70%
	ResNet (✗)	10	90~95%	≥ 80%
	ResNet (✓)	10	90~95%	~15%

* *Fun* achieved by HyperTheft under the weakest attack; see Table 3.
✗: training the student model (i.e., Conv) from scratch.
✓: initializing the student model (i.e., Conv) with our generated weights.

Task Domain & Query. As generally assumed by prior query-based attack [12, 32, 61], even if the victim DNN's training data are private, it is possible to have some other data that cover the victim DNN's task domain. For instance, attackers have some public cat and dog images when attacking a DNN classifying cat vs. dog. Therefore, we also evaluate how HyperTheft can be further enhanced in this scenario.

To ease the comparison with previous works, we follow their settings where attackers query the victim DNN using in-task-domain data, but only use the predictions (*without* confidence scores) as labels to train a student model. Differently, we do not train the student model from scratch — the student model is trained based on HyperTheft's generated weights (the generation does not use in-task-domain data). Since prior query-based attacks primarily focus on classification, we evaluate classification tasks.

Results are given in Table 7. Aligned to existing works, we report the query budget as its relative percentage to the number of victim DNN's training data. By cross-comparing Table 3 with the 8th, 9th, and 14th rows in Table 7, we see that, in order to achieve comparable results with HyperTheft's weakest-knowledge attack, query-based attacks require at least 70% query budget. In contrast, while HyperTheft's generated weights from the weakest-knowledge attack are not as good as the victim DNN, they can reach the same *Fun* as the victim DNN with only ~15% query budget, significantly reducing the cost. In that sense, HyperTheft enables query-based attacks for TEE-shielded DNNs since TEE's mitigation aims to largely increase the query budget (see Sec. 2.2).

8.4 Enabled Attacks

Recall that as introduced in Sec. 4, attackers can also leverage the recovered weights to enable white-box attacks towards the victim DNN. This section accordingly evaluates how HyperTheft can enable two popular attacks, membership inference attack (MIA) and bif-flip attack (BFA).

Table 8: Attack success rate (ASR) of membership inference attacks enabled by HyperTheft. Upper bound (UB) denotes ASR on the white-box victim DNN. Baseline is 50%.

Dataset	DNN	Surrogate	ASR	UB
MNIST	LeNet	Conv	65.7%	80.1%
	ResNet	Conv	65.9%	80.8%
Chest X-ray	LeNet	Conv	60.8%	72.7%
	ResNet	Conv	60.2%	71.6%
CIFAR10	LeNet	Conv	57.7%	66.5%
	ResNet	Conv	57.0%	67.3%
ImageNet	LeNet	Conv	57.8%	67.3%
	ResNet	Conv	59.7%	66.6%

8.4.1 Membership Inference Attack. This section evaluates how HyperTheft's recovered DNN weights, which give attackers white-box surrogate models, can enable/enhance MIA towards the (black-box) victim DNN. Following the setup in previous works [80], we construct a test suite where 50% of its data are the victim DNN's training data (i.e., only data from the corresponding sub-training split) and the remaining 50% are non-training data. Therefore, the baseline attack success rate (ASR) is 50% [80]. Note that training and non-training data in the test suite have the same class, such that MIA will not downgrade to class-wise classification. We adopt MLDoctor [49] to conduct MIA. Each time we feed an input from the test suite into HyperTheft's generated surrogate model, and record outputs from all layers of the surrogate model. These outputs are then concatenated and fed into MLDoctor to predict the membership. In this setting, the surrogate model is generated under the weakest-knowledge attack in Sec. 8.2.

Results & Analysis. Table 8 lists the MIA results. Interestingly, despite that the surrogate model is never trained with victim DNN's training data, it still significantly improves the ASR (from 50%) to ~65% for MNIST and ~57% for CIFAR10 and ImageNet. As a reference, the ASR of directly applying MLDoctor on the white-box victim DNN (i.e., upper bound ASR) is 80% for MNIST and 67% for CIFAR10 and ImageNet. Recall that as evaluated in Sec. 8.2, besides stealing the overall functionality from the victim DNN, HyperTheft also infers some of its specific behaviors (e.g., predictions for specific inputs), which explains why HyperTheft's generated surrogate model is useful for MIA towards the victim DNN.

