# CIPHERSTEAL: Stealing Input Data from TEE-Shielded Neural Networks with Ciphertext Side Channels

Yuanyuan Yuan*, Zhibo Liu*, Sen Deng*, Yanzuo Chen*, Shuai Wang*, Yinqian Zhang†, Zhendong Su◇.
*The Hong Kong University of Science and Technology
†Southern University of Science and Technology
◇ETH Zurich
{yyuanaq, zliudc, sdengan, ychenjo, shuaiw}@cse.ust.hk, yinqianz@acm.org, zhendong.su@inf.ethz.ch

***Abstract*—Shielding neural networks (NNs) from untrusted hosts with Trusted Execution Environments (TEEs) has been increasingly adopted. Nevertheless, this paper shows that the confidentiality of NNs and user data is compromised by the recently disclosed ciphertext side channels in TEEs, which leak memory write patterns of TEE-shielded NNs to malicious hosts. While recent works have used ciphertext side channels to recover cryptographic key bits, the technique does not apply to NN inputs which are more complex and only have partial information leaked. We propose an automated input recovery framework, CIPHERSTEAL, and for the first time demonstrate the severe threat of ciphertext side channels to NN inputs.**

**CIPHERSTEAL novelly recasts the input recovery as a two-step approach — information transformation and reconstruction — and proposes optimizations to fully utilize partial input information leaked in ciphertext side channels. We evaluate CIPHERSTEAL on diverse NNs (e.g., Transformer) and image/video inputs, and successfully recover visually identical inputs under different levels of attacker's pre-knowledge towards the target NNs and their inputs. We comprehensively evaluate two popular NN frameworks, TensorFlow and PyTorch, and NN executables generated by two recent NN compilers, TVM and Glow, and study their different attack surfaces. Moreover, we further steal the target NN's functionality by training a surrogate NN with our recovered inputs, and also leverage the surrogate NN to generate "white-box" adversarial examples, effectively manipulating the target NN's predictions.**

## 1. Introduction

Untrusted hosts constitute one major threat to the confidentiality of neural networks (NNs) and user data. Training NNs on malicious host platforms may leak the intellectual properties (IPs) of NN developers. Similarly, querying NNs from adversarial providers can expose user's private data. Trusted Execution Environments (TEEs) have been emerging as a promising and perhaps the most practical solution to shield NN training and achieve secure NN inference on untrusted hosts [56, 43, 39, 49, 30]. TEEs are hardware-based security mechanisms that encrypt sensitive data into ciphertexts via memory encryption. They are often implemented as a secure co-processor (e.g., Intel SGX [32]) or a secure virtual machine (e.g., AMD SEV [35]).

A well-known security concern of TEEs is their vulnerability to micro-architectural side channels such as cache or timing attacks [21, 58, 80], where attackers exploit secret-dependent data or control flows of TEE-shielded programs. Nevertheless, inputs of TEE-shielded NNs are unexploitable through micro-architectural side channels [27]: an NN is essentially a sequence of matrix computations, which exhibits a constant-time computation paradigm, i.e., data and control flows in NNs are *fixed* regardless of its inputs.

**A New Leakage.** Despite the general security belief, this paper shows that input data of TEE-shielded NNs can be leaked via ciphertext side channels, and the recovered inputs can be further leveraged to steal the NN's functionality. Ciphertext side channels denote a recently disclosed fine-grained information leakage that particularly exists in TEEs. It can leak memory write patterns (that are unexploitable through micro-architecture side channels) of TEE-shielded NNs to the malicious host. Since commercial TEEs widely adopt deterministic encryption, when secrets are stored at fixed physical locations in TEE's encrypted memory region (e.g., the VM save area, kernel data structures, user-land stacks, etc.), an identical ciphertext is generated for the same plaintext. Consequently, an adversary (e.g., hypervisors, the host OS) having read access to the ciphertext (either via software access [47] or via memory bus snooping [42]) can recover informative patterns in plaintext.

This paper for the first time recovers NN inputs from ciphertext side channels of TEE-shielded NNs. The recovery does *not* rely on the structure or model weights of the target NN. Moreover, unlike existing NN attacks that require full knowledge of the target NN's input domain (i.e., having data that cover all classes in the target NN's training data; see detailed clarifications in Sec. 4), we successfully recover NN inputs with only partial- or zero-knowledge of the input domain. By attacking TEE-based secure inference, user privacy in typical machine-learning-as-a-service (MLaaS) can be largely jeopardized, e.g., in cloud medical image diagnosis. Further, by attacking the TEE-protected training phase, private training inputs can be gathered to subsequently train

another NN to steal the target NN's functionality or boost adversarial example (AE) attacks.

**Technical Challenges.** Recent works have tentatively illustrated the threat of ciphertext side channels to semi-automatically recover cryptographic keys [47, 44]. NN inputs (e.g., images), which decide NN functionality or indicate user privacy, are fundamentally different from cryptographic keys. In fact, cryptographic key bits are either 0 or 1, and each bit often directly determines one ciphertext collision record — as a result, each record (i.e., collide or not) manifests a one-to-one mapping to each key bit. Nevertheless, during NN's computation, ciphertext collisions are induced by writing features (extracted from NN's inputs) to memory, and certain input information has been lost during the feature extraction stage. Moreover, a unit of the written feature typically corresponds to an image region, which has multiple pixels and each pixel's value is between 0 and 255. The large search space of NN inputs and the limited observation in ciphertext side channels make the input recovery inherently challenging.

This paper presents a generic and automated framework, CIPHERSTEAL, to address key challenges in recovering NN inputs from ciphertext side channels. CIPHERSTEAL recasts the input recovery as a two-step approach: transformation $\mathcal{T}$ and reconstruction $\mathcal{R}$. Given that certain information of NN input $x$ is lost in side-channel observation $c$, CIPHERSTEAL first transforms the remaining information in $c$ to $h = \mathcal{T}(c)$ where $h$ is presented in an aligned form with $x$ (e.g., $h$ is a partially recovered image). Then, with $h$ as the basis, CIPHERSTEAL reconstructs the lost information via $\mathcal{R}$. By optimizing the reconstruction with Bayesian theorem, CIPHERSTEAL eventually yields $\mathcal{R}(h) = x^* \approx x$.

**Results.** We employ CIPHERSTEAL to attack both the training and inference phases of TEE-shielded NNs. We consider two NN execution modes: the interpreter mode (running NNs in PyTorch or TensorFlow) and the executable mode (compiling NNs using compilers like TVM [11] or Glow [67]). CIPHERSTEAL is evaluated on 13 real-world and large-scale NNs of various structures (e.g., Vision Transformer [15]), training algorithms and tasks. We benchmark the recovery over five popular datasets of two representative input formats: image and video. In more than 100 different settings, we observe a consistently encouraging success rate (e.g., $> 90\%$ in half of the settings). We also demonstrate the high quality of inputs recovered from the TEE-shielded NN. By training another NN with these stolen training inputs, we obtain a surrogate NN exhibiting comparable performance (e.g., $> 98\%$ consistency) with the target NN. Also, using "white-box" adversarial examples generated over the surrogate NN, we largely enhance adversarial attacks towards the TEE-shielded NN (e.g., from 0 to a $30\%$ attack success rate). In sum, we make the following contributions:

- (**Concepts**) While NN inputs were believed free of micro-architecture side channels, we unveil the new attack surface of ciphertext side channels in TEE-shielded NNs, where malicious hosts can recover NN inputs to steal user privacy and NN functionality.
- (**Techniques**) We design a generic and automated framework, CIPHERSTEAL, to deliver practical NN input recovery with negligible knowledge of the target NN. We recast the recovery as a two-step approach and propose optimizations to improve its efficiency and accuracy.
- (**Attacks**) We constantly achieve promising input recovery on different NNs, input formats, datasets, runtimes, side-channel observations, and levels of attacker's pre-knowledge, etc. Our recovered inputs can be further leveraged to steal the target NN's functionality and boost adversarial example attacks.
- (**Findings**) We systematically study the leakage sites in different NN runtimes and summarize the lessons we learned. Our findings can provide insights for deploying NNs and designing NN interpreters/compilers.

**Artifact:** The code, data, and more examples of recovered images and videos, are provided at https://github.com/Yuanyuan-Yuan/CipherSteal [1].

## 2. Preliminaries

### 2.1. Neural Networks (NNs) and NN Runtimes

Unlike traditional programs (e.g., an RSA implementation) whose internal logics are hardcoded with human-written instructions, decision logics in NNs are formed by implicitly "learning" rules from manually annotated data. The execution of an NN is implemented as a sequence of matrix computation, with each computation operator propagating its output to the subsequent operator. Such computations are constant-time: the data and control flows of an NN's computation are fixed regardless of its inputs. NNs also feature a bidirectional computation: forward propagation (FP) and backward propagation (BP). The FP extracts features from inputs whereas the BP adjusts the NN's weights to tune the decision logics.

Modern NNs often run with the following two runtimes.

**Interpreter-Based.** Typical interpreter-based NN frameworks such as PyTorch [64] and TensorFlow [2] are software libraries that provide a programming interface for developers to build and run NNs. Their runtime system consists of two components: (1) the Python interfaces that parse and interpret the high-level NN into a set of matrix operations, and (2) third-party linear algebra libraries (e.g., OpenBLAS [97], MKL [77], and Eigen [23]) that implement such operations with efficient low-level binary code. NN frameworks usually allow users to run NNs in both forward and backward directions. For inference, an NN is executed in the forward mode. To train an NN, the computational graph of an NN is first constructed with intermediate results generated during the FP. Then, the computational graph is traversed in reverse order during BP, and the intermediate results are used to compute gradients for updating weights.

**Compiler-Based.** NNs are increasingly compiled into executables for better performance across different platforms. Two mature and actively maintained NN compilers,

TVM [11] and Glow [67], emit standalone executables that can run with minimal external dependencies. Both compilers follow similar design principles: they start from a high-level graph representation of the NN and progressively lower it to intermediate representations (IRs), allowing more platform-specific optimizations, and eventually emit machine code.

### 2.2. TEEs and Ciphertext Side Channel

TEEs aim to protect sensitive data by providing isolated environments, namely enclaves, for program execution. Modern TEE systems like AMD SEV, assisted by trusted hardware, encrypt each program's memory with a unique AES encryption key, thus preventing malicious hypervisors from accessing or modifying data used during execution. There is a growing trend of deploying NNs in TEEs [43, 39, 49, 30]. With TEEs, developers/users could protect NNs and preserve their data privacy while deploying/querying NNs in hardware devices controlled by untrusted hosts.

**Deterministic Encryption.** Encryption mode in TEEs is constrained by two factors. First, to support efficient random memory access that requires independently encrypted memory blocks, chaining mode (e.g., CBC mode) is inapplicable. Second, to encrypt large memory, the encryption mode should not support freshness (e.g., CTR mode) given the additional space required by counters. Therefore, AES encryption with *deterministic*, block-based mode is widely used by TEEs with large encrypted memory, such as AMD SEV [35], Intel TDX [32], Intel SGX on Ice Lake SP [32, 33], and ARM CCA [4]. For instance, the memory of TEE-shielded NN can be encrypted using 128-bit AES symmetric encryption, i.e., each aligned 16-byte memory block $m$ is encrypted independently. Although a tweak function $T(P_m)$ is used to calculate a mask value to be XORed with $m$ before encryption, $T(P_m)$ takes the physical address $P_m$ of $m$ as the only input. In short, the ciphertext of $m$ is calculated as $c = ENC(m \oplus T(P_m)) \oplus T(P_m)$, where the same $m$ stored in the same physical address $P_m$ is always encrypted into identical ciphertext [47].

**Ciphertext Side Channel.** Recent studies [47, 44] uncover ciphertext side channels to steal sensitive data (e.g., private keys). Ciphertext side-channel attacks exploit the deterministic encryption to infer the equality relations of consequent memory written values, which should be protected by TEEs. Suppose the ciphertext does not change after a memory write, the attacker easily infers that the written value equals the value previously stored in the target memory address. In contrast, a different ciphertext indicates a changed written value. With such capability, it is often possible to recover certain plaintext bits in the private keys [44, 14]. In terms of real-world exploitation, Li et al. propose the first ciphertext side-channel attack targeting AMD SEV-SNP [47] and exploit cryptographic libraries like RSA. While most discovered vulnerabilities living in AMD TEE [60, 84, 48, 46, 45] are promptly fixed, the ciphertext side channel, due to the design limitation of SEV-SNP, cannot be easily mitigated and is still exploitable by attackers [44].

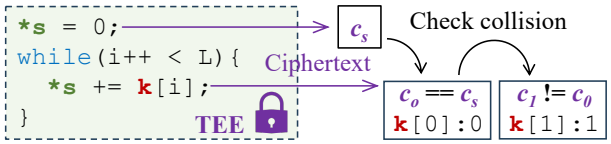



Figure 1. Ciphertext side-channel leakage of cryptographic keys. Since each key bit (i.e., $k[i]$) is either 0 or 1, attackers can directly infer the exact key bit values based on ciphertext collision information.

