SIDE CHANNEL ANALYSIS FOR AI INFRASTRUCTURES

by

Yuanyuan YUAN

A Thesis Submitted to
The Hong Kong University of Science and Technology
in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy
in Computer Science and Engineering

August 2024, Hong Kong

ACKNOWLEDGMENTS

This Ph.D. journey has been a long and winding road, and I am grateful to many people who have supported me along the way. Without their help, I could not make it to the end.

First and foremost, I would like to express my deepest gratitude to my supervisor, Dr. Shuai Wang. I met Shuai in Shanghai in 2019 when I was a junior undergraduate. Shuai generously accepted me to the group even when I was a novice in research. I have been fortunately working with him for five years and learned almost all my research skills from him. Shuai's guidance during the first few years of my Ph.D. study was invaluable, which shaped my research taste and the way of identifying, analyzing, and solving problems. Shuai is also highly supportive on my career development. He provided me opportunities to visit top research groups and collaborate with top researchers, and taught me how to mentor junior students. I am grateful for his patience, encouragement, and support. I am glad that I have gradually become a more mature and independent researcher under his supervision. Without any doubt, Shuai is the best mentor I could ever have and sets an exceptional role model for me.

Second, I would like to extend my sincere gratitude to Prof. Zhendong Su, who hosted me as a visiting student at the AST lab, ETH Zurich, from 2022 to 2023. Zhendong always encouraged me to think bigger and broader. His sharp insights and constructive feedback have helped me jump out of concrete technical details and see the big picture. As a pure researcher, his enthusiasm and persistence showed me what life-long learning and research should be. Zhendong always patiently listened to my ideas, no matter how naive they were, and gave me suggestions to improve them. I really enjoyed the discussions and chats with him. Moreover, Zhendong was always the first person to comfort me and encrouaged me to march forward every time I encountered problems. I am happy that I am becoming more confident and my research vision has been greatly expanded under Zhendong's mentorship.

Third, my gratitude also belongs to all committee members of my thesis defense, proposal, and qualifying exam. They are Prof. Zhiqiang Lin from the Ohio State University, Dr. Zili Meng from the department of ECE at HKUST, as well as others from my major

department, including Prof. Nevin Zhang, Dr. Wei Wang, Prof. Shing-Chi Cheung, Dr. Yangqiu Song, Dr. Long Chen, Dr. Dongdong She, and Dr. Binhang Yuan. Without them, I could not make the final thesis.

I am greateful to all group members at HKUST, including Dr. Daoyuan Wu, Dr. Zhibo Liu, Dr. Huaijin Wang, Qi Pang, Pingchuan Ma, Yanzuo Chen, Zhenlan Ji, Zongjie Li, Wai Kin Wong, Dongwei Xiao, Dong Chen, Hongyi Lu, Sen Deng, Yiteng Peng, Ao Sun, Liwen Wang, and many others. Our experiences of exchanging ideas and fighting for paper deadlines are unforgettable. I want to also thank my collaborators out of HKUST, including Prof. Yinqian Zhang from SUSTech and Prof. Tsong Yueh Chen from Swinburne University of Technology. They are also mentors to me and their passion for research has always encouraged and inspired me.

I would like to thank people I met in the AST lab at ETH Zurich, including Dr. Shaohua Li, Dr. Chengyu Zhang, Dr. Cong Li, Dr. Yifei Lu, Zuming Jiang, Shengcheng Yu, Yuchen Gu, Yann Girsberger, Theo Theodoridis, Dominik Winterer, and many others. Their help made my research and life in a new country much easier and more enjoyable. Moreover, I would like to also express my thanks to all my friends in Hong Kong; they are Dr. Changyang He, Dr. Jiacheng Shen, Chaokun Chang, Qihui Zhou, and Kai Chen. We have been friends since we were in Fudan, and I am glad that we can meet again in Hong Kong. Settling down in a new city is not easy, but their help made it much easier. Our dinners on weekends are always unforgettable memories.

I am sure that I must forget to mention many people who have helped me during this Ph.D. journey. Please forgive me for my poor memory and know that I am always grateful for your help.

Finally, my deepest gratitude goes to my wife, my parents, and all my family members. Their constant and unconditional love is not only the source of my courage but also the most precious thing in my life. I am deeply grateful for their support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

SIDE CHANNEL ANALYSIS FOR AI INFRASTRUCTURES

by

Yuanyuan YUAN

Department of Computer Science and Engineering

The Hong Kong University of Science and Technology

ABSTRACT

Side channel analysis (SCA) investigates the unintended secret leakage in a system's non-functional characteristics such as execution time or memory access patterns. Given the growing adoption of AI systems in security-critical and privacy-preserving applications, this thesis comprehensively studies side channel leakages in infrastructures that underpin the entire life cycle of modern AI systems, including data processing libraries, trusted execution environments (TEEs), runtime interpreters, executables on edge devices, etc. It also proposes highly practical solutions for SCA.

On the offensive side, this thesis demonstrates end-to-end attacks that recover user's inputs (i.e., the user's privacy) and the underlying neural networks (i.e., the intellectual property) of AI systems from various side channels. We first consider a non-privileged co-process as the attacker and exploit cache side channels. Modern AI systems adopt data processing libraries (e.g., Libjpeg, FFmpeg) to handle various formats of user inputs like images and audios. Our work for the first time recovers such complex inputs from these libraries' cache side channels. Then, we consider untrusted hosts as adversaries. While TEEs are widely employed to shield AI systems from malicious host platforms,

our works exploit the ciphertext side channels in TEEs and reconstruct both inputs and neural networks from TEE-shielded AI systems, breaking the security belief of TEEs. We also propose the first unified recovery scheme for complex input data like images, audios, text, videos, etc., and the first practical reconstruction technique for model weights of real-world neural networks.

On the defensive side, this thesis localizes hundreds of new leakage sources in the exploited systems. Prior works often adopt program analysis techniques to model leakage patterns, whose localization is leakage-specific and suffers from the scalability issue. We recast the information leakage as a cooperative game among all leakage sources and reduce the cost from exponential to nearly constant. Our work presents a generic localization pipeline and supports analyzing production-size programs and AI infrastructures for the first time. We systematically examine the leakage in AI runtime interpreters including TensorFlow and PyTorch, and study how their different computation paradigms affect the leakage. Our analysis also reveals that optimizations in AI compilers (e.g., TVM, Glow) enlarge the leakage in compiled neural network executables.

CHAPTER 1

INTRODUCTION

Security is the fundamental requirement for modern computer systems. Over the years, security exploitation and defense have been playing a cat-and-mouse game, with tremendous techniques continuously proposed and evolving. Among them, side channel leakage stands out as one severe yet stealthy threat to computer security. In general, side channels leak secrets via side effects and non-functional characteristics of a system's execution. Unlike typical security flaws that are induced by algorithmic defects, side channels denote unintended secret leakage from the system's physical activities and are often exploited through the interplay between a system and the outer world.

Side channel analysis (SCA) is the process of analyzing secret leakage in an attacker's observed side channels, such as power consumption [207], electromagnetic radiation [169], cache access [236], and memory activities [149], etc. A typical SCA covers two main aspects: 1) *secret analysis* and 2) *leakage analysis*. The secret analysis investigates how secrets can be reconstructed from the observed side channels, while the leakage analysis examines how secrets are leaked through side channel observations. With the era of artificial intelligence (AI) approaching, it is urgent to understand the security implications of side channel leakage in AI systems. To this end, this thesis comprehensively studies the opportunities and challenges of SCA in infrastructures that span the entire life cycle of modern AI systems, and also provides practical solutions.

1.1 AI Infrastructures and Secrets

Fig. 1.1 presents infrastructures in modern AI systems, including data processing libraries, the underlying neural network, and the runtime environments for running the neural network. Since the neural network relies on semantics in inputs (e.g., a face in a portrait image) to make predictions, the AI system adopts data processing libraries like `libjpeg` to handle user inputs in various format specifications (e.g., `JPEG`, `PNG`, etc.). The processed

inputs are then represented as vectors or matrices, and taken by the neural network to predict outputs. The prediction is performed as a sequence of matrix operations between an input and the neural network's weights.

The neural network can run in different runtime environments. The most popular way is running neural networks in interpreter-based frameworks like TensorFlow [12] and PyTorch [199]. Recently, neural networks are increasingly compiled into executables by AI compilers, such as TVM [51] and Glow [210], for better performance across different end devices. In cases of confidential computing (e.g., querying an untrusted AI service provider, or deploying the AI system on an untrusted host), the whole system is often put into Trusted Execution Environments (TEEs) to protect secrets [143, 125, 156].

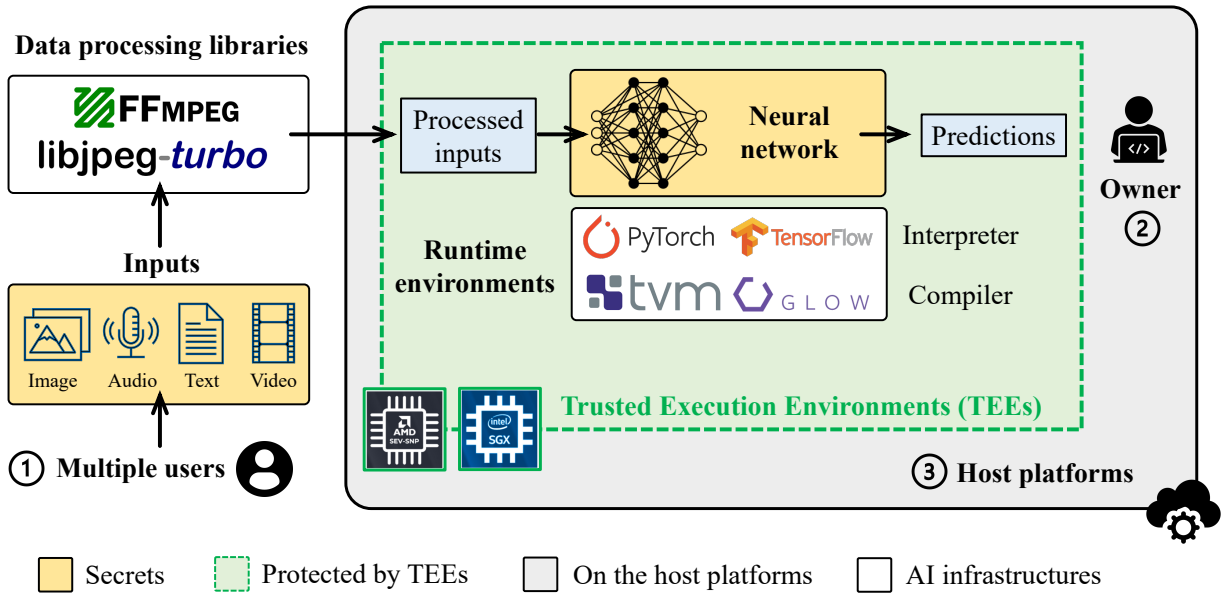


Figure 1.1: Illustration of the workflow, infrastructures, secrets, and participants in modern AI systems.

Secrets and Attacker's Incentives

Two types of secrets are of interest in AI systems: 1) the *input* which denotes user's private data, and 2) the *neural network* which represents the intellectual property of the AI system's owner. In particular, since the intelligence of a neural network is encoded in its weights and obtaining well-formed weights requires considerable (private) labeled data, extensive computing resources, and human expertise for designing the training algorithm, this the-

sis focuses on leakages of inputs and neural network weights in AI systems. In general, attackers have the following incentives to steal the above secrets:

1. **Spying User's Privacy.** Leaking inputs lead to privacy violations of users. For example, the input image of an AI-based disease diagnosis system may contain sensitive information about the user's health condition. The input text of an AI assistant may contain the user's private conversations.

2. **Copying Intellectual Property.** Leaking neural network weights enables attackers to clone the AI system's functionality without training neural networks from scratch. This significantly reduces the cost of developing AI systems and brings commercial benefits to the attackers.

3. **Enabling New Attacks.** The recovered inputs help attackers scope possible inputs of the AI system, while the recovered weights make the neural networks white-box accessible to attackers. This information can be leveraged to launch attacks to manipulate and collapse the AI system's functionality.

1.2 Adversaries and Side Channel Leakage

As illustrated in Fig. 1.1, the real-world usage of AI systems involves three groups of participants: ① multiple users, ② the owner of the AI system, and ③ the host machine where the AI system deploys. Aligned to the two types of secrets mentioned in Sec. 1.1, these groups may pose the following threats to the AI system:

① **User: Input Leakage.** When multiple users are using the same AI system (e.g., querying a cloud-based AI service), a malicious user may exploit the AI system to infer other users' private inputs.

② **Owner: Input Leakage.** When querying an AI system from an untrusted service provider, users may expose their privacy (e.g., a chest X-ray image containing disease information) to the AI system's owner.

③ **Host: Neural Network Leakage.** When deploying the AI system on an untrusted host platform, the host may duplicate the underlying neural network to copy the AI system's intellectual properties.

In the scenario of ①, the adversary is *non-privileged* and does not have access to the AI system's internals. Hence, the exploitable threat lies in the shared hardware resources between the AI system and its co-process launched by the adversary. This thesis identifies input leakage induced by cache side channels in the data processing libraries of AI systems: since the cache is shared by the AI system and the adversary, the adversary can record the accessed cache units when the data processing library is processing an input, and leverages cache access patterns to recover the input.

In the context of ②, the AI system's owner also owns the host platform; in that sense, both ② and ③ render privacy concerns from untrusted hosts. Given the high privilege of the malicious host, the most mature and widely adopted (in both academia and industry) mitigation is putting the whole AI system into Trusted Execution Environments (TEEs), so that the AI system's owner cannot view user's inputs and the host cannot duplicate neural network's weights from the deployed AI system. However, despite that TEEs have delivered wide security belief via memory encryption (i.e., data in memory are encrypted as ciphertexts), their deterministic encryption introduces a new threat — ciphertext side channels: when a TEE-shielded program is consecutively writing to the same memory address, the generated ciphertexts collide if identical plaintext (i.e., secrets) are written. This thesis shows that both inputs and the neural network's weights can be leaked via TEE's ciphertext collision patterns.

1.3 Challenges and Solutions

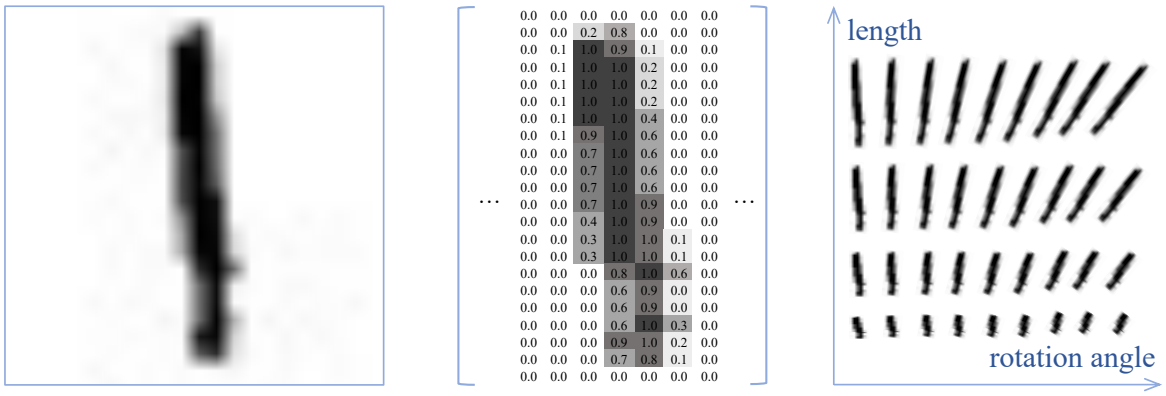
1.3.1 Secret Analysis

We first elaborate on the challenges and our contributions in recovering secrets from different side channels w.r.t. the three adversaries mentioned in Sec. 1.2.

***Challenge One*: High-dimension Inputs and Large Search Space (①)**

While the secret analysis has been widely studied in cryptography systems to recover private keys, their techniques do not apply to AI systems due to the large search space. Inputs of AI systems are often high-dimensional media data such as images, audios, and texts. For instance, a common RGB image of 1024×1024 size contains more than 3M pixels and each pixel's value ranges from 0 to 255. A private key in modern RSA encryption, however, only has 2048 binary key bits (i.e., either 0 or 1).

To tackle the issue of large search space, this thesis identifies the following opportunities. To ease the understanding, we use images as representative inputs of AI systems; our observations are also applicable to other types of inputs (see details in Chap. 3). First, different from cryptographic keys where key bits are private, not all pixels in an image are secrets — 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 (e.g., being visually consistent to user's inputs). Second, unlike cryptographic key bits that are independently sampled, pixel values are highly correlated, e.g., the digit one images in Fig. 1.2(a)-(b), where pixels are "constrained" (not randomly sampled) to form a valid digit one. With such correlations, the search space can be further reduced.



(a) Digit "1" (b) 1 - pixel values / 255 (c) Digit "1" of varied rotation angle and length.

Figure 1.2: Pixel values of a digit "1" image and the illustration of the semantic-wise viewpoint.

***Solution*: A Semantic-wise Viewpoint (①)**

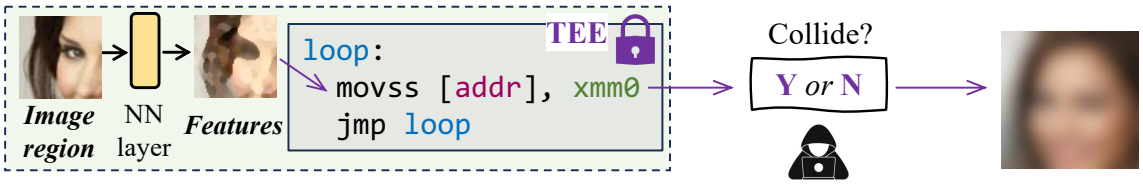
Motivated by the above observations, this thesis therefore focuses on semantical contents in user's inputs (e.g., a face in an image), rather than exact pixel values. This semantic-wise viewpoint is also aligned with AI systems, as they are designed to perceive semantics in their inputs. The semantics of media data are characterized by data manifolds, which are low-dimensional representations of media data that capture semantics in their contents. Consider again the digit one image in Fig. 1.2(a), whose size is 32×32, to represent this image using pixels, 32×32 integers from $[0, 255]$ are required. Alternatively, we can simplify the digit one as a segment (i.e., only focusing on the visual appearance of the digit one); this way, only two dimensions[1] are needed to represent different (simplified) digit ones, as illustrated in Fig. 1.2(c). Later in Chap. 3, we will show that, for more complex inputs like face photos, our semantic-wise viewpoint can reduce the search space by more than 90%. This viewpoint also enables automated and unified recovery for inputs of different types and formats, as will be introduced in Chap. 3.

***Challenge Two*: Partial Leakage and Insufficient Observations (②)**

With the above semantic-wise viewpoint, we can recover AI system's inputs from cache side channels in data processing libraries. However, the recovery pipeline cannot be directly employed in the context of TEE and ciphertext side channels: as illustrated in Fig. 1.3(c), only blurry images of missing details are recovered. This issue is due to the partial leakage and insufficient observations of ciphertext side channels in TEE-shielded neural networks.

As shown in Fig. 1.3(b), ciphertext side channels are records of ciphertext collisions (i.e., the ciphertexts collide or not when writing secrets into the memory) of the victim, providing only binary observations. Cache side channels, in contrast, record the accessed cache units (e.g., cache sets, cache lines, etc.) of the victim; merely a L1 cache can have 64 cache sets, resulting in more informative observations to attackers. Moreover, in threat ① which is identified in data processing libraries, the leakages are induced by the libraries' direct operations on image pixels, whereas in threat ②'s scenario, intermediate results

[1] We clarify that a digit one's semantics are more complicated than that of the simplified segment, but the additional dimensions required for the concrete representation should not be too many.



(a) Ciphertext collisions induced by writing features (b) Binary observations (c) Recovered image

Figure 1.3: Blurry images are recovered due to the partial leakage and insufficient observations issues in ciphertext side channels.

written by a neural network (that trigger ciphertext collisions) are features extracted from its input; these features are highly abstracted such that certain input information is inevitably lost, as shown in Fig. 1.3(a).

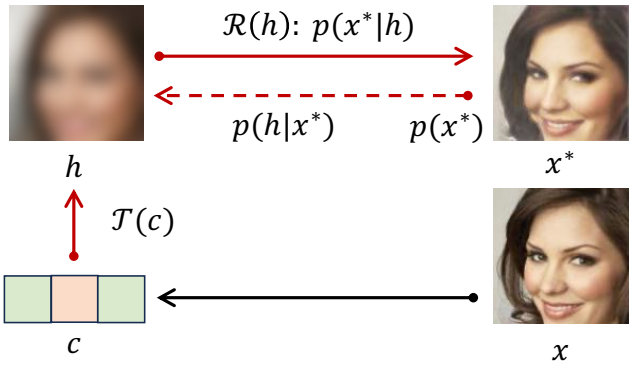


Figure 1.4: Illustration of our information reconstruction that is through the Bayesian perspective.

***Solution*: Information Reconstruction through Bayesian Perspective (②)**

As illustrated in Fig. 1.4, suppose the input x leads to ciphertext side channel observation c, the input recovery (as exploited in ①) essentially transforms input information leaked in c into a proper form that is aligned to the original input (e.g., a blurry image $h = \mathcal{T}(c)$ shown in Fig. 1.4). The key hurdle in ②'s context is reconstructing x's lost information from h (e.g., a partially recovered image). This should be feasible by leveraging the correlations (that constrain the image to have meaningful contents) among pixels. The same applies to other input types; see details in Chap. 4.

From the Bayesian perspective, the objective of the reconstruction is:

$$\arg\max_{x^*} p(x^*|h), \tag{1.1}$$

where $p(x^*|h)$, which infers x^* based on h, is maximized when x^* equals the target input x. According to Bayesian theorem, $p(x^*|h)$ can be reformulated as:

$$p(x^*|h) = \frac{p(h|x^*)p(x^*)}{p(h)}. \tag{1.2}$$

Since h is known after transforming the leaked information in c, $p(h)$ is accordingly fixed. As illustrated in Fig. 1.4, the reconstruction's objective is therefore equivalent to:

$$\arg\max_{x^*} p(h|x^*)p(x^*). \tag{1.3}$$

Here, $p(x^*)$ indicates how likely x^* is semantically meaningful (e.g., a valid face image rather than random pixels). Estimating $p(x^*)$ has been widely studied and existing research can provide out-of-the-box solutions via generative models (e.g., GANs [89], VAEs [128]). The $p(h|x^*)$, which is the inversion of the reconstruction process, mimics the information loss from the target input x to the observed h. Estimating $p(h|x^*)$ is inherently easier than estimating $p(x^*|h)$ as it "removes" information from x^*. Based on this reformulation, this thesis for the time demonstrates successful input recovery from TEE-shielded neural networks, as will be detailed in Chap. 4.

***Challenge Three*: Indirect Leakage and High Integrity Requirement (③)**

With the recovered inputs on hand, one may expect to train an equivalent neural network. However, this is less preferred in practice. Training a neural network often requires tens of thousands of inputs, that is, it requires performing the above input stealing attacks for the same amount of times. Frequently launching such attacks is not only time-consuming but also raises the risk of being noticed by the victim.

Therefore, this thesis focuses on directly recovering neural network weights from ciphertext side channels without using relevant inputs. The challenges of recovering neural network weights primarily lie in three folds. First, neural network weights have far more dimensions than that of user inputs, e.g., modern neural networks usually have billions of weight elements, and each weight element is a floating-point number of (theoretically) infinite range. Second, neural network weights are preloaded before computation and

TEE-shielded neural networks do not write their weights into memory during computing; the leakages in ciphertext side channels are induced by intermediate computation results indirectly derived from weights. Third and most importantly, neural network weights exhibit a high integrity requirement — a few incorrectly recovered weight elements (e.g., 1∼5 incorrect ones out of ∼10M weight elements [206, 268, 151]) can make the weights non-functional; this is not like the case in recovery inputs, where two visually identical images may have many pixels different.

***Solution*: Extracting Neural Network's Functionality (③)**

Recall as mentioned in Sec. 1.1, the key incentive of stealing neural network weights is to clone its functionality, as it is the intellectual property of the AI system. In this regard, it is unnecessary to recover the exact weights; this is supported by the training procedure of neural networks, where different but functionality-equivalent weights can be obtained in multiple training attempts. Moreover, extracting the neural network's functionality from its ciphertext side channels is feasible: the ciphertext side channels are induced by the intermediate computation results, which are features extracted by the neural network and reflect its functionality.

Hence, we exploit to directly recover functionality-equivalent weights from a TEE-shielded neural network's ciphertext side channels, and to make the attack stealthy, we propose to recover functional weights by observing the neural network's one execution without interacting with it (i.e., querying outputs from the neural network). We achieve this by leveraging the hyper-network, a special neural network that generates neural network weights. We also design a novel training algorithm for the hyper-network, delivering task-wise generalizable weight generation: our hyper-network can generate weights for an unseen functionality without using relevant inputs. The detailed algorithm and design considerations will be introduced in Chap. 5.

1.3.2 Leakage Analysis

The leakage analysis primarily localizes program modules (e.g., an instruction accessing the memory, or a matrix operation in a neural network) that induce secret-dependent side

channel observations. The key hurdles are the precision and scalability of the localization.

Conceptually, previous works have proposed various models to characterize the leakage's patterns, e.g., an array index that is secret, or a branch condition determined by a secret. However, with more leakage patterns increasingly discovered, existing models are not sufficient to cover all leakage sources. For instance, our work in Chap. 6 shows cache side channel leakage due to implicit data dependencies (e.g., an array access whose index is not secret, but is within a loop whose iteration count is secret). Such leakage patterns are not modeled previously, leaving considerable leakage sources unnoticed by developers. In addition, previous leakage analysis techniques exclusively focus on cache side channel leakage, and are inapplicable to ciphertext side channels. The first ciphertext side channel localization was proposed in 2023 [69], despite that ciphertext side channels have been known for years since 2021 [149].

Technically, most prior localization works rely on program analysis techniques (e.g., abstract interpretation [63] or symbolic execution [127]) to identify leakage sources. These techniques are computationally expensive and are limited to small programs. Existing works thus first pinpoint potentially vulnerable program segments and then apply the analysis to these selected program segments. As will be introduced in Chap. 6, previous works have neglected considerable vulnerable modules in their analyzed software, due to the limited scalability. Moreover, AI infrastructures are implemented as programs of production size, for instance, a single execution of a neural network running with PyTorch can generate billions of side channel observations, significantly exceeding the scale of prior works (e.g., analysis with abstract interpretation may only support programs with hundreds of memory accesses [73]).

***Solution*: A Cooperative Game for Leakage Localization**

This thesis solves the localization problem by forming the secret leakage as a cooperative game among different leakage sources. Since we have demonstrated recovering secrets from different side channels in the three threats, we can localize the leakage sources by investigating how much information each side channel record contributes to the recovery; the corresponding program points that generate these records are the leakage sources. Essentially, prior works explore possible patterns of leakage sources and leverage pre-

defined patterns to identify leakage sources. This thesis terminates this endless exploration (considering that new leakage patterns and new side channels are continuously discovered) by focusing the consequences of the leakage sources, i.e., contributing to the secret recovery. Extending our analysis to new leakage patterns or side channels has no technical hurdle, as long as they can be used to recover secrets.

This recast to a cooperative game is well-addressed by the game theory via Shapley value [220], which accurately attributes the recovered information to each leakage source. Nevertheless, the cost of computing Shapley value is exponential to the number of side channel records, making it infeasible to compute in practice. This thesis identifies a key property — sparsity — in side channels (i.e., only a few side channel records contribute to the secret recovery), and proposes optimizations to reduce the cost to nearly constant. Our techniques further enables quantitative analysis, so that developers can prioritize patches to the most severe leakage sources. As will be elaborate in Chap. 6, we localize hundreds of new leakage sources in software that have been extensively analyzed by prior works.

1.4 Contributions

This section summarizes the key contributions this thesis has made to side channel analysis and AI system security research. In the corresponding chapter of each presented work, we will show more detailed contributions regarding techniques, findings, impacts, etc.

Overall, this thesis's contributions primarily lie in the following aspects:

- **Attack & Threat.** Our works for the first time demonstrate two severe threats in modern AI systems. (1) We show that a malicious user, though without any system-level privilege, can recover other user's inputs from cache side channels of various data processing libraries. (2) We also reveal the "false trust" that Trust Execution Environments (TEEs) provide to AI systems. Despite that TEEs have been widely adopted to protect users when querying untrusted AI service providers, and prevent malicious host machines from stealing the deployed neural networks, our works show that both inputs and neural networks of AI system can be leaked via TEE's ciphertext side channels.

- **Secret Recovery.** We propose the first practical techniques for recovering user's inputs and neural network's weights from different side channels. (1) Our attack for the first time demonstrates that inputs of different types (including images, text, audios, and videos) and complex content (e.g., face photos, chest X-ray images) can be accurately recovered with secret information retained. The whole recovery process is fully automated without any human intervention. (2) Our attack also for the first time recovers weights of real-world neural networks. The attack is highly stealthy and efficient; it does not require querying the target neural network and can directly recover functional weights without further training or tuning.

- **Leakage Localization.** Our work proposes a new paradigm of side channel leakage localization. Prior techniques primarily model leakage patterns and are limited to specific side channels of pre-defined patterns. We turn our focus to the consequences of the leakage sources, and recast the leakage localization as identifying program points that contribute to the secret recovery, delivering a generic localization paradigm that is agnostic to leakage patterns or side channels. By re-examining software extensively studied by prior localization works, we have identified hundreds of new leakage sources. With meticulous optimization, our localization for the first time supports production-size software in AI infrastructures.

1.5 Thesis Organization

Chap. 1 has briefly introduced the studied problem, challenges, and our solutions and contributions in side channel analysis for AI infrastructures. The rest of this thesis is organized as follows.

Chap. 2 introduces preliminaries of AI infrastructures, their security implications, and the observable side channels; it also reviews related works in both secret and leakage analysis. Then, following the three threats discussed in Sec. 1.2, Chap. 3 presents our work (published in **USENIX Security 2022** [278]) that shows how a malicious user can recover other user's inputs from cache side channels in data processing libraries. Chap. 4 shows our work (published in **IEEE S&P 2025** [275]) on breaking TEE's security guarantees and demonstrating how a malicious AI service provider can steal user inputs. Chap. 5 intro-

duces our work (published in **CCS 2024** [276]) that exploits how weights of TEE-shielded neural networks can be leaked to the host platform. All three works accordingly propose countermeasures to our demonstrated attacks. Later in Chap. 6, we introduce our work (published in **USENIX Security 2023** [277]) on quantitative analysis of leakage sources; this work delivers scalable and generic techniques for localizing leakage sources in different side channels of production-size software. Finally, Chap. 7 summarizes key problems, solutions, and findings in this thesis.

CHAPTER 2

PRELIMINARIES AND RELATED WORKS

This chapter briefly introduces preliminaries and related works of the thesis's research topics. More detailed background knowledge of each research work will be presented in Chap. 3, Chap. 4, Chap. 5, and Chap. 6 based on their contexts.