8.4.2 Bit-Flip Attack. We also evaluate how HyperTheft's recovered DNN weights can enable BFA. As introduced in Sec. 2.2, to conduct BFA, the main prerequisite is localizing elements in a DNN's weight that are critical to the intelligence (which is infeasible in TEE-shielded DNNs), such that bits can be flipped efficiently. Note that BFA requires knowing the victim DNN's structure and launching rowhammer in TEEs may have additional challenges [10, 11]. However, they are out of the scope of this paper; we primarily focus on weight-related requirements that are enabled by HyperTheft. To assess how HyperTheft boosts BFA, we generate the surrogate model under the weakest-knowledge + structure setting. We follow the localization strategy in DeepHammer [75], the state of the art in this field, to localize critical weight elements[9] in the surrogate model and measure their overlapping (in terms of locations in the DNN's structure) with critical weight elements in the victim DNN. Two metrics, precision and recall, are adopted in this evaluation. Precision quantifies the percentage of victim DNN's critical elements that are localized in the surrogate model. Recall, in contrast, measures how many weight elements localized in the surrogate model are also critical in the victim DNN.

Table 9: Results of bit-flip attack (BFA) enabled by HyperTheft. BFA requires knowing the victim DNN's structure.

Dataset	DNN	Surrogate	Precision	Recall
MNIST	LeNet	LeNet	12.0%	93.1%
	ResNet	ResNet	13.7%	94.0%
Chest X-ray	LeNet	LeNet	33.2%	95.7%
	ResNet	ResNet	34.4%	96.3%
CIFAR10	LeNet	LeNet	63.7%	99.4%
	ResNet	ResNet	55.6%	99.1%
ImageNet	LeNet	LeNet	57.2%	98.8%
	ResNet	ResNet	59.3%	99.2%

Results & Analysis. Table 9 reports the results. The recall values are promising: weight elements identified in the surrogate model are highly likely to be critical in the victim DNN. While the precision is relatively lower (i.e., not all critical weight elements in the victim DNN can be identified via the surrogate model), it should not be a concern. In fact, launching BFA does not require identifying all the critical weight elements [62, 75]; usually ~10 flips can completely deplete a DNN's intelligence. The identification

[9]Most existing BFA works focus on quantized DNNs and identify weight bits [62, 75]. Since our evaluated DNNs are general floating-point DNNs, our evaluations mainly focus on the weight element level.

step primarily helps attackers to filter out non-critical bits since flipping bits via rowhammer is costly [37, 62, 75]. With this regard, recall should be a more important metric than precision. Thus, we can conclude that our current results are sufficient to deliver the prerequisite of BFA against TEE-shielded DNNs.

9 DISCUSSION AND MITIGATION

Non-Linearity Enlarges Leakage. For cryptographic software studied in prior works, ciphertext collisions can easily occur since private keys only have bits 0/1. However, general DNNs evaluated in this paper have floating-point intermediate outputs: since the probability of sampling two identical floating-point values is negligible, ciphertext collisions should rarely occur.

With manual inspection, we find that DNN's non-linearity (i.e., the basis of DNN's intelligence) is the primary root cause of the frequent ciphertext collisions. Given continuous values within a certain range, non-linear functions often map them into smaller ranges. For instance, the `Softmax` function in DNNs maps $[-\infty, +\infty]$ into $[0, 1]$. Also, the derivatives of DNN's non-linear functions (e.g., `Sigmoid`) usually approach zero for large or tiny inputs, i.e., the output values change negligibly with such inputs. Moreover, some non-linear functions only output discrete values for certain inputs, e.g., `ReLU` always outputs 0 if its input is negative. Overall, since floating-point numbers have limited precision in modern computers (e.g., 32-bit), these non-linear mappings greatly increase the frequency of identical intermediate outputs, leading to frequent ciphertext collisions.

Hardware and Software Mitigations. Some recent TEEs (e.g., Intel TDX [14], ARM CCA [45]) redesign hardware architectures to mitigate ciphertext side channels. For example, Intel TDX always returns zeros when outer programs read the encrypted memory. However, such hardware mitigations require modifying the current hardware design, which is impractical for TEEs that have been broadly used (e.g., AMD SEV [35]). Importantly, they cannot fully eliminate ciphertext side channels because attackers can still access the ciphertext via DMA devices [67] or physical attacks such as memory bus snooping [40], cold boot attack [25], etc.