Fig. 1 illustrates a schematic view of the ciphertext side-channel attack towards the cryptographic key `k`, where a TEE-shielded program consecutively writes to the address `s`. The ciphertext $c_s$ is generated when the program initializes `s` with 0. Because the ciphertext $c_0$ (when writing `k[0]` to `s`) equals $c_s$, attackers can infer that `k[0]` is 0. Accordingly, since key bits are either 0 or 1, attackers also know that `k[1]` is 1 given that $c_1 \neq c_0$.

## 3. Motivations

This section elaborates on challenges and our insights on recovering NN inputs from ciphertext side channels. To ease the presentation, we use images as representative NN inputs; however, our techniques are generic and apply to other types of NN inputs like videos.

**Significance of Input Data in NN.** As introduced in Sec. 2.1, NNs essentially enable a data-driven programming paradigm. Depending on the phase of being fed into NNs, input data act as the following key roles:

*Intellectual Property:* During the training phase, an NN learns rules from training inputs to form its decision logics. Preparing training inputs requires considerable manual effort and human expertise. In that sense, training inputs denote the intellectual property of the NN owner. Since attackers can train equivalent NNs of the same functionality using the recovered training inputs, leaking training inputs also compromises the confidentiality of TEE-shielded NNs.

*User Privacy*: In modern MLaaS, users often query cloud NNs with their private data (e.g., medical images of certain diseases). Since TEEs are widely adopted to ensure secure inference on NNs hosted by untrusted service providers [56], leaking user inputs and prediction results from TEE-shielded NNs largely violates the privacy guarantee.

TEE-shielded NNs should incur ciphertext collisions due to the following reasons. First, the matrix computations in NNs are implemented as nested loops, which frequently write intermediate results to fixed memory addresses. Besides, each NN layer has a non-linear activation function; it often maps the intermediate results into a smaller region (e.g., `Sigmoid`) or discrete values (e.g., `ReLU`), largely increasing the chance of ciphertext collisions (see Sec. 7.1).

**Crypto. Keys vs. NN Inputs.** Nevertheless, recovering NN inputs from ciphertext side channels is fundamentally more challenging than cryptographic keys. As illustrated in Fig. 1, each ciphertext collision in cryptographic software is often induced by writing a key bit. Since key bits are either 0

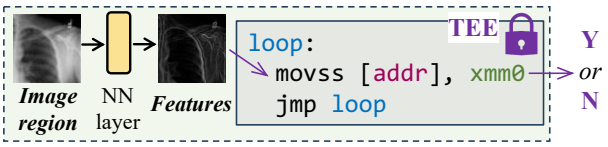



Figure 2. Ciphertext collisions are induced by writing intermediate features (often 32-bit in floating-point NNs) extracted from inputs into memory.

or 1, the collision information can be accurately mapped to each key bit. In contrast, as shown in Fig. 2, intermediate results written by an NN are features extracted from its input; these features are highly abstracted such that *certain input information is inevitably lost*. Moreover, a ciphertext collision record usually corresponds to features of an image region (e.g., in convolutional NNs). Given that multiple pixels exist in an image region and each pixel value ranges from 0 to 255, *the collision information is extremely limited*, let alone the lost information during feature extraction.

**Existing NN Input Recovery.** One may expect to adopt prior input recovery methods or recover NN inputs from other side channels. We clarify the infeasiblities below.

Yuan et al. have recovered NN inputs from cache side channels of the data pre-processing modules in MLaaS [94]. However, their context does not suffer from the information loss and limited observation issues: cache side channels in pre-processing modules are directly induced by image pixels, and the observable cache states are highly informative (e.g., a L1 cache may have 64 cache sets, resulting in 64 different states). Moreover, as clarified in Sec. 2.1, NNs do not have cache side channels due to their fixed data/control flows, and the leakage in pre-processing modules can be easily evaded by using already-processed inputs.

Several previous works tried to recover NN inputs from power side channels of NNs [59, 82]. They require binarized NNs (i.e., whose weights are either -1 or 1) and assume white-box access to target NNs. Importantly, they can only recover the coarse shape in images and only apply to black-and-white images of clean backgrounds. Nonetheless, modern NNs have floating-point weights and take diverse real-world images, making those recovery methods inapplicable. TEE-shielded NNs are also fully black-box to attackers.

**Observations & Insights.** This paper identifies the following insights to achieve NN input recovery from ciphertext side channels of TEE-shielded NNs. (1) Different from cryptographic keys where key bits are private, not all pixels in an image are secrets. E.g., failing to recover a few pixels in an image's background still indicates a successful input recovery, as long as the recovered inputs leak user privacy and enable stealing NN functionality. Moreover, (2) unlike cryptographic key bits that are independently sampled, pixel values are highly correlated. As pointed out by [94], pixel values have implicit constraints to form meaningful contents (e.g., randomly sampled pixels usually do not constitute meaningful images). Such constraints can be leveraged to reconstruct the lost information (see our solutions in Sec. 5).

Taking the above challenges and insights, we therefore do not aim to recover exact pixels in images, but recover image contents that are visually identical to the original ones. As evaluated in Sec. 7, our recovered inputs significantly leak user privacy, and enable effective functionality stealing and downstream attacks. Our techniques reconstruct the lost information even from limited observations, and are highly practical and effective under different levels of attacker's pre-knowledge, as will be introduced below in Sec. 4.

## 4. Threat Model and Positioning

**Attacker's Capability.** Aligned with existing works that attack/harden TEEs or shield NNs with TEEs [44, 14, 86], we follow the established threat model where adversaries are host OS or hypervisors: the adversary is assumed to have full system privilege on the machine and is also capable of performing physical attacks, including inferring address and content of every memory read via memory bus snooping [42], reading remnant data from the DRAM via cold boot attack [22], and accessing memory directly via DMA devices [73]. Nevertheless, attackers can only read the encrypted data and are unable to decrypt the ciphertext. Therefore, attackers cannot directly inspect the content of the deployed NN (e.g., reading its structure/weights). Also, when a normal user is using the NN, its inputs and predictions are *unknown* to attackers since they are encrypted.

**TEE & TEE-Shielded NNs.** Following existing works that deploy NNs in TEEs [57, 28, 70, 74], we assume that attackers can query the deployed NNs with their own data. However, to mitigate query-based NN cloning [61, 63, 75], TEE-shielded NNs only return the final prediction label to those who issue the queries; all intermediate results and the prediction confidence (i.e., the probability of the input belonging to each class) are not returned [98]. We also assume that the target NN is either already well-trained before deployed in TEEs or the NN can be trained/fine-tuned inside TEEs; both are common in practice.

The software stack inside the VM, including the OS, the NN runtime, and the NN itself, is secure and bug-free, such that the adversary cannot voluntarily alter its control flow or force it to leak secrets. The encryption algorithms of TEEs are also secure; adversaries cannot decrypt the ciphertext. We assume the hardware and microcode of the processor are up-to-date: known attacks against SEV, SEV-ES, and SEV-SNP [34] have all been fixed, leaving only generalized ciphertext side-channel leakage discovered in [47] for use.

**Attacker's Knowledge of the Target NN.** Our input recovery has much weaker requirements than previous attacks.

NN Structure and Weights. Previous NN attacks [9, 20] (see details in Sec. 4.1) often require full implementation details of NNs, including the structure and trained weights. In contrast, when recovering NN inputs, we do not require knowing the target NN's structure or weights.

Input Format & Input Domain. Aligned with existing NN attacks and side-channel attacks [75, 51, 9, 94, 17], we assume attackers can query the deployed NN with their own data and observe ciphertext side channels. We first define two terms related to NN inputs.

**Definition 1** (Input Domain)**.** *An NN's input domain denotes the set of its supported classes.*

Similar to C/C++ software that has input type restrictions (e.g., `int` vs. `float`), NNs also have constraints on their *valid* inputs, which are formed over the semantics level. These valid inputs constitute the *input domain* of the NN, as defined in Def. 1. For example, the input domain of an NN classifying digit one and zero consists of the class "`zero`" and "`one`." Similarly, for medical image diagnosis, the input domain is formed by all disease classes an NN can diagnose.

**Definition 2** (Input Format)**.** *Input Format denotes the union over input domains of different NNs serving the same usage.*

Beyond the input domain, we further define the *input format* in Def. 2, given that NNs having the same usage may have different input domains. For instance, although two NNs capable of diagnosing different chest diseases have different input domains, they both accept chest X-ray images as inputs. Here, chest X-ray image is their input format. Nevertheless, face photos and chest X-ray images are of different input formats, as they often correspond to NNs serving different usages. Note that our definition of input format is different from the conventional "format" in file extensions (e.g., `.PNG` vs. `.JPEG`). When processing inputs, NNs do not distinguish input of different extensions; raw input files are decoded first and then converted into floating-point matrices as NN inputs.

Existing NN attacks [75, 72, 8, 9] (see detailed explanations in Sec. 4.1) require having data covering the full input domain, which may not be always feasible. For example, to attack a disease-diagnosing NN, attackers may not have medical images covering all diseases supported by the NN. However, having data of the same input format is often feasible, e.g., it is practical to collect some benign medical images. The overly strong requirement on input domain limits the application scope of existing NN attacks. Our input recovery, in contrast, has a much weaker requirement that only assumes having data of the same input format. This way, we can recover data covering the target NN's input domain to enable previous attacks, rendering the severity of the leakage and the superiority of our techniques.

In some cases, an NN's input domain may be covered by public data (e.g., a classifier for cat and dog images). Therefore, to comprehensively assess attackers' (potential) capabilities and the attack surfaces of data leakage in TEE-shielded NNs, we evaluate our input recovery under different knowledge of the target NN's input domain: ① a zero-knowledge (**ZK**) attacker who does not have input from the target NN's input domain; ② a partial-knowledge (**PK**) attacker having inputs from a subset of the input domain; and ③ a full-knowledge (**FK**) attacker whose inputs cover the full input domain. Noting that having data from the same domain does *not* indicate having the same inputs. The attacker's data and target NN's inputs may be from the same class but are *always different* in our setting; otherwise, stealing the target NN's inputs is unnecessary.

## 4.1. Positioning w.r.t. Previous Attacks

CIPHERSTEAL, for the first time, recovers high-quality NN inputs from side channels of TEE-shielded NNs; it can complement existing side-channel attacks towards NNs and largely augments algorithmic attacks on TEE-shielded NNs.

**Completing Side-Channel Attacks Towards NNs.** Our input recovery is orthogonal to, and can complement existing side-channel attacks that recover NN structures [31, 91, 27, 90, 50, 17, 16]. Moreover, we argue that recovering NN inputs generally denotes more severe and new threats, because NN structures may be derived from public backbones. Importantly, despite that recovering an NN's weights is still hardly achievable,[1] attackers can leverage our recovered inputs to steal the target NN's functionality (a.k.a., obtaining different but equivalent weights).

TABLE 1. REQUIREMENTS OF PREVIOUS NN ATTACKS AND CIPHERSTEAL. ● AND ○ INDICATE NEEDED AND NOT NEEDED.

| Attack | Pre-Knowledge of the Target NN | | | | |
|---|---|---|---|---|---|
| | *Weights* | *Gradients* | *Prediction Confidence* | ***Input Domain*** | *Input Format* |
| Steal Functionality | ○ | ○ | ● | ● | ● |
| Fool Prediction | ● | ● | ○ | ● | ● |
| **CIPHERSTEAL** | ○ | ○ | ○ | ○ | ● |

**Augmenting Algorithmic NN Attacks.** As in Table 1, previous NN attacks can be divided into two categories.

Steal Functionality. Since recovering exact NN weights is challenging, query-based inference attacks [75, 51, 63] are proposed to steal NN functionality. In short, attackers query the target NN and let their own NN duplicate the prediction confidences (i.e., probabilities of the input belonging to all possible classes). However, such attacks are mitigated by TEE-shielded NNs which do not return prediction confidences. Moreover, the stealing is confined by the attacker's queried data: to steal the full functionality, attackers must have data covering the target NN's *full* input domain. For instance, it is infeasible to steal an NN's disease-diagnosing capability without images containing diseases. Also, to precisely steal the functionality, queried inputs are expected to be close to the target NN's training inputs.

Our input recovery, in the PK and ZK settings (as discussed in Sec. 4), can boost query-based attacks by recovering input in the full input domain. Further, we successfully recovered NN inputs during the training phase; with the recovered training inputs, CIPHERSTEAL facilitates more precise functionality stealing.

Fool Prediction. Previous works fool an NN's prediction by generating adversarial examples (AEs) [20, 54, 9]. The goal is to manipulate the target NN's prediction (e.g., let the NN always predict "benign" for all diseases) or downgrade the accuracy (i.e., deplete the NN's functionality). AEs are generated by slightly perturbing an NN's inputs which often rely on white-box access to the target NN (e.g., computing gradients). However, these white-box attacks are mitigated

1. Existing works steal NN weights by reading plaintext transmitted through PCI bus [100]; TEEs mitigate this via traffic encryption.

by TEE-shielded NNs whose weights are encrypted. Nevertheless, since an NN's vulnerabilities to AEs are mostly inherited from training data [20, 89, 92], our disclosed data leakage can enable these white-box attacks by generating AEs over a surrogate NN trained with CIPHERSTEAL's recovered training inputs (see results in Sec. 7.4).