2.1 Cache Side Channels

2.1.1 Caching Mechanisms and Exploitation

Modern computer systems have employed various caching mechanisms to accelerate data retrieval and enhance their overall performance. In essence, caches serve as high-speed buffers that store frequently accessed data and instructions, bridging the gap between fast processor and slow main memory access. Nevertheless, this optimization also introduces security vulnerabilities known as cache side channels.

Cache side channels are induced by two factors: 1) the shared cache between victims and adversaries, such that adversary's and victim's cache accesses interfere with each other, and 2) the measurable variations of cache hit (i.e., the accessed data are found in cache) and cache miss (i.e., the accessed data are not in cache and need to be fetched from the main memory). By running a program co-located with the victim on the same device, the attacker can frequently access the shared cache and infer the victim's cache access patterns during his/her access intervals. Cache side channels are severe threats since they can be exploited remotely (without physical access to the target) by non-privileged adversaries; the exploitation is also highly stealthy.

Flush+Reload [270] and Prime+Probe [164, 195] are the two most well-studied cache side channel exploitation techniques. Flush+Reload assumes sensitive code or data is shared between the attacker and the victim. Thus, the attacker can clean the shared cache

with the `clflush` instruction and observe the victim's subsequent behaviors by measuring the re-access time to the cache later. A fast re-access means the victim has accessed the target memory address, and a slow re-access means the opposite. On the other hand, Prime+Probe does not require any shared user-space memory pages and is more potent and widely applicable. However, Prime+Probe only observes activities in more coarse-grained cache units: it cannot recognize specific accesses to a cache line and only detects accesses to the monitored cache set. Instead of flushing the cache in a ready-made manner, Prime+Probe evicts all the data in the target cache set by filling it with the attacker's data. When the attacker re-accesses a cache set, a slow re-access indicates that the victim has accessed this cache set.

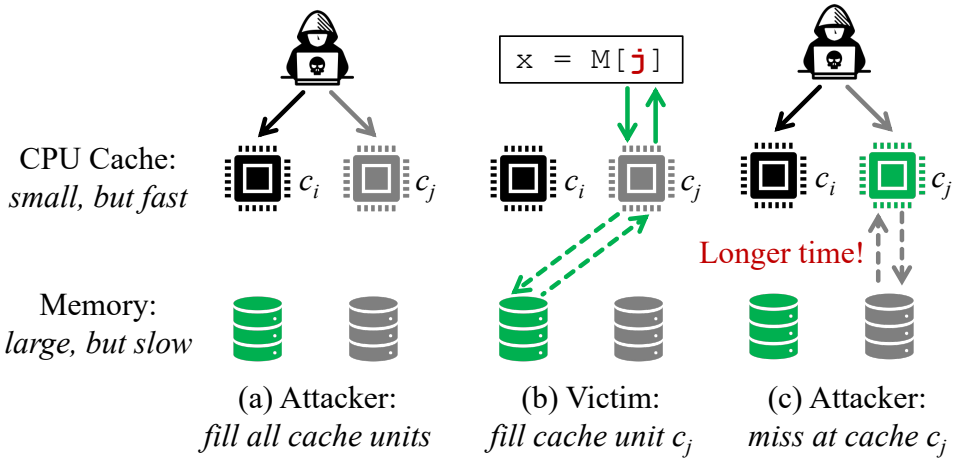


Figure 2.1: Illustration of a typical workflow of Prime+Probe.

A typical exploitation procedure of Prime+Probe is shown in Fig. 2.1. Each exploitation iteration consists of two steps. First, the attacker fills all cache sets with his/her own data. Then, the attacker re-accesses all cache sets after a time interval. If the victim has accessed a cache set c_j during the time interval, the attacker triggers a cache miss, which can be reflected as a longer access time. Otherwise, the attacker knows that the cache set c_j was not accessed by the victim. By repeatedly performing the access & re-access process, the attacker can infer the victim's cache access patterns, which depend on the victim program's execution and data access behaviors.

2.1.2 Data Processing Libraries

Inputs of AI systems are media data like images, text, audios, etc., which are often stored in different format specifications. For instance, an image of the same content may be stored in `JPEG`, `PNG`, or `BMP` formats. To handle the diverse formats and extract the semantic information (which is format-agnostic), modern AI systems adopt data processing libraries like `libjpeg` and `FFmpeg` to preprocess the inputs. The preprocessing typically parses the data bytes (e.g., pixels of an image, tokens of a sentence) and rearranges them into a proper form (e.g., an image is usually converted into a matrix of $channels \times width \times height$ size).

In practice, `libjpeg` denotes one representative image processing library; it is widely adopted in frameworks like PyTorch and TensorFlow. `FFmpeg` is broadly employed to process audios and videos. `Hunspell` maintains a dictionary and frequently checks words in the dictionary, representing a popular paradigm of handling words in modern AI systems. Overall, this thesis finds cache side channel leakages in all these popular data processing libraries.

We find that, when processing inputs, those libraries have extensive cache accesses that depend on data bytes of the input. For instance, we notice that in `libjpeg`, pixel values frequently involve in array index, so that the accessed cache unit is determined by the pixel values. As a consequence, these libraries' cache access patterns are highly dependent on the processed input. By exploiting the cache side channels in those libraries (e.g., following the procedure depicted in Fig. 2.1), attackers can record a sequence of their accessed cache units. Chap. 3 will present the concrete techniques of automatically recovering inputs from the corresponding cache access sequence.

2.1.3 Constant-Time Computations of Neural Networks

Neural network's computations are naturally free of cache side channels. Formally, a neural network $F = \ldots f_{i+1} \circ f_i \circ f_{i-1} \ldots$ consists of multiple connected layers and each layer is a function $f(x) = \sigma(\theta x + b)$. The σ is the non-linear activation function which does not have input-dependent conditions to support efficient computations for batched inputs. The layer f's weights $[\theta, b]$ and input x are all matrices. The internal computation of a

neural network is therefore a sequence of matrix operations of a fixed computing routine, whose data accesses and control branches are fixed. Accordingly, the neural network's accessed cache units are constant regardless of the input x or weights $[\theta, b]$.

With this regard, the cache side channel leakage of user inputs can be seen as a new threat brought by those data processing libraries. While a few prior works have exploited the input leakage in these libraries [265, 95], they focus on privileged adversaries like the operating system (OS). Our work presented in Chap. 3 for the first time demonstrates this threat from a remote and non-privileged malicious user's perspective, rendering a higher risk of privacy leakage.

2.2 Ciphertext Side Channels

2.2.1 Trusted Execution Environments

To prevent the host[1] from reading the AI system's inputs or neural network weights, the most mature and practical way is deploying the AI system inside Trusted Execution Environments (TEEs); this mitigation has been widely adopted in both academia [143, 125, 156] and industry [122, 20, 114], and its security guarantee is well acknowledged [285]. TEEs leverage memory encryption to create isolated execution environments for the AI system, so that the AI system's inputs, outputs, and computation results are encrypted. As a result, despite that the host can read data from the AI system's memory, he/she can only view the ciphertexts. In practice, TEE's encryption engine encrypts/decrypts memory data on-the-fly and is implemented as a hardware module between the CPU chip and DRAM.

TEE's memory encryption is constrained by two factors. First, to enable efficient random memory access, different memory blocks should be encrypted independently. Second, to support encrypting large memory, additional space and latency are needed for counters. To meet these two requirements, AES encryption with *deterministic*, block-based mode is widely used by TEEs with large encrypted memory, such as AMD SEV [122], Intel TDX [114], Intel SGX on Ice Lake SP [114, 119], and ARM CCA [20]. Specifically, given a

[1]Note that in threat ② introduced in Sec. 1.2, the malicious AI service provider is also the host.

memory block, to encrypt its memory value v, the encryption first takes a tweak function T to calculate a mask $m = T(a)$, where a is the address of the block. The encrypted ciphertext is generated as $c = P(v \oplus m) \oplus m$, where P is the encryption function. Therefore, when the same value v is stored at the same address a, the generated ciphertext is always identical (i.e., ciphertext collision).

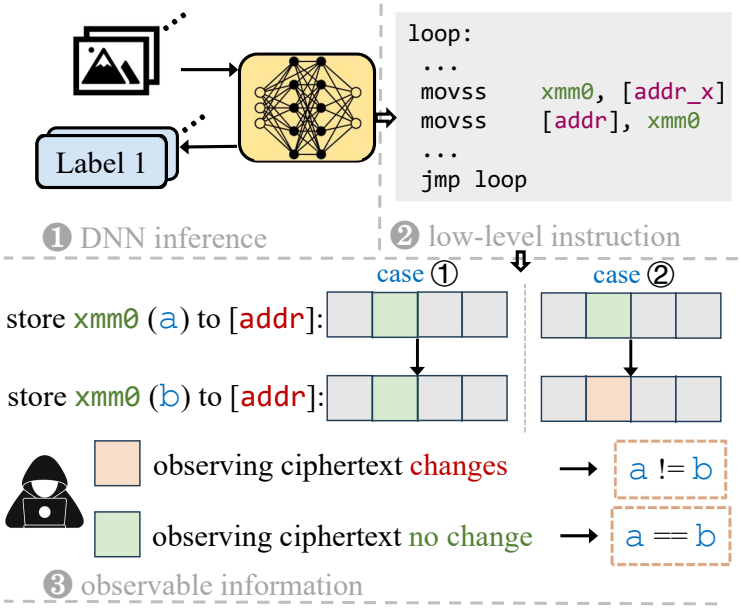


Figure 2.2: Illustration of ciphertext collision and its induced leakage.

Ciphertext side channels exploit TEE's deterministic encryption to infer the equality relation of consecutive memory writes. As illustrated in Fig. 2.2, 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.

2.2.2 Nested Loops and Non-Linearity in Neural Networks

This thesis reveals the leakage of inputs and neural network weights in TEE-shielded neural networks due to ciphertext side channels. Essentially, the computations inside a neural network are matrix operations between inputs and weights, which are implemented as nested loops. Therefore, a neural network's computations can lead to repeated memory writes to the same memory address. In addition, although intermediate computation results of a neural network are floating-point numbers which may rarely incur

collisions, neural network's non-linear activation functions[2] often output identical intermediate values. These non-linear activation functions either map inputs to a smaller range (e.g., `Sigmoid` that maps values from $[-\infty, +\infty]$ into $[0, 1]$), or convert continuous values into discrete ones (e.g., `ReLU` maps all negative values into 0). Considering that modern computer systems have a limited precision (e.g., 64-bit) for floating-point numbers, the neural network's non-linearity can largely increase the probability of identical written values.

The above two factors (i.e., repetitive writes to the same memory address and considerable identical written values) make ciphertext side channels highly exploitable in TEE-shielded neural networks. Since the intermediate values written into the memory are only determined by the neural network's inputs and weights (i.e., changing inputs or weights can lead to different intermediate values), the ciphertext collisions induced by the neural network's memory writes correlate to both inputs and weights, leaking their information to the host. Modern neural networks can run in different ways inside TEEs, leading to distinct memory write patterns and thus different leakage behaviors. We introduce different runtime environments of neural networks below in Sec. 2.2.3.

2.2.3 Runtimes of Neural Networks

Interpreter-Based Deep Learning Frameworks

The most popular way is running neural networks in interpreter-based deep learning (DL) frameworks like TensorFlow [12] (maintained by Google) and PyTorch [199] (maintained by Meta). Essentially, a DL framework is a software library that provides a programming interface for developers to build and run neural networks. The runtime system of typical DL frameworks consists of two components: (1) the Python interfaces that parse and interpret the high-level neural network into a set of matrix operations, and (2) third-party linear algebra libraries (e.g., OpenBLAS [282], MKL [242], and Eigen [93]) that implement such operations with efficient low-level binary code.

Modern DL frameworks support both forward propagation (i.e., inferencing with user's input) and backward propagation (e.g., computing gradients during training). However,

[2]The non-linearity is the core of neural network's intelligence.

different DL frameworks may implement distinct computation paradigms. For instance, PyTorch maintains a dynamic computational graph on the fly, and when computing gradients during backward propagation, it actively deletes the computational graph to save computing resources. TensorFlow, in contrast, maintains a static computational graph; this graph is fixed after initialization and does not change during the entire execution. The different computation paradigms of PyTorch and TensorFlow lead to different amounts of information leakage via ciphertext side channels, as will be discussed in Chap. 4.

Executables Compiled via Neural Network Compilers

Neural networks are increasingly compiled into executables for better performance across different platforms. Two mature neural network compilers, TVM [51] and Glow [210], emit standalone executables that can be run with minimal external dependencies. The real-world usage of neural network compilers has been illustrated in recent research [51, 210, 172, 117] and industry practice. The TVM community has reported that TVM has received code contributions from companies including Amazon, Facebook (Meta), Microsoft, and Qualcomm [57]. As a complement to DL framework's usage in GPUs, neural network compilers fulfill the emerging demand for a wide range of other platforms, e.g., TVM has been used to compile neural networks for CPUs [165, 117]. Facebook has also deployed Glow-compiled neural networks on CPUs [185]. Overall, neural network compilers are increasingly vital to boost neural network's applications on CPUs, embedded devices, and other heterogeneous hardware backends [18, 248].

Neural network compilers typically accept a high-level description of a well-trained neural network, exported from DL frameworks like PyTorch, as their input. During compilation, these compilers often convert the neural network into intermediate representations (IRs) for optimizations. High-level, platform-agnostic IRs are often graph-based, specifying the neural network's computation flow. Platform-specific IRs such as TVM's TensorIR and Glow's High Level Optimizer (HLO) specify how the neural network is implemented on a specific hardware backend and support hardware-specific optimizations. Optimizations performed by neural network compilers often include constant folding, operator fusion (e.g., fusing a ReLU operator with a preceding convolution operator), platform-aware scheduling, and others. Finally, these compilers convert their low-level

IRs into assembly code (or first into standard LLVM/CUDA IR [139, 189]).

2.3 Secret Analysis Techniques

2.3.1 Recovery Techniques for Inputs

Recovering inputs of AI systems via side channel leakage is inherently challenging as discussed in Sec. 1.3. Only a few prior works have demonstrated successful attacks, but with limited applicability and precision. For instance, Xu et al. [265] studied how a malicious OS can infer images from `libjpeg` via controlled-channel attacks (i.e., a strong side channel that is only exploitable by the OS). Despite that the attackers are highly privileged, they can only recover outlines of the images, leaving the rich details (which are of interest in AI systems) unrevealed. This thesis however considers non-privileged attackers which renders a more severe threat to the AI system. As will be shown in Chap. 3 and Chap. 4, our recovered inputs are visually consistent with the original ones; they retain sensitive information (e.g., face identities, disease information) to a great extent and can be even leveraged to train a new neural network and enable downstream attacks to the AI system.

Several prior works have turned the focus on side channels induced by neural network's computations [182, 251]. They require the attacked neural networks to be binarized (i.e., weights elements are either -1 or 1) and assume white-box access to the attacked neural networks. They also have strong requirements for the inputs: the inputs are assumed to be black-and-white and have clean backgrounds. Despite that strong assumptions are made, these attacks only recover coarse shape in images. In practice, real-world neural networks have floating-point weight elements and take colorful and complex images as inputs, making these techniques inapplicable. Our work in Chap. 4 is the first to recover real-life inputs from side channels of general neural networks; we also demonstrate successful attacks for other more complex input types like audios and videos.

2.3.2 Recovery Techniques for Neural Network Weights

Neural Network Structure

Most existing works explored how to recover a neural network's structure (e.g., how many layers a neural network has, how different layers are connected, etc.) from different side channels [112, 267, 105, 266, 167, 81, 75]. While they had received wide attention previously, the structure information is becoming less valuable in practice, as neural network's structure is often public in modern AI systems, and neural networks in most commercial AI systems are built upon public backbones. Moreover, stealing a neural network's structure does not bring much benefit to attackers, because without well-trained weights, the neural network is unusable. As we have discussed in Sec. 1.1, neural network's weights are crucial to the AI system's intelligence and considerable costs are spent on training weights (e.g., collecting the labeled data, running the training algorithm, etc.) when developing an AI system. Later, we will show in Chap. 5 that a neural network's structure information is not needed when recovering its weights. In that sense, this thesis focuses on recovering neural network's weights from side channels, and proposes the first successful weight recovery solution for real-world neural networks.

Neural Network Weights

Similar to the case of input recovery, recovering neural network weights from side channels was previously believed hardly doable (see challenges discussed in Sec. 1.3). Only limited works show successful recovery, but with too strong assumptions.

For instance, some works assume the neural network have secret-dependent computation paradigms (e.g., a neural network prunes its weight for different inputs [113]); however, this is not the case in reality. Some other works assume most weight elements of a neural network are public [36, 263], e.g., only weights in the last layer are secret, so that the attacked (sub) neural network becomes fully linear and attackers can directly calculate the weights by dividing the output with the input. This assumption also does not hold in practice. Some attacks also perform brute-force guesswork [28, 75]. Essentially, they treat side channel observations as "signatures" of neural network weights, and enumerate possible weights to find the best match. However, this approach is not feasible for

real-world neural networks with millions of weight elements. As will be demonstrated in Chap. 5, our weight recovery technique applies to large real-world neural networks like Vision Transformer [72] and works when the full weight elements are private. It also does not require any special computation paradigms of neural networks and is applicable to general neural networks.

2.4 Leakage Analysis Techniques

The leakage analysis has been a long-standing research topic in the security community, and a series of works have been proposed to detect leakages due to cache side channels. As we have discussed in Sec. 1.3.2, most previous works require modeling the leakage patterns, and thus are only applicable to one specific side channel and detect leakage sources exhibiting the modeled patterns. Moreover, they are primarily designed for cryptographic software (e.g., an RSA encryption implement by OpenSSL); extending them to the data processing libraries and neural network runtimes (e.g., DL frameworks or neural network executables) in AI systems is non-trivial and remains unclear.

The majority of previous leakage analysis works focuses on cache side channels due to the high severity and the wide threat that cache side channels bring. Below, we briefly introduce technical details of representative works. More leakage analysis works and their limitations that motivate our research will be discussed in Chap. 6.

CacheAudit [73] denotes one of the earliest leakage analysis works. It leverages abstract interpretation [63] to over-estimate the information leakage of private keys in cryptographic software. Hence, CacheAudit delivers sound analysis: it can rigorously verify that a program is free of cache side channels, but cannot decide whether a program has cache side channel leakage. Given that most real-world programs do not fully eliminate secret-dependent data access and control branches, CacheAudit's application scope is largely limited. The abstract interpretation is also known for its low scalability, such that CacheAudit only supports small programs with a few hundred of cache accesses.

CacheD [244] for the first time supports localizing leakage sources of cache side channels in a program. This improvement is achieved by the trace-based analysis, which trades the soundness guarantee for more precise analysis over each instruction's cache access.

Specifically, given a target program g, CacheD first executes it with one secret k (i.e., a private key) and then monitors secret-related instructions via taint analysis. With the secret k as the taint source, CacheD propagates the taint information along g's execution to mark instructions that use k. Then, for every secret-related memory access, CacheD formulates its accessed cache lines as a symbol constraint (with k as the symbol) and checks whether different cache lines can be accessed with different secrets k. If so, CacheD flags the corresponding instruction as a leakage source of cache side channels.

Despite the promise, CacheD suffers from the coverage issues: since it is traced-based, it can only localize leakage sources covered by the executed trace. To alleviate this issue, CacheS [243] combines the symbolic execution in CacheD with abstract interpretation. And to further improve the low scalability of abstract interpretation, a novel abstract domain named Secret-Augmented Symbolic (SAS) domain is proposed. The SAS domain categorizes program executions as secret-dependent and secret-independent; it only precisely tracks those secret-dependent executions to maintain the precision of analysis. The remaining secret-independent executions are coarsely modeled to improve the scalability. With these optimizations, CacheS for the first time supports nearly sound[3] cache side channel analysis for real-world programs.

Besides localizing leakage sources, one recent work, Abacus [26], also examines how much information is leaked via a program's cache side channels. Abacus is extended based on CacheD, and it leverages the mutual information between each leakage source and the secret to quantify the information leakage. The mutual information reflects how much search space of the secret can be reduced by observing a leakage source's resulting cache side channel observation. In this regard, for each leakage source localized by CacheD, Abacus samples different secrets (i.e., private keys in its scenario) and counts the percentage of secrets that result in the same cache side channel observation — this percentage quantifies the reduced search space and is leveraged to calculate the mutual information.

[3]Its implementation has unsound issues; we refer interested readers to the detailed discussions in the CacheS paper [243]

CHAPTER 3

AUTOMATED SIDE CHANNEL ANALYSIS OF MEDIA SOFTWARE WITH MANIFOLD LEARNING

This chapter presents MANIFOLD-SCA, *a framework that automatically reconstructs confidential media inputs from side channel traces of media software (i.e., data processing libraries adopted in AI systems). It reveals the severe threat that a malicious user can pose to the data privacy of modern AI systems.* MANIFOLD-SCA *is based on manifold learning: it forms the reconstruction of media inputs from side channel traces as cross-modality manifold learning task. This chapter also presents a novel and highly effective defensive scheme towards our attack. The defense, dubbed perception blinding, is motivated by the popular cryptographic blinding and perturbs media inputs with perception masks. The defense can effectively mitigate our attack but brings negligible cost to the normal usage of media software.*

3.1 Introduction

Side channel analysis (SCA) infers program secrets by analyzing the target software's influence on physical computational characteristics, such as the execution time, accessed cache units, and power consumption. Practical SCA attacks have been launched on real-world crypto systems [271, 164, 262] to recover crypto keys. With the adoption of cloud computing and machine learning as a service (MLaaS), media software, a type of application software used for processing media files like images and text, is commonly involved in processing private data uploaded to cloud (e.g., for medical diagnosis). Existing works have exploited media software with extensive manual efforts or reconstruct only certain media data [265, 95, 279]. However, the community lacks a systematic and thorough understanding of SCA attack vectors for media software and of the ways that private user inputs of various types (e.g., images or text) can be reconstructed in a unified and automated manner. Hence, this is the first study toward media software of various input

formats to assess how their inputs, which represent private user data, can be leaked via SCA in a fully automatic way.

Recent advances in representation learning and perceptual learning [33, 289] inspired us to recast SCA of media software as a cross-modality manifold learning task in which an autoencoder [101] is used to learn the mapping between confidential media inputs and the derived side channel traces in an end-to-end manner. The autoencoder framework can learn a low-dimensional joint manifold of media data and side channel observations to capture a highly expressive representation that is generally immune to noise.

Our proposed autoencoder framework is highly flexible. It converts side channel traces into latent representations with an encoder module ϕ_θ, and the media data in image, audio and text formats can be reconstructed by assembling decoders ψ_θ that correspond to various media data formats to ϕ_θ. By enhancing encoder ϕ_θ with attention [260], the autoencoder framework can automatically localize program points that primarily contribute to the reconstruction of media inputs. That is, the attention mechanism delivers a "bug detector" to locate program points at which information can leak.

The observation that manifold learning captures key perceptions of high-dimensional data in a low-dimensional space [289] inspired us to propose the use of *perception blinding* to mitigate manifold learning–based SCA. Well-designed perception blinding "dominates" the projected low-dimensional perceptions and thus confines adversaries to only generate media data perceptually bounded to the mask. In contrast, media software that is typically used to process data bytes of media data experiences no extra difficulty in processing the blinded data and recovering the original outputs.

Our evaluation exploits media software, including `libjpeg` [158], `FFmpeg` [3], and `Hunspell` [4], which process media data in image, audio, and text formats. We assess these media software using a common threat model where *trace-based* attackers [244, 124, 73, 37] can log a trace of CPU cache banks, cache lines, or OS page-table entries accessed when executing the media software. We also launch standard `Prime+Probe` attack [237] in userspace-only scenarios and use the logged cache side channels to reconstruct media data. We conduct qualitative and quantitative evaluations of six datasets that represent daily media data whose user privacy can be violated if leaked to adversaries. Our findings show that user inputs can be reconstructed automatically and that the recovered

media content, such as images or text, shows considerable (visual) similarity to user inputs. The attention modules facilitate localizing program points that incur input leakage; some have been disclosed before [265, 95, 279], but many, to the best of our knowledge, were previously unknown. Further, we find that perception blinding is highly effective in mitigating manifold learning-based SCA. We also demonstrate the noise resiliency of our attack, and how oblivious RAM [86, 225] can mitigate our attack, though it incurs high cost. In summary, the work presented in this chapter makes the following contributions:

- Advances in cross-modality manifold learning inspired us to advocate SCA of media software as a supervised task that learns a joint manifold of media data and side channel traces. High-quality media data can be reconstructed from side channel traces in a noise-resilient manner without knowledge of the underlying media software implementation or media data formats.
- We enhance autoencoder with attention to localize program points that make notable contributions to information leakage. Furthermore, we design a low-cost mitigation technique called perception-blinding that effectively mitigates the proposed SCA exploitation.
- Our evaluation subsumes widely used media software used to process images, audio, and text. We demonstrate that high-quality user inputs in various formats can be reconstructed and that perception blinding predominantly impedes our SCA. Our attention-based error-localization technique confirms some program points that have been reported as vulnerable and flags many previously unknown problems in media software.

Research Artifact. All code and data for this chapter is available at: `https://github.com/Yuanyuan-Yuan/Manifold-SCA` [9].

3.2 Background

We introduce the high-level procedure of launching SCA in which program inputs are assumed confidential. Let a deterministic and terminating program be P. Executing an input $i \in I$ can be modeled as $P : I \rightarrow R$, where R denotes the program behavior during the runtime. Although modern computer architectures prohibit attackers from directly

recording R and inferring input $i \in I$, attackers can leverage various *side channels*, which map the runtime behavior of R into an adversarial observation O of certain properties (e.g., cache status) in the execution context of P. The attacker's view can be represented as $view : R \rightarrow O$, where given side channel observation O, the attackers leverage composite inverse function $(view \circ P)^{-1} : O \rightarrow I$ to map O back to input $i \in I$.

Promising progress has been made by logging (high-resolution) side channels such as accessed cache line, cache bank, or page table entries in an automated manner [55, 262, 164, 265, 95]. Nevertheless, reconstruction of i from logged side channels requires attackers to infer the composite inverse function $(view \circ P)^{-1} : O \rightarrow I$. Recovery of such mappings requires an in-depth understanding of how program secrets are propagated (i.e., secret information flow), which could require considerable manual efforts [265, 95] or conducting formal analysis [244, 73, 37]. Note that high-resolution side channels (e.g., cache line access) usually contain millions of records, but only a tiny portion o^* is indeed *input-dependent* [243].

SCA on Media Software. Despite the widespread adoption of MLaaS to process users' private data, the SCA of media software has not been thoroughly examined. For instance, media software is commonly used to process X-ray images because it allows cost-efficient disease diagnosis with cloud resources. However, the leakage of such images on the cloud (e.g., via cache-based side channels [162]) involves a high risk of violating patient privacy. An immense demand exists to gain insights into the extent of privacy problems in media software, given its pervasive use in processing private data. Therefore, we examine real-world media software used to process media data such as photos and daily conversations.

Threat Model and Attack Scenarios. This study reconstructs confidential inputs of media software from side channels. We thus reasonably assume that different inputs of targeted media software can induce distinguishable memory access traces. Otherwise, no information regarding inputs would be leaked.

Profiled SCA [100, 174, 39, 98, 124, 99] commonly assumes that side channel logs have been prepared for training and data reconstruction. For our scenario, we generally assume a standard *trace-based* attacker. We assume that a trace of system side channel accesses made by the victim software has been prepared for use. Our evaluated system side channels include cache line, cache bank, and page table entries. The feasibility

of logging such fine-grained information has been demonstrated in real-world scenarios [265, 271, 95, 70], and this assumption has been consistently made by many previous works [244, 74, 115, 255, 37, 243]. In this study, we use Intel Pin [170] to log memory access traces and convert them into corresponding side channel traces (see Sec. 3.5).

We also benchmark userspace-only scenarios where attackers can launch `Prime+Probe` attack [237] to log cache activities when media software is processing a secret input. We use Mastik [269], a micro-architectural side channel toolkit, to launch "out-of-the-box" `Prime+Probe` and log victim's L1I and L1D cache activities. We pin victim process and `spy` process on the same CPU core; see attack details in Sec. 3.6.4.

Exploiting new side channels is *not* our focus. We demonstrate our attack over commonly-used side channels. This way, our attack is shown as practically approachable, indicating its high impact and threats under real-world scenarios. Unlike previous SCA on media software [265, 95] or on crypto libraries [284], we do *not* require a "white-box" view (i.e., source code) of victim software. We automatically analyze media software *executables* with different input types. As will be discussed in Sec. 3.6, we launch manifold learning to reconstruct media data with excellent (visual) similarity to user inputs. Many studies have only flagged program points of information leakage with (unscalable) abstract interpretation or symbolic execution [244, 37, 73]. Direct reconstruction of media data is beyond the scope of such formal method-based techniques, and these studies did not propose SCA mitigation.

3.3 A Manifold View on SCA of Media Software

This study recasts the SCA of media software as a cross-modality manifold learning task that can be well addressed with supervised learning. We train an autoencoder [101] that maps side channel observations O to the media inputs I of media software. Our threat model (Sec. 3.2) assumes that attackers can profile the target media software and collect side channel traces derived from many inputs. Therefore, our autoencoder framework is trained to learn from historical data and implicitly forms a low-dimensional joint manifold between the side channel logs and media inputs. We first introduce the concept of manifold, which will help to clarify critical design decisions of our framework (see Sec. 3.4).

Manifold Learning. The use of manifold underlies the feasibility of dimensionality reduction [142]. The key premise of manifold is the *manifold hypothesis*, which states that real-world data in high-dimensional space are concentrated near a low-dimensional manifold [76]. That is, real-world data often lie in a manifold $\mathcal{M}$ of much lower dimensionality d, which is embedded in its high-dimensional space $\mathcal{R}$ of dimensionality D ($d \ll D$). *Manifold learning* aims to find a projection $f : \mathcal{R} \rightarrow \mathcal{M}$ that converts data $x \in \mathcal{R}$ into y in an intrinsic coordinate system of $\mathcal{M}$.[1] f^{-1} projects $f(x) \in \mathcal{M}$ back onto representation x in the high-dimensional space $\mathcal{R}$.

PCA [13] is a linear manifold learning algorithm that aims to find $\mathcal{M}$ by extracting "principal components" of data points [33]. However, most real-world manifolds are nonlinear, and manifold learning algorithms (e.g., ISOMAP) are proposed to project data x onto nonlinear $\mathcal{M}$ [24].

Manifold learning views high-dimensional media data $x \in \mathcal{R}$ as a composite of perceptually meaningful contents that are shown as robust to noise or other input perturbations [77, 290, 289]. Manifold learning algorithms extract expressive representations of high-dimensional data such as images, audio, and text [48, 97], which explains why AI models can make accurate predictions pertaining to high-dimensional data [33]. It is shown that data of the same class (e.g., face photos) generally lie in the same manifold, whereas data of different classes (face vs. vehicle photos) are concentrated on separate manifolds in low-dimensional space [233]. Manifold learning clarifies the inherent difficulty of designing *universal* encoding and generative models applicable to high-dimensional data from different manifolds. The manifold hypothesis has been verified theoretically and empirically [77, 290, 289].