Li et al. [42] propose to mitigate the leakage via VMSA randomization, so that ciphertext is no longer deterministic. However, this scheme incurs considerable performance overheads and is not adopted by vendors. On the other hand, we foresee the high feasibility of implementing software-level randomization to specifically mitigate the leakage in TEE-shielded DNNs. In fact, DNNs exhibit robustness to random small perturbations [23] (not carefully crafted adversarial perturbations [9]) on their intermediate outputs. Hence, every time a DNN is executing, we can add random noise to DNN's intermediate outputs before they are written into memory. Since the main purpose is diversifying the written values and reducing collisions, we can make the noise negligible. To further minimize the impact on DNN's accuracy, we expect to accordingly refine conventional training algorithms to make DNNs robust to such noise. For example, existing robust training schemes [60] (which improve DNN's robustness to input perturbations) can be adapted for noise in DNN's intermediate outputs.

10 CONCLUSION

This paper presents HyperTheft to steal DNN weights from ciphertext side channels of TEE-shielded DNNs. We propose to generate functionality-equivalent weights and demonstrate its effectiveness and practicality. Weights generated by HyperTheft constantly achieve high accuracy under various DNNs, datasets, scenarios, and platforms, and can enable severe downstream DNN attacks.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. The HKUST authors were supported in part by an NSFC/RGC JRS grant under the contract N_HKUST605/23, a Google Research Scholar Award, and a RGC CRF grant under the contract C6015-23G. Yinqian Zhang was supported by NSFC grant number 62361166633.

REFERENCES

[1] [n. d.]. Research Artifact. https://github.com/Yuanyuan-Yuan/HyperTheft.
[2] Shun-ichi Amari. 1993. Backpropagation and stochastic gradient descent method. *Neurocomputing* (1993).
[3] ARM. 2023. Arm Confidential Compute Architecture software stack. https://developer.arm.com/documentation/den0127/latest.
[4] Saeid Asgari Taghanaki, Kumar Abhishek, Joseph Paul Cohen, Julien Cohen-Adad, and Ghassan Hamarneh. 2021. Deep semantic segmentation of natural and medical images: a review. *Artificial Intelligence Review* (2021).
[5] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. 2019. CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel. In *USENIX Security*.
[6] Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation learning: A review and new perspectives. *IEEE T-PAMI* (2013).
[7] Jakub Breier, Dirmanto Jap, Xiaolu Hou, Shivam Bhasin, and Yang Liu. 2021. SNIFF: reverse engineering of neural networks with fault attacks. *IEEE Transactions on Reliability* 71, 4 (2021), 1527–1539.
[8] Nicholas Carlini, Steve Chien, Milad Nasr, Shuang Song, Andreas Terzis, and Florian Tramer. 2022. Membership inference attacks from first principles. In *IEEE S&P*.
[9] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *IEEE S&P*.
[10] Pierre Carru. 2017. Attack ARM TrustZone using Rowhammer. In *GreHack*.
[11] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. 2020. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *IEEE S&P*.
[12] Varun Chandrasekaran, Kamalika Chaudhuri, Irene Giacomelli, Somesh Jha, and Songbai Yan. 2020. Exploring connections between active learning and model extraction. In *USENIX Security*.
[13] Vinod Kumar Chauhan, Jiandong Zhou, Ping Lu, Soheila Molaei, and David A Clifton. 2023. A brief review of hypernetworks in deep learning. *arXiv preprint arXiv:2306.06955* (2023).
[14] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. 2023. Intel TDX Demystified: A Top-Down Approach. *preprint arXiv:2303.15540* (2023).
[15] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.
[16] Li Deng. 2012. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE signal processing magazine* 29, 6 (2012), 141–142.
[17] Sen Deng, Mengyuan Li, Yining Tang, Shuai Wang, Shoumeng Yan, and Yinqian Zhang. 2023. CipherH: Automated Detection of Ciphertext Side-channel Vulnerabilities in Cryptographic Implementations. In *USENIX Security*.
[18] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *International Conference on Learning Representations*.
[19] Anuj Dubey, Rosario Cammarota, and Aydin Aysu. 2020. Maskednet: The first hardware inference engine aiming power side-channel protection. In *IEEE HOST*.