## 5. Recovering NN Inputs

When deployed in TEEs, an NN's trained weights, inputs, outputs, and all intermediate computation results are encrypted. However, as introduced in Sec. 2, due to the deterministic encryption in TEEs, ciphertext encrypted at a fixed physical address is only decided by the plaintext stored in memory. That is, whenever new plaintext is written at a certain physical address, by observing whether the ciphertext changes, we can infer if the plaintext is different from the historical content stored at that address. This way, when the target NN is taking an input, we can generate a binary sequence (each binary value "0/1" flags whether the ciphertext changes), which *depends* on the plaintext input, for each physical address during one execution of the target NN. These binary sequences, after being concatenated, denote one ciphertext side-channel trace used by CIPHERSTEAL.

Similar to existing profiling-based side-channel attacks [55, 38, 25, 94, 17], CIPHERSTEAL also consists of an offline profiling and an online attack stage.

**Offline Stage.** Given a TEE-shielded NN $\mathcal{F}$, the attacker prepares some data $\mathbb{X}'$ and uses them to query $\mathcal{F}$ for profiling. When querying, the attacker simultaneously logs the ciphertext side-channel traces $\mathbb{C}'_{\mathcal{F}}$. Finally, with the collected $\mathbb{C}'_{\mathcal{F}}$ and the corresponding $\mathbb{X}'$, CIPHERSTEAL established a mapping $\mathcal{A} : \mathbb{C}'_{\mathcal{F}} \rightarrow \mathbb{X}'$ for input recovery. The $\mathcal{A}$ should generalize well to unknown NN inputs.

Different from previous query-based attacks [75, 51, 63], CIPHERSTEAL does not use the queried prediction; it only infers how ciphertext side channels change with inputs. Thus, querying the target NN with data out of its input domain is still feasible (e.g., under the ZK setting), despite that the predictions are no longer meaningful.

**Online Stage.** During the online attack, whenever the target NN takes an unknown input $x \in \mathbb{X}$ (either for inference or training), the attacker logs the ciphertext side-channel trace $c$ and uses CIPHERSTEAL to recover $x$ from $c$. Note that $\mathbb{X}' \cap \mathbb{X} = \emptyset$. $\mathbb{X}'$ and $\mathbb{X}$ have the same input format (defined in Def. 2) but may cover different input domains. CIPHERSTEAL is agnostic to the specific TEE platform or side-channel logging tools (e.g., `CipherLeak` [47]); as evaluated in Sec. 7.3, CIPHERSTEAL works well for different ciphertext side channels.

### 5.1. Problem Reformulation

**Information Loss and Decomposition of $\mathcal{A}$.** As mentioned in Sec. 3, ciphertext collisions in TEE-shielded NNs are due to writing intermediate results to memory, which are highly abstracted features of NN inputs. Extracted features may vary with the task an NN performs. For example, an NN may focus on outlines of faces for image segmentation but eyes for face recognition. Therefore, certain information in the input is inevitably lost during this feature extraction process. In addition, ciphertext side channels, which characterize if two consecutive memory writes to the same addresses have the same content, only provide an incomplete and coarse-grained observation of intermediate outputs.

Thus, it is infeasible to recover the *exact same* input from ciphertext side channels due to the information loss mentioned above. The key challenges that CIPHERSTEAL addresses are 1) extracting the information leaked in ciphertext side channels and 2) utilizing the extracted (incomplete) information to reconstruct the lost information. Accordingly, $\mathcal{A}$ is decomposed into two phases: a transformation $\mathcal{T}$ and a reconstruction $\mathcal{R}$. The transformation $\mathcal{T}$ transforms the form of the information retained in ciphertext side channel $c$ (which is generated when the target NN is executing with input $x$) to get $h = \mathcal{T}(c)$, where $h$ denotes the re-formed information from $c$ that is presented in an aligned form with NN inputs (e.g., a burred image whose details are lost; see Fig. 3). Then, the reconstruction $\mathcal{R}$ aims to reconstruct the lost information in $h$ to get $x^* = \mathcal{R}(h)$ which is close to $x$.

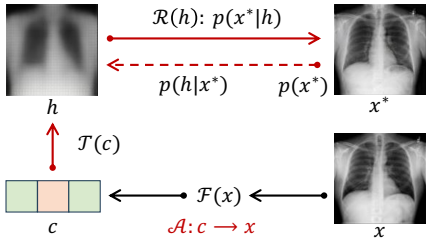



Figure 3. Decompose $\mathcal{A} : c \rightarrow x$ as transformation $\mathcal{T}$ and reconstruction $\mathcal{R}$. $\mathcal{R}$ is implemented via its inversion $p(h|x^*)$ and the realism term $p(x^*)$.

**Transformation $\mathcal{T}$.** Obtaining $\mathcal{T}$ is straightforward. Attackers can directly force $\mathcal{T}$ to output $x$ when $\mathcal{T}$ takes the corresponding $c$. By minimizing the distance between $\mathcal{T}(c)$ and $x$, $\mathcal{T}(c)$ is guided to represent in an aligned form with $x$. For example, if $x$ is a chest X-ray image, $\mathcal{T}(c)$ will also be a (blurred) chest X-ray image. However, $\mathcal{T}(c)$ often cannot generate the original $x$, because information of $x$ has been unavoidably lost in $c$. Instead, $\mathcal{T}(c)$ should output $h$ which properly re-forms the remaining information in $c$.

**Reconstruction $\mathcal{R}$.** Building $\mathcal{R}$ is inherently challenging since it requires "creating" information and refining $h = \mathcal{T}(c)$. While attackers may expect to infer the lost information, this process, if at all possible, should be data-intensive and not practical due to the transferability and generalizability issues. First, different NNs may have distinct preferences when extracting features from inputs; the $\mathcal{R}$ built for one target NN can hardly be transferred for another different target NN, and building one $\mathcal{R}$ for each target NN is costly. Second, inferring lost information may rely on the contents of each input and is input-dependent (e.g., the rule of inferring missed ears for cat images does not apply to inferring wheels in car images). Given that the target NN's inputs are unknown, the inferred $\mathcal{R}$ using the

attacker's own data may not generalize to them (especially PK or ZK settings discussed in Sec. 4).

**Revisit the Reconstruction: A Bayesian Perspective.** To alleviate the above hurdles, we view $\mathcal{R}$ from the Bayesian perspective. Given $h = \mathcal{T}(c)$ transformed from a ciphertext side-channel observation $c$, $\mathcal{R}$ aims to achieve the objective:

$$\arg\max_{x^*} p(x^*|h), \tag{1}$$

where $p(x^*|h)$, which infers $x^*$ based on $h$, is maximized when $x^*$ equals the NN input $x$ that produces $c$. According to Bayesian theorem, $p(x^*|h)$ in Eq. 1 can be re-formed as:

$$p(x^*|h) = \frac{p(h|x^*)p(x^*)}{p(h)} \tag{2}$$

Since $h$ is known, $p(h)$ is accordingly constant. As illustrated in Fig. 3, the objective in Eq. 1 is equivalent to

$$\arg\max_{x^*} p(h|x^*)p(x^*), \tag{3}$$

where $p(h|x^*)$, which infers $h$ based on $x^*$, is the inversion of the reconstruction $\mathcal{R}$. $p(x^*)$ denotes the realism of $x^*$, i.e., how likely $x^*$ is semantically meaningful (e.g., a valid medical image rather than random pixels).

Estimating $p(x^*)$ has been widely studied, and existing research can provide out-of-the-box solutions [19, 40]. Moreover, since $p(x^*)$ is only related to the attacker's data $\mathbb{X}'$, once estimated, it can be applied to any target NNs regardless of their tasks. This way, an attacker only needs to estimate $p(h|x^*)$ for each target NN.

Estimating $p(h|x^*)$ is inherently easier than estimating $p(x^*|h)$ as it "removes" information from $x^*$. Intuitively, if $x^*$ is an image, $p(h|x^*)$ aims to answer the question: "*What details should be removed from $x^*$ to mimic the feature extraction of $\mathcal{F}$?*" Thus, $p(h|x^*)$ manifests better generalizability: the pattern of information removal is mostly decided by the target NN's task (i.e., the feature extraction mentioned in the question) instead of a specific input. For example, if $x^*$ is a face photo and the target NN recognizes human identity, $p(h|x^*)$ simply ignores the face orientation but $p(x^*|h)$, which aims to reconstruct the orientation, depends on $x^*$ because face orientations vary in different $x^*$.

## 5.2. Implementation Considerations

This section introduces how different procedures formulated in Sec. 5.1 are implemented in CIPHERSTEAL.

**Implementation using Neural Networks.** In practice, we find that ciphertext side channels logged during one NN execution may be lengthy due to the matrix computations (i.e., nested loops) in NNs. Also, NN inputs are high-dimensional data like images and videos, which have semantically meaningful contents. Hence, we implement $\mathcal{T}$, $p(h|x^*)$, and $p(x^*)$ using neural networks given their capabilities of processing lengthy side-channel traces and understanding complex NN inputs [94, 71, 19]. Therefore, our offline stage trains $\mathcal{T}$, $p(h|x^*)$, and $p(x^*)$ using the attacker's own data $\mathbb{X}'$ and the corresponding ciphertext side channels. The online stage directly applies them to the target NN.

**Training Objective of $\mathcal{T}$.** The transformation is implemented as an NN $\mathcal{T}_\theta$, where $\theta$ denotes its weights. $\mathcal{T}_\theta$ is trained during the offline stage using the attacker's data $\mathbb{X}'$ and their derived ciphertext side channels $\mathbb{C}'_{\mathcal{F}}$. For each $x \in \mathbb{X}'$ and its corresponding $c \in \mathbb{C}'_{\mathcal{F}}$, the training of $\mathcal{T}_\theta(c)$ is guided with the objective:

$$\arg\min_{\theta} L(x, \mathcal{T}_\theta(c)) \tag{4}$$

where $L$ denotes the distance between $x$ and the transformed information in $c$. As mentioned in Sec. 5.1, with the objective of minimizing $L$, $\theta$ is optimized such that information in $c$ is re-formed as $h = \mathcal{T}_\theta(c)$, whose form is aligned to $x$. $L$ can be set as the mean squared error (MSE) or other advanced loss functions if applicable; see Appx. A.

**Time Series.** Besides recovering images, we also consider video as one representative sequential data of NN inputs. Compared with images, videos additionally include time-series information. A video can be viewed as a sequence of image frames where two adjacent frames are correlated. Recovering videos is conceptually similar to recovering sentences, which is a sequence of words, but is technically harder. To recover the video $x$ from its ciphertext side-channel trace $c$, the $\mathcal{T}_\theta$ is recurrently called. At each time step $i$ for the frame $f_i$ (i.e., an image) in the video $x$, $\mathcal{T}_\theta$ takes two inputs: 1) $c$ and 2) the recovered image frame $f_{i-1}$ at the previous step. This way, $\mathcal{T}$ recurrently outputs video frames which eventually constitute the video $x$. In particular, when reconstructing the first frame $f_0$, $\mathcal{T}$ takes $c$ and an empty variable $f_\emptyset = 0$ as inputs.

**Inverting Reconstruction: $p(h|x)$.** As shown in Sec. 5.1, we decompose the reconstruction as $p(h|x)$ and $p(x)$. The $p(x|h)$, which is the inversion of the reconstruction, is implemented with an NN $\mathcal{I}_\omega$ of weights $\omega$. $\mathcal{I}_\omega$ is simultaneously trained with $\mathcal{T}_\theta$, but from a different direction: $\mathcal{I}_\omega$ takes $x \in \mathbb{X}'$ as inputs and is expected to output $\mathcal{T}_\theta(c)$, where $c$ is the corresponding ciphertext side channel of $x$. Accordingly, the training objective in Eq. 4 is extended as:

$$\arg\min_{\theta,\omega} L(x, \mathcal{T}_\theta(c)) + L(\mathcal{I}_\omega(x), \mathcal{T}_\theta(c)) \tag{5}$$

**Ensuring the Realism: $p(x)$.** Estimating $p(x)$ has been widely studied via generative models (e.g., GANs [19], Diffusion models [40]). Given a generative model $G$, $p(x)$ can be estimated by establishing a mapping between different $x$ and points sampled from a continuous latent space $\mathbb{Z}$. Since $\mathbb{Z}$ is continuous, infinite and diverse (new) $x$ can be represented as the results of interpolation and exploitation in $\mathbb{Z}$ [19]. Therefore, by randomly sampling $z$ from $\mathbb{Z}$, vivid and new samples can be generated by $G(z)$. Note that $p(x)$ is estimated using the attacker's own data $\mathbb{X}'$.