In Fig. 3.1, we project a set of real-world face images to manifold $\mathcal{M}_{img}$ of two dimensions. To draw this projection, we tweak our autoencoder framework (see Sec. 3.4) to convert each face image into a latent representation of two dimensions. We observe that images are generally separated by skin and hair colors; in addition, face orientations are roughly decomposed into two orthogonal directions (green and red lines). In fact, recent studies have shown the effectiveness of conducting image editing by first projecting image

[1] "Intrinsic coordinate" denotes the coordinate system of the low-dimensional manifold space for each high-dimensional data sample [159, 60].

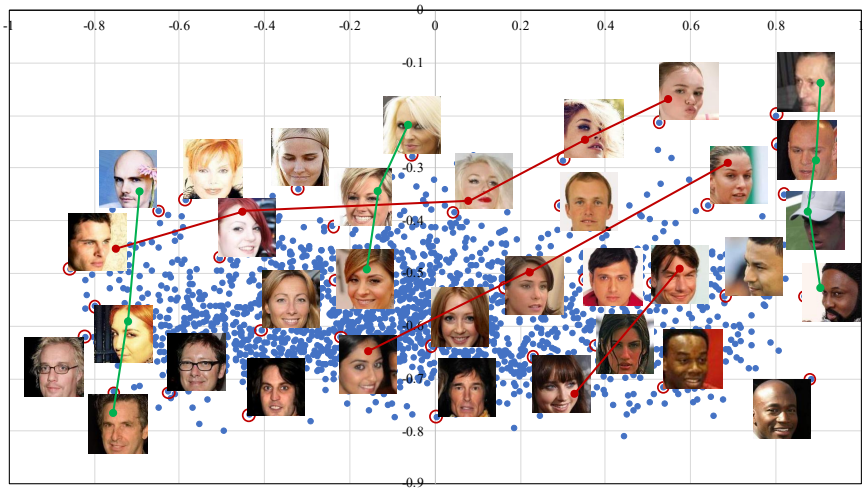


Figure 3.1: Project face photos to a 2-dimensional manifold.

samples into the manifold space [290, 224]. Editing images in the high-dimensional space involves a large search space $[0, 255]$ for every pixel; the random selection of pixel values from $[0, 255]$ struggles to create realistic images because arbitrary editing could "fall off" the manifold of natural images. In contrast, manifold learning facilitates sampling within $\mathcal{M}$, and the perceptually meaningful contents in $\mathcal{M}$ confine manipulations to generate mostly realistic images [290, 224].

Parametric Manifold. Most manifold learning schemes adopt non-parametric approaches. Despite the simplicity, non-parametric approaches cannot be used to project *new* data points in $\mathcal{R}$ onto $\mathcal{M}$. Recent advances in deep neural networks, particularly autoencoders, have enabled a parametric nonlinear manifold projection $f_\theta : \mathcal{R} \rightarrow \mathcal{M}$ [289]. Manifold learning can thus process unknown data points of high-dimensional media data [289, 290, 79, 103, 145, 33] and facilitate downstream tasks like face recognition [76].

High-Level Research Overview

Processing media data has an observable influence on the underlying computing environment; it thus induces side channel traces that can be logged by attackers to infer private inputs. Previous SCA studies, from a holistic view, attempted to (manually) map side channel logs to *data bytes* in media data (similar to how media software treats media data) [265, 95]; reconstruction of media data in a per-pixel manner is thus error-prone and likely requires expertise and manual efforts.

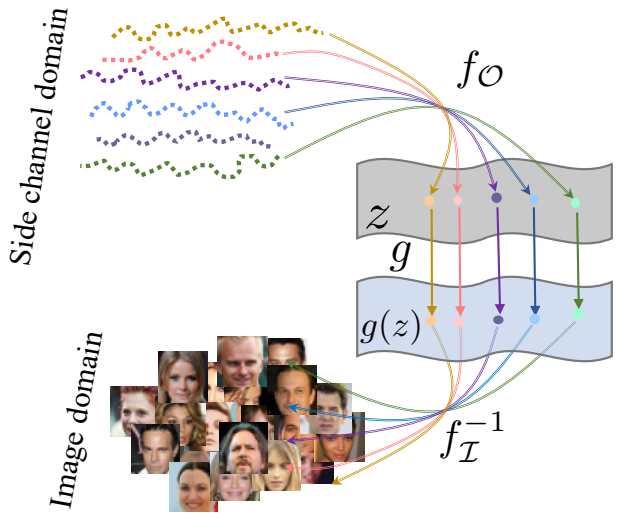


Figure 3.2: Mapping between side channels and images via a low-dimensional joint manifold $\mathcal{M}_{\mathcal{I},\mathcal{O}} = \mathcal{I} \times \mathcal{O}$.

The success of manifold learning in tasks like image editing and cross-modality reconstruction [290, 289] led us to construct a joint, *perception-level* connection between side channel logs and high-dimensional media inputs.[2] Therefore, instead of deciding value of each byte, reconstructing media data is recast into exploring the manifold of media data that satisfies the perception-level combinatory constraints.

Overall, we view SCA as a cross-modality high-dimensional data reconstruction task that is addressed with joint manifold learning in this work [289]. Aligned with the notations in Sec. 3.2, let the media software inputs be I; the attacker's observation on executing each input $i \in I$ can be represented as $(view \circ P) : I \to O$, where $o \in O$ denotes the observation of side channel traces. Let F be the composite function $view \circ P$. According to the manifold hypothesis, we assume that I and O also lie in the unknown manifolds $\mathcal{I}$ and $\mathcal{O}$, respectively. As mentioned in our threat model (Sec. 3.2), we assume that side channel observations depend on the inputs of media software; therefore, the entire joint dataset $\{i_i, o_i\}$ formed by the ith media input $i_i \in I$ and the corresponding ith observation $o_i \in O$ lies in a joint manifold

$$\mathcal{M}_{\mathcal{I},\mathcal{O}} = \{(i, F(i)) | i \in \mathcal{I}, F(i) \in \mathcal{O}\}$$

[2]Perception-level connection means constraints on data bytes formed by perceptual contents (e.g., gender, hair style) in media data are extracted from side channels.

where $(i, F(i))$ is described with the regular high-dimensional coordinate system. Since I and O also lie in the corresponding manifolds $\mathcal{I}$ and $\mathcal{O}$, the data points in $\mathcal{M}_{\mathcal{I},\mathcal{O}}$ should be equivalently described using an intrinsic coordinate system $(z, g(z))$. Hence, we assume the existence of a homomorphic mapping $(f_{\mathcal{I}}, f_{\mathcal{O}})$ over $(z, g(z))$ such that $z = f_{\mathcal{O}}(o)$ and $g(z) = f_{\mathcal{I}} \circ F^{-1}(o)$. $f_{\mathcal{O}}$ maps side channel observation $\mathcal{O}$ onto the intrinsic coordinate z, whereas $f_{\mathcal{I}}$ maps high-dimensional media data I onto $g(z)$. Note that g denotes the diffeomorphism (i.e., an isomorphism of two manifolds) between the $\mathcal{I}$ and $\mathcal{O}$ manifolds [289]. Hence, instead of computing $F^{-1} = (view \circ P)^{-1} : O \rightarrow I$ to map the side channel observation back onto the media inputs, we leverage the joint manifold to constitute the following composite function:

$$F^{-1}(o) = f_{\mathcal{I}}^{-1} \circ g \circ f_{\mathcal{O}}(o) \tag{3.1}$$

$i \in I$ can thus be reconstructed using the inverse composite function $f_{\mathcal{I}}^{-1} \circ g \circ f_{\mathcal{O}}$ over the joint manifold $\mathcal{M}_{\mathcal{I},\mathcal{O}}$. Fig. 3.2 provides a summary and presents a schematic view of how I and O of high-dimensional data are mapped via $\mathcal{M}_{\mathcal{I},\mathcal{O}}$.

The feasibility of using neural networks, especially autoencoders, to facilitate parametric manifold learning has been discussed [289, 290, 103, 177]. Accordingly, we train an autoencoder by encoding side channel traces O onto the latent space with encoder ϕ_θ and by generating media data I with decoder ψ_θ from the latent space. Therefore, Eq. 3.1 can be learned in an end-to-end manner [289, 33]. Holistically, ϕ_θ and ψ_θ correspond to $f_{\mathcal{O}}$ and $f_{\mathcal{I}}^{-1}$, respectively, whereas g is implicitly constructed in the encoded latent space.

3.4 Framework Design

We describe the design of our autoencoder in Sec. 3.4.1. Sec. 3.4.2 clarifies the usage of attention to localize code fragments inducing information leakage. Sec. 3.4.3 introduces perception blinding to mitigate our SCA.

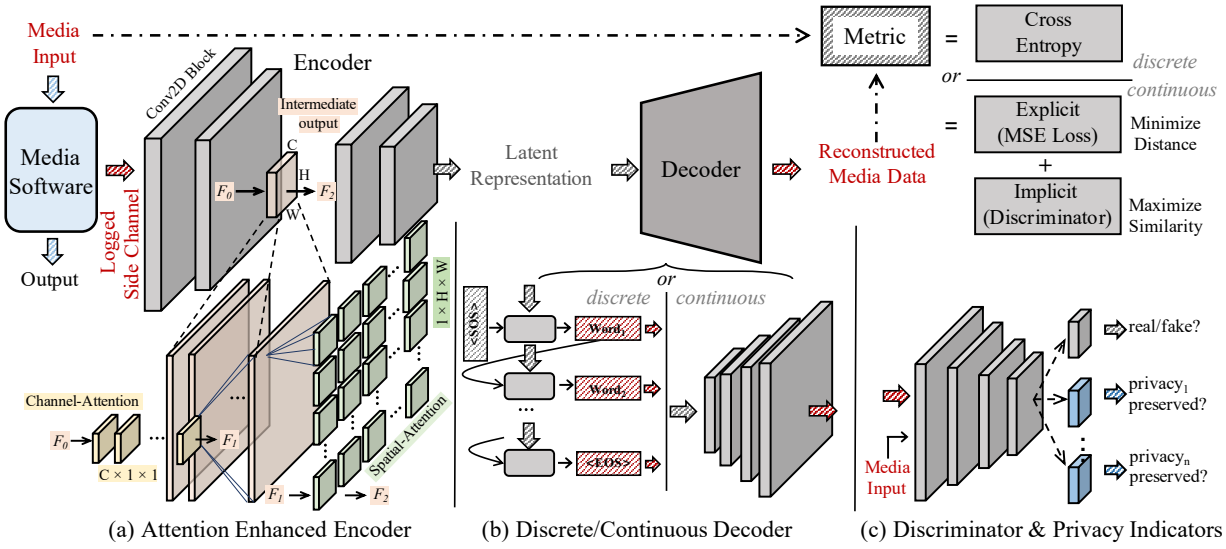


Figure 3.3: Reconstructing media data of different types with a unified autoencoder framework.

3.4.1 SCA with Autoencoder

We propose a general and highly-flexible design in which an autoencoder is used to facilitate SCA of various media data, including images, audio and text. The autoencoder framework [101] defines a parametric feature-extracting function f_θ, named *encoder*, that enables the projection of the input x onto a latent vector $h = \phi_\theta(x)$. Similarly, autoencoder frameworks also use ψ_θ as a *decoder* that reconstructs input $\hat{x}$ from a latent vector $\hat{x} = \psi_\theta(h)$. A well-trained autoencoder framework gradually identifies a parameter vector θ to minimize the reconstruction error as follows:

$$L(\theta) = \sum_t L(x_t, \psi_\theta \circ \phi_\theta(x_t))$$

where x_t is a training sample. Minimal errors can be found with statistical methods like stochastic gradient descent.

The first row of Fig. 3.3 depicts the workflow. We clarify that our focus is *not* to propose novel model architectures; rather, we show that high-quality inputs can be synthesized by assembling standard models, which indicates severity and effectiveness of our attack. We now discuss the high-level workflow and present the model structures and training details in Sec. 3.5. Given a logged side channel trace $o \in O$, encoder $\phi_\theta(o)$ converts o

into the corresponding latent representation. We prepare three decoders ψ_θ^i, ψ_θ^a, ψ_θ^t to reconstruct these types of media data (i.e., image, audio, and text) from the encoded latent representation. We pair encoder $\phi_\theta(o)$ with each ψ_θ^* and train the assembled pipeline for our customized objective functions $L(\theta)$. Our proposed framework is task-agnostic. Generating media data of various types requires only assembling corresponding decoders to the unified encoder ϕ_θ.

Encoder ϕ_θ. A logged side channel trace will first be folded into a $K \times N \times N$ matrix (see Table 3.3 for the detailed configuration of each trace). We then feed this matrix as the input of encoder ϕ_θ. The encoder ϕ_θ comprises several stacked 2D convolutional neural networks (CNNs). For the current implementation, ϕ_θ converts the high-dimensional inputs into latent vectors of 128 dimensions, given that the dimensions of our media inputs are all over 10K. Moreover, we find that increasing the dimension of latent vectors (i.e., from 128 to 256) does not make an observable improvement. This observation is consistent with the manifold hypothesis [142], such that only *limited* "perceptions" exist in normal media data. In contrast, reducing the number of dimensions (e.g., 32) makes the outputs (visually) much worse. However, users who strive to recover media data of lower-dimensions can configure our framework with smaller latent vectors (e.g., 32 dimensions).

Fig. 2(a) shows that we enhance encoder ϕ_θ with attention. Indeed, we insert one attention module between every two stacked CNN layers in the encoder. Attention generally improves output quality of autoencoder [240]. More importantly, attention facilitates localizing program points of information leakage. We elaborate on Fig. 2(a) in Sec. 3.4.2.

Decoder ψ_θ^*. We categorize the media data exploited by this study into two types: continuous and discrete. Image and audio data are represented as a continuous floating-point matrix and reconstructed by ψ_θ^i and ψ_θ^a in a continuous manner. In contrast, textual data comprise word sequences, and because there is no "intermediate word," textual data are regarded as sequences of discrete values and handled by ψ_θ^t.

As shown in Fig. 2(b), we use a common approach to stacking 2D CNNs to design ψ_θ^i. A 2D CNN has several convolutional kernels; each kernel focuses on one feature dimension of its input and captures the spatial information of this feature dimension. Images can thus be reconstructed from vectors in the low-dimensional latent space with stacked 2D CNNs, as each 2D CNN upsamples from the output of the previous layer. For audio data,

we first convert raw audio into the log-amplitude of Mel spectrum (LMS), a common 2D representation of audio data. This way, audio data are represented as 2D images (see some examples in [9]), in which the x-axis denotes time and the y-axis denotes the log scale of amplitudes at different frequencies. Herein, like ψ^i_θ, ψ^a_θ uses stacked 2D CNNs to process each converted 2D image, gradually upsamples from the latent representation, and reconstruct the LMSs of the audio data. Because the LMSs usually are not in square-shape, we append a fully connected layer to transform the shape of the reconstructed LMSs. These LMSs are then converted to raw audio losslessly.

Textual data, however, are reconstructed sequentially "word by word" due to their discrete nature. As shown in Fig. 2(b), to reconstruct a sentence from the latent space of a side channel trace o, a single word is gradually inferred based on words already inferred from sentence i. Following a common practice of training sequence-to-sequence autoencoders, we add a start-of-sequence (SOS) token before each sentence i and an end-of-sequence (EOS) token after i. Then, given a side channel trace o that corresponds to unknown text i, the trained decoder ψ^t_θ starts from the SOS token and predicts a word $w \in i$ sequentially until it yields the EOS token. From a holistic perspective, the trained model projects a sentence i into a low-dimensional manifold space of *word dependency*, which facilitates the gradual inference of each word w on i.

Designing Objective Functions. As depicted in the first row of Fig. 3.3, for discrete data (i.e., text), each decode step is a multi-class classification task where the output is classified as one element in a pre-defined dictionary. Thus, we use *cross entropy* as the training objective. For continuous data, we design the training objective L_θ *composing both explicit and implicit metrics*. We now introduce each component in detail.

Explicit Metrics A common practice in training an autoencoder is to explicitly assess the point-wise distance between the reconstructed media input i' and reference input i with metrics such as MSE loss, L1 loss, and KL divergence [136, 54]. The autoencoder will be guided to gradually minimize the point-wise distance $L_\theta(i, \psi_\theta \circ \phi_\theta(o))$ during training. Nevertheless, a major drawback of such explicit metrics is that the loss of each data point is calculated *independently* and contributes *equally* to update θ and minimize L_θ. Our preliminary study [9] shows that such explicit metrics suffer from "over-smoothing" [212], a well-known problem that leads to quality degradation of the reconstructed data.

Table 3.1: Privacy-aware indicators. Table 3.3 introduces each dataset.

Dataset	Indicator
CelebA	Is the celebrity's identity preserved?
ChestX-ray	Is the disease information preserved?
SC09	Is the speaker's identity preserved?
Sub-URMP	Is the musical instrument's type preserved?

Implicit Metrics Another popular approach is to assess the "distributed similarity" [212] of reconstructed i' and reference input i. Viewing the general difficulty of extracting the distribution of arbitrary media data, a common practice is to leverage a neural discriminator D. Discriminator D and decoder ψ_θ play a zero-sum game, in which D aims to distinguish the reconstructed input i' from normal media data i. In contrast, decoder ψ_θ tries to make its output i' indistinguishable with i to fool D. Although this paradigm generally alleviates the obstacle of "over-smoothing" [212], it creates the new challenge of mode collapse; that is, ψ_θ generates realistic (albeit very limited) i' from any inference inputs. From a holistic perspective, the use of a discriminator mainly ensures that the reconstructed i' is near i from a *distribution* perspective; no guarantee is provided from the view of a single data point.

Privacy-Aware Indicators In addition to the two standard objective functions mentioned above, we further take into account a set of *privacy-aware indicators*. As shown in Fig. 2(c), we extend discriminator D such that it checks whether the reconstructed outputs preserve the "privacy" in an explicit manner. Table 3.1 lists the privacy indicators used in our framework, which correspond to exploited media data of different types. For instance, for face photos (CelebA), we specify checking the identity. Hence, the enhanced discriminator D serves as a classifier to check whether the identity of the person is preserved, which thus forces decoder ψ_θ to decode the identity information in the zero-sum game. A specific fully connected layer is appended to the discriminator D in accordance with each privacy indicator.

Comparison with Generative Model-Based SCA [279]. One contemporary study [279] uses generative models (e.g., GANs [88]) to conduct SCA towards image libraries by capturing image distribution from side channels. Nevertheless, their work is particularly designed to recover images instead of proposing a general and flexible framework to exploit media software of various input types. In addition, we explicitly use privacy indicators

when designing objective functions, while [279] focuses on polishing the *visual appearance* of the reconstructed images.

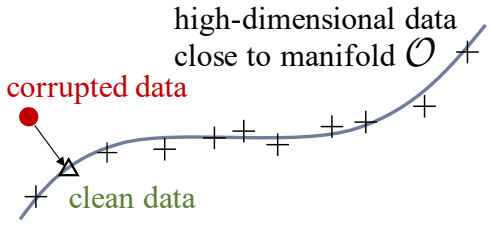


Figure 3.4: Denoising corrupted data during manifold learning.

Noisy Side Channel. Reconstructing media data from noisy side channels is of particular importance, because adversaries often face considerable noise in real-world attack scenarios. Manifold learning features denoising by design, the schematic view of which is presented in Fig. 3.4. Overall, manifold learning forces side channel traces O to concentrate near the learned low-dimensional manifold $\mathcal{O}$, where a corrupted high-dimensional data point $\tilde{o}$ (● in Fig. 3.4) should typically remain *orthogonal* to the manifold $\mathcal{O}$ [33]. Thus, when the decoder ψ_θ learns to reconstruct media data $i \in I$ from the representations lying on the joint manifold, corrupted $\tilde{o}$ can be fixed by first being projected onto the Δ in the manifold for denoising; and i can then be reconstructed from the Δ [103].

3.4.2 Fault Localization with Neural Attention

Some studies have detected software vulnerabilities that lead to side channel attacks [73, 244, 37, 243]. However, we note that such studies typically use heavyweight program-analysis techniques, such as abstract interpretation, symbolic execution, and constraint solving. Thus, performing scalable program analysis of real-world media software could prove a great challenge, given that such media software usually contains complex program structures (e.g., nested loops) and a large code base. Furthermore, the primary focus of previous studies has been crypto libraries (e.g., OpenSSL [192]), whose "sensitive data" are private key bytes or random numbers. In contrast, modeling potentially lengthy media data with various strictly defined formats could impose a further challenge (e.g., symbolizing such complex input formats) that may require the incorporation of domain-specific knowledge.

Inspired by advances in program neural smoothing [222, 221] and SCA based on neural networks [279, 201, 124], we seek to overcome question "which program point leaks side channel information" by answering the following question:

> "Which records on a logged side channel trace contribute most to the reconstruction of media data?"

Although answering the former question often requires rigorous and unscalable static analysis, the second question can be addressed smoothly by extending the encoder ϕ_θ with *attention* [260], a well-established mechanism that improves the representation of interest by telling the neural network where and upon what to focus. In particular, by enhancing the autoencoder with attention, our framework *automatically* flags side channel logs that make a primary contribution to input reconstruction. These logs are *automatically* mapped to the corresponding memory access instructions. We can then *manually* identify the corresponding "buggy" source code. For the last step, our current experiments rely on symbol information in the assembly programs to first identify corresponding functions in source code, and then narrow down to code fragments inducing input leakage.

Despite attention being a standard mechanism to boost deep learning models [240, 260], attention in our new scenario acts like a "bug detector" to principally ease localizing vulnerable program points. In contrast to program analysis-based approaches [73, 243, 244, 37], our solution is highly scalable and incurs no extra cost during exploitation. Moreover, it analyzes software in a black-box setting that is agnostic to media software implementation details or input formats.

Fig. 2(a) depicts the enhanced trace encoder with attention. An attention module (we follow the design in [260] given its simplicity and efficiency) is inserted within every two stacked CNN layers. Let the intermediate input of a CNN layer as $C \times H \times W$, the "Channel-Attention" module $A_{channel}$ processes each segment of $1 \times H \times W$ data points from C channels and tells the encoder "where" to focus on by assigning different weights to each segment. The "Spatial-Attention" module $A_{spatial}$ processes each segment of $C \times 1 \times 1$ records and advises the encoder "what to locate" by assigning different weight on each record. From a holistic perspective, attention module $A_{channel}$ projects a

coarse-grained focus on potentially interesting segments, while $A_{spatial}$ further identifies interesting side channel records in a segment.[3]

3.4.3 Mitigation with Perception Blinding

This section presents mitigation against manifold learning–enabled SCA (Sec. 3.4.1). Consistent with our attack and fault-localization, mitigation is also agnostic to particular media software and input types. We only need to perturb the media input I with pre-defined perception blinding masks.

We first introduce blinding images of the human face, and then explain how to extend perception blinding toward other input types.

A Working Example. As introduced in Sec. 3.2, manifold learning casts images of the human face into a set of perceptually meaningful representations; typical representations include hair style, age, and skin color. Hence, we define a *universal* mask i_{mask} of human face, such that by perturbing arbitrary images i of human face with i_{mask}, the produced output $i_{blinded}$ will be primarily projected to the same intrinsic coordinates z_{mask} in the manifold space $\mathcal{M}$. To use perception blinding, users only need to pick *one* mask i_{mask} to blind all input images i. Consequently, adversaries are restricted to the generation of media data perceptually correlated to z_{mask}. Particularly, to perturb i, we add i_{mask} as follows:

$$i_{blinded} = \alpha \times i \oplus \beta \times i_{mask}$$

where we require $\beta \gg \alpha$ and $\alpha + \beta = 1$. Perceptual contents of i_{mask} thus "dominates" the projected low-dimensional perceptions in $\mathcal{M}$. Let $P(i_{blinded})$ be the output of media software after processing $i_{blinded}$, and the user can recover the desired output by subtracting $P(i_{mask})$ from the output as follows:

$$P(i_{private}) = \frac{1}{\alpha} \times (P(i_{blinded}) \ominus \beta \times P(i_{mask}))$$

[3] It is well accepted that a CNN is organized in the form of `num_channels` × `width` × `height`. Therefore, we name two attention components as $A_{channel}$ and $A_{spatial}$, which are aligned with the convention.

Table 3.2: Side channels derived from a memory access made by victim media software using address *addr*.

Side Channel Name	Side Channel Record Calculation
CPU Cache Bank Index [271]	$addr \gg L$ where L, denoting cache bank size, is usually 2 on modern computer architectures.
CPU Cache Line Index	$addr \gg L$ where L, denoting cache line size, is usually 6 on modern computer architectures.
OS Page Table Index [95]	$addr$ & $(\sim M)$ where M, denoting `PAGE_MASK`, is usually 4095 on modern computer architectures.

Table 3.3: Statistics of side channel traces and media software. There is *no* overlapping between training and testing data.

Dataset	Information	Training Split	Testing Split	Trace Length	Matrix Encoding	Media Software
CelebA [168]	Large-scale celebrity face photos	162,770	19,962	$338,123 \pm 14,264$	$6 \times 256 \times 256$	`libjpeg` (ver. 2.0.6)
ChestX-ray [246]	Hospital-scale chest X-ray images	86,524	25,596	$329,155 \pm 10,186$	$6 \times 256 \times 256$	LOC: 103,273
SC09 [249]	Human voice of saying number 0–9	18,620	2,552	$1,835,067 \pm 103,328$	$8 \times 512 \times 512$	`ffmpeg` (ver. 4.3)
Sub-URMP [144]	Sound clips of 13 instruments	71,230	9,575	$1,678,485 \pm 36,122$	$8 \times 512 \times 512$	LOC: 1,236,079
COCO [160]	Image captions	414,113	202,654	$77,796 \pm 14$	$6 \times 128 \times 128$	`hunspell` (vers. 1.7.0)
DailyDialog [155]	Sentences of daily chats	11,118	1,000	$77,799 \pm 102$	$6 \times 128 \times 128$	LOC: 39,096

where $P(i_{private})$ is the desired output, and $P(i_{mask})$ can be pre-computed. $\oplus$ and $\ominus$ directly operate $i \in I$ of various formats, as will be defined later in this section. Because typical operations of media software (e.g., compression) are *independent* of the perceptual meaning of media inputs, the proposed blinding scheme introduces no extra hurdle for media software. In contrast, as shown in Sec. 3.6.3, SCA based on manifold learning can be mitigated in a highly effective manner.

Requirement of i_{mask}. Comparable to how RSA blinding is used to mitigate timing channels [38], perception blinding is specifically designed to mitigate manifold learning-based SCA. We require that i_{mask} must lie in the same low-dimensional manifold with the private data. Thus, i_{mask} must manifest high **perception correlation** with media software inputs $i_{private} \in I$. This shall generally ensure two properties: 1) the privacy (in terms of certain perceptions, such as gender and skin color) in $i_{private}$ can be successfully "covered" by i_{mask}, and 2) i_{mask} imposes nearly no information loss on recovering $P(i_{private})$ from $P(i_{blinded})$ except a mild computational cost due to mask operations. Considering Fig. 3.4, when violating this requirement of *perception correlation*, for instance, such as by using random noise to craft i_{mask}, the intrinsic coordinate of the original input (Δ) can likely drift to a "corrupted input" ($\bullet$) that is mostly orthogonal to the manifold of I. As explained in Sec. 3.4.1, due to the inherent noise resilience of manifold learning, crafting such a corrupted input can cause less challenge to attackers when recovering i from the low-level manifold space. Although $i_{private}$ is of low weight in $i_{blinded}$, it can still be reconstructed to some extent, as will be shown in Fig. 3.8 of Sec. 3.6.3.

Extension to other data types. For image and audio data, we recommend using a normal image $i \in I$ as the mask i_{mask}. Intuitively, by amplifying i_{mask} with a large coefficient β in generating $i_{blinded}$, i_{mask} is presumed to dominate the perceptual features in $i_{private}$. Hence, we stealthily hide the private perceptual features of $i_{private}$ in $i_{blinded}$, whose contents are difficult for adversaries to disentangle without knowing i_{mask}. For textual data, we recommend inserting notional words of high frequency to blind $i_{private}$. We present empirical results on how various choices of i_{mask} can influence the mitigation effectiveness in Sec. 3.6.3.

Implementation of Operators $\oplus$ And $\ominus$. For image and audio data, we use floating-point number addition and subtraction to implement $\oplus$ and $\ominus$. Textual data are discrete: considering that media software often manipulates textual data at the word level, simply "adding" or altering words in the input text will likely trigger some error handling routines of the corresponding media software, which is not desirable. Sec. 3.4.1 clarifies that our autoencoder framework essentially captures the "word dependency" between words in a sentence; accordingly, we define the $\oplus$ operation as inserting words in a sentence, whereas the $\ominus$ operation is implemented to remove previously inserted words. As shown in Sec. 3.6.3, this strategy effectively breaks the word dependency in the original text.

3.5 Attack Setup

We leverage three high-resolution side channels, as shown in Table 3.2. As clarified in our threat model (Sec. 3.2), these side channel are commonly adopted in many existing works in this field. We clarify that exploiting new side channels is *not* our focus. We use common side channels in the era of cloud computing, implying the severity and effectiveness of our proposed attack. The resolution when performing attacks on those side channels are 4B, 64B, and 4096B, respectively. Higher-resolution side channels should enable recovering media data with more vivid details. Media data of better quality, however, does not necessarily enhance privacy stealing (e.g., determining whether chest X-Ray images indicate pneumonia). See quantitative evaluation of privacy inference in Sec. 3.6.1.

For evaluation in Sec. 3.6, we use `Pin` [170] to collect memory access traces and map each trace into three side channel traces following mapping rules in Table 3.2. Sec. 3.6.4

further demonstrates attack in an userspace-only scenario, i.e., we collect cache side channels via `Prime+Probe` [237].

Media Software and Media Dataset. Table 3.3 reports evaluated media software and statistics of side channel traces. We pick media software following previous works [265, 95, 279]. All media software are complex real-world software, e.g., `FFmpeg` contains 1M LOC. We prepare two common datasets for each media software to comprehensively evaluate our attack. All datasets contain daily media data that, once exposed to adversaries, would result in privacy leakage. We compile all three media software into 64-bit binary code using `gcc` on a 64-bit Ubuntu 18.04 machine.

Implementation

We implement our framework in PyTorch (ver. 1.4.0). We use the Adam optimizer with learning rate as 0.0002 for all models. Batch size is 64. For continuous decoders, we set the loss function as $\lambda L_{exlicit} + L_{implicit} + \sum_{i=1}^{n} L_{privacy}$, where $\lambda = 50$ and n is the number of privacy-aware indicators. We ran experiments on Intel Xeon CPU E5-2683 with 256 GB RAM and one Nvidia GeForce RTX 2080 GPU. For experiments based on Pin-logged traces (Table 3.3), the training is completed at 100 epochs and takes less than 24 hours. For experiments using `Prime+Probe`-logged traces, training takes 9 to 27 hours (after excluding preprocessing time; see details in [9]). Table 3.3 reports the dataset size and training/testing splits. See our released codebase [9] for result verification.

3.6 Evaluation

We present the SCA exploitation toward media software in Sec. 3.6.1. We discuss program points that induce information leakage in Sec. 3.6.2, and demonstrate the effectiveness of perception blinding in Sec. 3.6.3.