[20] Jonathan Frankle and Michael Carbin. 2018. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *ICLR*.

[21] Mudasir A Ganaie, Minghui Hu, AK Malik, M Tanveer, and PN Suganthan. 2022. Ensemble deep learning: A review. *Engineering Applications of Artificial Intelligence* (2022).

[22] Yansong Gao, Huming Qiu, Zhi Zhang, Binghui Wang, Hua Ma, Alsharif Abuadbba, Minhui Xue, Anmin Fu, and Surya Nepal. 2024. DeepTheft: Stealing DNN Model Architectures through Power Side Channel. In *IEEE S&P*.

[23] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. 2018. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *IEEE S&P*.

[24] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*. 1–6.

[25] Michael Gruhn and Tilo Müller. 2013. On the practicability of cold boot attacks. In *2013 International Conference on Availability, Reliability and Security*. IEEE, 390–397.

[26] David Ha, Andrew M Dai, and Quoc V Le. 2016. HyperNetworks. In *International Conference on Learning Representations*.

[27] Sanghyun Hong, Michael Davinroy, Yiğitcan Kaya, Stuart Nevans Locke, Ian Rackow, Kevin Kulda, Dana Dachman-Soled, and Tudor Dumitraş. 2018. Security analysis of deep neural networks operating in the presence of cache side-channel attacks. *arXiv preprint arXiv:1810.03487* (2018).

[28] Jiahui Hou, Huiqi Liu, Yunxin Liu, Yu Wang, Peng-Jun Wan, and Xiang-Yang Li. 2022. Model Protection: Real-Time Privacy-Preserving Inference Service for Model Privacy at the Edge. *IEEE Trans. Dependable Secur. Comput.* 19, 6 (2022), 4270–4284. https://doi.org/10.1109/TDSC.2021.3126315

[29] Bin Hu, Yan Wang, Jerry Cheng, Tianming Zhao, Yucheng Xie, Xiaonan Guo, and Yingying Chen. 2023. Secure and Efficient Mobile DNN Using Trusted Execution Environments. In *Asia CCS*.

[30] Weizhe Hua, Zhiru Zhang, and G Edward Suh. 2018. Reverse engineering convolutional neural networks through side-channel information leaks. In *DAC*.

[31] Intel. 2023. Product brief, 3rd gen intel xeon scalable processor for iot. https://www.intel.com/content/www/us/en/products/docs/processors/embedded/3rd-gen-xeon-scalable-iot-product-brief.html.

[32] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. 2020. High accuracy and high fidelity extraction of neural networks. In *USENIX Security*.

[33] Simon Johnson, Raghunandan Makaram, Amy Santoni, and Vinnie Scarlata. 2021. Supporting intel sgx on multi-socket platforms. *Intel Corp* (2021).

[34] Kaggle. 2017. Stock Dataset. https://www.kaggle.com/datasets/borismarjanovic/price-volume-data-for-all-us-stocks-etfs.

[35] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. *White paper* (2016), 13.

[36] Kyungtae Kim, Chung Hwan Kim, Junghwan John Rhee, Xiao Yu, Haifeng Chen, Dave (Jing) Tian, and Byoungyoung Lee. 2020. Vessels: efficient and scalable deep learning prediction on trusted processors. In *SoCC*.

[37] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 361–372.

[38] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[39] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[40] Dayeol Lee, Dongha Jung, Ian T Fang, Chia-Che Tsai, and Raluca Ada Popa. 2020. An Off-Chip attack on hardware enclaves via the memory bus. In *29th USENIX Security Symposium (USENIX Security 20)*.

[41] Taegyeong Lee, Zhiqi Lin, Saumay Pushp, Caihua Li, Yunxin Liu, Youngki Lee, Fengyuan Xu, Chenren Xu, Lintao Zhang, and Junehwa Song. 2019. Occlumency: Privacy-preserving Remote Deep-learning Inference Using SGX. In *MobiCom*.

[42] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. 2022. A systematic look at ciphertext side channels on AMD SEV-SNP. In *IEEE S&P*.

[43] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *USENIX Security*.