**High-Level Attack Pipeline.** Once $\mathcal{T}_\theta$, $\mathcal{I}_\omega$, and $G$ are well-trained, they can be employed to recover NN inputs during the online attack. Suppose a ciphertext side-channel trace $c$ is logged when the target NN is taking an unknown input $x$, the attacker first transforms $c$ to $h = \mathcal{T}_\theta(c)$. To reconstruct the lost information in $h$, the attacker needs to optimize the following objective:

$$\arg\min_{z^*} L(h, \mathcal{I} \circ G(z^*)) \tag{6}$$

which can be achieved via optimization such as stochastic gradient descent (SGD) [37]; see detailed discussions in Appx. A. The objective in Eq. 6 updates $z^*$ to generate different $G(z^*)$. This process is equivalent to searching for a valid input (of rich details) that can lead to the same $h$. Since partial information in $x$ is retained in $h$ and pixels in images (or video frames) are highly correlated [94], $h$ can guide $G(z^*)$ to be close to $x$.

## 6. Evaluation Setup and Configuration

**NNs, Datasets, Tasks, and Input Domain.** Table 2 lists our evaluated NNs and datasets. These NNs are representative and diverse in structures. We follow their standard configurations and the trained NNs are provided in our artifact [1] for reproducibility. We consider both classification and regression tasks. The datasets are also diverse, including images and videos that cover representative real-life scenarios. For different datasets and NNs, we construct different experiments where attackers have varied knowledge of the input domain, as highlighted in Table 2. The experiment IDs Ⓐ-Ⓜ in Table 2 are consistently used in the rest of this paper to ease finding the setups.

For FK and PK cases, to ensure that NN owners, users, and attackers do *not* have overlapped data, we use half of the data in the original training split as NN owners' data to train the target NN. The remaining data in the training split are used as the attacker's query data; accordingly, data in the original test split are treated as user inputs. For ZK cases, attacker's data and owner's/user's data are from different classes, ensuring that they do not overlap. Noteworthy, for human action (video) classification (Ⓗ), despite that attacker's data cover all actions (i.e., FK), the human identities of attacker's videos and owner's/user's videos do *not* overlap.

**CIPHERSTEAL Configuration.** Following [94], both $\mathcal{T}$ and $p(h|x)$ in Fig. 3 are implemented as auto-encoders. Since $\mathcal{T}$ transforms side-channel traces to images (or video frames), its auto-encoder consists of a side-channel trace encoder and an image decoder. $p(h|x)$, in contrast, takes images as inputs and outputs their coarse-grained versions. Hence, the auto-encoder of $p(h|x)$ has an image encoder and an image decoder. We reuse the image/side-channel encoders and image decoders from [94]; more configuration suggestions for different scenarios are provided in Appx. A.

For each attacked NN, CIPHERSTEAL requires training the two corresponding auto-encoders. Our training is conducted on one NVIDIA GeForce RTX 2080 Ti GPU, which takes ∼15 minutes for MNIST-trained NNs (Ⓐ-Ⓒ, Ⓘ-Ⓚ), ∼5 hours for ImageNet-trained NNs (Ⓖ), ∼3 hours for each of the remaining target NNs. The offline training only needs to be conducted once. During the online attack, whenever the target NN is taking an input $x$, attackers can log a ciphertext side-channel trace $c$ and feed it to CIPHERSTEAL. CIPHERSTEAL will subsequently recover $x$, which takes less than one second for ∼100 NN inputs.

**Runtimes.** In line with Sec. 2.1, NNs running in both interpreter-based frameworks and executable forms are evaluated. We consider two most popular frameworks, PyTorch (version 2.0) and TensorFlow (version 2.13). We also consider the two most popular NN compilers, TVM (version 0.12) and Glow (the commit 2dcde3f).

**Execution Phases.** We consider the following three different execution phases of NNs.

***Inference***: Feature Extraction. The inference phase of an NN only has forward propagation (FP). Therefore, we study if ciphertext side-channel leakage exists in the computations of common NN operations such as convolution, pooling, and layout transforms. We evaluate the inference phase of both interpreter-based frameworks and executables given their different computational paradigms.

***Training***: Gradient Computation. The training phase consists of an FP followed by a backward propagation (BP), which performs gradient computation. Currently, only interpretation-based frameworks support BP. Thus, we study the additional leakage of the gradient computations in BP of PyTorch and TensorFlow.

***Fine-Tuning***: Updated Weights. NNs can be fine-tuned (i.e., slightly trained) after being deployed in TEEs due to security hardening (e.g., NN slicing [99, 98]). As a result, the weights of the NN will be updated, and the patterns of ciphertext side-channel collisions (which are jointly decided by NN weights and inputs) will be accordingly changed. Since attackers may be unaware of the fine-tuning and the query (during offline profiling) is conducted over the initially deployed NN, we study whether the input recovery applies if the target NN updates its weights.

**Evaluation Metrics.** As discussed in Sec. 3, NN inputs play distinct roles (e.g., user's privacy, or NN owner's intellectual properties) in different execution phases. Therefore, we jointly use three metrics to evaluate the recovered inputs from different aspects.

Prediction Consistency (**PC**). Since input information related to the prediction is critical (e.g., the disease in a medical image), we evaluate if a recovered input can result in the same prediction as the ground truth input when fed into the target NN. Because an NN's output is chosen from a predefined set of predictions, *the baseline of classification tasks is* $1/\#Classes$. For regression tasks (Ⓓ, Ⓖ, Ⓜ in Table 2), since NN outputs are vectors of continuous values, we check if the recovered input has a smaller Cosine distance with its ground truth input than one randomly selected input. Thus, *the baseline of PC for regression tasks is* $50\%$.

Training Consistency (**TC**). Training inputs further decide the functionality of the target NN. Thus, we also evaluate, when a new surrogate NN is trained using the recovered *training inputs*, whether it has consistent functionality with the target NN. Following TEE's protection, we annotate recovered training inputs using the predicted labels (w/o confidences) when querying the target NN with them. The PC is measured as the percentage of *user test inputs* for which the newly trained surrogate NN has the same prediction as the target NN. *The baseline of TC is the same as PC:* $1/\#Classes$ *for classification and* $50\%$ *for regression.*

TABLE 2. STUDIED NNS AND DATASETS FOR VARIOUS TASKS UNDER DIFFERENT ATTACKER'S KNOWLEDGE. MARKERS Ⓐ-Ⓚ ARE CONSISTENTLY USED IN THE REST OF THIS PAPER TO EASE FINDING THE SETUPS.

| Exp. ID | NN | Dataset | Task | Usage | Input Domain | | Logging Tool |
|---|---|---|---|---|---|---|---|
| | | | | | *Owner/User* | *Attacker* | |
| Ⓐ | LeNet [41] | MNIST [13] | Classification | Digit recognition | Digit 0-9 | **FK**: Digit 0-9 | Emulated SEV-Step* |
| Ⓑ | | | | | Digit 0-9 | **PK**: Digit 0-4 | |
| Ⓒ | | | | | Digit 0-4 | **ZK**: Digit 5-9 | |
| Ⓓ | FaceNet [68] | CelebA [52] | Regression | Face recognition | Face Photos | **ZK**: Face photos of other identities | |
| Ⓔ | MobileNet [29] | Chest X-ray [81] | Classification | Disease diagnosis | 14 Diseases | **ZK**: Benign | |
| Ⓕ | | | | | | **PK**: 7 Diseases | |
| Ⓖ | ResNet [24] | ImageNet [12] | Regression | Image compression | 100 classes in ImageNet | **FK**: 100 classes in ImageNet | |
| Ⓗ | ConvLSTM [71] | KTH Actions [69] | Classification | Video understanding | 6 Actions | **FK**: 6 Actions† | |
| Ⓘ | ViT [15] | MNIST [13] | Classification | Digit recognition | Digit 0-9 | **FK**: Digit 0-9 | CipherLeak |
| Ⓙ | | | | | Digit 0-9 | **PK**: Digit 0-4 | |
| Ⓚ | | | | | Digit 0-4 | **ZK**: Digit 5-9 | |
| Ⓛ | ViT | Chest X-ray [81] | Classification | Disease diagnosis | 14 Diseases | **PK**: 7 Diseases | |
| Ⓜ | ViT | CelebA [52] | Regression | Face recognition | Face Photos | **ZK**: Face photos of other identities | |

* We emulate SEV-Step using Intel Pin [53] due to the incompatibility issue; see details in Sec. 6.
† In the setup of Ⓗ, the human IDs of attacker's videos and owner's/user's videos do *not* overlap.

Similarity (**SIM**). Besides the distinct roles of inputs in NNs under different contexts, we also conduct a similarity evaluation exclusively on each recovered input. We use the LPIPS [96] as the similarity metric given its high expressiveness of capturing image semantics. For each recovered input, we use the ground truth NN input and $M-1$ randomly selected (different) NN inputs to construct a candidate set. We then compute all candidates' similarities with the recovered input. To assess the recovered information beyond the input's label (as already evaluated via PC), all candidates are from the *same class* of the ground truth input. Results are reported as the percentage of recovered inputs whose ground truth inputs are among the top-$K$ similar candidates. To reduce randomness, we repeat the similarity evaluation five times and report the average results in Sec. 7. We set $K = 1$ and $M = 100$. Thus, *the baseline of SIM is* $1/100 = 1\%$.

**Logging Side Channels.** Two mature logging tools have been proposed by previous works to collect ciphertext side channels: CipherLeak [47] and SEV-Step [44, 88]. In short, CipherLeak only logs ciphertexts of *last writes* in a memory page, and checks page-wise ciphertext collisions. SEV-Step, in contrast, is finer-grained to track each instruction's memory write and record ciphertext collisions between instructions. Therefore, we use SEV-Step to log ciphertext side channels from (classical) moderately sized NNs, and employ CipherLeak for larger NNs (i.e., ViT, as indicated in Table 2) where using SEV-Step is too costly.

We follow the default configurations of CipherLeak. When configuring SEV-Step in our experiments, we found that it is based on Linux kernel 5.14 which is outdated and incompatible with the latest SEV-SNP firmware (version 1.55). This poses a conflict since the latest firmware is required to launch a SEV-SNP guest VM to run the target NNs. Porting SEV-Step to newer kernel versions requires considerable manual effort and is impractical on our end. We have contacted developers of SEV-Step for help; by the time of submission, the upgrade is still in progress.

Thus, we mimic SEV-Step by using Intel Pin [53], an instrumentation tool, to record each instruction's memory write in TEE-shielded NNs. Our experience on SEV-Step shows that its outputs are "clean" and precise to track each memory write, and our logging results using SEV-Step and Pin are identical on programs currently supported by SEV-Step. However, since SEV-Step is timer-based, it may neglect memory writes occurred during a time interval. We therefore also benchmark our input recovery towards this impact. Overall, our input recovery is promising even when 63 of every 64 memory writes are unrecorded; see Sec. 7.3.

# 7. Evaluation

We consider four research questions (RQs). **RQ1** studies the leakage sites and attack surfaces under various settings. **RQ2** assesses recovering complex and diverse NN inputs. **RQ3** evaluates how our input recovery is affected by different ciphertext side channels. **RQ4** demonstrates NN attacks (mentioned in Sec. 4.1) enhanced by our results.

## 7.1. RQ1: Leakage Sites and Attack Surface

We first analyze how the vulnerable functions distribute among different NN executables and interpreters. We then show the recovery results w.r.t. different settings. To ease the setup of controlled experiments, we focus on MNIST cases in this section and mainly discuss the prediction and training consistency. Similarity results and other input data and formats are given in Sec. 7.2.

TABLE 3. VULNERABLE MODULES AND KEYWORDS OF SAMPLE FUNCTIONS. MORE CASES ARE PROVIDED IN OUR ARTIFACT [1].

| Runtime | Module | Example Keywords |
|---|---|---|
| PyTorch | Conv/Matrix/Kernel<br>Auto-grad | _conv_, _bmm_<br>_autograd_ |
| TensorFlow | GEMM | _sgemm_<br>gemm_ |
| TVM | Layout Transformation<br>Each layer | layout_transform<br>fused_ |
| Glow | Each layer | conv2d_f_3_<br>matmul_f_21_ |

#### 7.1.1 Vulnerable Modules

Due to the constant-time computations of NNs (i.e., the accessed memory addresses of NN computations are fixed), localizing vulnerable modules that have ciphertext side-channel leakage in NNs is straightforward. Similar to the trace differentiation in existing side-channel detection works [83, 85], we can simply check if the ciphertext collisions of each address change with inputs.

```
; xmm2: 0, [addr]: zero-initialized
vaddss xmm3, xmm3, dword ptr [bias]
vmaxss xmm3, xmm3, xmm2
vmovss dword ptr [addr], xmm3
```

(a) ReLU operation (Glow).

```
; [addr2]: zero-initialized
movss  xmm0, dword ptr [addr1]
movss  dword ptr [addr2], xmm0
```

(b) Layout transformation (TVM).

```
; [bias]: bias
; [addr1]: Conv result
; [addr2]: zero-initialized
; xmm4: 0
movss xmm1, dword ptr [bias]
movss xmm3, dword ptr [addr1]
addss xmm3, xmm1
maxss xmm3, xmm4
movss dword ptr [addr2], xmm3
```

(c) Fused Conv & ReLU (TVM).

Figure 4. Code patterns in Glow and TVM executables.