3.6.1 Side Channel Attack

This section reports key evaluation results. Full setups and evaluation results are present in our artifact [9].

Qualitative Evaluation

This section presents and compares the reconstructed media data with the reference inputs in terms of various settings. Fig. 3.5 demonstrates that the reconstructed images and the references show highly aligned visual appearances, including gender, eyebrow shapes, skin color, and hair styles. Images constructed from different side channels manifest comparable visual quality. Fig. 3.6 further reports the text reconstruction results of daily dialogs by comparison with the reference inputs. The reconstructed sentences, although are not fully aligned with the reference, still retain considerable correct contents and the original intents.

Figure 3.5: Qualitative evaluation of CelebA.

Reconstructed Text	Reference Input
*I think it **'** be better **for find** a good babysitter here . It **' be cost , an** or three days .*	*I think it would be better to have a good babysitter here . It might even be for two or three days .*
*She **is** a **single** cold , and **it** don ' t want to take **care to** us . But we don ' t like **how** can stay with our .*	*She has a bad cold , and we don ' t want to take her with us . But we don ' t know who can stay with her .*
*I ' m sorry **, say that** . What ' s wrong with her ?*	*I ' m sorry to hear it . What ' s wrong with her ?*
*I <UNK>' t want to insult Jill or her **brother** . I think Jill **,** could be it . But I ' **ll** rather have some **to** little older .*	*I don ' t want to insult Jill or her mother . I think Jill maybe could do it . But I ' d rather have someone a little older .*

Figure 3.6: Qualitative evaluation of DailyDialog. We mark ***inconsistent reconstructions***.

We interpret the overall qualitative evaluation results, in terms of images and text, as highly encouraging. We present reconstructed chest X-ray images, sub-URMP/SC09

Table 3.4: CelebA face image matching evaluation.

	Cache bank	**Cache line**	**Page table**
same face	45.4%	43.5%	44.5%
non-face	2.0%	2.0%	2.1%

Table 3.5: SC09 human voice matching evaluation.

	Cache bank	**Cache line**	**Page table**
ID accuracy	29.1%	28.8%	23.2%
Content accuracy	21.6%	24.2%	22.6%

Table 3.6: Text data inference evaluation.

Dataset	**Cache bank**	**Cache line**	**Page table**	**Baseline**
COCO Caption	43.4%	42.6%	42.1%	0.0000%
Daily Diolgue	38.1%	37.4%	37.6%	0.0183%

audio data, and COCO text at [9]. Encouraging results can be consistently observed.

Quantitative Evaluation

Image Data. For CelebA, we leverage commercial face recognition APIs, Face++ [11], to decide whether a reconstructed face and its reference input can be considered as from the *same person* with over 99.9% confidence scores. We thus launch a de-anonymization attack of user identity with reconstructed images. Table 3.4 reports the evaluation results; for all three exploited side channels, more than 43% of the reconstructed faces can be correctly matched to their reference inputs, showing a high success rate of face matching. Only 2% of the reconstructed images are deemed as "non-face," which indicates the negligible chance of generating corrupted faces. We report the quantitative evaluation of chest X-ray in [9].

Our attack achieves plausible accuracy. The quantitative results are *not* noticeably affected by differences in side channels, which indicates that face matching evaluation extracts representative attributes from images for matching. As mentioned in Sec. 3.5, the three side channels manifest different resolutions: although higher-resolution side channels enable reconstruction of more vivid images, this does not necessarily promote privacy stealing. However, enabled by manifold learning-based autoencoder and our objective functions which explicitly account for privacy indicators (Table 3.1), privacy-related fac-

Table 3.7: Localized program points in `libjpeg`.

Module	#Functions	Frequency	Sample Func. Names
MCU	2	7,060	`encode_mcu_huff` `decode_mcu`
Transform	1	5866	`jtransform_execute_transform`
IDCT	13	4,027	`jpeg_idct_15x15` `jsimd_idct_ifas`
Upsample	15	2,033	`h2v1_merged_upsample` `h2v1_fancy_upsample`
Decompress	6	1,352	`tjDecompress2` `tjDecompressHeader3`
Dump	4	8,23	`write_bmp_header` `start_input_bmp`

tors are extracted in the reconstructed images across side channels of various resolutions. Similar observations are made for media data of other formats; our discussion follows.

Audio Data. Table 3.5 reports the voice matching results for SC09. Using the reconstructed voice commands (number 0–9), we train two classifiers for speaker identity and command 0–9 classification.[4] The evaluation results largely outperform the baseline (i.e., random guessing). With a total of 184 speakers, we achieve greater than 20% accuracy in matching correct speaker identities across all settings. We also exceed 20% accuracy in content matching (0–9). We observed decreasing accuracy in speaker identity matching, which is reasonable given that the cache bank side channel only "kicks off" two least significant bits, while cache line and page table side channels retain less amount of information. We also report the matching rate of musical instruments in [9], which yields consistent and promising findings.

Text Data. To reconstruct text data, we gradually predict each word based on previously predicted words in the sentence. Hence, for the quantitative evaluation, we adopt an attack strategy mostly aligned with [41] to measure the average accuracy of word-level prediction accuracy. Table 3.6 reports the evaluation results for the COCO and Dailydialog datasets. To prepare a baseline for comparison, we feed a random input to the decoder ψ_θ instead of using the latent vector of an input side channel trace. As expected, our exploitation of both datasets achieves much greater accuracy than the baseline regarding all side channels.

[4]Please refer to [9] for details of these classifiers.

```
1  static void idct32(int *coeffs, int col_limit) {
2    int limit = min(H, col_limit + 4);
3    for (int i = 0; i < H; i++)
4      TR_32(src, src, H, H, limit);
5  }
6  static void TR_32(int *dst, int *src, int dstep,
7                    int sstep, int end) {
8    int o_32[16] = { 0 };
9    for (int i = 0; i < 16; i++)
10     // loading pre-calculated matrix ''transform''
11     for (int j = 1; j < end; j += 2)
12       o_32[i] += transform[j][i] * src[j * sstep];
13   // TR_16 calls TR_8, and TR_8 calls TR_4.
14   TR_16(e_32, src, 1, 2 * sstep, SET, end/2);
15 }
```

Figure 3.7: Vulnerable code components in `FFmpeg`. We mark variables depending on `FFmpeg`'s input in red, and **bold** input-dependent memory accesses (line 12).

Table 3.8: Localized program points in `FFmpeg`.

Module	#Functions	Frequency	Sample Func. Names
Encode	50+	10K+	`encode_frame`
Decode	50+	10K+	`decode_frame`
Filter	50+	10K+	`filter_frame`
IDCT	10+	5K+	`idct_idct_32x32_add_ce` `iadst_idct_16x16_add_c`
Dump	10+	5K+	`wav_write_trailer` `wav_write_header`

3.6.2 Program Point Localization

We present a representative buggy code fragment of `FFmpeg` in Fig. 3.7. We present representative buggy code localized in `libjpeg` and `Hunspell` in [9]. We also list **all** localized program points in term of assembly code in [9].

`libjpeg`. We analyze 2,000 media inputs from the CelebA and Chest X-ray datasets. Table 3.7 reports the localization results of `libjpeg`. For instance, we identify 7,060 side channel points from 4,000 traces, which can be mapped back to two functions (`encode_mcu_huff` and `decode_mcu`) performing minimum coded unit (MCU)-related operations. Similarly, we find information leakage points in modules related to decompression, IDCT, and also output dumping.

Table 3.9: Localized program points in `Hunspell`.

Module	#Functions	Frequency	Sample Func. Names
Interface	1	1,333	`pipe_interface`
Parser	4	1,230	`next_token`, `alloc_token` `get_parser`, `TextParser::init`
Look up	2	213	`check`, `insertion_sort`
Insert	5	1,076	`putdic`, `allocate_string` `chenc`, `allocate_char_vector`

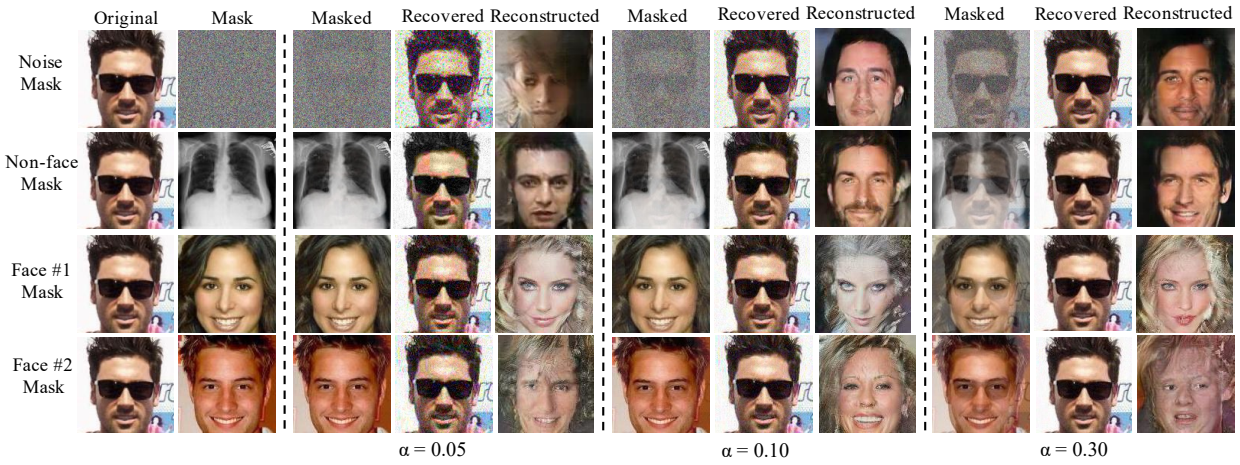


Figure 3.8: Qualitative evaluation results of perception blinding.

Table 3.3 shows that *one* trace has approximately 400K data points. In other words, Table 3.7 reveals that a tiny portion of "informative" points on a side channel trace make a primary contribution to information reconstruction. Given an image compressed in JPEG, `libjpeg` decompresses the image into a bitmap. It is pointed out that the decompression process introduces side channels. IDCT-related functions that were noted by [265] are automatically re-discovered by us. In addition, we identify functions in other image transformation routines (e.g., MCU, upsampling) and output dumping routines that leak inputs. We manually inspected the corresponding implementation of `libjpeg` and confirmed our findings. Note that our approach is automated and treats the entire `libjpeg` software as a blackbox, whereas previous studies [265, 95, 141] could rely on expert knowledge to first localize the vulnerable program points before launching SCA.

`FFmpeg`. We use 2,000 inputs from SC09 and Sub-URMP as the inputs of `FFmpeg`. Our findings, as reported in Table 3.8, can be mapped to five modules of `FFmpeg`, each of which contains many functions. `FFmpeg` processes audio inputs with audio sampling

(the sampling frequency is set as 4,000 in our experiments). Fig. 3.7 presents a vulnerable program component in the IDCT module of `FFmpeg` where the input "taints" certain variables (marked in red) and eventually influences the memory accesses at line 12. Given different input values, different memory cells are visited at line 12, resulting in access to different cache units or page table entries. In addition, `TR_32`, `TR_8` and `TR_4` also suffer from similar patterns (line 14). To the best of our knowledge, side channel issues on `FFmpeg` are rarely studied, but the findings in `FFmpeg` are conceptually similar to other media software; for example, IDCT algorithms and output dumping in both `FFmpeg` and `libjpeg` are flagged as vulnerable by us.

`Hunspell`. We use 2,000 inputs from each dataset (i.e., DailyDialog and COCO) to run `Hunspell` and analyze the logged side channels. Table 3.9 reports the results, where information leakage points are found from the interface, parser, and also spell checking . In fact, previous works [265, 141] have pointed out side channel issues of `Hunspell`. `Hunspell` performs spell check, where a dictionary of words is maintained as a hash table. `Hunspell` iterates each word w in the input sentence to check if w is in the hash table, thus deciding the correctness of its spelling. When checking each w, `Hunspell` computes the hash value of w and looks up the corresponding hash bucket of words. This would lead to a sequence of memory accesses, which can be potentially used to map back to word w. Note that while previous works attacking `Hunspell` assume the knowledge of the dictionary [265, 141] before attack, such pre-knowledge is *not* needed for our attack. Instead, we use side channel traces and their corresponding sentences fed to `Hunspell` as the training data to implicitly learn a mapping in the low-dimensional joint manifold space. [265, 141] reports that functions `lookup` and `add_word` primarily leak inputs. Our manual confirmation shows that our findings (e.g., `putdic`, `chenc`, `check`, `insertion_sort`) indeed invoke `lookup` and `add_word` functions. We also find that the parser and interface (we use Linux utility `echo` to feed `Hunspell`) of `Hunspell` also influence side channels, both of which are not disclosed by previous works.

Confirmation with the Developers. We have reported our localized program points to the developers. By the time of writing, the FFmpeg developers confirmed our findings. Nevertheless, they mentioned that software-level fixing is undesirable, given the difficulty of writing side channel-free code and the incurred extra performance penalty. From his

perspective, OS-level or hardware-level fixing seems more practical.

3.6.3 Mitigation with Perception Blinding

We benchmark the mitigation effectiveness in terms of quantitative and qualitative analysis. We also discuss how different masks can influence the mitigation.

Qualitative Evaluation

We report qualitative evaluation by comparing the reference inputs with the reconstructed inputs after applying blinding. Due to the limited space, Fig. 3.8 only reports the perception blinding over a private face image $i_{private}$ in terms of different settings. The original image $i_{private}$ is presented in the "Original" column, and applied perception masks are presented in the "Mask" column. For each masked image $i_{blinded}$, the adversarial recovered images are presented in the "Reconstructed" columns, and the final media software outputs after unblinding are given in the "Recovered" columns.

"Noise mask" (the first row) and "non-face mask" (the second row) do not seem helpful in blinding $i_{private}$ because features such as face orientation are still preserved in the reconstructed images. However, the use of real face images as the mask, as shown in the third and fourth columns, gives promising results to blind key perceptual-level contents like hair color and skin color. Overall, after blinding, the adversary-reconstructed images seem to show a correlation with i_{mask} instead of $i_{private}$. This is intuitive; as clarified in Sec. 3.4.3, a large coefficient β is assigned to i_{mask} such that the perception contents of i_{mask} largely determine the projected intrinsic coordinate in the manifold. This way, the reconstructed images incline to manifest the perception of i_{mask}.

Additionally, although a small α value (e.g., 0.05) introduces a non-trivial amount of noise in the final output, outputs of much better quality can be recovered when α is set to 0.10 or even higher. We thus recommend that users adopt a reasonably high α when constituting $i_{blinded}$. More reconstructed cases are given in the artifact [9].

Table 3.10: Face matching results after blinding in terms of (cache bank/cache line/page table).

Mask	$\alpha = 0.05$	$\alpha = 0.1$	$\alpha = 0.3$
Noise	27.5/28.6/27.8%	25.2/26.9/28.2%	26.6/27.5/29.0%
Non-face	28.8/28.8/26.5%	26.2/27.6/27.4%	28.7/31.4/26.2%
Face#1	1.4/1.2/2.4%	1.8/1.4/2.7%	2.0/1.5/3.1%
Face#2	0.6/1.3/1.6%	0.7/1.7/1.9%	1.2/1.6/2.2%

Table 3.11: Mitigating COCO text inference attack in terms of (cache bank/cache line/-page table). $\alpha = 0.05$, $\alpha = 0.1$, $\alpha = 0.3$ denote each word are appended with 19, 9, and 2 masks, respectively.

Mask	$\alpha = 0.05$	$\alpha = 0.1$	$\alpha = 0.3$
"man"	0.39/0.40/0.36%	0.68/0.69/0.67%	2.46/2.45/2.13%
"sitting"	0.16/0.16/0.22%	0.30/0.30/0.39%	1.36/1.40/1.39%

Quantitative Evaluation

We launch quantitative evaluation following the procedures in Sec. 3.6.1. The perception blinding of "Noise" and "Non-face" masks, as shown in Table 3.10, reduces the average success rates of face matching from approximately 44% (Table 3.4) to 27.4%, but still has non-negligible privacy leakage. Compared with "Face#1" and "Face#2", Fig. 3.8 shows that images reconstructed from "Noise"- and "Non-face"-blinded images manifest better visual similarity with the reference inputs. In addition, Table 3.10 reports that "Face#1" and "Face#2" exhibit much better mitigation (less than 3.1% matching rates) in terms of quantitative metrics. We find that the value of α does *not* notably influence the results but is still positively correlated with privacy leakage. Overall, for images, we mask the perception contents using blinding. However, the privacy indicator for this scenario, i.e., celebrity's identity, is *not* simply a linear sum of all perception contents. Overall, identity recognition depends on subtle features of a human face: changing α not necessarily impedes capturing informative features.

Table 3.11 reports the mitigation results of `Hunspell` using COCO. We use two notional words of high frequency, "man" and "sitting", for blinding. We insert N notional words after each word in an input sentence, where N ($\frac{1}{\alpha} - 1$) is 19, 9, and 2 given different α. When more notional words are used, we observe a higher decrease in inference accuracy. However, with blinding, the inference accuracy decreases from more than 40.0% (see

Table 3.12: Quantitative evaluation results using cache side channels logged by `Prime+Probe`. We also provide the processing time (*ms*) when launching `Prime+Probe` (normal → with `Prime+Probe`).

	`FFmpeg` & SC09 voice matching	
	Intel	**AMD**
L1I Cache	12.6% (36 → 580)	11.6% (10 → 420)
L1D Cache	81.8% (36 → 590)	15.7% (10 → 420)
	`libjpeg` & CelebA face matching/non-face	
	Intel	**AMD**
L1I Cache	38.0/0.85% (5 → 18)	35.9/1.2% (2 → 22)
L1D Cache	36.9/0.80% (5 → 20)	33.9/0.90% (2 → 24)
	`Hunspell` & DDialog text matching	
	Intel	**AMD**
L1I Cache	33.9% (60 → 130)	33.2% (23 → 60)
L1D Cache	32.2% (60 → 130)	31.8% (23 → 60)

Table 3.6) to less than 2.5% (close to baseline; see Table 3.6) even two notional words are inserted after each normal word. Different from masking images, the privacy of text is assessed by word dependency (introduced in Sec. 3.6.1). α decides #notional words inserted to break word dependency. Therefore, the results changes notably w.r.t. values of α.

In sum, our quantitative evaluation demonstrates the effectiveness of our proposed mitigation despite differences in the media data formats or exploited side channels. See our artifact [9] for more results; for instance, blinding chest X-ray images can drastically reduce the disease diagnosis F1 score from an average of 0.73 to less than 0.1.

3.6.4 Real-World Attack with `Prime+Probe`

This section explores collecting cache access traces via a practical cache attack, `Prime+Probe` [237, 284], in *userspace-only scenarios*. To do so, we conduct an end-to-end experiment, by leveraging Mastik [269], a micro-architectural side channel toolkit, to perform `Prime+Probe` and log victim's access toward L1D and L1I cache. We use Linux `taskset` to pin the victim software and the `spy` process on the same CPU core. We now report key setup and results in this section.

We assume that attackers know when the victim media software begins and ends to process an unknown input [237]. The `spy` process primes and probes the cache. Technically, there is another "coordinator" process on the same core which computes victim process's cache activities and logs the cache side channels to disk. Nevertheless, according

to our observation, this coordinator process has generally consistent cache access patterns, thus its mostly fixed cache access does not interfere with our well-trained autoencoder.

We launch experiments on both Intel Xeon CPU and AMD Ryzen CPU. The thresholds of deciding cache hit and cache miss are 120 CPU cycles on Intel Xeon CPU and 100 CPU cycles on AMD Ryzen CPU. `Prime+Probe` is performed in the following manner:

PRIME: The `spy` process fills all cache sets.

IDLE: The attacker logs the access time of all cache sets for the previous `Prime+Probe` iteration. As a result, the idle phase *interval* equals the duration of performing one file I/O operation. Meanwhile, the cache is utilized by the `victim`.

PROBE: The `spy` process refills all cache sets and times the duration to refill the same cache sets to learn how `victim` accesses cache sets.

For a cache set, the logged cache status flip, from hit to miss, indicates at least one cache access of `victim`. We are thus particularly interested in logging such status flip. We name such cache status flips as "cache activity" in the rest of this chapter. Whenever a cache activity is observed, we record the cache activities of the all cache sets into a vector V, whose length equals to the number of cache sets. $V[i] = 1$ indicates there is a cache activity in i-th cache set. In other words, the i-th cache set is accessed at least one time by the `victim`. If no cache activity is observed from any cache set, we omit to generate a new V. See full details of this end-to-end attack in our artifact [9].

The quantitative evaluation results, as reported in Table 3.12, are generally encouraging. Attacks toward `libjpeg` and `Hunspell` manifest high accuracy comparable with attacks over `Pin`-logged traces (Sec. 3.6.1). While `Prime+Probe` logs relatively noisier cache side channels, our attack shows promising noise resilience, as trace encoder (and manifold learning by design) is noise resilient. Further, the logged side channels are *sparse*; given only a few records are secret-dependent, noise introduced by `Prime+Probe` and other workloads do not primarily impede our attack.

`FFmpeg` reports high attack accuracy (over 80%) on Intel L1D cache but lower accuracy for other settings. As in Table 3.13, the logged side channel traces are unstable (and challenging to comprehend), where stddev is about half of the average trace length. To verify the high attack accuracy on Intel L1D cache, we manually checked all the reconstructed

Table 3.13: Statistics (*mean*±*stddev*) and matrix encoding of cache side channel traces logged by `Prime+Probe`. Each trace consists of a 64-element vector sequence. The #training and #test splits in each setting are the same as those shown in Table 3.3.

	`libjpeg`(×8)		`FFmpeg`(×4)		`Hunspell`(×8)	
	Intel	**AMD**	**Intel**	**AMD**	**Intel**	**AMD**
L1I Cache	2719 ± 745 $1 \times 256 \times 256$	1114 ± 290 $1 \times 256 \times 256$	44315 ± 23374 $8 \times 512 \times 512$	7013 ± 2540 $2 \times 512 \times 512$	4834 ± 1788 $4 \times 256 \times 256$	1865 ± 432 $2 \times 256 \times 256$
L1D Cache	780 ± 137 $1 \times 256 \times 256$	5750 ± 931 $4 \times 256 \times 256$	8837 ± 3050 $2 \times 512 \times 512$	46698 ± 35163 $8 \times 512 \times 512$	3739 ± 164 $4 \times 256 \times 256$	9732 ± 1498 $8 \times 256 \times 256$

2,552 audio clips (also uploaded at [9] for reference). The reconstructed audio clips on Intel L1D cache manifest high quality. However, while the original audio clips of the same class are produced by different persons (and sound very different), all reconstructed audio clips of the same class sound *indistinguishable*. This indicates that the trained model stealthily simplifies the task of reconstruction into a task of ten class-conditional generation (recall this dataset has "0–9" labels). We then manually checked the collected traces: we find that for this particular case, Intel L1D cache "amplifies" the distance of inter-class traces while reduces the distance of intra-class traces. As a result, intra-class differences are not well learned using training data. However, since our quantitative metrics only check if the reconstructed audio can be classified correctly, the attack accuracy is high, indicating privacy leakage. We use attention (Sec. 3.4.2) and compared all the localized code components contributing side channels: we report that localized functions are the *same* on Intel/AMD CPUs. However, they manifest different frequencies, which result in this subtle model decaying. We confirm that this stealthy issue *only* occurs for this case. To solve this issue (and therefore reconstruct diverse outputs within the same class), users can opt for more complex models or larger training data, if needed.

Overall, inspired by recent work [250] exploring timing-based microarchitectural side channels, we deem it an interesting future work to benchmark the microarchitectural side channel differences. Our tool can automatically check whether side channels are informative enough to reconstruct secrets, when it largely outperforms the baseline.

We report the slowdown incurred by `Prime+Probe` attack in Table 3.12. Overall, media software are highly complex, and processing media data can usually induce a large volume of cache accesses. This way, frequent cache misses due to `Prime+Probe` can cause a reasonably high slowdown.

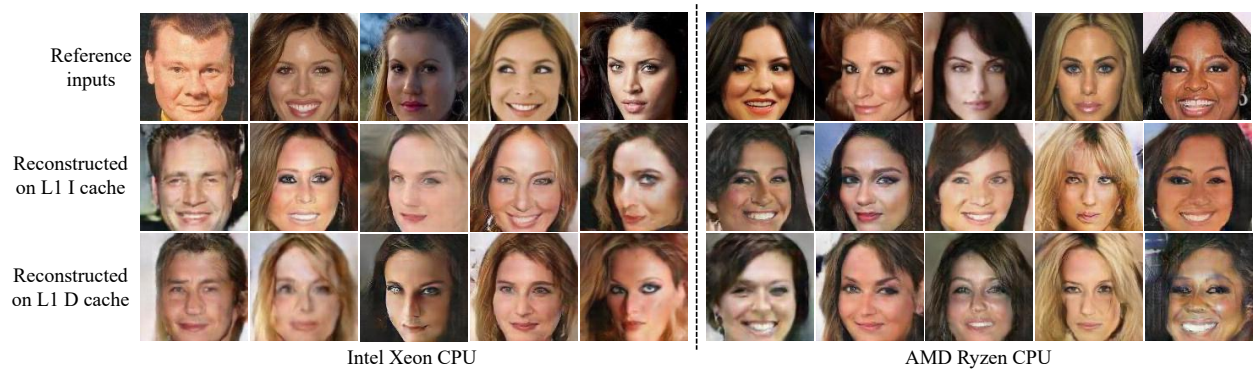


Figure 3.9: Reconstructed images on L1 cache of Intel Xeon and AMD Ryzen CPU. We can observe highly correlated visual appearances, including gender, face orientation, skin color, nose shape, and hair styles.

Table 3.14: Attack PathOHeap.

Function	`IDCT`	`MCU`
w/o ORAM	40.3%	38.0%
with ORAM	0.2%	0.2%

Qualitative Evaluation Results. Fig. 3.9 reports reconstructing CelebA face photos on different CPUs. We interpret the reconstruction results as generally promising: a considerable number of visual perceptions are faithfully retained in the reconstructed images, which include gender, face orientation, skin color, nose shape, and hair styles. There is a bit quality degradation on some images, for instance, facial details (e.g., the appearance of teeth) are not always precisely aligned with reference inputs. However, we emphasize that images reconstructed under four settings all manifest comparable visual quality, indicating high feasibility of applying our framework on the basis of commonly-used side channels. The qualitative results are also consistent with results reported in Table 3.12 — though the attack is launched on different CPUs and different caches, privacy leakage is steadily notable.

3.6.5 Mitigation Using ORAM

Besides perception blinding, this section assesses other mitigations. Existing mitigations aim at adding randomness, making it constant, or directly masking inputs. Nearly all of them are particularly designed to protect crypto software [234, 59, 38]. Raccoon [208]

proposes general mitigation using software obfuscation; however, its implementation is not available. That said, oblivious RAM (ORAM) [86] conceal memory access sequences of a program by continuously shuffling data as they are accessed. We study whether a representative ORAM, PathOHeap [225], can mitigate our attack. Due to the limited space, we report the key evaluation results in Table 3.14. PathOHeap takes several hours to process one memory access made by `libjpeg`. We thus focus on two critical functions localized by our framework (see Sec. 3.6.2), `IDCT` and `MCU`, separately.[5] We measure attack success rates with and w/o first converting memory traces using PathOHeap.

Cache line side channels derived from either `IDCT` or `MCU` are sufficient for attack. Nevertheless, ORAM eliminates information leak: memory access traces, after being processed by PathOHeap, do *not* depend on input images. We report that our autoencoder does not even reach convergence during training, and the reconstructed images (using poorly trained autoencoder) show indistinguishable and meaningless visual appearances. The non-zero result (i.e., 0.2%) is because that many face photos look like an "average" face. In other words, 0.2% implies the *baseline* of face matching.

Comparison. PathOHeap is very costly: while `libjpeg` can process an image within 100ms, PathOHeap takes several hours to convert the corresponding memory trace. The obfuscator, Racoon [208], has an average overhead of 16.1×. In contrast, perception-blinding delivers negligible extra cost (i.e., processing masked data using media software once) albeit its mitigation is specific for manifold learning-based SCA. This underlines the key novelty of our technique.

3.6.6 Noise Resilience

We have discussed the general immunity to noise of manifold learning in Fig. 3.4. This section empirically assesses our attacks under various scenarios where noise is introduced in side channels. We summarize our noise insertion schemes in Table 3.15. The first three schemes are launched to mutate cache line access traces logged by `Pin`, whereas the latter three mutate the cache set hit/miss records logged via `Prime+Probe`. **NA** means no noise is inserted, whereas **Low/High** denotes to what extent side channel logs are perturbed.

[5] Focusing functions with known information leakage (i.e., `IDCT`) [265, 95] demonstrates a "white-box" attacker using our technique.

Table 3.15: Quantitative evaluation results (same face/non-face) of face images reconstructed from noisy side channels.

Setting	Noise Insertion Scheme	NA	Low	High
`Pin` logged trace	**Gaussian**	43.5/2.0%	33.8/2.1%	28.0/1.5%
	Shifting	43.5/2.0%	42.9/1.7%	39.1/1.8%
	Removal	43.5/2.0%	30.0/1.9%	29.3/4.3%
`Prime+Probe` logged trace	**Leave out**	36.9/0.8%	36.8/1.1%	36.8/1.1%
	False hit/miss	36.9/0.8%	36.4/1.2%	36.1/1.2%
	Wrong order	36.9/0.8%	36.8/1.0%	36.7/1.0%
Workload under `Prime+Probe`	**Bzip2**	36.9/0.8%	27.6/1.0%	
	Victim$_1$	36.9/0.8%	30.7/1.1%	
	Victim$_2$	36.9/0.8%	29.0/1.0%	

We also benchmark how real-world workload, i.e., by launching `bzip2` or another victim software (e.g., `libjpeg`) on the same CPU core, can undermine our attack. **Victim$_1$** and **Victim$_2$** represent launching another victim software on the same core and processing the same or different inputs.

We only report the quantitative evaluation results of `libjpeg` on CelebA in Table 3.15. See our artifact [9] for other quantitative and qualitative results: despite the applied noise, perceptual features are still retained in the reconstructed data (e.g., face photos), illustrating the feasibility of privacy stealing under noisy scenarios.

The reconstructed images are more resilient toward **Shifting**. **Removal** and **Gaussian** noise, by extensively leaving out or perturbing data points on the logged trace (e.g., **Removal/High** removes *half* records on a trace), show greater influence on data reconstruction. As for noise inserted in `Prime+Probe` logged side channel records, none of them primarily affect the attack accuracy. We note that `Prime+Probe` logged side channel traces, even without applying these noise insertion schemes, are of high stddev. That is, our autoencoder will be trained with more "diverse" side channel logs, which enhance the robustness but undermines accuracy. Similar findings are obtained in launching extra real-world workloads.