[44] Shaofeng Li, Xinyu Wang, Minhui Xue, Haojin Zhu, Zhi Zhang, Yansong Gao, Wen Wu, and Xuemin Sherman Shen. 2024. Yes, One-Bit-Flip Matters! Universal DNN Model Inference Depletion with Runtime Code Fault Injection. In *USENIX Security*.

[45] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. 2022. Design and verification of the arm confidential compute architecture. In *OSDI*.

[46] Yuepeng Li, Deze Zeng, Lin Gu, Quan Chen, Song Guo, Albert Y. Zomaya, and Minyi Guo. 2021. Lasagna: Accelerating Secure Deep Learning Inference in SGX-enabled Edge Cloud. In *SoCC*.

[47] Zheng Li, Yiyong Liu, Xinlei He, Ning Yu, Michael Backes, and Yang Zhang. 2022. Auditing membership leakages of multi-exit networks. In *CCS*.

[48] Yuntao Liu and Ankur Srivastava. 2020. Ganred: Gan-based reverse engineering of dnns via cache side-channel. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*.

[49] Yugeng Liu, Rui Wen, Xinlei He, Ahmed Salem, Zhikun Zhang, Michael Backes, Emiliano De Cristofaro, Mario Fritz, and Yang Zhang. 2022. ML-Doctor: Holistic Risk Assessment of Inference Attacks Against Machine Learning Models. In *USENIX Security*.

[50] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*.

[51] Eran Malach, Gilad Yehudai, Shai Shalev-Schwartz, and Ohad Shamir. 2020. Proving the lottery ticket hypothesis: Pruning is all you need. In *International Conference on Machine Learning*. PMLR, 6682–6691.

[52] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Soteris Demetriou, Ilias Leontiadis, Andrea Cavallaro, and Hamed Haddadi. 2020. DarkneTZ: towards model privacy at the edge using trusted execution environments. In *MobiSys '20: The 18th Annual International Conference on Mobile Systems, Applications, and Services, Toronto, Ontario, Canada, June 15-19, 2020*, Eyal de Lara, Iqbal Mohomed, Jason Nieh, and Elizabeth M. Belding (Eds.). ACM, 161–174. https://doi.org/10.1145/3386901.3388946

[53] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. Cachezoom: How SGX amplifies the power of cache attacks. In *Cryptographic Hardware and Embedded Systems–CHES*.

[54] Phong Q Nguyen and Jacques Stern. 2000. Lattice reduction in cryptology: An update. In *International Algorithmic Number Theory Symposium*. Springer, 85–112.

[55] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *USENIX ATC*.

[56] Daryna Oliynyk, Rudolf Mayer, and Andreas Rauber. 2023. I know what you trained last summer: A survey on stealing machine learning models and defences. *Comput. Surveys* (2023).

[57] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. 2019. Knockoff nets: Stealing functionality of black-box models. In *CVPR*. 4954–4963.

[58] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. 2017. Practical black-box attacks against machine learning. In *ACM Asia CCS*. 506–519.

[59] Karl Pearson. 1901. LIII. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science* 2, 11 (1901), 559–572.

[60] Zhuang Qian, Kaizhu Huang, Qiu-Feng Wang, and Xu-Yao Zhang. 2022. A survey of robust adversarial training in pattern recognition: Fundamental, theory, and methodologies. *Pattern Recognition* 131 (2022), 108889.

[61] Adnan Siraj Rakin, Md Hafizul Islam Chowdhuryy, Fan Yao, and Deliang Fan. 2022. Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories. In *IEEE S&P*.

[62] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. 2019. Bit-flip attack: Crushing neural network with progressive bit search. In *ICCV*.

[63] Neale Ratzlaff and Li Fuxin. 2019. Hypergan: A generative model for diverse, performant neural networks. In *ICML*.

[64] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint* (2018).

[65] Shai Shalev-Shwartz and Shai Ben-David. 2014. *Understanding machine learning: From theory to algorithms*. Cambridge university press.

[66] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *IEEE S&P*.

[67] Patrick Stewin and Iurii Bystrov. 2012. Understanding DMA malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 21–41.

[68] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. [n. d.]. Stealing machine learning models via prediction apis. In *USENIX Sec'16*.

[69] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *CCS*.