**Executables.** Table 3 summarizes our localized vulnerable modules. In compiled NN executables, each NN layer is implemented as a standalone function. For executables generated by Glow, we find that almost all layers have ciphertext side-channel leakages. We analyze all leakage-incurring instructions in executables and attribute these leakages to the compiled activation and pooling functions. Activation and pooling are non-linear functions converting continuous values into a smaller range or discrete ones. Indeed, an NN's intelligence is based on its non-linearity. Fig. 4(a) shows an example of $ReLU(x) = \max(0, x)$ in Glow-emitted executables, which writes the results into the output memory region (i.e., `addr` in Fig. 4(a)). However, since the output region is zero-initialized, when a negative value is fed into ReLU, the output 0 written to the output region will trigger an observable memory ciphertext collision. Note that such leakage instructions are repeatedly called in loops of matrix computations; thus even a single leakage point can reveal a large amount of input information.

Differently, while similar operations (activation functions, pooling) exist in executables compiled by TVM, we do not observe pervasive leakage sites as in Glow executables. Note that multiple operators in the target NN may be optimized as one function by NN compilers (e.g., via operator fusion [11]). As shown in Fig. 4(c), a ReLU is fused into its preceding Conv layer. In that case, the results of Conv operations (which have fewer zeros) are stored in the output memory region; thus, collisions between ReLU's output zeros and the zero-initialized memory are largely reduced. Nevertheless, we find that the layout transformation modules of TVM contribute to many ciphertext collisions. The zeros from ReLU still exist in the consequent computation. As shown in Fig. 4(b), whenever these zeros are moved to a zero-initialized memory region (i.e., `addr2`), which happens frequently due to memory layout optimizations, ciphertext collisions still occur.

**Interpreter Frameworks.** PyTorch and TensorFlow have leakages in similar modules. In particular, for PyTorch, most ciphertext collisions occur in the convolution, matrix, and kernel computation modules (e.g., `conv`-, `kernel`-, and `bmm`-related functions). During BP, the auto-grad modules also have ciphertext side-channel leakage. Similar modules in TensorFlow, which are implemented via GEMM (i.e., general matrix multiply) functions, also induce leakages.

Due to the just-in-time (JIT) compilation paradigm of PyTorch and TensorFlow, NN layers/modules are actively constructed via primitive operators when the NN is running. We notice that primitive operators, such as `sum`, `copy`, etc., induce considerable ciphertext collisions, leading to pervasive leakage sites. After investigating the patterns of ciphertext collisions, we find that the root cause of leakages in interpreters is similar to that in executables: the non-linear functions map floating-point values to a smaller range or fixed ones, greatly increasing the possibility of collisions.

Overall, the convolution modules are popular in classical NNs. Matrix multiplication functions like `bmm` are building blocks of fully connected layers and self-attention modules in Transformer-based NNs. Similarly, kernel computation is extensively used in max-pooling, average-pooling modules, etc. These modules exist in nearly all modern NNs, indicating the *severity* and the *pervasiveness* of the attack surface.

#### 7.1.2 Attack Surfaces under Various Scenarios

**Setup.** This section presents input recovery towards our localized modules in Sec. 7.1.1 and studies how the results are affected by different attack scenarios. For the FP of PyTorch and TensorFlow, since most NN layers share the same primitive operators, we do not observe notable differences due to the choice of the target primitive operator. For Pytorch BP, we choose auto-grad functions to study BP's specific leakage. In TensorFlow, because both FP and BP adopt GEMM functions, to specifically study BP's leakage, we choose GEMM functions that are not involved in FP.

In executables generated by TVM and Glow, NN layers are implemented as standalone functions. For Glow executables, this section reports results on functions derived from deeper layers. Since layers at different depths contribute differently to the NN's predictions [95, 18], we further evaluate how the depths of layers affect the input recovery in Sec. 7.3. For TVM executables, we focus on layout transformation functions which primarily induce the leakage.

Table 4 presents our results. Below, we analyze them from several aspects and summarize eight key findings.

**Knowledge of Input Domain.** As in Table 4, the attack results can be improved with more knowledge of the input domain. Note that for ZK cases, the target NN performs 5-class classification, whereas the NN classifies 10 classes in PK and FK cases. Although the PC and TC results of ZK are comparable to PK in some settings, PK cases should have better results. Overall, our attack achieves encouraging results even in PK and ZK settings. To steal the target NN's functionality, previous attacks (even when the prediction confidences are available) are "upper-bounded" by the knowledge of input domain. Specifically, attackers only steal the target NN's *partial* functionality on their *known* input domain. E.g., if attackers only have digit 1, their own NN trained with queried predictions can only predict 1. That is, query-based attacks at most achieve 50% and 0% TC in

TABLE 4. RECOVERY RESULTS FOR STUDYING ATTACK SURFACES. PC AND TC DENOTE PREDICTION AND TRAINING CONSISTENCY.

| | Runtime | Training Input | | | | User Test Input | | Runtime | Training Input | | | | User Test Input | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | *Forward* | | *Backward* | | *Forward* | *Fine-Tuning* | | *Forward* | | *Backward* | | *Forward* | *Fine-Tuning* |
| | | PC | TC | PC | TC | PC | PC | | PC | TC | PC | TC | PC | PC |
| Ⓐ **FK** | | 97.18% | 98.13% | N/A | N/A | 97.33% | 97.18% | | 82.08% | 97.82% | 40.37% | 68.07% | 66.83% | 70.67% |
| Ⓑ **PK** | **TVM** | 90.90% | 98.17% | N/A | N/A | 90.62% | 90.71% | **PyTorch** | 70.57% | 96.98% | 30.35% | 68.07% | 60.03% | 60.15% |
| Ⓒ **ZK** | | 96.81% | 99.65% | N/A | N/A | 97.60% | 97.66% | | 75.14% | 99.55% | 39.61% | 76.70% | 63.31% | 68.52% |
| Ⓐ **FK** | | 98.20% | 98.28% | N/A | N/A | 97.88% | 97.87% | | 70.45% | 89.61% | 55.14% | 97.26% | 60.45% | 60.40% |
| Ⓑ **PK** | **Glow** | 95.65% | 98.26% | N/A | N/A | 95.33% | 95.33% | **TensorFlow** | 61.03% | 87.87% | 45.36% | 96.66% | 51.22% | 51.41% |
| Ⓒ **ZK** | | 97.96% | 99.68% | N/A | N/A | 98.11% | 98.40% | | 69.50% | 81.86% | 49.33% | 96.71% | 52.23% | 59.01% |

our PK and ZK settings, respectively. In contrast, ❶ *our recovered inputs can steal the NN functionality with more than 90% TC even in the ZK cases*.

**Fine-Tuning.** We also study if the input recovery still applies after weights of the target NN have been fine-tuned (as mentioned in Sec. 6). Compared with the initially deployed NNs (which are queried during the offline preparation), these fine-tuned NNs have 79.6% (on average) weights changed.

As shown in the 8th and last columns of Table 4, our input recovery is not affected in all cases. Note that the internal decision logics of the fine-tuned NN remain unchanged despite that weight values are updated (otherwise, the fine-tuning failed). Thus, we infer that the patterns, which exist in the ciphertext side channels to facilitate recovering NN inputs, primarily depend on the decision logic. This is reasonable since an NN's decision logics, to some extent, decide what information to be extracted from inputs. However, we find that our input recovery is inapplicable to new NNs that are different from the NN we queried offline. Therefore, we conclude that ❷ *updating NN weights (due to hardening) does not affect our input recovery unless the target NN is replaced with a new different one.*

**Interpreter vs. Executable.** In all settings, the input recovery has better results on NN executables than NNs running in interpreters. After investigating the logged ciphertext side channels, we find more ciphertext collisions occur in NN executables. In fact, NN executables are highly optimized; their memory accesses are more compact, which increases the chance of ciphertext collisions. Also, NN interpreters have non-determinism in some functions (e.g., the OpenMP multi-threading [76] in PyTorch), such that some ciphertext collisions are due to randomness, which negatively impacts the recovery. Thus, we infer that ❸ *optimizations in NN compilers have introduced substantially new ciphertext side-channel leakage of NN inputs.*

**Training vs. Test Inputs.** For PyTorch and TensorFlow, the recovered training inputs have higher PC than user test inputs. Note that existing works find that NNs can memorize some training inputs [72, 8], we suspect that such memorization eases recovering training inputs. This gap may not be obvious in cases of higher leakage (e.g., in executables), but is enlarged when the leaked information is slimmer (e.g., in interpreters). To conclude, ❹ *compared with test inputs, training inputs are more likely to be leaked via ciphertext side channels.*

**Functionality vs. Input.** The recovered training inputs usually have higher TC than PC in Table 4. We interpret the result from two aspects. First, our technique ensures the realism of recovered inputs by modeling $p(x)$; see Sec. 5. Despite that some recovered inputs have inconsistent predictions with ground truth inputs, they are still valid and meaningful NN inputs. This highlights the merit of our design considerations. Second, PC may have more restrictive requirements: to retain the prediction, full details of the input should be recovered. Nevertheless, even if some details are missed in the recovered input (thus making the NN change its prediction), the recovered input is still useful as one training sample because it reflects the target NN's decision logics on the partially recovered image details. Therefore, we conclude that ❺ *NN functionality has more severe leakage via ciphertext side channels.*

**FP (Forward) vs. BP (Backward).** For both PC and TC, inputs recovered from the FP phase have better results. Intuitively, FP primarily extracts features from inputs, whereas BP computes gradients which reflect the NN's decision logics. Thus, the leakage in FP is more informative to recover NN inputs. We conclude that ❻ *NN input leakage during FP is more informative than BP.*

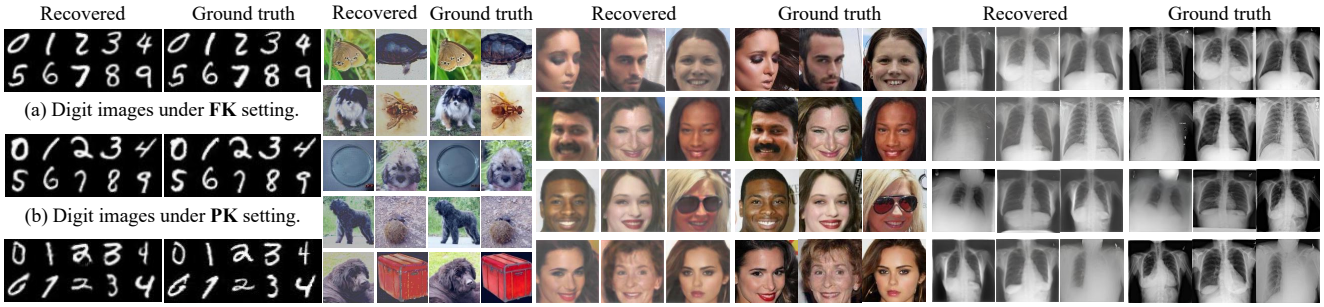
**PyTorch vs. TensorFlow.** As in Table 4, attack results over PyTorch and TensorFlow exhibit varying trends on FP and BP. We discuss them below.

Forward: Memory Usage. Compared with TensorFlow, inputs recovered from PyTorch's FP have better PC and TC. By cross-comparing their FP, our experiments show that PyTorch consumes more memory. Accordingly, more memory writes occur, increasing the chance of ciphertext collisions. In sum, ❼ *PyTorch has more leakage than TensorFlow during FP due to its higher memory usage.*

Backward: Static vs. Dynamic Computational Graph. Different from the FP, inputs recovered during TensorFlow's BP have higher PC and TC. Also, by cross-comparing the gaps between FP's and BP's results in PyTorch and TensorFlow, PyTorch cases have larger gaps. Note that PyTorch maintains a dynamic computational graph on the fly, and it deletes the graph when back-propagating gradients to save computing resources. Thus, with the graph and historical computation results gradually deleted, ciphertext becomes less likely to trigger collision (and leakage) in BP. In contrast, TensorFlow maintains a static computational graph during runtime which is fixed after initialization. Thus, the results indicate that ❽ *the static computational graph in TensorFlow induces more leakage during the BP.*

## 7.2. RQ2: Complex and Diverse Inputs

This section evaluates our input recovery for more complex NN inputs. We focus on the FP (forward) since it is




(a) Digit images under **FK** setting. (b) Digit images under **PK** setting. (c) Digit images under **ZK** setting. (d) ImageNet cases. (e) Face photo cases. (f) Chest X-ray image cases.

Figure 5. Examples of recovered images and ground truth. For the ZK case in Fig. 5(c), the target NN only processes digits 0-4. Recovered videos and more image examples are provided in [1]. Failure cases (whose frequency is very low) and their root causes are discussed in Appx. B.

involved during both inference and training. We consider Glow and PyTorch as the representative NN compilers and interpreters in this section.

**Qualitative Examples.** Fig. 5 presents examples of recovered input images and their ground truth. The recovered digits are almost identical to the ground truth, and CIPHERSTEAL is able to recover digits 0-4 with only digits 5-9 under the PK and ZK cases. In cases of face photos, these people's identities are accurately recovered, despite that attackers do not have face photos of the same identities. In addition, facial attributes, such as gender, skin color, eye status, expressions, orientations, etc., are also highly consistent between recovered face photos and ground truth. The recovered chest X-ray images also match the size, number, and position of lung lobes and ribs in the ground truth inputs. Recovered videos are displayed on our artifact website [1]. Overall, the recovered videos are smooth, and each frame matches that in the ground truth videos. The person (which is unknown) and the performed action in each video are also precisely recovered.