3.7 Related Work

Side Channel Analysis. Kocher proposes to use timing side channel to exploit crypto systems [130]. To date, side channels have been used to exploit crypto systems under different scenarios [15, 25, 71, 262], including trusted computing environments like Intel SGX [141, 34, 180, 217]. [38] demonstrates that timing side channel can be launched remotely through network. The CPU cache are particularly exploited given its indispensable role in boosting modern computing platforms [95, 194, 271, 164]. Controlled side channel assumes an adversarial-controlled OS to log page table access of victim software [265]. DNNs have been used to infer secret keys from crypto libraries [100, 174, 99]. These works, usually referred to as "profiled SCA", share the same assumption with our research that models are trained using historical data. Most existing DNN-based SCA focuses on attacking crypto systems; they typically perform low-level *bit-wise classification* to gradually infer key bits. In contrast, we show that attackers in black-box scenarios can use manifold learning to reconstruct media data of various types in an end-to-end manner. [137, 261] also use autoencoders in the context of SCA. However, they use autoencoder to denoise side channel traces as a *preprocessing* step for SCA of crypto software.

Countermeasures. Software-based techniques include constant-time techniques which ensure that software behavior is independent with its confidential data [58, 204, 216, 131, 183]. Techniques have also been proposed to blind secrets or randomize side channel access patterns [22, 35, 65, 111, 133, 208]. ORAM [87, 161, 229, 225] translates memory access into identical or indistinguishable traces, which can provably eliminate many side channels but incur high performance penalty. Program analysis methods such as information flow tracking [138, 186], model checking [16], type system [14, 211], abstract interpretation [132, 73, 243], and constraint solving [244, 243, 37] are used to check crypto software and detect side channels. In contrast, our study delivers a *neural attention*-based approach to detecting code fragments inducing information leakage. Hardware-based countermeasures include randomizing side channel access or enforcing fine-grained resource isolation [247, 162, 196]. Compared with system- and hardware-based countermeasures, software-based approaches usually do not require to modify the underlying hardware design. Nevertheless, software-based countermeasures are generally high cost and low scalable in analyzing real-world software.

3.8 Conclusion

This research proposes SCA for media software. We perform cross-modality manifold learning to reconstruct media data from side channel traces. We also use attention to localize program points leading information leakage. We design perception blinding to mitigate the proposed SCA. Our evaluation on real-world media software reports promising results.

CHAPTER 4

TRUSTED EXECUTION BRINGS FALSE TRUST: DISCLOSING USER DATA LEAKAGE IN TEE-SHIELDED NEURAL NETWORKS

This chapter shows that the confidentiality of neural networks (NNs) and user data is compromised by the recently disclosed ciphertext side channels in Trust Execution Environments (TEEs), which leak memory write patterns of TEE-shielded NNs to malicious hosts. This chapter presents our work, CIPHERSTEAL, *that for the first time demonstrates 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.*

In this chapter, we comprehensively evaluate two popular deep learning frameworks, TensorFlow and PyTorch, and NN executables generated by two recent NN compilers, TVM and Glow, and summarize our key findings about their distinct 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.

4.1 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 [176, 143, 125, 156, 110]. 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 [114]) or a secure virtual machine (e.g., AMD SEV [122]).

A well-known security concern of TEEs is their vulnerability to micro-architectural side channels such as cache or timing attacks [91, 180, 245], where attackers exploit secret-dependent data or control flows of TEE-shielded programs. Nevertheless, unlike traditional programs that express their internal logics via conditioned branches or data accesses, an NN is essentially a sequence of matrix computations, which exhibits a constant-time computation paradigm: data and control flows in NNs are *fixed* regardless of its inputs, making NN inputs free of micro-architectural side channels.

A New Leakage. Despite the general security belief, this chapter 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 [149] or via memory bus snooping [141]) can recover informative patterns in plaintext.

The work presented in this chapter 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. 4.4), 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 [149, 146]. 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 chapter 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 [51] or Glow [210]). CIPHERSTEAL is evaluated on 13 real-world and large-scale NNs of various structures (e.g., Vision Transformer [72]), 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, the work presented in this chapter makes 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.

Research Artifact. The code, data, and thousands of recovered images and videos, are available at: https://sites.google.com/view/cipher-steal [7].

4.2 Preliminaries

4.2.1 Neural Networks (NNs)

An NN $\mathcal{F}$'s execution can be expressed as $f_n \circ \cdots \circ f_2 \circ f_1(x)$, which propagates the input x through a series of layer f_i. Each layer $f : y = \sigma(Wx + b)$ consists of a matrix multiplication and addition followed by a non-linear function σ. Unlike traditional programs whose logics are hardcoded with human-written instructions, NNs enable a data-driven programming paradigm: internal logics in NNs are formed by implicitly "learning" rules

from data (i.e., training data) and are encoded in its weights W and b. Different from traditional programs, control and data flow in NN executions are typically *fixed* and do *not* change with input x or weights W and b.

The life cycle of an NN can be divided into two phases.

Training Phase. When training an NN, both forward and backward propagations are involved. The normal scheme is to train NN weights once and then use the weights without updates. Nevertheless, an NN may still update its trained weights in scenarios like fine-tuning [52] or security hardening [286] after being deployed in TEE.

Forward Propagation (FP). To develop NNs, developers first collect training inputs and manually annotate their solutions (ground truth). Then, the NN takes the training inputs and performs FP to compute the outputs, which are then compared with ground truth to adjust the NN logic (weights). *Intermediate computing results* in FP reflect how different elements in an input are processed by the NN [280, 218].

Backward Propagation (BP). When NN outputs deviate from ground truth, a loss penalty is computed and backpropagated. Intermediate results generated during BP are known as "gradients" and guide the NN to update its weights, such that the NN's logics gradually matches the ground truth marked in training inputs. Gradients are believed as more precise indicators of NN behaviors. Many attacks use gradients to downgrade or control NN predictions [268].

Inference Phase. A trained NN infers predictions for unknown inputs. The inference phase only has FP, with intermediate outputs characterizing the NN's inputs. Inference inputs are user secrets because NNs are commonly adopted for privacy-sensitive uses, e.g., medical image diagnosis. NN structures often fall into two groups.

Acyclic Structure. NNs are widely used for non-sequential data like images. Representative structures include convolutional (Conv) and fully connected (FC) NNs, which have distinct patterns when processing images. For example, in Conv, a kernel slides over the image and more *nested loops* (usually four) are required to implement the computations. However, in FC, inputs are often reshaped as vectors (e.g., a $3 \times 32 \times 32$ image is reshaped as a 3072-dimension vector); the computation usually requires only two *nested loops*.

Recurrent Structure. Sequential data like sentences and videos are processed with NNs

having recurrent structures (e.g., LSTM [102]). During the execution, an NN is recurrently applied to each sequence element (e.g., a video frame). The basic NNs in these recurrent structures are the same as in acyclic structures. Typically, FC is recurrently applied on words in a sentence whereas Conv is applied on video frames.

4.2.2 Runtime Modes

Interpreter-Based. An NN framework is a software library that provides a programming interface for developers to build and run NNs. Typical NN frameworks include PyTorch [199] and TensorFlow [12]. The runtime system of typical NN frameworks 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 [282], MKL [242], and Eigen [93]) 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 where the intermediate results are used to compute gradients for weight updates.

Compiler-Based. NNs are increasingly compiled into executables for better performance across different platforms. Two mature and actively maintained NN compilers, TVM [51] and Glow [210], emit standalone executables that can be 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) to allow more target-specific optimizations and eventually emit machine code.

4.2.3 TEEs and Ciphertext Side Channel

TEEs aim to protect sensitive data processing by providing isolated environments, namely enclaves, for program execution. Specifically, 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 [143, 125, 156, 110].

With TEEs, developers could protect NNs and preserve their data privacy while deploying 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 [122], Intel TDX [114], Intel SGX on Ice Lake SP [114, 119], and ARM CCA [20]. 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 [149].

Ciphertext Side Channel. Recent studies [146] uncover ciphertext side channels to steal sensitive data (e.g., private keys). Specifically, ciphertext side channel attacks exploit such 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 [146, 69]. In terms of real-world exploitation, Li et al. propose the first ciphertext side channel attack targeting AMD SEV-SNP [149] and exploiting cryptographic libraries like RSA. While most discovered vulnerabilities living in AMD TEE [184, 254, 150, 148, 147] 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 [146].

Fig. 4.1 shows 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 cipher-

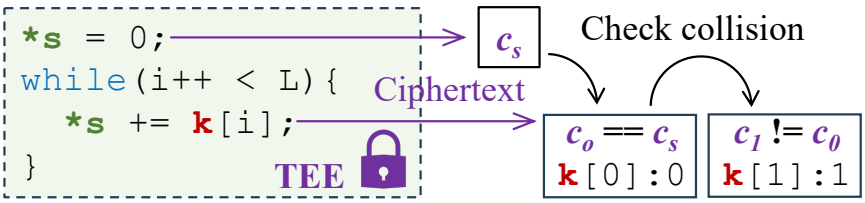


Figure 4.1: Ciphertext side channel leakage of cryptographic keys.

text 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 and $c_1 \neq c_0$, attackers also know that `k[1]` is 1.

4.3 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. 4.2.1, 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 [176], 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. 4.7.1).

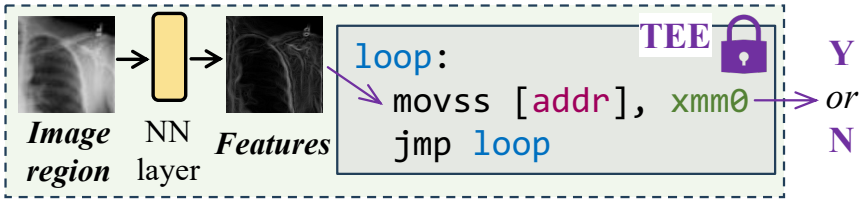


Figure 4.2: Ciphertext side channel leakage of NN inputs.

Crypto. Keys vs. NN Inputs. Nevertheless, recovering NN inputs from ciphertext side channels is fundamentally more challenging than cryptographic keys. As illustrated in Fig. 4.1, each ciphertext collision in cryptographic software is often induced by writing a key bit. Since key bits are either 0 or 1, the collision information can be accurately mapped to each key bit. In contrast, as shown in Fig. 4.2, 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 [278]. 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. 4.2.1, 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 prior works tried to recover NN inputs from power side channels of NNs [182,

251]. 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 chapter 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 [278], 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. 4.5).

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. 4.7, 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.

4.4 Threat Model and Assumptions

Assumptions. Following existing works that deploy NNs in TEEs [179, 108, 223, 231], we assume that the deployed NNs only output the final predicted label to users who issue the queries (note that malicious hosts *cannot* view the predicted label), whereas all remaining intermediate results are kept in the TEE and not returned to the users. Also, to mitigate query-based NN cloning [193, 198, 235], TEE-shielded NNs do not return the prediction confidence (i.e., the probability of the input belonging to each class). 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 [121] have all been fixed, leaving only generalized ciphertext side channel leakage discovered in [149] for use.

Attacker's Capability. Aligned with existing works that attack/harden TEEs or shield NNs with TEEs [146, 69, 256], 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 [141], reading remnant data from the DRAM via cold boot attack [92], and accessing memory directly via DMA devices [230]. 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.

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

NN Structure and Weights. Previous NN attacks [42, 90] (see details in Sec. 4.4.1) 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 [235, 166, 42, 278, 81], 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 Definition 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 Definition 2, given that NNs having the same usage may have different input domains. For instance, although two NNs capable of diagnosing different diseases have different input domains, they both accept medical images as inputs. Here, medical images are their input format. 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 [235, 227, 40, 42] (see detailed explanations in Sec. 4.4.1) 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.

4.4.1 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 [112, 267, 105, 266, 167, 81, 75]. 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 4.1: 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 4.1, previous NN attacks can be divided into two categories.

Steal Functionality. Since recovering exact NN weights is challenging, query-based inference attacks [235, 166, 198] are proposed to steal NN functionality. In short, attackers query the target NN and let their own NN duplicate the prediction confidences (i.e., prob-

[1] Existing works steal NN weights by reading plaintext transmitted through PCI bus [291]; TEEs mitigate this via traffic encryption.

abilities 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. 4.4), 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) [90, 173, 42]. 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 by TEE-shielded NNs whose weights are encrypted. Nevertheless, since an NN's vulnerabilities to AEs are mostly inherited from training data [90, 264, 274], 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. 4.7.4).

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. 4.2, 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 [174, 124, 99, 278, 81], 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 queried-based attacks [235, 166, 198], 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 Definition 2) but may cover different input domains. CIPHERSTEAL is agnostic to the specific TEE platform or side channel logging tools (e.g., `SEV-Step` [146, 258] or `CipherLeak` [149]); as evaluated in Sec. 4.7.3, CIPHERSTEAL works well for different ciphertext side channels.

4.5.1 Problem Reformulation

Information Loss and Decomposition of $\mathcal{A}$. As mentioned in Sec. 4.3, 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. 4.3). Then, the reconstruction $\mathcal{R}$ aims to reconstruct the lost information in h to get $x^* = \mathcal{R}(h)$ which is close to x.

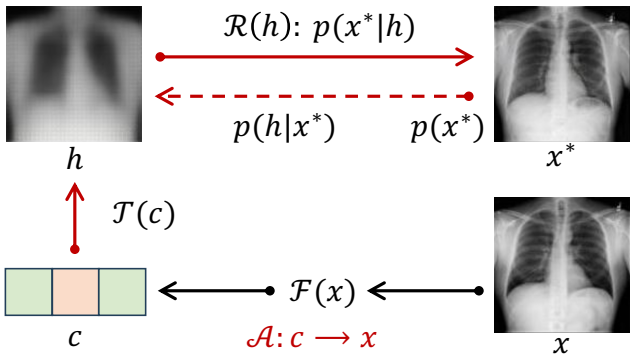


Figure 4.3: Decompose $\mathcal{A} : c \rightarrow x$ as transformation $\mathcal{T}$ and reconstruction $\mathcal{R}$. $\mathcal{R}$ is implemented via its inversion $p(h|x^*)$ and the realism term $p(x^*)$.

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 trans-

ferred 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. 4.4).

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:

$$\arg\max_{x^*} p(x^*|h), \tag{4.1}$$

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. 4.1 can be re-formed as:

$$p(x^*|h) = \frac{p(h|x^*)p(x^*)}{p(h)} \tag{4.2}$$

Since h is known, $p(h)$ is accordingly constant. As illustrated in Fig. 4.3, the objective in Eq. 4.1 is equivalent to

$$\arg\max_{x^*} p(h|x^*)p(x^*), \tag{4.3}$$

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 [89, 128]. 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^*.

4.5.2 Implementation Considerations

This section introduces how different modules in Sec. 4.5.1 are implemented.

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 [278, 226, 89]. 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 θ 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:

$$\arg\min_{\theta} L(x, \mathcal{T}_\theta(c)) \tag{4.4}$$

where L denotes the distance between x and the transformed information in c. As mentioned in Sec. 4.5.1, with the objective of minimizing L, θ 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.

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. 4.5.1, 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 ω. $\mathcal{I}_\omega$ 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. 4.4 is extended as:

$$\arg\min_{\theta,\omega} L(x, \mathcal{T}_\theta(c)) + L(\mathcal{I}_\omega(x), \mathcal{T}_\theta(c)) \tag{4.5}$$

Ensuring the Realism: $p(x)$. Estimating $p(x)$ has been widely studied via generative models (e.g., GANs [89], Diffusion models [128]). In short, given a generative model G, $p(x)$ can be estimated by projecting a collection x into a continuous latent space $\mathbb{Z}$. Since $\mathbb{Z}$ is continuous, infinite and diverse x can be represented as the results of interpolation and exploitation in $\mathbb{Z}$ [89]. 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 $\mathbb{X}'$.

Once $\mathcal{T}_\theta$, $\mathcal{I}_\omega$, and G are well-trained, suppose during the online attack, the attacker logs a ciphertext side channel trace c 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:

$$\arg\min_{z^*} L(h, \mathcal{I} \circ G(z^*)) \tag{4.6}$$

which can be achieved via optimization such as stochastic gradient descent (SGD) [123]. Note that solving Eq. 4.6 is similar to training a neural network. Nevertheless, the weights of neural networks are fixed in Eq. 4.6 and only the input of G, namely z, is updated. 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 [278], h can guide $G(z^*)$ to be close to x.

4.6 Evaluation Setup

Table 4.2: Studied NNs and datasets for various tasks under different attacker's knowledge.

Exp. ID	NN	Dataset	Task	Usage	Input Domain		Logging Tool
					Owner/User	*Attacker*	
Ⓐ	LeNet [140]	MNIST [68]	Classification	Digit recognition	Digit 0-9	**FK**: Digit 0-9	SEV-Step
Ⓑ					Digit 0-9	**PK**: Digit 0-4	
Ⓒ					Digit 0-4	**ZK**: Digit 5-9	
Ⓓ	FaceNet [213]	CelebA [168]	Regression	Face recognition	Face Photos	**ZK**: Face photos of other identities	
Ⓔ	MobileNet [109]	Chest X-ray [246]	Classification	Disease diagnosis	14 Diseases	**ZK**: Benign	
Ⓕ						**PK**: 7 Diseases	
Ⓖ	ResNet [96]	ImageNet [67]	Regression	Image compression	100 classes in ImageNet	**FK**: 100 classes in ImageNet	
Ⓗ	ConvLSTM [226]	KTH Actions [214]	Classification	Video understanding	6 Actions	**FK**: 6 Actions†	
Ⓘ	ViT [72]	MNIST [68]	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 [246]	Classification	Disease diagnosis	14 Diseases	**PK**: 7 Diseases	
Ⓜ	ViT	CelebA [168]	Regression	Face recognition	Face Photos	**ZK**: Face photos of other identities	

* Markers Ⓐ-Ⓚ are consistently used in the rest of this chapter to ease finding the setups.
† In the setup of Ⓗ, the human IDs of attacker's videos and owner's/user's videos do *not* overlap.

NNs, Datasets, Tasks, and Input Domain. Table 4.2 lists our evaluated NNs and datasets. These NNs are representative and diverse in structures. They are widely adopted as the backbone of modern NNs, e.g., the Vision Transformer (ViT) [72] is the backbone of modern multi-modal LLMs. 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 4.2. The experiment IDs Ⓐ-Ⓜ in Table 4.2 are consistently used in the rest of this chapter 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.

Runtimes. In line with Sec. 4.2.2, 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. Only interpretation-based frameworks currently 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 [286, 285]). 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. 4.3, 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 pre-defined set of predictions, *the baseline of classification tasks is*

$1/\#Classes$. For regression tasks (Ⓓ, Ⓖ, Ⓜ in Table 4.2), 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.*

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 [281] 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. 4.7. We set $K=1$ and $M=100$. Thus, *the baseline of SIM is* $1/100 = 1\%$.

Side Channel Logging. Two mature logging tools have been proposed by previous works to collect ciphertext side channels: `CipherLeak` [149] and `SEV-Step` [146, 258]. 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 4.2) 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 [170], 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. 4.7.3.

4.7 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. 4.4.1) enhanced by our results.

4.7.1 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. 4.7.2.

Table 4.3: Vulnerable modules and keywords of sample functions. More cases are provided in our artifact [7].

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

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 [253, 255], we can simply check if the ciphertext collisions of each address change with inputs.

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

(a) ReLU operation (Glow).

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

(b) Layout transformation (TVM).

```
1 ; [bias]: bias
2 ; [addr1]: Conv result
3 ; [addr2]: zero-initialized
4 ; xmm4: 0
5 movss xmm1, dword ptr [bias]
6 movss xmm3, dword ptr [addr1]
7 addss xmm3, xmm1
8 maxss xmm3, xmm4
9 movss dword ptr [addr2], xmm3
```

(c) Fused Conv & ReLU (TVM).

Figure 4.4: Code patterns in Glow and TVM executables.

Executables. Table 4.3 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. 4.4(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. 4.4(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 [51]). As shown in Fig. 4.4(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. 4.4(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.

Table 4.4: 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%

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.

Attack Surfaces under Various Scenarios

Setup. This section presents input recovery towards our localized modules in Sec. 4.7.1 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 [280, 84], we further evaluate how the depths of layers affect the input recovery in Sec. 4.7.3. For TVM executables, we focus on layout transformation functions which primarily induce the leakage.

Table 4.4 presents our results. Below, we analyze them from several aspects and summarize eight key findings.

Knowledge of Input Domain. As in Table 4.4, 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 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. 4.6). 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 4.4, 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 [239] 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 [227, 40], 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 4.4. We interpret the result from two aspects. First, our technique ensures the realism of recovered inputs by modeling $p(x)$; see Sec. 4.5. 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 4.4, 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.*

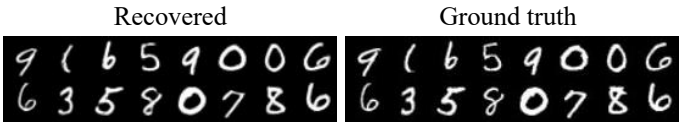
Backward: Static vs. Dynamic Computational Graph. Different from the FP, inputs recovered during TensorFlow's BP have higher PC and TC. Also, by cross-comparing the gaps between FP's and BP's results in PyTorch and TensorFlow, PyTorch cases have larger gaps. Note that PyTorch maintains a dynamic computational 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.*

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

Qualitative Examples. Fig. 4.5 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 [7]. 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.

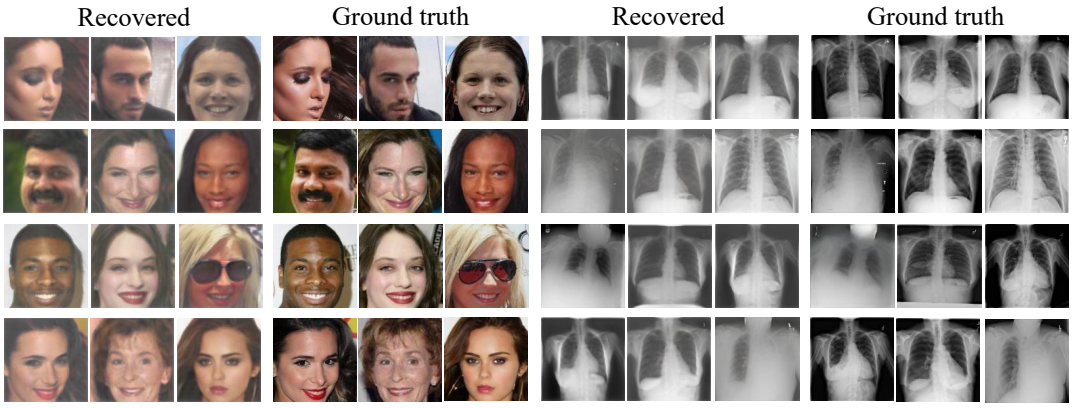
Quantitative Analysis. As reported in Table 4.5, we achieve encouraging prediction consistency (PC) and training consistency (TC) results for diverse and more complex input formats, indicating that our recovered inputs are capable of stealing the predictions and functionalities of the target NNs. Table 4.6 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



(a) Digit images under **FK** setting.

(b) Digit images under **PK** setting.

(c) Digit images under **ZK** setting.



(d) Recovered face photos.

(e) Recovered chest X-ray images.

Figure 4.5: Examples of recovered images and ground truth. For the ZK case in Fig. 4.5(c), the target NN only processes digits 0-4. Recovered videos and more images are in [7].

Table 4.5: Attack results of other input formats. PC and TC denote prediction consistency and training consistency.

	Input		**Training Input**		**User Test Input**
			PC	TC	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 4.6: Similarity evaluation (SIM). Baseline is 1%.

Input		Training Input		User Test Input	
		Glow	*PyTorch*	*Glow*	*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%

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 [251]. 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 [278]. 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. 4.5.2, recovering videos is conceptually similar to, but technically harder than recovering text.

4.7.3 RQ3: Side Channel Observations

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

Logging with `CipherLeak`. Besides `SEV-Step`, we also evaluate CIPHERSTEAL by logging ciphertext collisions with `CipherLeak`. Results are given in Table 4.7. 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

Table 4.7: 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%

using SEV-Step. Nevertheless, the results are still promising, and remain largely higher than the baselines.

We note that the results over interpreters and executables are very close. In Sec. 4.7.1, 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 4.8: 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 [258], such that some memory writes are periodically missed. To benchmark such impacts on CIPHERSTEAL, we consider SEV-Step of different gran-

ularity T, i.e., only recording every T-th memory write. We consider $T = 16$ and 64.

As in Table 4.8, 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. 4.7.1: 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 [280, 29], we study how the input recovery is affected by the choice of layers. As shown in Table 4.8, 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.

4.7.4 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. 4.4.1, 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 [19], which is directly applied to target NNs. We use PGD [23] 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 ℓ_∞-norm of 0.3. In each setting, we generate 2,000 AEs for the attack.

Results. Table 4.9 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 in-

Table 4.9: 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%

herited 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 ∼170s for MNIST and ∼20 min for X-ray images; however, this is a one-time effort.

4.8 Discussion on 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 [256] may not be directly applicable to our attack, given that [256] primarily fixes vulnerable cryptographic code patterns that do not exist in NNs. [146] advocates to achieve non-deterministic ciphertexts in TEEs via VMSA randomization, and [257] 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 [146, 257]. 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 [104] may be a good starting point.

In addition, inspired by our findings in Sec. 4.7.1, 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. 4.4, 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.

4.9 Conclusion

This chapter 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.

CHAPTER 5

THIEVING MODEL WEIGHTS FROM TEE-SHIELDED NEURAL NETWORKS VIA CIPHERTEXT SIDE CHANNELS

Following Chap. 4 that recovers inputs of TEE-shielded deep neural networks (DNNs), this chapter shows how weights of a TEE-shielded DNN can be recovered from its ciphertext side channels. Overall, we propose a novel viewpoint that focuses on the functionality of DNN weights, rather than each weight element's exact value. Accordingly, we design HYPERTHEFT *to directly generate weights that are functionality-equivalent to the victim DNN using ciphertext side channels.* HYPERTHEFT *is established for highly practical settings; it exhibits the weakest requirement compared to prior methods. When only knowing a victim DNN's input type and task type (which are public and denote the minimal information required to use a DNN),* HYPERTHEFT *can recover its weight using ciphertext side channels logged during the victim DNN's one execution. The whole procedure does not require attackers to 1) query the victim DNN, 2) have valid data that the DNN accepts, or 3) know the victim DNN's structure. Our evaluations generate more than 8K DNN weights which constantly achieve 77%∼97% test accuracy in different DNN runtimes, including various versions of PyTorch and DNN executables. Our recovered weights can subsequently enable training data leakage and severe bit-flip attacks.*

5.1 Introduction

Deep neural networks (DNNs) have been exponentially deployed on various platforms (e.g., cloud servers, edge devices) given their high intelligence in solving real-life tasks. DNNs' intelligence is encoded in their weights which are trained over humongous datasets, with extensive human expertise and computing resources required. Nevertheless, as DNNs are white-box accessible on the host machine, a malicious host can directly copy their weights to steal DNN intelligence and launch white-box attacks [227, 285], posing a major security and privacy threat to modern DNNs [125].

To address the trust concern, Trusted Execution Environment (TEE), such as AMD SEV [122], Intel SGX [114, 119], etc., is proposed to protect DNNs [143, 156, 110]. With memory encryption, TEE provides isolated execution to shield a DNN as fully black-box on the host machine. Although recent studies launched successful side-channel attacks on TEE-shielded programs based on their secret-dependent information flows [91, 180, 245], DNN weights are believed secure under TEE protection: modern DNNs implement a constant-time computation paradigm, where the computations are achieved as a sequence of matrix operations with constant[1] data/control flows, eliminating mainstream micro-architecture side channels [285, 105].

Despite the promise, this chpater uncovers severe DNN weight leakage due to the ciphertext side channel [149, 146] recently disclosed in AMD SEV.[2] Essentially, modern TEEs adopt *deterministic* encryption to support efficient random memory access and large memory encryption [146, 69], and the ciphertext of each memory write value only depends on the plaintext and the written address. Hence, if attackers observe that the ciphertexts of two consecutive memory writes at the same address do not change (i.e., ciphertext collision), they can infer the equality relation between two plaintext written values [149, 146]. Intuitively, as matrix computations inside a DNN involve nested loops with DNN weights, which often repeatedly access the same memory address, ciphertext collisions should correlate with DNN weights to a large extent.

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

① **A More Challenging Threat Model.** Unlike cryptographic software (i.e., the attack target of prior ciphertext side-channel attacks [149, 146]) whose implementation details are known to attackers (e.g., the RSA algorithm is public), the computation graphs of DNNs are often *private*. Thus, attackers only have a ciphertext side channel trace consisting of all ciphertext collisions logged from the victim DNN's whole execution. They *cannot* investigate how each weight element is involved in DNN computations and is consequently

[1] Recent works have proposed the multi-exit DNN [157] whose execution may terminate early for some inputs. However, multi-exit DNN's control and data flows only depend on the final predictions and do not leak DNN weights.

[2] Practical attacks have been demonstrated on AMD SEV and reported to the vendor; yet, the attacks are feasible to any deterministic-encryption-based TEEs via hardware attacks such as memory bus snooping [141], cold boot attack [92], etc.

leaked via each ciphertext collision.

② **Partial Leakage of DNN Weights.** The ciphertext side-channel leakage in cryptographic software is lossless, as it is due to memory writes *directly* determined by each private key bit [149, 146]. However, memory writes of a DNN's execution rely on *intermediate computing results* derived from the weight, ciphertext collisions therefore do not leak all weight elements. Since a DNN often has millions of weight elements which are *highly correlated*, failing to recover a few weight elements can result in non-functional DNNs whose predictions are purely random [206, 268].

③ **Over-Strong Requirements of Query Attacks.** One may conduct query-based attacks, which use a student model to duplicate confidence scores of a DNN's predictions over query inputs [198, 235, 116, 47], to imitate the victim DNN's behaviors. However, TEE-shielded DNNs do not return confidence scores, greatly increasing the cost of query-based attacks [285]. While recent hardware attacks can be adopted to reduce the cost by recovering partial DNN weights, they require knowing the DNN's structure (which can be private) and are limited to quantized DNNs [205]. Importantly, the query inputs must cover all classes of the victim DNN's training data, which is impractical if the DNN is trained on private datasets like medical images. Noteworthy, training a student model to generate identical ciphertext collisions as the victim DNN is also infeasible, as the generation of ciphertext is non-differentiable.

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

HYPERTHEFT delivers highly stealthy and generic attacks. It considers both regression and classification, two fundamental tasks of all modern DNN applications such as image recognition, disease diagnosis, financial forecast, etc. To attack a DNN performing regression or binary classification, HYPERTHEFT only requires ciphertext collisions from its one execution. For k-class classification ($k > 2$) — in case k is unknown — HYPERTHEFT decouples it as k different binary classifications (i.e., belonging to the k-th class or not), and generates weights for k surrogate models (with each one for a binary classification) by observing the victim DNN's (minimal) k executions. Further, inspired by stochastic training algorithms of DNNs (e.g., SGD [17]), we introduce stochasticity into HYPERTHEFT's weight generation, such that *multiple* functionality-equality weights can be generated using only one side-channel trace; the corresponding surrogate models (for the same task) can form a majority voting to further improve their accuracy.