[70] Xiaosong Wang, Yifan Peng, Le Lu, Zhiyong Lu, Mohammadhadi Bagheri, and Ronald M Summers. 2017. Chestx-ray8: Hospital-scale chest x-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2097–2106.

[71] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. 2020. Big Numbers-Big Troubles: Systematically Analyzing Nonce Leakage in (EC) DSA Implementations. In *29th USENIX Security Symposium (USENIX Security 20)*. 1767–1784.

[72] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. 2023. SEV Step. https://github.com/sev-step/sev-step.

[73] Yun Xiang, Zhuangzhi Chen, Zuohui Chen, Zebin Fang, Haiyang Hao, Jinyin Chen, Yi Liu, Zhefu Wu, Qi Xuan, and Xiaoniu Yang. 2020. Open dnn box by power side-channel attack. *IEEE Transactions on Circuits and Systems II: Express Briefs* (2020).

[74] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. [n. d.]. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *USENIX Sec'20*.

[75] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. 2020. {DeepHammer}: Depleting the intelligence of deep neural networks through targeted chain of bit flips. In *USENIX Security*.

[76] Honggang Yu, Haocheng Ma, Kaichen Yang, Yiqiang Zhao, and Yier Jin. 2020. Deepem: Deep neural networks model recovery through em side-channel information leakage. In *2020 IEEE HOST*.

[77] Honggang Yu, Kaichen Yang, Teng Zhang, Yun-Yun Tsai, Tsung-Yi Ho, and Yier Jin. 2020. CloudLeak: Large-Scale Deep Learning Models Stealing Through Adversarial Examples.. In *NDSS*.

[78] Yuanyuan Yuan, Zhibo Liu, Sen Deng, Shuai Wang, Yinqian Zhang, and Zhendong Su. 2024. HyperTheft: Thieving Model Weights from TEE-Shielded Neural Networks via Ciphertext Side Channels. In *31st ACM Conference on Computer and Communications Security (CCS)*.

[79] Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Automated side channel analysis of media software with manifold learning. In *USENIX Security*.

[80] Ziqi Zhang, Chen Gong, Yifeng Cai, Yuanyuan Yuan, Bingyan Liu, Ding Li, Yao Guo, and Xiangqun Chen. 2024. No Privacy Left Outside: On the (In-) Security of TEE-Shielded DNN Partition for On-Device ML. In *IEEE S&P*.

[81] Andrey Zhmoginov, Mark Sandler, and Maksym Vladymyrov. 2022. Hypertransformer: Model generation for supervised and semi-supervised few-shot learning. In *ICML*.

[82] Hattie Zhou, Janice Lan, Rosanne Liu, and Jason Yosinski. 2019. Deconstructing lottery tickets: Zeros, signs, and the supermask. *NeurIPS* (2019).

A PROOF FOR SEC. 5

For the case mentioned in Sec. 5, where a DNN's last layer is implemented as $y = Sigmoid(\theta x + b)$, suppose the input is $x = [x_0, x_1]^\intercal$ and the output is $y = [y_0, y_1, y_2]^\intercal$. Given weight $W = [\theta, b]$:

$$\theta = \begin{bmatrix} \theta_{0,0} & \theta_{0,1} \\ \theta_{1,0} & \theta_{1,1} \\ \theta_{2,0} & \theta_{2,1} \end{bmatrix}, b = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}$$

we have $y_0 = Sigmoid(h_0)$ and $h_0 = \theta_{0,0}x_0 + \theta_{0,1}x_1 + b_0$.

From Weight to Lines. In practice, a binary classification predicts "yes" if $Sigmoid(h_0) > 0.5$, i.e., $h_0 = \theta_{0,0}x_0 + \theta_{0,1}x_1 + b_0 > 0$. Thus, in the two-dimensional space constituted by x_0 and x_1, the decision boundary is depicted by the line $\theta_{0,0}x_0 + \theta_{0,1}x_1 + b_0 = 0$. The same applies to y_1 and y_2, and we can draw the three lines as in Fig. 3.

Distance Between Dot and Line. As illustrated in Fig. 5, for an input (i.e., a dot) $A : [x_0^*, x_1^*]^\intercal$, its distance to the line $l : \theta_{0,0}x_0 + \theta_{0,1}x_1 + b_0 = 0$ equals the length of segment $\overline{AC}$, whose direction is orthogonal to the line l.