TABLE 5. ATTACK RESULTS OF OTHER INPUT FORMATS. PC AND TC DENOTE PREDICTION CONSISTENCY AND TRAINING CONSISTENCY.

| | Input | | Training Input PC | Training Input TC | User Test Input PC |
|---|---|---|---|---|---|
| **Glow** | Ⓓ Face photos | **ZK** | 98.6% | 98.8% | 98.3% |
| | Ⓔ Chest X-ray images | **ZK** | 78.2% | 90.4% | 78.1% |
| | Ⓕ Chest X-ray images | **PK** | 92.5% | 94.3% | 90.1% |
| | Ⓖ 100-class images | **FK** | 95.4% | 96.7% | 94.7% |
| | Ⓗ Human action videos | **FK** | 50.8% | 79.0% | 43.4% |
| **PyTorch** | Ⓓ Face photos | **ZK** | 88.0% | 96.8% | 82.4% |
| | Ⓔ Chest X-ray images | **ZK** | 68.7% | 82.1% | 68.8% |
| | Ⓕ Chest X-ray images | **PK** | 78.9% | 90.4% | 77.1% |
| | Ⓖ 100-class images | **FK** | 89.3% | 91.6% | 86.5% |
| | Ⓗ Human action videos | **FK** | 34.1% | 51.1% | 33.9% |

TABLE 6. SIMILARITY EVALUATION (SIM). BASELINE IS 1%.

| Input | | Training Input *Glow* | Training Input *PyTorch* | User Test Input *Glow* | User Test Input *PyTorch* |
|---|---|---|---|---|---|
| Ⓐ MNIST | **FK** | 98.5% | 85.3% | 98.3% | 69.3% |
| Ⓑ MNIST | **PK** | 95.9% | 76.7% | 96.5% | 63.8% |
| Ⓒ MNIST | **ZK** | 98.2% | 81.8% | 97.9% | 65.0% |
| Ⓓ Face photos | **ZK** | 88.2% | 70.7% | 87.3% | 70.1% |
| Ⓔ Chest X-ray images | **ZK** | 67.7% | 58.8% | 66.5% | 55.4% |
| Ⓕ Chest X-ray images | **PK** | 85.6% | 66.4% | 84.5% | 63.9% |
| Ⓖ 100-class images | **FK** | 92.1% | 82.4% | 92.7% | 81.9% |
| Ⓗ Human action videos | **FK** | 51.1% | 38.1% | 51.5% | 37.2% |

**Quantitative Analysis.** As reported in Table 5, we achieve encouraging prediction consistency (PC) and training consistency (TC) results for diverse and more complex input formats, indicating that our recovered inputs are capable of stealing the predictions and functionalities of the target NNs. Table 6 presents results of the similarity evaluation (SIM). Note that our similarity evaluations are conducted among inputs of the same class, these high results (compared with the 1% baseline) demonstrate that rich details in each image are successfully recovered. Previous techniques only apply to black-and-white images such as digit images in MNIST, and the recovered digits lose details [82]. CIPHERSTEAL, in contrast, is *not* limited to specific input formats: our technique and the promising results highlight the severity of ciphertext side-channel leakage in TEE-shielded NNs.

Overall, results of chest X-ray images and face photos reflect the *precision* of CIPHERSTEAL and *fine-grained details* leaked: IDs/disease information can be recovered from ciphertext side channels when the attacker does not know the face ID or has only benign chest X-ray images. Besides, recovering images of 100 classes benchmarks the *scalability* of CIPHERSTEAL, i.e., simultaneously handling all of them. Note that previous attacks towards data processing modules in NNs can only handle images of one class each time [94]. Moreover, recovering videos demonstrates the *generalizability* of CIPHERSTEAL since videos are sequential data and are processed by NNs having special recurrent structures; as clarified in Sec. 5.2, recovering videos is conceptually similar to, but technically harder than recovering text.

## 7.3. RQ3: Side-Channel Observations

This section evaluates how different ciphertext side-channel observations and noises affect our input recovery.

**Logging with `CipherLeak`.** Besides our emulated `SEV-Step`, we also evaluate CIPHERSTEAL by logging ciphertext collisions with `CipherLeak`. Results are given in Table 7. Since `CipherLeak` only records the last writes at each memory page and checks ciphertext collisions when different pages are accessed, it is not surprising that the results are relatively lower than using `SEV-Step`. Nevertheless, the results are still promising, and remain largely higher than the baselines.

TABLE 7. INPUT RECOVERY RESULTS OF USING CIPHERLEAK.

| | Input | Training Input | | | User Test Input | |
|---|---|---|---|---|---|---|
| | | PC | TC | SIM | PC | SIM |
| Glow | Ⓘ MNIST **FK** | 67.2% | 85.4% | 38.8% | 67.1% | 37.9% |
| | Ⓙ MNIST **PK** | 58.7% | 80.2% | 37.6% | 57.8% | 37.7% |
| | Ⓚ MNIST **ZK** | 66.1% | 84.4% | 38.5% | 64.3% | 38.1% |
| | Ⓛ Chest **PK** | 52.6% | 77.3% | 26.7% | 52.2% | 26.3% |
| | Ⓜ Face **ZK** | 78.5% | 86.8% | 50.5% | 77.7% | 51.2% |
| PyTorch | Ⓘ MNIST **FK** | 66.7% | 82.8% | 37.6% | 65.6% | 38.0% |
| | Ⓙ MNIST **PK** | 53.4% | 79.7% | 36.1% | 53.2% | 35.8% |
| | Ⓚ MNIST **ZK** | 64.3% | 81.5% | 38.2% | 63.2% | 37.5% |
| | Ⓛ Chest **PK** | 50.3% | 76.7% | 25.9% | 50.6% | 26.1% |
| | Ⓜ Face **ZK** | 73.1% | 84.9% | 49.4% | 72.8% | 49.3% |

We note that the results over interpreters and executables are very close. In Sec. 7.1.2, we reveal that optimizations in NN executables enlarge the leakage due to more compact memory accesses. However, the enlarged leakage is not significant when using CipherLeak. Note that optimizations in NN executables primarily fuse adjacent operators, such that their memory writes are more likely to the access same addresses. However, the accessed memory addresses of adjacent operators are presumably located on the same page, whose collisions are likely not logged as CipherLeak only records the last writes that occurred on the same page.

TABLE 8. EVALUATIONS OF DIFFERENT NN LAYERS AND GRANULARITIES OF SEV-STEP.

| Layer | Input | Training Input | | | | User Test Input | |
|---|---|---|---|---|---|---|---|
| | | PC | | TC | | PC | |
| | | $T=16$ | $T=64$ | $T=16$ | $T=64$ | $T=16$ | $T=64$ |
| Shallow | Ⓐ MNIST **FK** | 92.4% | 83.4% | 98.1% | 98.1% | 92.9% | 83.9% |
| | Ⓑ MNIST **PK** | 73.9% | 58.6% | 97.9% | 97.8% | 74.9% | 58.4% |
| | Ⓒ MNIST **ZK** | 87.4% | 71.3% | 99.4% | 99.6% | 87.7% | 73.1% |
| Deep | Ⓐ MNIST **FK** | 90.7% | 59.1% | 98.2% | 98.2% | 90.9% | 60.3% |
| | Ⓑ MNIST **PK** | 69.1% | 46.7% | 97.5% | 97.2% | 68.9% | 46.5% |
| | Ⓒ MNIST **ZK** | 85.0% | 47.3% | 99.7% | 98.9% | 86.1% | 46.7% |

**Different Granularity of SEV-Step.** While ciphertext side channels can be logged in a single-step granularity via SEV-Step, multiple instructions could be executed during the given APIC timer interval [88], such that some memory writes are periodically missed. To benchmark such impacts on CIPHERSTEAL, we consider emulating SEV-Step with different granularity $T$, i.e., only recording every $T$-th memory write. We consider $T = 16$ and $64$.

As in Table 8, even when $T = 64$ (i.e., 63 records are missed among every $64$ memory writes), the PC is still promising, e.g., over $70\%$ in the ZK cases of shallow layer, whose baseline is $20\%$. Differently, the TC is almost not affected by the granularity. This observation is consistent with our finding ❺ in Sec. 7.1.2: despite leading to different predictions, recovered inputs are still valid and useful training samples since we ensure their realism.

**Picking Leakage Sites.** Different from interpreters where different layers are constructed via primitive operators, Glow executables implement each NN layer as one standalone function. Considering that different layers (i.e., shallow vs. deep) often contribute differently to NN predictions [95, 6], we study how the input recovery is affected by the choice of layers. As shown in Table 8, better input recovery is achieved on shallow layers. This is reasonable because NNs propagate inputs from shallow to deep layers, with more abstracted features gradually extracted. Hence, shallow layers should retain more information about NN inputs.

## 7.4. RQ4: Enabled Attacks

Training consistency (TC) results presented in previous sections show that our input recovery can largely enhance attacks that steal NN functionality. This section evaluates how our results bring white-box adversarial examples (AEs) to fool NN predictions. As mentioned in Sec. 4.1, we train a surrogate NN using the recovered training inputs. We then generate AEs over the surrogate NN and use these AEs to manipulate or downgrade the target NN's predictions.

**Setup & Baseline.** The manipulation attack forces the victim NNs to always predict "0" or "benign" for digit recognition (Ⓐ-Ⓒ) and chest X-ray image diagnosis (Ⓔ), respectively. Our attack is compared with one state-of-the-art black-box adversarial attack, square attacks [3], which is directly applied to target NNs. We use PGD [5] to generate white-box AEs on surrogate NNs. We configure both algorithms to query its attacked NN (the target NN or surrogate NN) at most 20 times and the adversarial perturbations are bounded with the maximum $\ell_\infty$-norm of 0.3. In each setting, we generate 2,000 AEs for the attack.

TABLE 9. EVALUATION OF ENABLED ATTACKS.

| Input | Downgrade | | Manipulation | |
|---|---|---|---|---|
| | Black-Box | Ours | Black-Box | Ours |
| Ⓐ MNIST **FK** | 0 | 37.8% | 0 | 32.0% |
| Ⓑ MNIST **PK** | 0 | 33.7% | 0 | 29.7% |
| Ⓒ MNIST **ZK** | 0 | 21.7% | 0 | 22.2% |
| Ⓔ Chest **ZK** | 18.85% | 97.5% | 0 | 12.0% |

**Results.** Table 9 presents attack success rates. Black-box AEs are less effective and never succeed in 7 over 8 settings. Our attack, by leveraging the adversarial vulnerabilities inherited from training data, can successfully downgrade and manipulate the target NN's prediction with around 30% success rate for digit recognition. Downgrading chest X-ray diagnosis (the only successful case of black-box AEs; our attack has $97\%$ success rate) is significantly easier than enforcing it to predict "benign" ($12\%$ success rate by our attack). To explain, the diagnosis relies on fine-grained details in X-ray images, and adding adversarial perturbations can easily break those details to mislead the prediction. Nevertheless, manipulating the prediction to benign requires hiding all disease-related details which is more challenging.

Our attack is also more efficient; it only takes half of the time spent in the black-box attack (63s vs. 122s). Training a surrogate NN takes about $\sim$170s for MNIST and $\sim$20 min for X-ray images; however, this is a one-time effort.

# 8. Discussions and Future Works

**Countermeasures.** CIPHERSTEAL, for the first time, enables recovering high-quality inputs from TEE-shielded NNs. The recovered inputs can be further used to steal NN functionality and generate more effective adversarial examples. As a result, the adoption of CIPHERSTEAL may raise potential privacy and security concerns, especially in the context of TEEs. Recent work in repairing ciphertext side channels [86] may not be directly applicable to our attack, given that [86] primarily fixes vulnerable cryptographic

code patterns that do not exist in NNs. [44] advocates to achieve non-deterministic ciphertexts in TEEs via VMSA randomization, and [87] proposes to obfuscate the memory access patterns or ciphertexts via oblivious RAMs. However, the randomization/obfuscation may bring considerable performance overheads.

Having that stated, we believe that there are several promising directions to mitigate our attack. First, from the algorithmic perspective, we can specifically design randomization/obfuscation schemes for NNs following the principles of [44, 87]. Unlike traditional software, the executions of NNs are essentially matrix computations, which are resilient to non-adversarial noise in intermediate computations. Since the collisions are induced by writing the same intermediate results to memory, negligible noise can be injected into NN's intermediate outputs on the fly to randomize the generated ciphertext. Such randomization should yield a low cost, and the key obstacle is finding the "sweet spot" between NN accuracy and the noise level. Existing profiling-based obfuscation techniques [26] may be a good starting point.

In addition, inspired by our findings in Sec. 7.1, the following software- and system-level countermeasures are also highly feasible. For runtime implementation, instead of directly writing to memory, using registers to hold as many intermediate values as possible should avoid many ciphertext collisions. For optimizations in NN computation, motivated by the TVM case in Fig. 4, we expect to leverage operator fusion to reduce memory writes for neighboring operators. Moreover, as we observe many collisions between written zeros and zero-initialized memory, we advocate initializing unused memory regions with non-deterministic values to reduce the frequency of ciphertext collisions.