Practicality: The Weakest Knowledge. Distinct from all prior weight stealing techniques, HYPERTHEFT does not interact with the victim DNN; it only passively observes ciphertext side channels *without* querying the victim DNN. Thanks to our well-designed training algorithm for hyper-network (see Sec. 5.6), HYPERTHEFT does *not* require valid data accepted by the victim DNN (compared with query-based attacks). Additionally, empowered by our functionality-centric view, HYPERTHEFT does not rely on the victim DNN's structure and is applicable to general DNNs (compared with prior hardware attacks [205, 272]). In Sec. 5.8.2, we employ HYPERTHEFT to generate more than 8K weights under this *weakest-knowledge* setup, and these weights constantly achieve 77%∼97% test accuracy. To comprehensively assess the real-world threats, Sec. 5.8.3 evaluates how those stronger assumptions in prior works, e.g., knowing the DNN structure or querying the DNN (which may hold in certain scenarios), can further boost HYPERTHEFT.

Findings: Broad Attack Surface. HYPERTHEFT can successfully steal weights from popular DNNs (e.g., Transformer, ResNet, etc.) performing various classification and regression tasks over representative datasets (e.g., ImageNet, Chest X-ray, etc.). We consider different runtimes of DNNs: the most popular deep learning (DL) framework PyTorch and the recent DL compiler, Glow [210], that compiles DNN models into executables. We also systematically evaluate various versions of PyTorch and consider different usages of TEEs (i.e., shielding full DNNs or DNN slices). Our recovered weights constantly achieve

the objectives of stealing DNN intelligence and enabling white-box attacks against the victim DNN. Although the recovered weights are never trained using the victim DNN's training data, they largely enhance membership inference attack [227, 40] to leak the training data. The recovered weights also bring bit-flip attack [206, 268], which can globally decimate DNN intelligence for nearly all (benign) inputs. In sum, the work presented in this chapter makes the following contributions:

- For the first time, we unveil the high risk of leaking DNN weights via ciphertext side channels of TEE-shielded DNNs, despite that weights in vanilla DNNs (unprotected by TEEs) are free of mainstream micro-architecture side channels. We demonstrate that such weight leakage subsequently enables stealing DNN intelligence and launching white-box DNN attacks.
- To overcome technical hurdles of recovering DNN weights, we propose to directly generate functionality-equivalent weights from ciphertext side channels. We design HYPERTHEFT, which can recover DNN weights passively with only negligible and the weakest knowledge of the victim DNN. HYPERTHEFT applies to general DNNs and is capable of recovering DNN weights by observing only a few executions of the victim DNN.
- We comprehensively evaluate diverse and representative DNNs, datasets, DNN runtimes, and TEE usages, where HYPERTHEFT can constantly recover DNN weights from ciphertext side channels. We also systematically assess how public knowledge in various scenarios can boost HYPERTHEFT's capability, and conduct membership inference and bit-flip attacks based on HYPERTHEFT's recovered weights.

Research Artifact. The code and data of this chapter are available at: `https://sites.google.com/view/hyper-theft` [8].

5.2 Preliminaries and Background

5.2.1 DNNs and Terminologies

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

definitions for terms related to this chapter.

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

Definitions. We define the following terms for this chapter.

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

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

• *DNN Structure & Parameters*: Following prior literature [266, 81], structure denotes the computation graph of a DNN, which includes 1) the number of layers, 2) how each layer is implemented, and 3) how different layers are connected. For instance, LeNet and ResNet are two different structures. Parameters indicate the hyperparameters in DNN structures, e.g., kernel sizes in convolutional layers.

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

[3]Text DNN's outputs are concatenated using words from a fixed vocabulary.

inputs must be cat or dog images. Here, `cat` & `dog` images form the "task domain" of the DNN.[4]

• *Input Type*: To distinguish the subtle difference between public and private information of DNN's valid inputs, we further define "input type" over the task domain. Consider two medical diagnosis DNNs that accept chest X-ray images. Suppose the two DNN developers have training images of *non-overlapping* diseases, these two DNNs will support diagnosing different diseases, leading to different task domains. However, they share the same input type of "`chest X-ray`" image. Note that the `cat` & `dog` images mentioned above are from a different input type of "`natural`" image [33, 21]; see more cases in Sec. 5.8. In practice, having data from the victim DNN's task domain may be impractical, as these data are often private (e.g., medical images of certain diseases). However, obtaining data sharing the same input type with the victim DNN is often feasible (e.g., medical images of benign cases). Previous query-based [235, 116, 47] and hardware attacks [205, 272] require data covering the victim DNN's full task domain, while our work loosens the requirement to only input type, delivering a more practical attack.

Hyper-Network. A hyper-network is a special DNN that generates weights for a target DNN [50, 94, 209]. The machine learning community has designed various hyper-networks to study the statistics of DNN weights for a specific task [209, 287]. Nevertheless, conventional hyper-networks require data from the same task domain of the target DNN, and only generate weights of identical functionality, impeding their application in stealing DNN weights. By carefully designing the training algorithm of hyper-networks (see Sec. 5.6.2), our work presents a *task-wise* generalizable hyper-network, which can generate weights for the target DNN (i.e., the attacked TEE-shielded DNN) without using data from its task domain. By leveraging ciphertext side channels from the target DNN, our generated weights can exhibit different *unseen* functionalities.

5.2.2 TEE Protection and Mitigated Attacks

Model Stealing. Besides preventing attackers from copying DNN weights, TEE also mitigates query-based model stealing [198, 235, 116, 47] (a.k.a., knowledge distilling). To

[4]The task domain is also referred to as "problem domain" in existing literature [191].

understand how the mitigation works, we first elaborate on motivations behind this attack. Typically, attackers first query the victim DNN using their own data and then train a student model to duplicate the victim DNN's prediction confidences (i.e., an input's probabilities of belonging to *all* possible predictions) on the queried data. Because DNN training is data-intensive and costly, query-based attacks aim to obtain a functionality-equivalent student model using fewer training data (i.e., the queried data) guided by the victim DNN's confidence scores. TEE-shielded DNNs mitigate such attacks by only returning the prediction *without* confidence scores [285]. Thus, despite that attackers can still query a TEE-shielded DNN, they only label their queried data and the attack's cost becomes comparable to training a new DNN from scratch [285]. Note that the query data must cover the TEE-shielded DNN's full task domain; if the query data are from a subset of the task domain, the student model only learns the victim DNN's partial functionality w.r.t. this subset [285, 116, 47, 235].

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

Data Privacy: Membership Inference. Membership inference attack (MIA) [227, 40] aims to infer if an input is included in the victim DNN's training data — a successful MIA indicates a severe training data leakage. Existing attacks primarily leverage DNN's prediction confidence, intermediate results, and/or gradients to infer membership of a given input. Since TEE-shielded DNNs are purely black-box[5] and only return final predictions, no available information can be leveraged to infer an input's membership.

DNN Integrity: Intelligence Depletion. Unlike adversarial attacks that only fool a DNN to mis-classify the crafted adversarial examples, bit-flip attack (BFA) can *globally* deplete DNN intelligence such that the victim DNN randomly guesses predictions for almost all (non-adversarial) inputs [206, 268, 151]. To launch BFA, attackers first require localizing weight elements that are critical to the DNN's intelligence and then leverage rowhammer attacks [126] to flip bits of these weight elements. The localization process is conducted by computing gradients over different weight elements, which is prohibited by the black-box view of TEE-shielded DNNs. As a result, attackers have to randomly flip bits in a

[5]Note that previous MIA works assume that "black-box" DNNs return prediction confidence [166], which is different from the black-box in our context.

DNN's weight, which is impractical due to the massive search space and the high cost of rowhammer attacks [206, 268].

5.2.3 TEE and Ciphertext Side Channel

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

Deterministic Encryption. Two factors must be considered by TEE. First, efficient random memory access that requires independently encrypted memory blocks. Second, encrypting large memory where additional space and latency are needed for counters. To meet these requirements, modern TEEs including AMD SEV [122], ARM CCA [20], Intel TDX [114], and Intel SGX on Ice Lake SP [114, 119], adopt the *deterministic*-mode AES encryption. Specifically, given a memory block, to encrypt its memory value v, the encryption first takes a tweak function T to calculate a mask $m = T(a)$, where a is the address of the block. The encrypted ciphertext is generated as $c = P(v \oplus m) \oplus m$, where P is the encryption function. Therefore, when the same value v is stored at the same address a, the generated ciphertext is always identical (i.e., ciphertext collision).

Leakage Due to Ciphertext Collision. Existing works have leveraged ciphertext collision to infer the plaintext private keys of TEE-shielded cryptographic software [149, 146]. The attack workflow is illustrated in Fig. 5.1: ⓐ the attacker observes ciphertexts generated from two consecutive memory writes at the same address. If ciphertexts do not change, the two written values should be identical. In contrast, if ciphertexts change, different values are written. ⓑ Based on the cryptographic software's implementation, the attacker manually investigates which instruction induces the ciphertext collisions and consequently infers the plaintext private keys.

Essentially, a similar procedure can be applied to TEE-shielded DNNs. When a TEE-shielded DNN is executing, the attacker observes ciphertext collisions of its memory writes. Nevertheless, as the victim DNN's computation graph is often private, the attacker can-

not map ciphertext collisions back to their corresponding DNN computations — only a ciphertext side-channel trace that records ciphertext collisions from all the victim DNN's memory writes is available. Our community still lacks techniques to extract DNN weights from ciphertext side-channel traces.

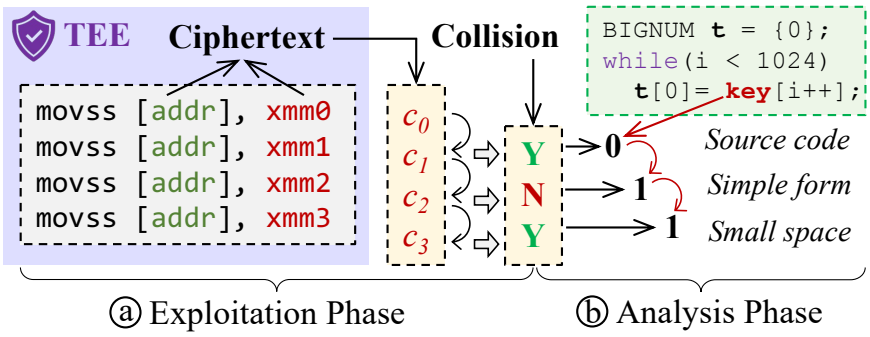


Figure 5.1: The workflow of ciphertext side-channel attacks.

5.3 Application Scope

As shown in Fig. 5.1, a ciphertext side-channel attack consists of two main phases: ⓐ an *exploitation* phase that collects side channels from the victim, and ⓑ an *analysis* phase that recovers secrets from the collected side channels. In previous attacks towards cryptographic software, the analysis phase is conducted by manually analyzing how different execution states of the victim affect ciphertext collisions [149, 146]. This is feasible given the public source code. Cryptographic keys also have a simple form and a small search space; each key only has 1∼2K *independent* binary bits and these bits *directly* determine the ciphertext collisions.

However, manually analyzing how DNN weights affect ciphertext collisions is hardly doable. In practice, the implementation of the victim DNN is often private [285], such that attackers cannot investigate how ciphertext collisions are induced by different computation operators. On the other hand, ciphertext collisions in TEE-shielded DNNs are *not* due to writing weight elements to memory since weights have been preloaded before the execution. Instead, the collisions are induced by a small portion of intermediate computation results of the DNN. As a result, ciphertext side channels only *leak partial and indirect information* of DNN weights, whereas cryptographic keys are fully leaked in prior works.

Moreover, modern DNNs have millions of weight elements, and each weight element is a floating-point number with an unconstrained range under the specified precision, leading to an almost infinite search space. Importantly, DNN weight elements are *highly correlated* — a few incorrectly recovered weight elements (since not all of them are leaked) can result in a non-functional DNN, e.g., 1∼5 incorrect ones out of ∼10M weight elements as shown in previous works [206, 268, 151], denoting a failed attack. This *high integrity requirement* of DNN weights is fundamentally different from cryptographic keys where each bit is independent: even partially recovered bits can be sufficient for practical attacks [188, 252].

Prior works have proposed side-channel analysis techniques for various DNN secrets, e.g., DNN structure [81], DNN input [278], etc. Nevertheless, since DNN weights are free of mainstream side channels, the corresponding analysis approach is rarely studied. Therefore, this chapter proposes a DNN weight analysis technique to complete the puzzle of DNN secret research, and bridges it with ciphertext side channels to study the threats of TEE-shielded DNNs. Aligned to previous analysis works [278, 81], we do *not* present a new exploitation tool (ⓐ) because existing tools are relatively mature. Rather, we focus on the analysis phase (ⓑ) and design HYPERTHEFT to automatically generate DNN weights from *already-prepared* ciphertext side channels. We aim to greatly ease the attack requirements and enhance the attack performance. HYPERTHEFT is designed to support any exploitation tools if available (e.g., `CipherLeak` [149], `SEV-Step` [259]).

Table 5.1: Knowledge required by existing attacks and HYPERTHEFT. **Alg.** indicates algorithmic query-based attacks and **HW** indicates hardware side-channel attacks. ✓ denotes public knowledge. **Task Type** and **Input Type** are public and required by all existing works. + and − indicate "required" and "not required" for private knowledge. Prediction confidence (**Conf.**) of TEE-shielded DNNs is not available in all cases.

		Observation	**Target DNN**	**DNN Knowledge**			**Data Knowledge**		**Query**
				Task Type	**Conf.**	**Struct.**	**Input Type**	**Task Domain**	
Alg.	[273, 198, 235] [116, 47, 193], etc.	Pred. Conf.	General DNN	✓	+	−	✓	+	+
HW	DeepEM [272]	Electromagnetic	Bin. DNN	✓	+	+	✓	+	+
	DeepSteal [205]	Rowhammer [126]	Quant. DNN	✓	−	+	✓	+	+
	HYPERTHEFT	Collision [149]	General DNN	✓	−	−	✓	−	−

5.4 Threat Model and Related Works

This section elaborates on the threat model and required knowledge of our work, and compares our technique with prior methods. We omit existing works that hypothesize secret-dependent computations of DNNs (e.g., a DNN prunes its weight for different inputs [113]), assume most DNN weight elements are public [36, 263], or perform brute-force guesswork [28, 75].

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

Attacker. Consistent with the objective of shielding DNNs with TEEs, we assume an untrusted host machine (e.g., a malicious hypervisor, or the host OS) which has full system privileges. Thus, attackers can read the content (i.e., encrypted ciphertext) and address of a memory write via direct software access (as demonstrated on AMD SEV [149, 146]). Besides, attackers are also capable of conducting physical attacks on TEE-shielded DNNs. For instance, attackers can leverage memory bus snooping to read the ciphertext, as exploited on Intel SGX [141]. Having that said, we do not assume a specific ciphertext side-channel exploitation approach; we aim to provide an out-of-the-box solution to automatically generate DNN weights from already-prepared ciphertext side channels.

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

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

2. Launching White-Box Attacks. As introduced in Sec. 5.2.2, the white-box access to DNN weights enables severe attack chances, compromising data privacy [227, 40] and breaking DNN integrity [206, 268, 151].

This chapter recovers DNN weights from ciphertext side channels by generating weights of equivalent functionality. Despite being different from the victim DNN's weights, our recovered weights fulfill the two attack goals (as evaluated in Sec. 5.8 and Sec. 5.9).

Target DNNs. Unlike existing attacks leveraging characteristics of certain specific DNNs (e.g., binarized DNNs whose weight elements are either 1 or -1) [272], ciphertext side-channel attacks exploit the defects in TEE's design. Therefore, our technique applies to any general DNNs as long as they are "protected" by TEEs.

5.4.1 Attacker's Knowledge and Actions

Our technique is established for highly practical attack scenarios and assumes attackers having the weakest knowledge of the victim DNN, i.e., only minimal information that specifies the DNN's basic usage — without them, attackers do not even know how to use the stolen DNN and the stealing accordingly becomes meaningless. Specifically, we consider that attackers only know the input type (as defined in Sec. 5.2.1) and the task type (i.e., classification or regression, which decides whether DNN predictions are discrete or continuous). We clarify that input type and task type are public in TEE-shielded DNNs and are also required by all existing attacks.

On the other hand, our technique does *not* only apply to this weakest-knowledge setting; it also supports incorporating stronger knowledge if available. Table 5.1 lists the attacker's knowledge required by existing works. Below, we discuss their (in-)availability under various considerations and how our technique can be further enhanced with them.

DNN Knowledge. As in Table 5.1, all existing works require knowing the victim DNN's task type. Here, the task type only distinguishes regression vs. classification and specifies the DNN's output format (i.e., continuous or discrete). It does *not* indicate the number of or which classes that a classifier can predict (since they are private). Existing query-based attacks require having the victim DNN's confidence scores, which are always unavailable in TEE-shielded DNNs [285]. Previous hardware attacks (those speed up query-based

attacks [272, 205]) rely on the victim DNN's structure. However, HYPERTHEFT works without the structure information given our functionality-centric view and the carefully developed hyper-network; see Sec. 5.6 and Sec. 5.8.

While DNN structure is protected by TEEs, considering that DNN structure can be leaked via cache side-channel attacks [105, 266, 167] and TEEs are exploitable through cache side channels as well [190, 91, 180, 245], it is reasonable to also evaluate a stronger attacker with the structure knowledge. Hence, to comprehensively assess the threat, Sec. 5.8.3 further study how the structure information may enlarge the weight leakage.

Data Knowledge. All previous works assume knowing the victim DNN's input type and having data from the victim DNN's task domain (see definitions in Sec. 5.2.1), because they must train the student model to make it functional. However, this assumption does not always hold. For example, when attacking medical DNNs, data containing certain diseases may not be publicly available. HYPERTHEFT directly generates functional DNN weights (from ciphertext side channels) without training them. Importantly, HYPERTHEFT is task-wise generalizable: it can generate weights of *unseen* functionalities *without* data from the corresponding task domain.

In case data from the victim DNN's task domain are available, HYPERTHEFT's generated weights can be leveraged to initialize the student model for queried-based attacks, significantly reducing the query cost when attacking TEE-shielded DNNs (see Sec. 5.8.3).

Attacker's Action. As marked in Table 5.1, all prior attacks assume an active attacker who can frequently query with the victim DNN. In practice, such action is often limited by the query budget (i.e., the number of queries). For instance, querying commercial DNNs incurs economic cost and DNN service providers may limit the number of queries. Our technique, in contrast, enables a passive attack: the attacker does not need to query the victim DNN, but only passively records ciphertext side channels when the victim DNN is executing. HYPERTHEFT is also highly stealthy: it only requires ciphertext side channel traces logged from the victim DNN's a few executions (with each one for a binary classification), and the cost of developing HYPERTHEFT is comparable to training a student model as in prior attacks (as elaborated in Sec. 5.6).

5.5 Explorations and Insights

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

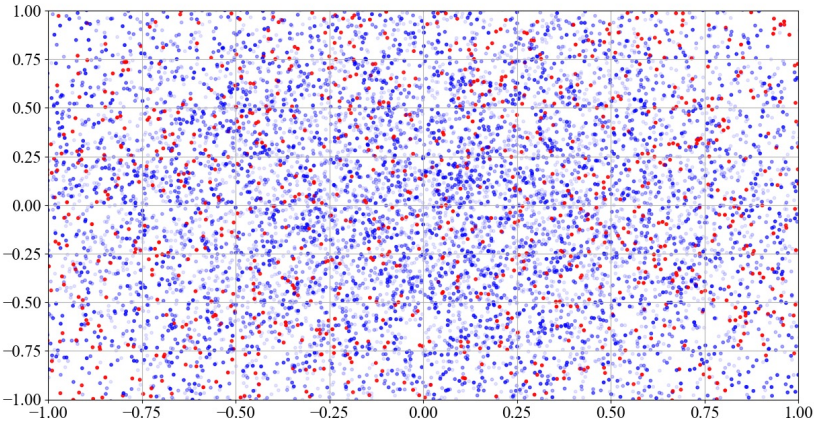
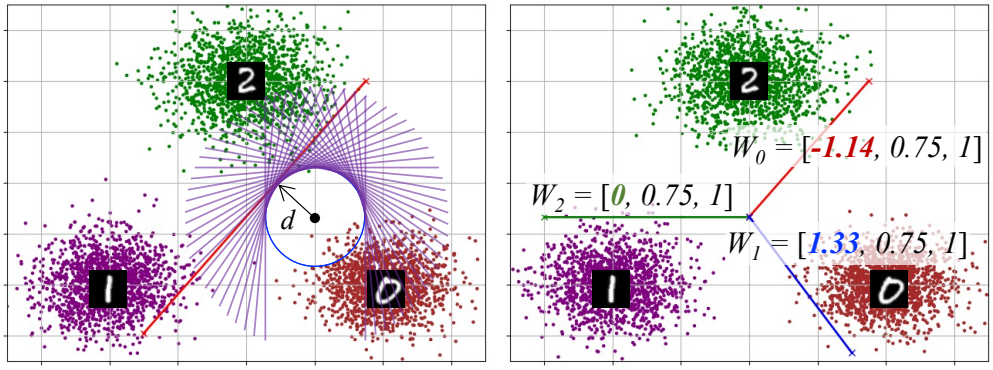


Figure 5.2: Visualization of different DNN weights w.r.t. their accuracy. Each dot denotes one DNN weight, and its coordinates (which are normalized into $[-1, 1]$) represent the values of weight elements. Weights of $> 80\%$ accuracy are marked in red. For blue dots (i.e., weights having $\leq 80\%$ accuracy), a more transparent color indicates lower accuracy.

We then project these weights onto a two-dimensional space via PCA [200] to ease the visualization. As shown in Fig. 5.2, each dot denotes one DNN weight and its coordinates indicate the values of weight elements. Red dots mark DNN weights having $> 80\%$ test accuracy, which can be deemed as functional DNN weights since they enable DNN intelligence. The remaining weights are blue-colored where higher transparency indicates lower accuracy. Fig. 5.2 reveals that, functional DNN weights (i.e., red dots) sparsely and discontinuously distribute in the whole space. Therefore, we have the following two conclusions.

First, slightly perturbing a few weight elements (i.e., changing a dot's coordinates in Fig. 5.2) can turn a functional weight (red dot) into a non-functional one (blue dot). Second, DNN weights of distinct elements (i.e., two far-flung red dots) can have equivalent functionality (i.e., solving the XOR task).

Motivation. The first conclusion is aligned to results in prior DNN attacks, which demonstrate that changing a few (out of millions) weight elements can totally deplete DNN intelligence [206, 268]. It also renders the impracticality of per-element recovery of DNN weights: as long as the exact value of a weight element is not recovered under this scheme (e.g., some elements are not leaked), the inferred DNN weight is likely non-functional. The second conclusion can be drawn from DNN training, whose different runs generate distinct but equivalent weights. It is also aligned to existing DNN pruning works, where a DNN's functionality remains same after replacing more than 90% of its weight elements with zeros [288]. This conclusion sheds light on the feasibility of *recovering different but functionality-equivalent DNN weights from partial observations of the victim DNN's weight.*



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

Figure 5.3: Decision boundaries for three classes.

DNN Functionality. A DNN's intended task uniquely decides its functionality. Fig. 5.3 shows an example where a DNN, whose last layer is $y = Sigmoid(\theta x + b)$[6], classifies three clusters of digits "0", "1", and "2". This task requires the DNN to split the input space (which consists of all valid digits "0", "1", and "2") to separate different digits. The DNN accordingly forms the required functionality by training θ and b. Each row in the concatenated matrix $W = [\theta, b]$ indicates a line drawn by the DNN. Given the trained

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

weight:

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix} = \begin{bmatrix} -1.14 & 0.75 \\ 1.33 & 0.75 \\ 0 & 0.75 \end{bmatrix}, \quad b = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \tag{5.1}$$

the first row $[\theta_0, b_0] = [-1.14, 0.75, 1]$ is marked as the red line, $\theta_0 \cdot x + b_0 = 0$, in Fig. 5.3(b). After drawing all three lines as in Fig. 5.3(b), the DNN's is capable of classifying digits. Overall, DNN structure (i.e., the dimension of W) reflects *how many* lines are drawn, and DNN weight decides *where and how to draw these lines*.

Intuitively, each row of $[\theta, b]$ also characterizes a binary classification. In Fig. 5.3(a), the red line separates "0" from other digits (i.e., classifying if a digit is "0"). Similarly, the blue and green lines in Fig. 5.3(b) (derived from the second and third rows of $[\theta, b]$) classifies if a digit belongs to "1" or "2", respectively. As in Eq. 5.1, despite that the three rows of $[\theta, b]$ only differ in the first element, they represent distinct functionalities for different binary classifications; this is consistent with our first conclusion delivered from Fig. 5.2.

Intermediate Outputs Reflect Functionality. Given an input x, each layer's output (a.k.a. the intermediate output) describes x's relative position w.r.t. the lines drawn by this layer. Considering Fig. 5.3(a) where an input is marked as the black dot, suppose the first element of its intermediate output (from the layer we discussed above) is $-d$ ($d > 0$), we know that x is below (since $-d < 0$) the red line $\theta_0 \cdot x + b_0 = 0$ and the distance is $\frac{d}{|\theta_0|}$ (see proof in Sec. 5.12.2). With this information, we can infer that the first row of $[\theta, b]$ corresponds to a line that locates in the region covered by the purple lines in Fig. 5.3(a). While in large DNNs, the above case may become more complicated due to layer propagations and the high non-linearity, we can safely conclude that a DNN's intermediate outputs reflect its functionality. Furthermore, given that ciphertext collisions are due to intermediate computations of TEE-shielded DNNs, it is therefore reasonable to believe that *DNN functionality can be reflected from ciphertext side channels*.

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

DNN functionality from ciphertext side channels, and how to limit the required victim's knowledge to only public information. Below, we introduce our solution in Sec. 5.6.

5.6 Solution and Technical Details

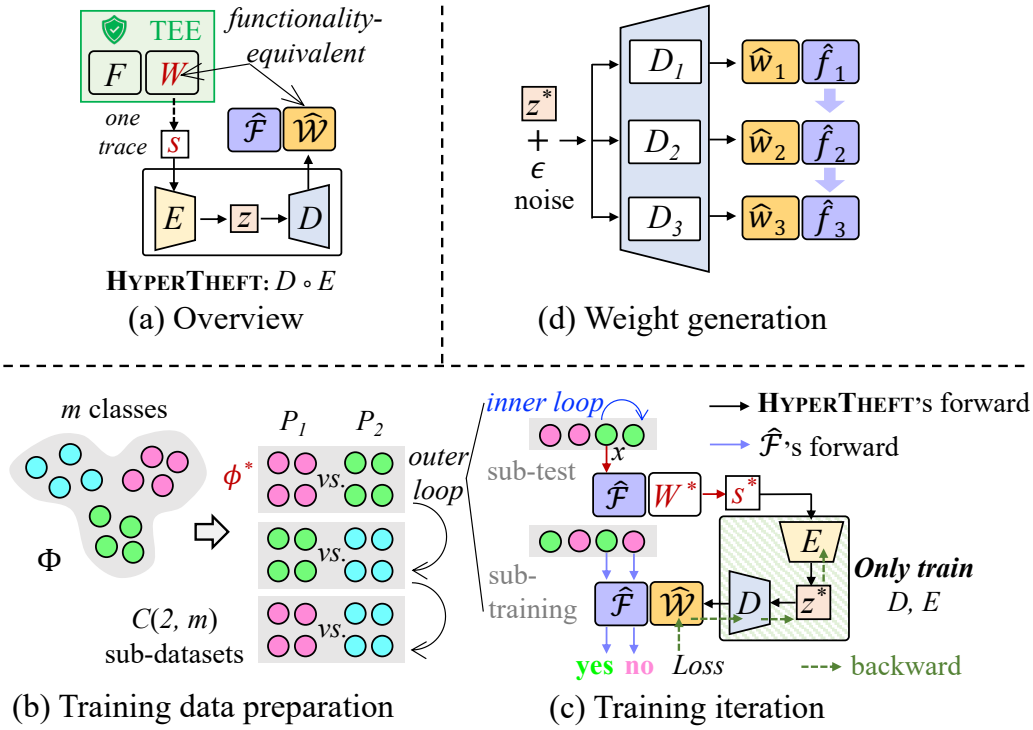


Figure 5.4: Workflow of HYPERTHEFT. In Fig. 5.4(a), F and its input, output, and weight W are protected by TEEs. HYPERTHEFT only takes F's one ciphertext side channel trace s as input and generates a different but functionality-equivalent weight $\hat{\mathcal{W}}$ for a surrogate model $\hat{\mathcal{F}}$. $\hat{\mathcal{F}}$ has a different structure from F. Fig. 5.4(b) illustrates how our training "data" are constructed as different binary classification (or regression) tasks. In Fig. 5.4(c), we illustrate how each training iteration is performed. Fig. 5.4(d) shows how we implement stochastic generation via random noise ϵ and separately generate weight $\hat{w}_i$ for layer $\hat{f}_i$.

5.6.1 Overview and Goals

Fig. 5.4 illustrates the workflow of HYPERTHEFT. Similar to existing automated analysis approaches [278, 81], our technical pipeline also consists of an offline and an online stage. As illustrated in Fig. 5.4(a), when attacking an unknown TEE-shielded DNN F of unknown weight W in the online stage, we collect a ciphertext side channel trace s from F's

execution. We then feed s to HYPERTHEFT to directly generate weight $\hat{\mathcal{W}}$ for a surrogate model $\hat{\mathcal{F}}$ (whose structure can be different from F), so that $\hat{\mathcal{F}}$ is functioning consistently with F.

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

❶ Handling the partially leaked weight information;

❷ Forming task-wise generalizable weight generation;

❸ Capturing functionalities and their equivalence;

❹ Generating functional weights from a single trace;

❺ Supporting both regression and classification tasks;

❻ Modeling correlations between weight elements;

❼ Maximizing performance with limited observations.

In the following, we first introduce how to build HYPERTHEFT and the training data, pipeline, and objective w.r.t. to the seven goals in Sec. 5.6.2. Then, we introduce our in-depth optimizations in Sec. 5.6.3.

5.6.2 Building and Training HYPERTHEFT

Encoder and Decoder (❶). As illustrated in Fig. 5.4(a), HYPERTHEFT consists of two components: an encoder E that converts ciphertext side channel s into a latent variable z, and a decoder D that generates DNN weights $\hat{\mathcal{W}}$ according to z. Here, we let z have a much lower dimension than both s and $\hat{\mathcal{W}}$ due to the following reasons: 1) considering the frequent memory accesses in DNNs which result in lengthy s (i.e., containing millions of ciphertext collision records), a lower-dimensional z can force E to neglect irrelevant records in s and to focus on functionality-related information (guided by proper training objectives, as will be introduced later). Besides, 2) the dimension expansion process in D (i.e., from z to $\hat{\mathcal{W}}$) encourages D to infer the *unleaked* information (as ciphertext side channels only leak partial weight information), instead of merely transmitting information in

s. For efficiency, we implement E and D as multilayer perceptrons (MLPs); empowered by our training algorithms, such simple forms of E and D work sufficiently well in practice.

Training "Data" Construction (❷, ❺). Unlike conventional DNN training data that is formed as a set of input-output pairs, HYPERTHEFT's "training data" is a set Φ of different binary classification or regression *tasks*. This setup helps the weight generation generalize from known (training) tasks to unknown (test) tasks. We first collect some data that have the same input type (e.g., merely natural images) with the victim DNN. Note that their task domain does *not* overlap with the victim DNN's task domain. As in Fig. 5.4(b), if the victim DNN performs a classification task and suppose the attacker's data have total m classes, we form $C(2, m)$ [7] (i.e., the number of 2-combinations for a set of m elements) sub-datasets with each for a binary classification task. For victim DNNs performing regression tasks, attackers can randomly divide their data into sub-datasets for different regression tasks.