To compute the length of $\overline{AC}$, we first randomly select one dot from l, as marked by $B : [x'_0, x'_1]^\intercal$ in Fig. 5. Therefore, the length of $\overline{AC}$ denotes the vector $\vec{AB}$'s projection onto the line l's orthogonal

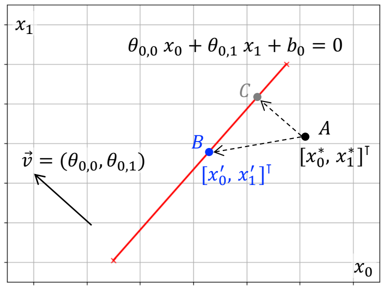

x_1
$\theta_{0,0}\, x_0 + \theta_{0,1}\, x_1 + b_0 = 0$
C
A
B
$[x_0^*, x_1^*]^\intercal$
$\vec{v} = (\theta_{0,0}, \theta_{0,1})$
$[x'_0, x'_1]^\intercal$
x_0

Figure 5: Distance between dot and line.

direction, namely $\vec{v} = (\theta_{0,0}, \theta_{0,1})$. Thus, we have

$$\begin{aligned} \left|\overline{AC}\right| &= \left|\vec{AB} \cdot \frac{\vec{v}}{|\vec{v}|}\right| \\ &= \left|(x'_0 - x_0^*, x'_1 - x_1^*) \cdot \frac{(\theta_{0,0}, \theta_{0,1})}{\sqrt{\theta_{0,0}^2, \theta_{0,1}^2}}\right| \\ &= \frac{1}{\sqrt{\theta_{0,0}^2, \theta_{0,1}^2}} \left|(x'_0 - x_0^*)\theta_{0,0} + (x'_1 - x_1^*)\theta_{0,1}\right|. \end{aligned}$$

Since B lies in the line l, we have

$$-b_0 = \theta_{0,0}x'_0 + \theta_{0,1}x'_1.$$

Therefore, the length of $\overline{AC}$ can be computed as

$$\left|\overline{AC}\right| = \frac{\left|\theta_{0,0}x_0^* + \theta_{0,1}x_1^* + b_0\right|}{\sqrt{\theta_{0,0}^2, \theta_{0,1}^2}}$$

Intermediate Output and Distance. When the dot $A : [x_0^*, x_1^*]^\intercal$ is taken by the layer $y = Sigmoid(\theta x + b)$, the intermediate output $h = [h_0, h_1, h_2]^\intercal = \theta[x_0^*, x_1^*]^\intercal + b$ is computed as

$$h = \begin{bmatrix} \theta_{0,0}x_0^* + \theta_{0,1}x_1^* + b_0 \\ \theta_{1,0}x_0^* + \theta_{1,1}x_1^* + b_1 \\ \theta_{2,0}x_0^* + \theta_{2,1}x_1^* + b_2 \end{bmatrix},$$

We can see that A's distance (i.e., $\left|\overline{AC}\right|$) to the line l (which corresponds to the first row of $[\theta, b]$) is

$$\frac{|h_0|}{\sqrt{\theta_{0,0}^2, \theta_{0,1}^2}} = \frac{|h_0|}{|[\theta_{0,0}, \theta_{0,1}]|}$$

Accordingly, the same rule also holds for the lines corresponding to other rows in $[\theta, b]$.

B IMPACTS OF APIC TIMER

As mentioned in Sec. 7, multiple memory writes can happen during a given APIC timer interval [72], which could differ SEV-Step from our Pin-based simulation. To benchmark the impact, we only record every *step*-th memory write and then measure ciphertext collisions. We consider *step* = 8, 16, 32, and 64.

Results are reported in Table 10 and Table 11. We note that when *step* ≤ 32, the *Fun* and *Fid* values (2nd-4th rows in Table 10 and Table 11) are comparable to our results in Sec. 8.2 and Sec. 8.3. Only *step* = 64 notably degrades the *Fun* and *Fid* values; however, the impacts are *not* significant. For example, when using Conv as the surrogate model's structure, even when *step* = 64 (i.e., only one memory write is logged among 64 memory writes), the *Fun* and *Fid* values are still around 80%. Moreover, we can further improve the HyperTheft's results under this extremely challenging scenario via majority voting; see the last two rows in Table 10 and Table 11.