**Detecting Ciphertext Side-Channel Leakage.** As introduced in Sec. 7.1.1, due to the constant-time computations of NNs, detecting NN modules vulnerable to ciphertext side channels is straightforward via trace differentiation. However, this approach is inaccurate for programs of varied data/control flows, where different ciphertext side-channel traces may be unaligned. The size of recent NNs is growing exponentially and optimizations are accordingly proposed for NN computations. For instance, dynamic control flows have been implemented in recent large NNs [66]. NN compilers have also supported compiling dynamic computation graphs [10]. Therefore, it is high time to propose scalable and more advanced ciphertext side-channel detection techniques for NNs.

CipherH [14] detects ciphertext side-channel leakages in cryptographic software and is able to handle varied control and data flows. However, it is inapplicable to NNs due to its heavy program analysis techniques. Considering that side-channel detection has been extensively studied for cache side channels [79, 78], we foresee related techniques (e.g., trace alignment [83], lengthy trace conversion [93], quantitative analysis [85, 93], etc.) can be adapted for ciphertext side channels in NNs; the primary challenge should be the engineering efforts on dealing with the Python interface in some NN runtimes (e.g., PyTorch).

**Exploiting New Side-Channel Leakage.** While this paper has illustrated the threat that untrusted hosts pose to user privacy, it is also vital to investigate the threat from other adversaries like malicious users. Since a malicious user is unprivileged, the attack surface mostly lies in the exploitation of user-space micro-architectural side channels. For instance, Yuan et al. [94] have shown that, due to the shared CPU caches among users in MLaaS, a malicious user can leverage cache side channels in data processing libraries (e.g., `libjpeg`, `ffmpeg`) to steal other user's inputs. However, this threat can be mitigated by locally processing inputs without using online data processing libraries.

Despite that NN's computations are mostly constant-time (user inputs provided to NNs are therefore free of mainstream micro-architectural side channels), similar to TEE's implementation issue (i.e., the deterministic encryption) attacked in this paper, their practical implementations in runtime environments may bring new exploitation opportunities. For instance, the distributed data-parallel computations of modern NNs often implement data-dependent optimizations, which may lead to input-dependent data flows that are observable to malicious users. Exploiting new side channels in NNs w.r.t. different adversaries should be a promising research direction.

**Extension of CIPHERSTEAL.** As discussed in Sec. 4.1, CIPHERSTEAL primarily focuses on the novel usage of ciphertext side channels; the two key procedures in CIPHERSTEAL, information transformation and reconstruction, are independent of specific characteristics of ciphertext side channels and should be applicable to input recovery in other contexts. Overall, extending CIPHERSTEAL to other NN side channels that leak inputs (if identified in the future) is straightforward: attackers only need to feed the collected side-channel traces to CIPHERSTEAL for input recovery.

## 9. Conclusion

This paper demonstrates that ciphertext side channels can be exploited to recover input data from TEE-shielded NNs. We propose CIPHERSTEAL to address the information loss and limited observation issues, and recover high-quality inputs under varied knowledge of the victim. Comprehensive evaluations show the superiority of CIPHERSTEAL in recovering NN inputs and augmenting downstream attacks.

## Acknowledgments

We thank the anonymous reviewers and our shepherd 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] Research Artifact. https://github.com/Yuanyuan-Yuan/CipherSteal.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[3] M. Andriushchenko, F. Croce, N. Flammarion, and M. Hein. Square attack: a query-efficient black-box adversarial attack via random search. In *European conference on computer vision*, pages 484–501. Springer, 2020.

[4] ARM. Arm confidential compute architecture software stack. https://developer.arm.com/documentation/den0127/latest, 2023.

[5] A. Athalye, N. Carlini, and D. Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *International conference on machine learning*, pages 274–283. PMLR, 2018.

[6] D. Bau, J.-Y. Zhu, H. Strobelt, A. Lapedriza, B. Zhou, and A. Torralba. Understanding the role of individual units in a deep neural network. *Proceedings of the National Academy of Sciences*, 117(48):30071–30078, 2020.

[7] A. Brock, J. Donahue, and K. Simonyan. Large scale gan training for high fidelity natural image synthesis. In *International Conference on Learning Representations*, 2018.

[8] N. Carlini, S. Chien, M. Nasr, S. Song, A. Terzis, and F. Tramer. Membership inference attacks from first principles. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1897–1914. IEEE, 2022.

[9] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, pages 39–57. Ieee, 2017.

[10] S. Chen, S. Wei, C. Liu, and W. Yang. Dycl: Dynamic neural network compilation via program rewriting and graph optimization. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 614–626, 2023.

[11] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX OSDI*, pages 578–594, 2018.

[12] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[13] L. Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE signal processing magazine*, 29(6):141–142, 2012.

[14] S. Deng, M. Li, Y. Tang, S. Wang, S. Yan, and Y. Zhang. CipherH: Automated detection of ciphertext side-channel vulnerabilities in cryptographic implementations. In *USENIX Security*, 2023.

[15] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2020.

[16] A. Dubey, R. Cammarota, and A. Aysu. Maskednet: The first hardware inference engine aiming power side-channel protection. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 197–208. IEEE, 2020.

[17] Y. Gao, H. Qiu, Z. Zhang, B. Wang, H. Ma, A. Abuadbba, M. Xue, A. Fu, and S. Nepal. Deeptheft: Stealing dnn model architectures through power side channel. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024.

[18] R. Geirhos, P. Rubisch, C. Michaelis, M. Bethge, F. A. Wichmann, and W. Brendel. Imagenet-trained cnns are biased towards texture; increasing shape bias improves accuracy and robustness. In *International Conference on Learning Representations*, 2018.

[19] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

[20] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[21] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[22] M. Gruhn and T. Müller. On the practicability of cold boot attacks. In *2013 International Conference on Availability, Reliability and Security*, pages 390–397. IEEE, 2013.

[23] G. Guennebaud, B. Jacob, et al. Eigen. *URl: http://eigen. tuxfamily. org*, 3(1), 2010.

[24] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[25] B. Hettwer, T. Horn, S. Gehrer, and T. Güneysu. Encoding power traces as images for efficient side-channel analysis. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 46–56. IEEE, 2020.

[26] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11. IEEE, 2013.

[27] S. Hong, M. Davinroy, Y. Kaya, S. N. Locke, I. Rackow, K. Kulda, D. Dachman-Soled, and T. Dumitraş. Security analysis of deep neural networks operating in the presence of cache side-channel attacks. *arXiv preprint arXiv:1810.03487*, 2018.

[28] J. Hou, H. Liu, Y. Liu, Y. Wang, P. Wan, and X. Li. Model protection: Real-time privacy-preserving inference service for model privacy at the edge. *IEEE Trans. Dependable Secur. Comput.*, 19(6):4270–4284, 2022.

[29] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[30] B. Hu, Y. Wang, J. Cheng, T. Zhao, Y. Xie, X. Guo, and Y. Chen. Secure and efficient mobile dnn using trusted execution environments. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, pages 274–285, 2023.

[31] X. Hu, L. Liang, S. Li, L. Deng, P. Zuo, Y. Ji, X. Xie, Y. Ding, C. Liu, T. Sherwood, et al. Deepsniffer: A dnn model extraction framework based on learning architectural hints. In *ASPLOS*, pages 385–399, 2020.

[32] Intel. 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, 2023.

[33] S. Johnson, R. Makaram, A. Santoni, and V. Scarlata. Supporting intel sgx on multi-socket platforms. *Intel Corp*, 2021.

[34] D. Kaplan. Upcoming x86 technologies for malicious hypervisor protection. https://static.sched.com/hosted_files/lsseu2019/65/SEV-SNP%20Slides%20Nov%201%202019.pdf, 2020.

[35] D. Kaplan, J. Powell, and T. Woller. Amd memory encryption. *White paper*, page 13, 2016.

[36] T. Karras, S. Laine, and T. Aila. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 4401–4410, 2019.

[37] N. Ketkar and N. Ketkar. Stochastic gradient descent. *Deep learning with Python: A hands-on introduction*, pages 113–132, 2017.

[38] J. Kim, S. Picek, A. Heuser, S. Bhasin, and A. Hanjalic. Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 148–179, 2019.

[39] K. Kim, C. H. Kim, J. J. Rhee, X. Yu, H. Chen, D. J. Tian, and B. Lee. Vessels: efficient and scalable deep learning prediction on trusted processors. In R. Fonseca, C. Delimitrou, and B. C. Ooi, editors, *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, pages 462–476. ACM, 2020.

[40] D. Kingma, T. Salimans, B. Poole, and J. Ho. Variational diffusion models. *Advances in neural information processing systems*, 34:21696–21707, 2021.

[41] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[42] D. Lee, D. Jung, I. T. Fang, C.-C. Tsai, and R. A. Popa. An off-chip attack on hardware enclaves via the memory bus. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[43] T. Lee, Z. Lin, S. Pushp, C. Li, Y. Liu, Y. Lee, F. Xu, C. Xu, L. Zhang, and J. Song. Occlumency: Privacy-preserving remote deep-learning inference using SGX. In S. A. Brewster, G. Fitzpatrick, A. L. Cox, and V. Kostakos, editors, *The 25th Annual International Conference on Mobile Computing and Networking, MobiCom 2019, Los Cabos, Mexico, October 21-25, 2019*, pages 46:1–46:17. ACM, 2019.

[44] M. Li, L. Wilke, J. Wichelmann, T. Eisenbarth, R. Teodorescu, and Y. Zhang. A systematic look at ciphertext side channels on amd sev-snp. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 337–351. IEEE, 2022.

[45] M. Li, Y. Zhang, and Z. Lin. CROSSLINE: Breaking "Security-by-Crash" based Memory Isolation in AMD SEV. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2937–2950, 2021.

[46] M. Li, Y. Zhang, Z. Lin, and Y. Solihin. Exploiting unprotected i/o operations in amd's secure encrypted virtualization. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1257–1272, 2019.

[47] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng. Cipherleaks: Breaking constant-time cryptography on amd sev via the ciphertext side channel. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 717–732, 2021.

[48] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng. TLB Poisoning Attacks on AMD Secure Encrypted Virtualization. In *Annual Computer Security Applications Conference*, 2021.

[49] Y. Li, D. Zeng, L. Gu, Q. Chen, S. Guo, A. Y. Zomaya, and M. Guo. Lasagna: Accelerating secure deep learning inference in sgx-enabled edge cloud. In C. Curino, G. Koutrika, and R. Netravali, editors, *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, pages 533–545. ACM, 2021.

[50] Y. Liu and A. Srivastava. Ganred: Gan-based reverse engineering of dnns via cache side-channel. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 41–52, 2020.

[51] Y. Liu, R. Wen, X. He, A. Salem, Z. Zhang, M. Backes, E. De Cristofaro, M. Fritz, and Y. Zhang. Ml-doctor: Holistic risk assessment of inference attacks against machine learning models. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4525–4542, 2022.

[52] Z. Liu, P. Luo, X. Wang, and X. Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

[53] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[54] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. In *ICLR*, 2018.

[55] H. Maghrebi, T. Portigliatti, and E. Prouff. Breaking cryptographic implementations using deep learning techniques. In *Security, Privacy, and Applied Cryptography Engineering: 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings 6*, pages 3–26. Springer, 2016.

[56] Z. Á. Mann, C. Weinert, D. Chabal, and J. W. Bos. Towards practical secure neural network inference: the journey so far and the road ahead. *ACM Computing Surveys*, 56(5):1–37, 2023.

[57] F. Mo, A. S. Shamsabadi, K. Katevas, S. Demetriou, I. Leontiadis, A. Cavallaro, and H. Haddadi. Darknetz: towards model privacy at the edge using trusted execution environments. In E. de Lara, I. Mohomed, J. Nieh, and E. M. Belding, editors, *MobiSys '20: The 18th Annual International Conference on Mobile Systems, Applications, and Services, Toronto, Ontario, Canada, June 15-19, 2020*, pages 161–174. ACM, 2020.

[58] A. Moghimi, G. Irazoqui, and T. Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In *Cryptographic Hardware and Embedded Systems–CHES 2017: 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 69–90. Springer, 2017.

[59] S. Moini, S. Tian, D. Holcomb, J. Szefer, and R. Tessier. Power side-channel attacks on bnn accelerators in remote fpgas. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 11(2):357–370, 2021.

[60] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel. Severed: Subverting amd's virtual machine encryption. In *Proceedings of the 11th European Workshop on Systems Security*, pages 1–6, 2018.

[61] T. Orekondy, B. Schiele, and M. Fritz. Knockoff nets: Stealing functionality of black-box models. In *CVPR*, pages 4954–4963, 2019.

[62] X. Pan, X. Zhan, B. Dai, D. Lin, C. C. Loy, and P. Luo. Exploiting deep generative prior for versatile image restoration and manipulation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(11):7474–7489, 2021.

[63] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. Practical black-box attacks against machine learning. In *ACM Asia CCS*, pages 506–519, 2017.