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

Per-Task Granularity and Single-Trace Input (❷, ❹, ❼). As shown in Fig. 5.4(b)-(c), in each training iteration, we randomly pick a task ϕ^* and its corresponding trained DNN $\hat{\mathcal{F}}_{W^*}$. We then randomly select one input data x from ϕ^*'s sub-test set and feed it to $\hat{\mathcal{F}}_{W^*}$. It's worth noting that using x from the sub-test set does *not* misuse the dataset because our "data" are split as training and test *tasks*; these sub-test sets belong to training tasks. When $\hat{\mathcal{F}}_{W^*}$ is executing with x, we collect a ciphertext side channel s^* as one training input of HYPERTHEFT. Then, HYPERTHEFT takes s^* and outputs $\hat{\mathcal{W}}$. HYPERTHEFT is optimized to generate $\hat{\mathcal{W}}$ that is functionality-equivalent to W^* (i.e., $\hat{\mathcal{F}}_{W^*}$'s weight).

Note that each input of HYPERTHEFT is a ciphertext side-channel trace logged from the victim DNN's only *one* execution. Recall as explored in Sec. 5.5, a DNN's one execution can reflect rich information of its functionality. The single-trace setup can make the

[7] In practice, using all $C(2, m)$ sub-datasets is usually unnecessary. Our results in Sec. 5.8 show that 25-30 sub-datasets are sufficient.

online attack stealthy and minimize the online attack's cost. Sec. 5.6.3 will introduce our optimizations for this single-trace setup.

Functionality-Centric Training Objective (❶, ❷, ❸, ❻). Our training objective aims to measure the equivalence between the generated weight $\hat{\mathcal{W}}$ and the target weight W^*. As elaborated in Sec. 5.5, element-wise distance metrics are inapplicable to DNN weights: a weight having similar elements with W^* may exhibit a distinct functionality and is even non-functional. Given that W^*'s functionality was formed when training W^* on ϕ^*'s sub-training set, we can also enable an equivalent functionality for $\hat{\mathcal{W}}$ using this sub-training set. Nevertheless, we should *not* directly train $\hat{\mathcal{W}}$ on the sub-training set, because the victim DNN's training data (or data from the same task domain) is unavailable during online attack.

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

$$\arg\min_{D,E} L(\hat{\mathcal{F}}_{\hat{\mathcal{W}}}, \phi^*), \quad \text{where} \quad \hat{\mathcal{W}} = D \circ E(s^*). \tag{5.2}$$

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

As illustrated in Fig. 5.4(c), similar to conventional DNN training, this objective minimizes the loss L by back-propagating through $\hat{\mathcal{F}}_{\hat{\mathcal{W}}} \rightarrow D \rightarrow E$. However, it does not directly train $\hat{\mathcal{F}}_{\hat{\mathcal{W}}}$, but only trains E and D to optimize their output $\hat{\mathcal{W}}$. As a result, $\hat{\mathcal{W}}$ generated from the subsequent iterations (after training E and D) can reduce the loss $L(\hat{\mathcal{F}}_{\hat{\mathcal{W}}}, \phi^*)$. Finally, the generated $\hat{\mathcal{W}}$ will have an equivalent functionality with W^* when it reduces $L(\hat{\mathcal{F}}_{\hat{\mathcal{W}}}, \phi^*)$ to a satisfactory level.

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

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

ferent unseen tasks, $D \circ E(s^*)$ generates weights that are capable of solving the corresponding tasks. Importantly, we find that HYPERTHEFT can generate functional weights for DNNs that are much larger than HYPERTHEFT. That is, we do *not* need to build a huge HYPERTHEFT whose weight subsumes all functionality-equivalent weights of the victim DNN. Essentially, the objective in Eq. 5.2, together with the dimension expansion of D, enable HYPERTHEFT to infer unleaked information in s^* through the functionality perspective. Moreover, representing functionality should require less information than representing the weight itself; this is also supported by existing DNN weight pruning works [78, 288, 175].

5.6.3 Optimizations for HYPERTHEFT

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

Worse, our tentative experiments show that HYPERTHEFT rarely converges under the above setup, because it poses conflicts between different training iterations. Suppose after one iteration, HYPERTHEFT is able to generate (nearly) functional weights for the task ϕ^*. The subsequent iteration, however, requires HYPERTHEFT to generate functional weights for another different task. Since universal weights (that can solve all tasks) do not exist [219], fulfilling the subsequent iteration's requirement may break HYPERTHEFT's generation ability formed in the current iteration.

To reduce the overhead and alleviate conflicts between different iterations, we adopt a sampling strategy. Instead of using ϕ^*'s whole sub-training set, we randomly sample one data instance from the sub-training set to optimize Eq. 5.2. Different from using the whole sub-training set that *independently and subsequently* trains HYPERTHEFT for each task, this sampling strategy *jointly* trains HYPERTHEFT for all tasks over multiple iterations. It, to some extent, explores the similarity between tasks and helps HYPERTHEFT to form the task-wise generalization. In addition, the sampling only reduces the cost of each training iteration; we still have sufficient iterations (that use different s^* as E's inputs) to train E.

Decoupling Functionality (❹, ❺, ❼). As discussed in Sec. 5.4, TEE-shielded DNNs only return the final prediction, and we do not assume knowing how many (i.e., reflected from DNN structure) or which (i.e., the task domain) classes the victim DNN can predict. Our current attack pipeline is adequate for binary classification and regression tasks. Below, we discuss how it can be applied to k-class classification ($k > 2$).

As explored in Sec. 5.5, the full functionality of k-class classification can be decoupled as k (sub-)functionalities of binary classification. Thus, attackers can steal the full functionality via k surrogate models. However, the binary classification decoupled from k-class classification (i.e., whether an input is from a class or not) is slightly different from our training binary classification (i.e., classifies two different classes). To address this, we actively flip the two labels of each sub-training set when training HYPERTHEFT.

As illustrated in Fig. 5.4(b)-(c), the sub-dataset has two classes P_1 and P_2. If the ciphertext side channel s^* is collected when $\hat{\mathcal{F}}_{W^*}$ is taking an input x from class P_2, we set P_2 in the sub-training set as the label "`yes`" whereas P_1 as the label "`no`", and vice versa if x is from class P_1. This way, every time a TEE-shielded DNN executes with an input x of class P_x, HYPERTHEFT can generate a surrogate model which is able to predict whether an input belongs to P_x or not. By collecting ciphertext side channels from the victim DNN's (minimal) k executions, HYPERTHEFT can generate k surrogate models where the i-th surrogate model predicts an input's confidence of belonging to the i-th class, thereby stealing the full functionality. Since HYPERTHEFT's weight generation generalizes across tasks, we only need to train HYPERTHEFT once.

Stochastic and Layer-Wise Generation (❹, ❻, ❼). Inspired by DNN's stochastic training algorithms (e.g., SGD [17], Adam [129]), we also implement a stochastic weight generation by adding a random noise ϵ to z^*, as illustrated in Fig. 5.4(d). This way, HYPERTHEFT can more extensively utilize one logged side channel to generate different but equivalent weights in different runs. The resulting surrogate models (for the same binary classification or regression task) can form a majority voting (i.e., using the most frequent prediction from these surrogate models as the final prediction) to improve the accuracy [80].

In addition, following advice from [209, 287], HYPERTHEFT uses one encoder E, but n independent decoders $D_1, \ldots, D_n$ to generate weights $\hat{\mathcal{W}} = \{\hat{w}_1, \ldots, \hat{w}_n\}$ for layers $\hat{f}_1, \ldots, \hat{f}_n$, respectively. As shown in Fig. 5.4(d), each D_i takes an identical input $z^* + \epsilon$

(i.e., the ϵ is fixed after sampled). Beyond our functionality-centric objective that implicitly captures correlations between weight elements, this layer-wise generation can explicitly model the layer propagation in DNNs, while considering the independent execution of each layer.

5.7 Implementation and Setup

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

Side Channel Preparation. Since a TEE-shielded DNN isolatedly operates in its memory, and given the high privilege of attackers (e.g., hypervisor, OS), ciphertext side channel distinguishes other conventional side channels by its clean and noise-free nature [149, 146]. Also, unlike prior DNN attacks, HYPERTHEFT does not need to interact with or log ciphertext side channels from the victim DNN in the offline stage. Hence, we can speed up the offline data preparation by mimicking an exploitation tool using Intel Pin [170] on attacker's own DNNs. During the online stage, we use these exploitation tools to collect ciphertext side channels from (unknown) TEE-shielded DNNs and steal their weights. To date, two mature exploitation tools, `CipherLeak` [149] and `SEV-Step` [146, 258] have been proposed. `CipherLeak` is coarse-grained but is more scalable by operating in a page granularity, while `SEV-Step` can precisely track memory write in an instruction granularity but is costly.

We configure `CipherLeak` following the default setup. However, when setting up `SEV-Step`, we note that its current implementation is based on Linux kernel Ver. 5.14, which is too old to be compatible with the latest SEV-SNP firmware (Ver. 1.55) required

to launch a guest VM for DNNs. We have contacted the developers for support, and the upgrading is still in process by the time of submission given the considerable manual efforts required. Hence, we simulate `SEV-Step` using Intel Pin. Our preliminary explorations show that ciphertext side channels collected using our Pin-based simulation and `SEV-Step` are identical (on those programs supported by both). As suggested by the developers, a potential (and might be the only) factor that could differ our Pin-based simulation from `SEV-Step` is due to the multiple memory writes occurred during a given APIC timer interval, where some memory writes can be periodically missed by `SEV-Step` [258]. Thus, for completeness, we also benchmark HYPERTHEFT towards this impact in Sec. 5.12.1. Overall, HYPERTHEFT constantly achieves promising performance even when considerable (e.g., $^{63}/_{64}$) memory writes are missed.

Considering the stealthy and efficiency, we employ `CipherLeak` to exploit large DNNs (e.g., ViT [72]). Note that it is often impractical to put these extremely large DNNs into TEEs. Existing works divide large DNNs into slices and only shield sensitive slices via TEEs [179, 108, 285], leaving other slices as public. Therefore, we adopt HYPERTHEFT to recover weights of the TEE-shielded DNN slices. For moderate-size DNNs (e.g., ResNet, LSTM) that can be fully shielded by TEEs, we use `SEV-Step` (simulated via Pin due to the compatibility issue) to collect ciphertext side channels and recover the full DNN weights.

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

Multi-Threading. While the matrix computations in DNNs are implemented as multi-threading, we do *not* observe non-deterministic ciphertext collisions because the memory management in modern DNN runtime is clean: each thread primarily performs computations of a specific region in the matrix, and its memory accesses are restricted to the assigned small memory region.

5.8 Evaluation

In this section, we first introduce the evaluation setup in Sec. 5.8.1. We then evaluate HYPERTHEFT under the weakest knowledge in Sec. 5.8.2, where victim DNNs are running with the latest version of PyTorch (ver. 2.1.0). Sec. 5.8.3 evaluates the attack surface by considering DNN executables and different versions of PyTorch, and studies how stronger knowledge (under certain possible scenarios) can enhance the weakest-knowledge attacks.

Sec. 5.9 further shows that attacks mitigated by TEEs can be largely enhanced by our recovered weights. In Sec. 5.12.3, we present DNN modules that induce the leakage. Overall, the leakages are due to basic computation operators shared by different DNNs.

5.8.1 Evaluation Setup

Table 5.2: Evaluated datasets and victim DNNs. ImageNet is evaluated under a *cross-dataset* setting. For ViT, we recover the weights of the multi-head self-attention layers.

Dataset	Input Type	Task Type	Remarks	DNNs	MSE/Acc.
Stock [120]	Stock price	Reg.	Sequence	LSTM	1.46∼1.95
Chest X-ray [246]	Medical image	Classif.	2-class	LeNet ResNet	90∼95% 90∼95%
MNIST [68]	Digit	Classif.	2-class	LeNet ResNet ViT	94∼98% 94∼98% 94∼98%
CIFAR10 [134]	Natural image	Classif.	2-class	LeNet ResNet ViT	90∼95% 90∼95% 90∼95%
ImageNet [67]	Natural image	Classif.	Multi-class & Cross-dataset	7 DNNs*	90∼95%

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

DNNs. Table 5.2 shows our evaluated DNNs, which are popular and representative. In particular, LeNet has a sequential structure whereas ResNet has a non-sequential structure. They are widely employed as (part of) modern DNNs backbone. LSTM has a recurrent structure, which usually processes discrete data sequences. The ViT and the multi-head self-attention mechanism are the building blocks of recent large language models (LLMs).

TEE Usage. For classical moderate-size DNNs (DNNs in Table 5.2 except for ViT), we put the full DNNs into TEE and generate their full weights using HYPERTHEFT. However, given the large size of Vision Transformer (ViT) [72], it is impractical to shield the full DNN via TEEs. Following recent works [179, 108, 285], we consider shielding only sensitive slices of ViT and using HYPERTHEFT to recover weights of the shielded DNN slices. Since ViT's effectiveness is due to the self-attention, we shield ViT's multi-head self-attention layers via TEEs and use HYPERTHEFT to generate the corresponding weights.

"Data" (i.e., Task) Construction. Table 5.2 lists our adopted datasets. Stock dataset is used to predict the stock price for different companies which is a regression task. We divide Stock dataset according to the company to form different regression tasks. We use MNIST, CIFAR10, and Chest X-ray to evaluate HYPERTHEFT for binary classification w.r.t. different input types. For each dataset, we randomly choose two classes to form a binary classification as our test "data". The remaining classes are used to construct $C(2,8) = 28$ binary classifications as tasks in our training "data", so that task domains of HYPERTHEFT's training and test data/tasks do not overlap.

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

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

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

much simpler than the victim DNNs.

Note that ViT (and all transformer-based DNNs) is implemented using only fully-connected (FC) layers. Also, in the slice-based protection, the structure of each TEE-shielded slice can be easily inferred from public DNN slices. Therefore, our surrogate model has the same structure as the ViT's self-attention layer. However, given the dense computations of FCs, training ViT requires careful regulations like dropout to avoid over-fitting [72], which can be private in practice. Thus, for the weakest-knowledge setting in Sec. 5.8.2, we assume attackers do not know the regulation. Sec. 5.8.3 then evaluates impacts of the regulation information.

Evaluation Criteria. Following existing works [116, 285], we use two criteria, fidelity and functionality, to evaluate DNN weights recovered by HYPERTHEFT. Fidelity (*Fid*) calculates the percentage of test inputs (from the sub-test set of each test task) where the surrogate model and the victim DNN have identical predictions (including incorrect prediction). For classification, *Fid* can be directly calculated via predicted labels, whereas for regression, because the prediction is continuous numbers, we deem two predictions as identical if their difference is less than 2 (the stock prices vary with a range around 100). Functionality (*Fun*) denotes the task's own evaluation metric. *Fun* is the MSE or accuracy of all test inputs for regression or classification tasks, respectively. Higher accuracy indicates better results whereas lower MSE is better.

5.8.2 The Weakest-Knowledge Attack

In this section, we generate 100×5 different weights for each victim DNN, leading to more than 8K weights and evaluation results (for 17 different victim DNNs). To better reflect the average and fluctuations, we report the ranges of these results in Table 5.3.

Regression & Binary Classification. As shown in Table 5.3, HYPERTHEFT can successfully recover DNN weights for both regression and binary classification using a *single* side-channel trace logged during the victim DNN's one execution. The recovered weights are functional (*Fun*) and also consistent (*Fid*) with the victim DNN's weights. We note that *Fid* is higher than *Fun* on average, because *Fid* also counts incorrect predictions. It also reflects that HYPERTHEFT infers specific behaviors of the victim DNN beyond its

Table 5.3: Results of the weakest-knowledge attack.

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

overall functionality, despite that HYPERTHEFT never queries the victim DNN. Recall that ciphertext collisions are generated due to the victim DNN's intermediate outputs. As explored in Sec. 5.5, these intermediate outputs specify both a DNN's functionality (i.e., how the input space is split) and its specific behaviors (i.e., where and how the splitting lines are drawn), rendering the superiority of HYPERTHEFT's weight generation scheme.

Multi-Class Classification. As shown in the 14-24 rows in Table 5.3, HYPERTHEFT is capable of (passively) generating weights for classification of more classes. Although classifying multiple classes is more challenging, HYPERTHEFT alleviates this hurdle by cleverly decoupling the k-class classification as k binary classifications. Nevertheless, we note that Conv (the 14th row) exhibits a relatively lower capability of classifying 10 classes due to its insufficient depth. Nevertheless, when using $\text{Conv}_{\text{deep}}$ (the 15th row) as the surrogate model's structure, the *Fun* and *Fid* are largely improved.

Majority Voting. Table 5.3's 17-24 rows show that majority voting among multiple surrogate models (for the same task) can further improve *Fid* and *Fun*. As observed, while a single surrogate model based on Conv performs worse when classifying 10 classes, the re-

sults can be improved as being comparable to $\text{Conv}_{\text{deep}}$ with multiple Conv-based surrogate models. Note that we introduce stochasticity into the weight generation for majority voting, these different DNN weights are therefore generated using a single side channel trace from the victim DNN's one execution.

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

5.8.3 Attack Surface in Different Cases

Table 5.4: Results of Glow and various PyTorch versions. We evaluate binary classification w/o using majority voting.

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

Different Runtimes. To disclose the widespread leakage, we evaluate different versions

of PyTorch. We also consider Glow, a popular deep learning compiler that compiles DNNs into executables. Since Glow does not evolve too much, we only evaluate its latest version. Overall, the same DNN's internal computations are implemented distinctly in PyTorch and Glow-executables.

As in Table 5.4, for both Glow-executables and different versions of PyTorch, HYPERTHEFT can constantly recover the victim DNN's weights, indicating HYPERTHEFT's superiority and the wide existence of ciphertext side-channel leakage in different runtimes. That is, the leakage is presumably due to issues in DNN's own design (see detailed discussion in Sec. 5.10), rather than implementation defects in a specific runtime or version.

Table 5.5: Generating weights using ciphertext side channel traces logged from the victim DNN's multiple executions.

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

Multiple Executions. To evaluate how multiple side-channel traces (derived from *different* executions of the victim DNN) can improve HYPERTHEFT's recovered weights, we let HYPERTHEFT generate one weight for each trace and conduct majority voting among these weights. Results are in Table 5.5. Compared with the 17-19th rows in Table 5.3, the improvements brought by majority voting among weights generated via, 1) multiple traces vs. 2) HYPERTHEFT's multiple runs using one trace, are comparable, indicating the merit of HYPERTHEFT's stochastic generation.

Knowledge of DNN Structure. We evaluate how structure information boosts the attack by using the victim DNN's structure for the surrogate model. By cross-comparing Table 5.6 with the 2nd-15th rows in Table 5.3, we see that the results w/ and w/o structure information are comparable for binary classification and regression. However, when knowing the structure information, the result of 10-class classification is better than using Conv as the surrogate model, but is comparable to the $\text{Conv}_{\text{deep}}$ case in Table 5.3. This observation is consistent with our conclusion derived from Table 5.3: the structure information primarily helps attackers to determine an appropriate depth for the surrogate model. However, this can be complemented by majority voting among multiple surrogate models (generated using single trace or multiple traces). For ViT cases, while the regula-

tion mechanism is critical when training a ViT from scratch, it does not notably affect HYPERTHEFT's performance. We infer that those public DNN slices' weights, which were jointly trained with private slices' weights under the same regulation, help HYPERTHEFT to encode the regulation into its generated weight.

Although structures may differ in terms of connectivity or hyperparameters, as shown in Sec. 5.12.3, their implementations share the same vulnerable computing operations. E.g., the `cascade_sum` function (which performs pairwise sum) is frequently called by different layers. This further highlights the severity of weight leakage in TEE-shield DNNs. Previous works often require the exact structure information to boost query-based model inference [205], whereas HYPERTHEFT can enhance query-based attacks *without* such information, as evaluated below.

Table 5.6: Attack using the victim DNN's structure.

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

Task Domain & Query. As generally assumed by prior query-based attack [205, 116, 47], even if the victim DNN's training data are private, it is possible to have some other data that cover the victim DNN's task domain. For instance, attackers have some public cat and dog images when attacking a DNN classifying cat vs. dog. Therefore, we also evaluate how HYPERTHEFT can be further enhanced in this scenario.

To ease the comparison with previous works, we follow their settings where attackers query the victim DNN using in-task-domain data, but only use the predictions (*without* confidence scores) as labels to train a student model. Differently, we do not train the student model from scratch — the student model is trained based on HYPERTHEFT's gener-

Table 5.7: Attack with victim DNN's in-task-domain data.

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

* *Fun* achieved by HYPERTHEFT under the weakest attack; see Table 5.3.
✗: training the student model (i.e., Conv) from scratch.
✓: initializing the student model (i.e., Conv) with our generated weights.

ated weights (the generation does not use in-task-domain data). Since prior query-based attacks primarily focus on classification, we evaluate classification tasks.

Results are given in Table 5.7. Aligned to existing works, we report the query budget as its relative percentage to the number of victim DNN's training data. By cross-comparing Table 5.3 with the 8th, 9th, and 14th rows in Table 5.7, we see that, in order to achieve comparable results with HYPERTHEFT's weakest-knowledge attack, query-based attacks require at least 70% query budget. In contrast, while HYPERTHEFT's generated weights from the weakest-knowledge attack are not as good as the victim DNN, they can reach the same *Fun* as the victim DNN with only ∼15% query budget, significantly reducing the cost. In that sense, HYPERTHEFT enables query-based attacks for TEE-shielded DNNs since TEE's mitigation aims to largely increase the query budget (see Sec. 5.2.2).

5.9 Enabled Attacks of HYPERTHEFT

Recall that as introduced in Sec. 5.4, attackers can also leverage the recovered weights to enable white-box attacks towards the victim DNN. This section accordingly evaluates how HYPERTHEFT can enable two popular attacks, membership inference attack (MIA) and bif-flip attack (BFA).

Table 5.8: Attack success rate (ASR) of membership inference attacks enabled by HYPERTHEFT. Upper bound (UB) denotes ASR on the white-box victim DNN. Baseline is 50%.

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

5.9.1 Membership Inference Attack

This section evaluates how HYPERTHEFT's recovered DNN weights, which give attackers white-box surrogate models, can enable/enhance MIA towards the (black-box) victim DNN.

Setup. Following the setup in previous works [285], we construct a test suite where 50% of its data are the victim DNN's training data (i.e., only data from the corresponding sub-training split) and the remaining 50% are non-training data. Therefore, the baseline attack success rate (ASR) is 50% [285]. Note that training and non-training data in the test suite have the same class, such that MIA will not downgrade to class-wise classification. We adopt MLDoctor [166] to conduct MIA. Each time we feed an input from the test suite into HYPERTHEFT's generated surrogate model, and record outputs from all layers of the surrogate model. These outputs are then concatenated and fed into MLDoctor to predict the membership. In this setting, the surrogate model is generated under the weakest-knowledge attack in Sec. 5.8.2.

Results & Analysis. Table 5.8 lists the MIA results. Interestingly, despite that the surrogate model is never trained with victim DNN's training data, it still significantly improves the ASR (from 50%) to ∼65% for MNIST and ∼57% for CIFAR10 and ImageNet. As a reference, the ASR of directly applying MLDoctor on the white-box victim DNN (i.e., upper bound ASR) is 80% for MNIST and 67% for CIFAR10 and ImageNet. Recall that as evaluated in Sec. 5.8.2, besides stealing the overall functionality from the victim DNN, HYPERTHEFT also infers some of its specific behaviors (e.g., predictions for specific inputs), which explains why HYPERTHEFT's generated surrogate model is useful for MIA towards

the victim DNN.

Table 5.9: Results of bit-flip attack (BFA) enabled by HYPERTHEFT. BFA requires knowing the victim DNN's structure.

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

5.9.2 Bit-Flip Attack

We also evaluate how HYPERTHEFT's recovered DNN weights can enable BFA. As introduced in Sec. 5.2.2, to conduct BFA, the main prerequisite is localizing elements in a DNN's weight that are critical to the intelligence (which is infeasible in TEE-shielded DNNs), such that bits can be flipped efficiently. Note that BFA requires knowing the victim DNN's structure and launching rowhammer in TEEs may have additional challenges [46, 43]. However, they are out of the scope of this chapter; we primarily focus on weight-related requirements that are enabled by HYPERTHEFT.

Setup. To assess how HYPERTHEFT boosts BFA, we generate the surrogate model under the weakest-knowledge + structure setting. We follow the localization strategy in DeepHammer [268], the state of the art in this field, to localize critical weight elements[9] in the surrogate model and measure their overlapping (in terms of locations in the DNN's structure) with critical weight elements in the victim DNN. Two metrics, precision and recall, are adopted in this evaluation. Precision quantifies the percentage of victim DNN's critical elements that are localized in the surrogate model. Recall, in contrast, measures how many weight elements localized in the surrogate model are also critical in the victim DNN.

Results & Analysis. Table 5.9 reports the results. The recall values are promising: weight

[9]Most existing BFA works focus on quantized DNNs and identify weight bits [206, 268]. Since our evaluated DNNs are general floating-point DNNs, our evaluations mainly focus on the weight element level.

elements identified in the surrogate model are highly likely to be critical in the victim DNN. While the precision is relatively lower (i.e., not all critical weight elements in the victim DNN can be identified via the surrogate model), it should not be a concern. In fact, launching BFA does not require identifying all the critical weight elements [206, 268]; usually $\sim$10 flips can completely deplete a DNN's intelligence. The identification step primarily helps attackers to filter out non-critical bits since flipping bits via rowhammer is costly [126, 206, 268]. With this regard, recall should be a more important metric than precision. Thus, we can conclude that our current results are sufficient to deliver the prerequisite of BFA against TEE-shielded DNNs.

5.10 Discussion and Mitigation

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

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

Hardware and Software Mitigations. Recent TEEs (e.g., Intel TDX [53], ARM CCA [153]) redesign hardware architectures to mitigate ciphertext side channels. For example, Intel TDX always returns zeros when outer programs read the encrypted memory. However,

such hardware mitigations require modifying the current hardware design, which is impractical for TEEs that have been broadly used (e.g., AMD SEV [122]). Importantly, they cannot fully eliminate ciphertext side channels because attackers can still access the ciphertext via DMA devices [230] or physical attacks such as memory bus snooping [141], cold boot attack [92], etc.

Li et al. [146] propose to mitigate the leakage via VMSA randomization, so that ciphertext is no longer deterministic. However, this scheme incurs considerable performance overheads and is not adopted by vendors. On the other hand, we foresee the high feasibility of implementing software-level randomization to specifically mitigate the leakage in TEE-shielded DNNs. In fact, DNNs exhibit robustness to random small perturbations [83] (not carefully crafted adversarial perturbations [42]) on their intermediate outputs. Hence, every time a DNN is executing, we can add random noise to DNN's intermediate outputs before they are written into memory. Since the main purpose is diversifying the written values and reducing collisions, we can make the noise negligible. To further minimize the impact on DNN's accuracy, we expect to accordingly refine conventional training algorithms to make DNNs robust to such noise. For example, existing robust training schemes [203] (which improve DNN's robustness to input perturbations) can be adapted for noise in DNN's intermediate outputs.

5.11 Conclusion

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

5.12 Appendix for Chap. 5

5.12.1 Impacts of APIC Timer

As mentioned in Sec. 5.7, multiple memory writes can happen during a given APIC timer interval [258], which could differ `SEV-Step` from our Pin-based simulation. To benchmark the impact, we only record every *step*-th memory write and then measure ciphertext collisions. We consider $step = 8$, 16, 32, and 64.

Results are reported in Table 5.10 and Table 5.11. We note that when $step \leq 32$, the *Fun* and *Fid* values (2nd-4th rows in Table 5.10 and Table 5.11) are comparable to our results in Sec. 5.8.2 and Sec. 5.8.3. Only $step = 64$ notably degrades the *Fun* and *Fid* values; however, the impacts are *not* significant. For example, when using Conv as the surrogate model's structure, even when $step = 64$ (i.e., only one memory write is logged among 64 memory writes), the *Fun* and *Fid* values are still around 80%. Moreover, we can further improve the HYPERTHEFT's results under this extremely challenging scenario via majority voting; see the last two rows in Table 5.10 and Table 5.11.

Table 5.10: Benchmarking impacts of APIC timer (*Fun*).

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

Table 5.11: Benchmarking impacts of APIC timer (*Fid*).

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

5.12.2 Proof for Sec. 5.5

For the case mentioned in Sec. 5.5, where a DNN's last layer is implemented as $y = Sigmoid(\theta x + b)$, suppose the input is $x = [x_0, x_1]^\intercal$ and the output is $y = [y_0, y_1, y_2]^\intercal$. Given weight $W = [\theta, b]$:

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

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

From Weight to Lines. In practice, a binary classification predicts "yes" if $Sigmoid(h_0) > 0.5$, i.e., $h_0 = \theta_{0,0}x_0 + \theta_{0,1}x_1 + b_0 > 0$. Thus, in the two-dimensional space constituted by x_0 and x_1, the decision boundary is depicted by the line $\theta_{0,0}x_0 + \theta_{0,1}x_1 + b_0 = 0$. The same applies to y_1 and y_2, and we can draw the three lines as in Fig. 5.3.

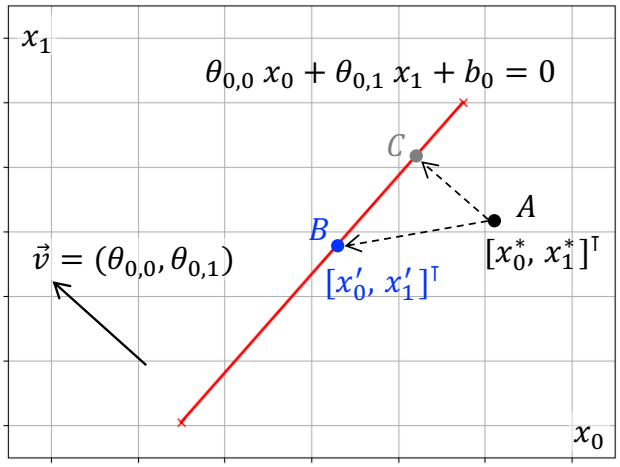


Figure 5.5: Distance between dot and line.

Distance Between Dot and Line. As illustrated in Fig. 5.5, for an input (i.e., a dot) $A : [x_0^*, x_1^*]^\intercal$, its distance to the line $l : \theta_{0,0}x_0 + \theta_{0,1}x_1 + b_0 = 0$ equals the length of segment $\overline{AC}$, whose direction is orthogonal to the line l.