Table 10: Benchmarking impacts of APIC timer (*Fun*).

Sur.	*step*	#Votes	*Fun*	Sur.	*step*	#Votes	*Fun*
Conv	8	1	85~89%	LeNet	8	1	84~92%
	16	1	83~86%		16	1	84~88%
	32	1	84~88%		32	1	81~89%
	64	1	72~83%		64	1	74~85%
	64	5	82~87%		64	5	80~86%
	64	11	86~89%		64	11	83~90%

Table 11: Benchmarking impacts of APIC timer (*Fid*).

Sur.	*step*	#Votes	*Fid*	Sur.	*step*	#Votes	*Fid*
Conv	8	1	86~91%	LeNet	8	1	87~92%
	16	1	85~88%		16	1	86~89%
	32	1	86~88%		32	1	85~90%
	64	1	73~85%		64	1	76~86%
	64	5	84~89%		64	5	83~88%
	64	11	87~92%		64	11	88~91%

C VULNERABLE FUNCTIONS IN PYTORCH

To localize vulnerable modules in DNNs, we use Pin to track functions whose memory writes trigger weight-dependent ciphertext collisions. In particular, Glow-executables implement each DNN layer as a function and *all* layers are vulnerable. We notice two types of collisions: the first one is caused by identical intermediate DNN outputs (whose possibilities are enlarged by the non-linearity). The second type is due to collisions between DNN's (intermediate) output zeros and the zero-initialized memory (e.g., `ReLU` can output many zeros). Different from Glow, PyTorch first implements basic computing operations (e.g., multiplication, sum), and uses these operations to further implement DNN layers.

We find that the leakage widely spans these computing operations (e.g., sum, pooling); an example list is given in Table 12. Interestingly, while we achieve comparable attacks over different versions of PyTorch, their vulnerable modules are slightly different. For instance, when running ResNet in PyTorch 2.1.0 (the latest version), the average pooling operation has leaks. However, such leakage is not identified in older versions when running the same ResNet.

D DATASETS AND DNNS

To present a self-contained paper, this section introduces statistics and characteristics of our evaluated datasets and DNNs. Overall, our evaluation comprehensively covers a wide range of representative datasets and DNNs.

Datasets. The Stock [34] dataset contains daily stock prices of various U.S. companies and is often used to train DNN for stock price prediction. MNIST dataset consists of single-channel (i.e., only black and white pixels) digit images. CIFAR10 dataset contains natural images (i.e., RGB images taken in real life). Both MNIST and CIFAR10 have 10 different classes. Chest X-ray has medical images of 14 diseases and the benign case; these images (especially those having disease information) are deemed as private. To align Chest X-ray with MNIST and CIFAR10, we randomly choose 9 disease classes and the benign class to form 10 classes.

For ImageNet, we randomly select 100 different classes and divide them into 10 groups for 10 different test tasks. Since both ImageNet and CIFAR10's input type is natural image, as mentioned in Sec. 8.2, our evaluations of ImageNet are conducted under a cross-dataset setting. That is, we train HyperTheft using binary classifications formed by CIFAR10, but test HyperTheft using 10-class classifications formed by ImageNet.

DNNs. Our evaluated DNNs in Table 2 are popular and representative. In particular, LeNet has a sequential structure whereas ResNet has a non-sequential structure. They are widely employed as (part of) modern DNNs backbone. LSTM has a recurrent structure, which usually processes discrete data sequences. The ViT and the multi-head self-attention mechanism are the building blocks of recent large language models (LLMs).

Table 12: Vulnerable Functions in PyTorch.

Version 2.1	Version 1.13	Version 1.10	Version 1.7
`cascade_sum`	`cascade_sum`	`cascad_sum`	`sum_kernel`
`as_strided`	`max_pool`	`max_pool`	`multi_row_sum`
`sgemm_mscale`	`segmm_pst`	`sgemm_mscale`	`conv2d_forward`
`conv2d_forward`	`unfolded2d_copy`	`unfolded2d_copy`	`sgemm_copyan`
`unfolded2d_copy`	`setStrided`	`as_strided`	`unfolded2d_copy`
`avg_pool2d`	`sgemm_copyan`	`sgemm_copyan`	`max_pool`