[64] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

[65] Y. Poirier-Ginter and J.-F. Lalonde. Robust unsupervised stylegan image restoration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 22292–22301, 2023.

[66] C. Riquelme, J. Puigcerver, B. Mustafa, M. Neumann, R. Jenatton, A. Susano Pinto, D. Keysers, and N. Houlsby. Scaling vision with sparse mixture of experts. *Advances in Neural Information Processing Systems*, 34:8583–8595, 2021.

[67] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint*, 2018.

[68] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[69] C. Schuldt, I. Laptev, and B. Caputo. Recognizing human actions: a local svm approach. In *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.*, volume 3, pages 32–36. IEEE, 2004.

[70] T. Shen, J. Qi, J. Jiang, X. Wang, S. Wen, X. Chen, S. Zhao, S. Wang, L. Chen, X. Luo, F. Zhang, and H. Cui. SOTER: guarding black-box inference for general neural networks at the edge. In J. Schindler and N. Zilberman, editors, *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 723–738. USENIX Association, 2022.

[71] X. Shi, Z. Chen, H. Wang, D.-Y. Yeung, W.-K. Wong, and W.-c. Woo. Convolutional lstm network: A machine learning approach for precipitation nowcasting. *Advances in neural information processing systems*, 28, 2015.

[72] R. Shokri, M. Stronati, C. Song, and V. Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE, 2017.

[73] P. Stewin and I. Bystrov. Understanding dma malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–41. Springer, 2012.

[74] Z. Sun, R. Sun, L. Lu, and S. Jha. Shadownet: A secure and efficient system for on-device model inference. *CoRR*, abs/2011.05905, 2020.

[75] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Stealing machine learning models via prediction apis. In *USENIX Sec'16*.

[76] P. tutorials. OpenMP in PyTorch. https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html#utilize-openmp, 2023.

[77] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang. Intel math kernel library. *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures*, pages 167–188, 2014.

[78] S. Wang, Y. Bao, X. Liu, P. Wang, D. Zhang, and D. Wu. Identifying Cache-Based side channels through Secret-Augmented abstract interpretation. In *28th USENIX security symposium (USENIX security 19)*, pages 657–674, 2019.

[79] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu. CacheD: Identifying cache-based timing channels in production software. In *USENIX Security*, 2017.

[80] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2421–2434, 2017.

[81] X. Wang, Y. Peng, L. Lu, Z. Lu, M. Bagheri, and R. M. Summers. 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*, pages 2097–2106, 2017.

[82] L. Wei, B. Luo, Y. Li, Y. Liu, and Q. Xu. I know what you see: Power side-channel attack on convolutional neural network accelerators. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 393–406, 2018.

[83] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl. Data–differential address trace analysis: Finding address-based side-channels in binaries. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 603–620, 2018.

[84] J. Werner, J. Mason, M. Antonakakis, M. Polychronakis, and F. Monrose. The severest of them all: Inference attacks against secure virtual enclaves. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 73–85, 2019.

[85] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar. Microwalk: A framework for finding side channels in binaries. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 161–173, 2018.

[86] J. Wichelmann, A. Pätschke, L. Wilke, and T. Eisenbarth. Cipherfix: Mitigating ciphertext {Side-Channel} attacks in software. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6789–6806, 2023.

[87] J. Wichelmann, A. Rabich, A. Pätschke, and T. Eisenbarth. Obelix: Mitigating side-channels through dynamic obfuscation. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 189–189. IEEE Computer Society, 2024.

[88] L. Wilke, J. Wichelmann, A. Rabich, and T. Eisenbarth. SEV Step. https://github.com/sev-step/sev-step, 2023.

[89] C. Xiao, B. Li, J.-Y. Zhu, W. He, M. Liu, and D. Song. Generating adversarial examples with adversarial networks. *arXiv preprint arXiv:1801.02610*, 2018.

[90] M. Yan, C. W. Fletcher, and J. Torrellas. Cache telepathy: Leveraging shared resource attacks to learn dnn architectures. In *USENIX Sec'20*.

[91] D. Yang, P. J. Nair, and M. Lis. Huffduff: Stealing pruned dnns from sparse accelerators. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 385–399, 2023.

[92] X. Yuan, P. He, Q. Zhu, and X. Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE transactions on neural networks and learning systems*, 30(9):2805–2824, 2019.

[93] Y. Yuan, Z. Liu, and S. Wang. CacheQL: Quantifying and localizing cache {Side-Channel} vulnerabilities in production software. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 2009–2026, 2023.

[94] Y. Yuan, Q. Pang, and S. Wang. Automated side channel analysis of media software with manifold learning. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4419–4436, 2022.

[95] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13*, pages 818–833. Springer, 2014.

[96] R. Zhang, P. Isola, A. A. Efros, E. Shechtman, and O. Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *CVPR*, 2018.

[97] X. Zhang, M. Kroeker, W. Saar, Q. Wang, and Z. Chothia. Openblas. http://www.openblas.net/, 2023.

[98] Z. Zhang, C. Gong, Y. Cai, Y. Yuan, B. Liu, D. Li, Y. Guo, and X. Chen. No privacy left outside: On the (in-) security of tee-shielded dnn partition for on-device ml. In *IEEE S&P*, 2024.

[99] Z. Zhang, Y. Li, Y. Guo, X. Chen, and Y. Liu. Dynamic slicing for deep neural networks. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 838–850, 2020.

[100] Y. Zhu, Y. Cheng, H. Zhou, and Y. Lu. Hermes attack: Steal dnn models with lossless inference accuracy. In *USENIX Security*, 2021.

# Appendix A. Configuration Suggestions of CIPHERSTEAL

As introduced in Sec. 5.2, the offline profiling of CIPHERSTEAL trains several neural networks. This section provides suggestions for configuring CIPHERSTEAL.

**Auto-Encoders.** We implement both the transformation $\mathcal{T}$ and the reconstruction's inversion $\mathcal{I}$ as auto-encoders. Since $\mathcal{T}$ maps side-channel traces to images (or video frames), the auto-encoder of $\mathcal{T}$ consists of a side-channel trace encoder and an image decoder. Similarly, the auto-encoder of $\mathcal{I}$, whose inputs and outputs are images, consists of an image encoder and an image decoder. Following the common practice [94], both image encoders and decoders are implemented as convolutional neural networks. The side-channel trace encoder is adopted from [94], which explores the sparsity of side channels to support handling lengthy (e.g., having millions of records) side-channel traces.

**Loss Function $L$.** The simplest loss function of images is the mean square error (MSE), which works sufficiently well for digit images in MNIST. However, since other complex images (i.e., those evaluated in Sec. 7.2) have richer details than digit images, we compute the loss in a multi-resolution manner following [65]. In short, the multi-resolution computation first down-samples images into multiple resolutions (e.g., down-sampling $128 \times 128$ images into various sizes of $64 \times 64$, $32 \times 32$, and $16 \times 16$) and computes the loss value for each resolution. The final loss value is the average of loss values at different resolutions.

Although the multi-resolution loss computation is more accurate, it multiplies the computing cost. Therefore, we suggest users of CIPHERSTEAL first adopt the default computation of loss function; if the results are not satisfactory, users can switch to the multi-resolution computation.

**Transformation and Reconstruction.** We find that not all datasets evaluated in Sec. 7 require reconstructing the lost information. For the MNIST dataset and digit recognition NNs, we notice that full details of the digits are leaked in ciphertext side channels, and only using the transformation is sufficient to accurately recover different digits (see examples in Fig. 4). However, for all the remaining dataset

(e.g., face photos, chest X-ray images, 100-class images in ImageNet), the reconstruction is necessary; otherwise, the recovered images will lose considerable details and are usually unrecognizable. Overall, digit images in MNIST have clean backgrounds and are distinguishable with only black and white pixels; it is reasonable that ciphertext side channels can reflect their details.

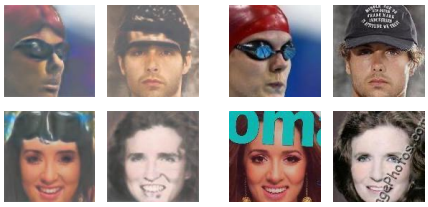
**Generative Model $G$.** To date, typical generative models include Generative Adversarial Networks (GANs), Variational Auto-Encoders (VAEs), and Diffusion models. We have explored using representative GANs, VAEs, and Diffusion models, and found that GANs are the most effective in reconstructing the lost information.

Despite that Diffusion models have shown better image generation capabilities than GANs recently, the fundamental difference between Diffusion models and GANs is that each GAN has an accompanying discriminator, which helps to check whether the generated images are valid. Since optimizing the objective in Eq. 6 actively modifies the generator's inputs, the discriminator in GANs can supervise the optimization process and limit the modification to ensure generating valid images. Diffusion models and VAEs, in some cases, may generate invalid images, guiding the follow-up optimization iterations to incorrect directions.

Regarding the choice of GANs, we suggest using StyleGAN [36] for domain-specific cases like face photos and chest X-ray images, and optimizing Eq. 6 can be conducted following [65]. In case the input domain is very large (e.g., the ImageNet case), we suggest using BigGAN [7] and following the regulations in [62] to optimize Eq. 6. Our pre-trained GANs are provided in our artifact [1].

## Appendix B. Analysis of Failure Cases

To comprehensively understand the limitations of CIPHERSTEAL and the incapabilities of exploiting ciphertext side channels on TEE-shielded NNs, this section analyzes the failure cases in our input recovery. We manually inspected 500 recovered face photos. While most recovered images are visually identical to the ground truth, we identify the following four cases where the recovered face photos largely deviate from the ground truth.



Recovered Ground truth

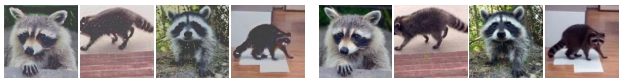
Figure 6. Failure cases in recovering images.

In the two cases shown in the first row of Fig. 6, CIPHERSTEAL fails to recover the caps in the ground truth images. In the remaining two cases in Fig. 6, the letters in the ground truth images are not recovered. Note that our attacked NNs perform face recognition, where the caps and letters may be neglected when extracting facial features. In addition, our information reconstruction leverages the implicit constraints over image contents (see Sec. 3). The caps (of diverse decorations) and letters are unusual contents in face photos, as a result, they are unlikely to be reconstructed by exploring constraints formed by common facial attributes.

Having that said, these failure cases do *not* undermine the threat of ciphertext side-channel leakage and the superiority of CIPHERSTEAL in recovering NN inputs. As discussed in Sec. 3, the core incentive behind the input recovery is stealing user secret and NN functionality. Details such as caps and letters in face photos are often *not* user privacy and usually do *not* decide the functionality of face recognition. However, secret facial attributes are accurately recovered by CIPHERSTEAL even in the cases in Fig. 6.

## Appendix C. Limitation and Potential Augmentation

As CIPHERSTEAL relies on constraints of image contents to reconstruct the lost information in NN inputs (see Sec. 3), its input recovery may be limited by the attacker's knowledge of the target input's content (i.e., contents of the attacker's inputs and user's inputs are expected to follow the same "format", as defined in Def. 2). For instance, CIPHERSTEAL is unable to recover inputs whose contents are completely different from attacker's inputs (e.g., face vs. chest X-ray images whose "input formats" are different).



Recovered Ground truth

Figure 7. Raccoon (a class not in ImageNet) images recovered using only ImageNet images.

To further explore CIPHERSTEAL's recovery capability when facing inputs of distinct contents, we use ImageNet images (as attacker's inputs) to recover raccoon images (a class not in ImageNet) following [62]; results are shown in Fig. 7. Although contents in raccoon images are largely distinct from all ImageNet images, some ImageNet images (e.g., dog images) and raccoon images share similar content-wise constraints (e.g., both dog and raccoon often have fur, a tail, and multiple legs). This result sheds light on expanding CIPHERSTEAL's application scope beyond inputs of the same "format": the shared content-wise constraints can be incorporated into CIPHERSTEAL to recover inputs of distinct contents. Recent multi-modal large models (e.g., GPT-4V and LLaVA which accept inputs from an open set) can effectively map diverse inputs into a unified embedding space that captures content-wise similarity. To augment the input recovery with them, CIPHERSTEAL can be employed to transform/reconstruct the embeddings instead of the original inputs; we leave this as future work.

## Appendix D.
## Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

### D.1. Summary of Paper

The work shows how side-channel attacks can be used to recover input data from TEE-shielded neural networks. The authors use a two-step approach where partial information is recovered using the ciphertext side channel, and then reconstructed into a valid input from the given domain.

### D.2. Scientific Contributions

- Identifies an Impactful Vulnerability.
- Provides a Valuable Step Forward in an Established Field.
- Establishes a New Research Direction.

### D.3. Reasons for Acceptance

- The paper demonstrates the novel use of ciphertext side channel information to reconstruct actual inputs to TEE-protected NNs, highlighting a limitation of TEEs. It extends work on crypto key recovery, showing how side-channel attacks can be mounted against low-entropy "soft targets". The work also contributes methods for reconstructing data (image/video) using the partial data received using the side channel attack.
- The program committee appreciated the work's objectives, the novelty of the attack target (neural network inputs), the thorough evaluation, and the fact that the problem was formally modeled.