To compute the length of $\overline{AC}$, we first randomly select one dot from l, as marked by $B : [x'_0, x'_1]^\intercal$ in Fig. 5.5. Therefore, the length of $\overline{AC}$ denotes the vector $\vec{AB}$'s projection

onto the line l's orthogonal direction, namely $\vec{v} = (\theta_{0,0}, \theta_{0,1})$. Thus, we have

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

Since B lies in the line l, we have

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

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

$$\left|\overline{AC}\right| = \frac{|\theta_{0,0}x^*_0 + \theta_{0,1}x^*_1 + b_0|}{\sqrt{\theta^2_{0,0}, \theta^2_{0,1}}}$$

Intermediate Output and Distance. When the dot $A : [x^*_0, x^*_1]^\intercal$ is taken by the layer $y = Sigmoid(\theta x + b)$, the intermediate output $h = [h_0, h_1, h_2]^\intercal = \theta[x^*_0, x^*_1]^\intercal + b$ is computed as

$$h = \begin{bmatrix} \theta_{0,0}x^*_0 + \theta_{0,1}x^*_1 + b_0 \\ \theta_{1,0}x^*_0 + \theta_{1,1}x^*_1 + b_1 \\ \theta_{2,0}x^*_0 + \theta_{2,1}x^*_1 + b_2 \end{bmatrix},$$

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

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

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

5.12.3 Vulnerable Functions in PyTorch

To localize vulnerable modules in DNNs, we use Pin to track functions whose memory writes trigger weight-dependent ciphertext collisions. In particular, Glow-executables implement each DNN layer as a function and *all* layers are vulnerable. We notice two types

of collisions: the first one is caused by identical intermediate DNN outputs (whose possibilities are enlarged by the non-linearity). The second type is due to collisions between DNN's (intermediate) output zeros and the zero-initialized memory (e.g., `ReLU` can output many zeros). Different from Glow, PyTorch first implements basic computing operations (e.g., multiplication, sum), and uses these operations to further implement DNN layers. We find that the leakage widely spans these computing operations (e.g., sum, pooling); an example list is given in Table 5.12. Interestingly, while we achieve comparable attacks over different versions of PyTorch, their vulnerable modules are slightly different. For instance, when running ResNet in PyTorch 2.1.0 (the latest version), the average pooling operation has leaks. However, such leakage is not identified in older versions when running the same ResNet.

Table 5.12: Vulnerable Functions in PyTorch.

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

CHAPTER 6

QUANTIFYING AND LOCALIZING SIDE-CHANNEL VULNERABILITIES IN PRODUCTION SOFTWARE

Chap. 3, Chap. 4, and Chap. 5 primarily focus on the offensive perspective by stealing various secrets from different side channels. This chapter presents our work on the defensive aspect that localizes leakage sources of side channels. We first review how to achieve full-fledged localization and propose eight criteria. Then we show CACHEQL *that meets all eight criteria.* CACHEQL *precisely quantifies information leaks by characterizing the distinguishability of logged side channel traces. Moreover,* CACHEQL *models leakage as a cooperative game, allowing information leakage to be precisely distributed to program points vulnerable to cache side channels.* CACHEQL *is meticulously optimized to analyze whole side channel traces logged from production software (where each trace can have millions of records), and it alleviates randomness introduced by defense modules or real-world noises.*

Since existing localization works mostly target cache side channels of cryptographic software. To present an aligned comparison, we adopt CACHEQL *to localize cache side channel leakages in OpenSSL, MbedTLS, Libgcrypt that were extensively studied previously, and identify a few hundred new leakages.* CACHEQL *is also employed to analyze data processing libraries like* `libjpeg`*, which is not supported previously. Overall, extending* CACHEQL *for new side channels and new leakages is straightforward, as it does not rely on specific side channels or leakage patterns.*

6.1 Introduction

Cache side channels enable confidential data leakage through shared data and instruction caches. Attackers can recover program secrets like secret keys and user inputs by monitoring how victim software accesses cache units. Exploiting cache side channels has been shown particularly effective for cryptographic systems such as AES, RSA, and ElGa-

mal [87, 236]. Recent attacks show that private user data including images and text can be reconstructed [265, 95, 278].

Both attackers and software developers are in demand to quantify and localize software information leakage. It is also vital to precisely distribute information leaks toward each vulnerable program point, given that exploiting program points that leak more information can enhance an attacker's success rate. Developers should also prioritize fixing the most vulnerable program points. Additionally, cyber defenders are interested in assessing subtle information leaks over cryptosystems already hardened by mitigation techniques (e.g., blinding). Nevertheless, most existing cache side channel detectors focus exclusively on qualitative analysis, determining whether programs are vulnerable without *quantifying* information that these flaws may leak [244, 243, 253, 37, 278]. Given the complexity of real-world cryptosystems and media libraries, scalable, automated, and precise vulnerability localization is lacking. As a result, developers may be likely reluctant (or unaware) to remedy vulnerabilities discovered by existing detectors. As shown in our evaluation (Sec. 6.8), attack vectors in production software are underestimated.

This work initializes a comprehensive view on detecting cache side-channel vulnerabilities. We propose *eight criteria* to design a full-fledged detector. These criteria are carefully chosen by considering various important aspects like scalability. Then, we propose CACHEQL, an automated detector for production software that meets all eight criteria. CACHEQL quantifies information leakage via mutual information (MI) between secrets and side channels. CACHEQL recasts MI computation as evaluating conditional probability (CP), characterizing *distinguishability* of side channel traces induced by different secrets. This re-formulation largely enhances computing efficiency and ensures that CACHEQL's quantification is more precise than existing works. It also principally alleviates *coverage issue* of conventional dynamic methods.

We also present a novel vulnerability localization method, by formulating information leak via a side channel trace as *a cooperative game* among all records on the trace. Then, Shapley value [220], a well-established solution in cooperative game theory, helps to localize program points leaking secrets. We rely on domain observations (e.g., side channel traces are often sparse) to reduce the computing cost of Shapley value from $\mathcal{O}(2^N)$ to

roughly constant with nearly no loss in precision.[1] CACHEQL directly analyzes binary code, and captures both explicit and implicit information flows. CACHEQL analyzes *entire* execution traces (existing works require traces to be cut to reduce complexity) and overcomes "non-determinism" introduced by noises or hardening techniques (e.g., cryptographic blinding, ORAM [87]).

We evaluate CACHEQL using production cryptosystems including the latest versions (by the time of writing) of OpenSSL, Libgcrypt and MbedTLS. We also evaluate Libjpeg by treating user inputs (images) as privacy. To mimic debugging [244], we collect memory access traces of target software using Intel Pin as inputs of CACHEQL.[2] We also mimic automated real attacks in userspace-only scenarios, where highly noisy side channel logs are obtained via `Prime+Probe` [236] and fed to CACHEQL. CACHEQL analyzed 10,000 traces in 6 minutes and found hundreds of bits of secret leaks per software. These results confirm CACHEQL's ability to pinpoint all known vulnerabilities reported by existing works [243, 253] and quantify those leakages. CACHEQL also discovers hundreds of unknown vulnerable program points in these cryptosystems, spread across hundreds of functions never reported by prior works. Developers promptly confirmed representative findings of CACHEQL. Particularly, despite the adoption of constant-time paradigms to harden sensitive components, cryptographic software is not fully constant-time, whose non-trivial secret leaks are found and quantified by CACHEQL. CACHEQL reveals the *pre-process* modules, such as key encoding/decoding and BIGNUM initialization, can leak many secrets and affect *all* modern cryptosystems evaluated. In summary, we have the following contributions:

- We propose eight criteria for systematic cache side-channel detectors, considering various objectives and restrictions. We design CACHEQL, satisfying all of them;
- CACHEQL reformulates mutual information (MI) with conditional probability (CP), which reduces the computing error and cost efficiently. It then estimates CP using neural network (NN). Our NN can properly handle lengthy side channel traces and analyze secrets of various types. Moreover, it does *not* require manual annotations of leakage in training data;

[1] N, the length of a side channel trace, reaches 5M in OpenSSL 3.0 RSA.

[2] Using Intel Pin to log memory access traces is a common setup in this line of works. CACHEQL, however, is not specific to Intel Pin [170].

- CACHEQL further uses Shapley value to localize program points leaking secrets by simulating leakage as a cooperative game. With domain-specific optimizations, Shapley value, which is computational infeasible, is calculated with a nearly constant cost;
- CACHEQL identifies subtle leaks (even with RSA blinding enabled), and its correctness has theoretical guarantee and empirical supports. CACHEQL also localizes all vulnerable program points reported by prior works and hundreds of unknown flaws in the latest cryptosystems. Our representative findings are confirmed by developers. It illustrates the general concern that BIGNUM and pre-processing modules are largely leaking secrets and undermining recent cryptographic libraries.

Research Artifact. To support follow-up research, we release the code, data, and all our findings at `https://github.com/Yuanyuan-Yuan/CacheQL` [6].

6.2 Background & Motivating Example

Application Scope. CACHEQL is designed as a *bug detector*. It shares the same design goal with previous detectors [244, 73, 243, 253, 255], whose main audiences are developers who aim to test and "debug" software. CACHEQL is incapable of synthesizing proof-of-concept (PoC) exploits and is hence incapable of launching real attacks. In general, exploiting cache side channels in the real world is often a multi-step procedure [162] that involves pre-knowledge of the target systems and manual efforts. It is challenging, if not impossible, to fully automate the process. For instance, exploitability may depend on the specific hardware details [163, 162, 271], and in cloud computing, the success of co-residency attacks denotes a key pre-condition of launching exploitations [283]. These aspects are not considered by CACHEQL which performs software analysis.

Threat Model. Aligned with prior works in this field [26, 73, 244, 243, 37], we assume attackers share the same hardware platforms with victim software. Attackers can observe cache being accessed when victim software is running. Attackers can log all cache lines (or other units) visited by the victim software as a side channel trace [162, 271].

Given a program g, we define the attacker's observation, a side channel trace, as $o \in O$ when g is executing $k \in K$. O and K are sets of all observations and secrets. K can be cryp-

tographic keys or user private inputs like photos. We consider a "debug" scenario where developers measure leakage when g executes k. Aligned with prior works [253, 255], we assume that developers can obtain noise-free o, e.g., o is execution trace logged by Pin. We also assume developers are interested in assessing leaks under real attacks. Indeed, OpenSSL by default only accepts side channel reports exploitable in real scenarios [5]. We thus also launch standard `Prime+Probe` attack to log cache set accesses. We aim to quantify information in k leaked via o. We also analyze leakage distribution across program points to localize flaws. Developers can prioritize patching vulnerabilities leaking more information.

Two Vulnerablities: Secret-Dependent Control Branch and Data Access. Our threat model focuses on two popular vulnerability patterns that are analyzed and exploited previously, namely, secret-dependent control branch (SCB) and secret-dependent data access (SDA) [164, 73, 74, 244, 243, 37, 26]. SDA implies that memory access is influenced by secrets, and therefore, monitoring which data cache unit is visited may likely reveal secrets [271]. SCB implies that program branches are decided by secrets, and monitoring which branch is taken via cache may likely reveal secrets [164]. CACHEQL captures both SCB and SDA, and it models secret information flow. That is, if a variable v is influenced ("tainted") by secrets via either explicit or implicit information flow, then control flow or data access that depends on v are also treated as SCB and SDA. The definition of SDA/SCB is standard and shared among previous detectors [244, 73, 243, 253, 255].

Detecting SDA Using CACHEQL.[3] Consider two vulnerable programs depicted in Fig. 6.1. In short, two program points in Fig. 1(a) have 128 (`L6`) and 512 (`L11`) memory accesses that are secret-dependent (i.e., SDA). Developer can use Pin to log one memory access trace o when executing Fig. 1(a), and by analyzing o, CACHEQL reports a total leakage of 768 bits. CACHEQL further apportions the SDA leaked bits as: 1) 2 bits for each of 128 memory accesses at `L6`, and 2) 1 bit for each of 512 memory accesses at `L11`. For Fig. 1(b), two array lookups at `L10` and `L12` depend on the secret. Given a memory access trace o, CACHEQL quantifies the leakage as 510 bits and apportions 255 bits for each SDA. We discuss technical details of CACHEQL in Sec. 6.4, Sec. 6.5, and Sec. 6.6.

[3] SCB can be detected in the same way, and is thus omitted here.

```
1  // s is a 1024-bit key
2  BIGNUM s;
3
4  for(i=0; i<512; i+=4) {
5    // secret dependent
6    x[s[i:i+4] % 4] = 0;
7  }
8  for(; i<1024; i++) {
9    // secret dependent
10   if(s[i])
11     y[1] = 0;
12 }
```

(a) SDA via implicit info. flow (L11).

```
1  // s is a 1024-bit key
2  BIGNUM s;
3
4  for(i=0;i<1024;i++) {
5    // random integer
6    x[urand() % 2048]=0;
7  }
8
9  // secret dependent
10 y[s[0:256] / 2] = 0;
11 // secret dependent
12 z[s[256:512] / 2] = 0;
```

(b) Prog. with random data access (L6).

Figure 6.1: Two pseudocode code of secret leakage. The secrets are 1024-bit keys. `s[i:j]` are bits between the i-th (included) and j-th bit (excluded).

Comparison with Existing Quantitative Analysis.[4] MicroWalk [255] measures information leakage via mutual information (MI). However, we find that its output is indeed mundane Shannon entropy rather than MI over different program execution traces, since both key and randomness like blinding can differ traces. MicroWalk has two computing strategies: whole-trace and per-instruction. For Fig. 1(b), MicroWalk reports 1024 leaked bits using the whole-trace strategy. The per-instruction strategy localizes three leakage program points, where each point leaks 1024 (L6), 255 (L10), and 255 (L12) bits, respectively. However, it is clear that those 1024 memory accesses at L6 are decided by *non-secret* randomness. Thus, both quantification and localization are inaccurate. Abacus [26] uses trace-based symbolic execution to measure leakage at each SDA, by estimating number of different secrets (`s`) that induces the access of different cache units. No implicit information flow is modelled, thereby omitting to "taint" the memory access at L11 of Fig. 1(a). Abacus quantifies leakage of Fig. 1(a) over o as 256 bits, since it only finds SDA at L6.

Program points may have dependencies. For instance, one branch may have its information leaked in its parent branch, and therefore, separately adding them together largely over-estimates the leakage: Abacus outputs a total leakage of 413.6 bits in AES-128, despite its 128-bit key length. CACHEQL precisely calculates the leakage as 128.0 bits (Sec. 6.8.2). Also, some static analyses [73, 74, 49] have limited scalability due to heavy-

[4] We discuss their analysis about SDA; SCB is conceptually the same.

Table 6.1: Benchmarking criteria for side channel detectors. ✓, ◇, ✗ denote support, partially support, and not support.

	❶	❷	❸	❹	❺	❻	❼	❽
CacheAudit [73, 74]	✗	✗	✗	◇	✗	✗	✗	✓
CacheD [244]	✗	✗	✓	✗	✓	✗	✗	✗
CaSym [37]	✗	✗	✗	✗	✓	✗	✗	✗
CacheS [243]	✗	✗	✓	✗	✓	✗	✗	✗
Abacus [26]	✗	✗	✓	◇	✓	✗	✗	✗
CHALICE [49]	✗	✗	✗	◇	✗	✗	✗	✓
DATA [253, 252]	✗	◇	✓	✗	✓	✗	✓	◇
MicroWalk [255]	✗	✗	✓	◇	◇	✗	✓	◇
CANAL [232]	✗	✗	✗	✗	✓	✗	✗	✓
Manifold [278]	✓	✗	✓	✗	✓	◇	✓	✓
CACHEQL	✓	✓	✓	✓	✓	✓	✓	✓

weight abstract interpretation or symbolic execution. Real-world cryptosystems and media software are complex, with millions of records per side channel trace. In addition, they are often unable to localize vulnerable points.

6.3 Related Works & Criteria

We propose eight criteria for a full-fledged detector. Accordingly, we review related works in this field and assess their suitability. Sec. 6.8.3 empirically compares them with CACHEQL. Also, many studies launch cache analysis on real-time systems and estimate worst-case execution time (WCET) [56, 152, 154, 178]; we omit those studies as they are mainly for measurement, not for vulnerability detection.

Execution Trace vs. Cache Attack Logs. Most existing detectors [244, 243, 37] assume access to execution traces. In addition to recording noise-free execution traces (e.g., via Intel Pin), considering real cache attack logs is equally important. Cryptosystem developers often require evidence under real-world scenarios to issue patches. For instance, OpenSSL by default only accepts side channel reports if they can be successfully exploited in real-world scenarios [5]. In sum, we advocate that a side channel detector should ❶ *analyze both execution traces and real-world cache attack logs*.

Deterministic vs. Non-deterministic Observations. Deterministic observations imply that,

for a given secret, the observed side channel is fixed. Decryption, however, may be non-deterministic due to various masking and blinding schemes used in cryptosystems. Furthermore, techniques like ORAM [87] can generate non-deterministic memory accesses and prevent information leakage. Thus, memory accesses or executed branches may differ between executions using one secret. Nearly all previous works [244, 26, 243, 37] only consider deterministic side channels, failing to analyze the protection offered by blinding/ORAM and may overestimate leaks (not just keys, blinding/ORAM also change side channel observations). We suggest that a detector should ❷ *analyze both deterministic and non-deterministic observations.* CACHEQL uses statistics to quantify information leaks from non-deterministic observations, as explained in Sec. 6.4.4.

Analyze Source Code vs. Binary. A detector should typically analyze software in executable format. This allows the analysis of legacy code and third-party libraries. More importantly, by analyzing assembly code, low-level details like memory allocation can be precisely considered. Studies [74, 228] reveal that compiler optimizations could introduce side channels not visible in high-level code representations. Thus, we argue that detectors should ❸ *be able to analyze program executables.*

Quantitative vs. Qualitative. Qualitative detectors decide whether software leaks information and pinpoint leakage program points [244, 243, 37]. Quantitative detectors further quantify leakage from each software execution [73, 49, 74], or at each vulnerable program point [26, 255]. We argue that a detector should ❹ *deliver both qualitative and quantitative analysis*. Developers are reluctant to fix certain vulnerabilities, as they may believe those defects leak negligible secrets [26]. However, identifying program points that leak large amounts of data can push developers to prioritize fixing them. To clarify, though quantitative analysis was previously deemed costly [118], CACHEQL features efficient quantification.

Localization. Along with determining information leaks, localizing vulnerable program points is critical. Precise localization helps developers debug and patch. Therefore, a detector should ❺ *localize vulnerable program points leaking secrets.* Most static detectors struggle to pinpoint leakage points [73, 74], as they measure the number of different cache statuses to quantify leakage. Trace-based analysis, including CACHEQL, can identify leakage instructions on the trace that can be mapped back to vulnerable program points [244, 26].

Key vs. Private Media Data. Most detectors analyze cryptosystems to detect key leakage [244, 73, 26]. Recent side channel attacks have targeted media data [265, 95]. Media data like images used in medical diagnosis may jeopardize user privacy once leaked. We thus advocate detectors to ❻ *analyze leakage of secret keys and media data*. Modeling information leakage of high-dimensional media data is often harder, because defining "information" contained in media data like images may be ambiguous. CACHEQL models image holistic content (rather than pixel values) leakage using neural networks.

Scalability: Whole Program/Trace vs. Program/Trace Cuts. Some prior trace-based analyses rely on expensive techniques (e.g., symbolic execution) that are not scalable. Given that one execution trace logged from cryptosystems can contain millions of instructions, existing works [244, 26] require to first *cut* a trace and analyze only a few functions on the cutted trace. Prior static analysis-based works may use abstract interpretation [73, 243], a costly technique with limited scalability. Only toy programs [73] or a few sensitive functions are analyzed [243, 74, 37]. This explains why most existing works overlook "non-deterministic" factors like blinding (criterion ❷), as blinding is applied *prior to* executing their analyzed program/trace cuts. Lacking whole-program/trace analysis limits the study scope of prior works. CACHEQL can analyze a whole trace logged by executing production software, and as shown in Sec. 6.8, CACHEQL identifies unknown vulnerabilities in pre-processing modules of cryptographic libraries that are not even covered by existing works due to scalability. In sum, we advocate that a detector should be ❼ *scalable for whole-program/whole-trace analysis.*

Implicit and Explicit Information Flow. Explicit information flow primarily denotes secret data flow propagation, whereas implicit information flow models subtle propagation by using secrets as code pointers or branch conditions [215]. Considering implicit information flow is challenging for existing works based on static analysis due to scalability. They thus do not *fully* analyze implicit information flow [244, 243, 37, 26]. We argue a detector should ❽ *consider both implicit and explicit information flow* to comprehensively model potential information leaks. CACHEQL delivers an "end-to-end" analysis and identifies changes in the trace due to either implicit or explicit information flow propagation of secrets.

Comparing with Existing Detectors. Table 6.1 compares existing detectors and CACHEQL

to the criteria. Abacus and MicroWalk cannot precisely quantify information leaks in many cases, due to either lacking implicit information flow modeling or neglecting dependency among leakage sites (hence repetitively counting leakage). CacheAudit only infers the upper bound of leakage. Thus, they partially satisfy ❹. MicroWalk quantifies per-instruction MI to localize vulnerable instructions, whose quantified leakage per instruction, when added up, should not equal quantification over the whole-trace MI (its another strategy) due to program dependencies. Also, MicroWalk cannot differ randomness (e.g., blinding) with secrets. It thus partially satisfies ❺.

DATA [253, 252] launches trace differentiation and statistical hypothesis testing to decide secret-dependency of an execution trace. Similar as CACHEQL, DATA can also analyze non-deterministic traces. Nevertheless, we find that DATA, by differentiating traces to detect leakage, may manifest low precision, given it would neglect secret leakage if a cryptographic module also uses blinding. It thus partially satisfies ❷. More importantly, DATA does not deliver quantitative analysis.

Recent research attempts to reconstruct media data like private user photos from side channels [278, 137, 261]. In Table 6.1, we compare CACHEQL with Manifold [278], the latest work in this field. Manifold leverages manifold learning to infer media data. Manifold learning is *not* applicable to infer secret keys (as admitted in [278]): unlike media data which contain perception constraints retained in manifold [101], each key bit is sampled independently and uniformly from 0 or 1. It thus partially fulfills ❻. CACHEQL is the first to quantify information leaks over cryptographic keys and media data.

Implicit information flow (❽) is not tracked by most existing static analyzers. Analyzing implicit information flow requires considering more program statements and data/-control flow propagations, which often largely increases the code chunk to be analyzed. This introduces extra hurdles for static analysis-based approaches. DATA and MicroWalk also do not systematically capture implicit information flow. DATA/MicroWalk first *align* traces and then compare aligned segments, meaning that they can overlook holistic differences (unalignment) on traces. CACHEQL satisfies ❽ as it directly observes and analyzes changes in the side channel traces. Given any information flow, either explicit or implicit, can differ traces, CACHEQL captures them in a unified manner. Nevertheless, it is evident that the implicit information flow cannot be captured by CACHEQL unless it is covered in

the dynamic traces.

Clarification. These criteria's importance may vary depending on the situations. Having only some of these criteria implemented is not necessarily "bad," which may suggest that the tool is targeted for specific use cases. Analyzing private image leakage (❻) may not be as important as others, especially for cryptosystem developers. We consider image leakage because recent works [265, 95, 278] consider recovering private media data. We present eight criteria for building full-fledged side-channel detectors. The future development of detectors can refer to this chapter and prioritize/select certain criteria, according to their domain-specific need. Also, we clarify that in parallel to research works that detect side channel leaks, another line of approaches (i.e., static verification) aims at deriving precise, certified guarantees [73, 66].

As clarified above, CACHEQL can analyze real attack logs (❶) and media data (❻). However, for the sake of presentation coherence, we discuss them in Sec. 6.9. In the rest of this chapter, we explain the design and findings of CACHEQL using Pin-logged traces from cryptosystems.

6.4 Quantifying Information Leakage

Overview. This section discusses quantitative measurement of information leaks over side channel observations. We start with preliminaries in Sec. 6.4.1. The overview of our approach is illustrated in Fig. 6.2. Sec. 6.4.2 introduces MI computation via Point-wise Dependence (PD). Then, Sec. 6.4.3 recasts calculating PD into computing conditional probability (CP). CACHEQL employs parameterized neural networks $\mathcal{F}_\theta$ (see Sec. 6.5) to estimate CP, which is carefully designed to quantify leakage of keys and private images from extremely lengthy side channel traces. The error of estimating CP with $\mathcal{F}_\theta$ is bounded by a negligible ϵ. In contrast, prior works use marginal probability (MP) to estimate MI. CP outperforms MP in terms of lower cost, fewer errors, and better coverage, as compared in Sec. 6.4.3. In Sec. 6.4.4, we extend the pipeline in Fig. 6.2 to handle non-deterministic side channel traces.

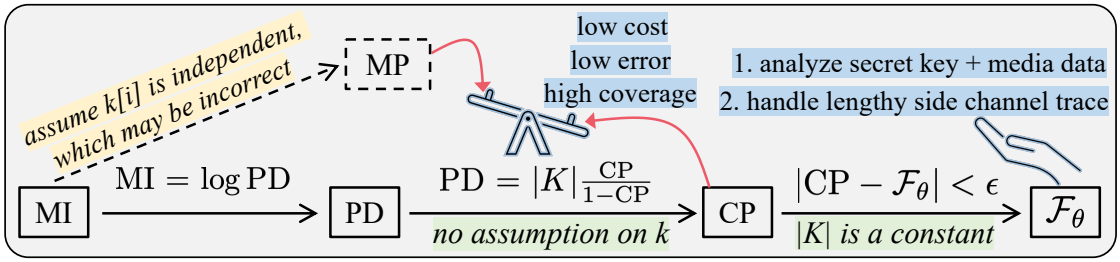


Figure 6.2: Overview. $|K|$ is the total number of possible keys. $|K|$ is assumed as known to detectors by all existing quantification tools including CACHEQL.

6.4.1 Problem Setting

In general, side channel analysis aims to infer k from o. The information leak of K in O can be defined as their MI:

$$I(K;O) = H(K) - H(K|O), \tag{6.1}$$

where $H(\cdot)$ denotes the entropy of an event. According to Shannon's information theory, $I(K;O)$ describes how much information about K can be obtained by observing O. Consider the program in Fig. 3(a), where the probability of correctly guessing each $k \in K$ (i.e., $\texttt{s} \in \{\texttt{0},\texttt{1},\texttt{2},\texttt{3}\}$), without any observation, is $\frac{1}{4}$. Thus, $H(K) = -\log\frac{1}{4} = 2$ bits.[5] Nevertheless, the observation $o = a[0]$ (L6) indicates that k must be "0" (i.e., the probability is 1), thus, $H(K|o{=}a[0]) = -\log 1 = 0$. Therefore, $a[0]$ leaks 2 bits of information. Similarly, the memory access $b[0]$ (L9) leaks $\log\frac{4}{3}$ bits of information since $H(K|o{=}b[0]) = -\log\frac{1}{3} = \log 3$. Ideally, a secure program should have $H(K) = H(K|O)$, indicating no information in K can be obtained from O. We continue discussing Fig. 3(b) in Sec. 6.4.3.

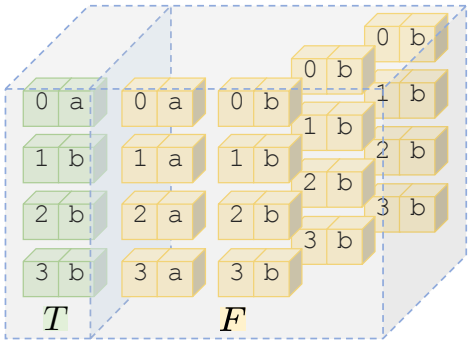
6.4.2 Computing MI via PD

Following Eq. 6.1, let k and o be random variables whose probability density functions (PDF) are $p(k)$ and $p(o)$. The MI $I(K;O)$ can be represented in the following way,

[5]Given log base 2 is used by default, the unit of information is *bit*.

```
1 //s: evenly {0,1,2,3}
2 int s;
3 array a[1], b[1];
4 // leak log4 = 2 bits
5 if(s == 0)
6   a[0] = 1;
7 // leak log(4/3) bits
8 else
9   b[0] = 1;
```

(a) A vulnerable program.



(b) An illustration of $P_{K\times O}$ and $P_K P_O$.

Each green block (i.e., two joint cubes in green) denotes a k and its induced o ($<k, o>$) and each yellow block is produced by randomly combining k and o. Each block (both green and yellow) has equal probability.

Figure 6.3: Quantification of side channel leaks.

$$
\begin{aligned}
I(K;O) &= \int\int_{K\times O} p(k,o)\log\frac{p(k,o)}{p(k)p(o)}\mathrm{d}k\mathrm{d}o \\
&= \mathbb{E}_{P_{K\times O}}\left[\log\frac{p(k,o)}{p(k)p(o)}\right] = \mathbb{E}_{P_{K\times O}}[\log c(k,o)],
\end{aligned}
\tag{6.2}
$$

where $p(k,o)$ is the joint PDF of K and O, and $P_{K\times O}$ is the joint distribution. $c(k,o) = \frac{p(k,o)}{p(k)p(o)}$ denotes *point-wise dependency* (PD), measuring discrepancy between the probability of k and o's co-occurrence and the product of their independent occurrences. Accordingly, $\log c(k,o)$ denotes the point-wise mutual information (PMI).

The MI of K and O, by definition, is the expectation of PMI. That is, Eq. 6.2 measures the dependence retained in the joint distribution (i.e., $\langle k,o\rangle \sim P_{K\times O}$) relative to the marginal distribution of K and O under the assumption of independence (i.e., $\langle k,o\rangle \sim P_K P_O$, where P_K and P_O are marginal distributions of K and O). When K and O are independent, we have $p(k,o) = p(k)\cdot p(o)$ and $\frac{p(k,o)}{p(k)p(o)}$ is 1, thus, the leakage is $\log 1$=0. Nevertheless, whenever o leaks k, k and o should co-occur more often than their independent occurrences, and therefore, $c(k,o) > 1$ and $\log c(k,o) > 0$.

Eq. 6.2 illustrates two aspects for quantitatively computing information leakage: 1) **PMI** $\log c(k{=}k^*, o{=}o^*)$, denoting per trace leakage for a specific k^* and its corresponding o^*, and 2) **MI** $I(K;O)$, denoting program-level leakage over all possible secrets $k \in K$. To

compute $I(K;O)$, we average PMI over a collection of $\langle k, o\rangle$, where sample-mean offers an unbiased estimation for the expectation $\mathbb{E}(\cdot)$ of a distribution [45].

Comparison with Prior Works. Abacus [26] launches symbolic execution on Pin-logged execution traces. It makes a strong assumption that k is uniformly distributed, i.e., $p(k{=}k^*) = \frac{1}{|K|}$.[6] It also assumes that each trace must be deterministic, such that $p(k{=}k^*, o{=}o^*) = p(k{=}k^*)$ for a given o^* and its corresponding k^*. This way, approximating MI in Eq. 6.2 is recasted to estimating the marginal probability (MP) $p(o{=}o^*)$. At a secret-dependent control transfer or data access point l, Abacus finds all $k \in K'$ that cover l. The leakage at l is computed as $-\log p(o{=}o^*) = -\log(\frac{|K'|}{|K|})$. Deciding $|K'|$ via constraint solving is costly, and therefore, Abacus uses *sampling* to approximate $|K'|$. Nevertheless, estimating MP with sampling is unstable and error-prone (see Sec. 6.4.3). MicroWalk also samples k to estimate MP; it thereby has similar issues. CacheAudit quantifies program-wide leakage. Using abstraction interpretation, it only analyzes small programs or code fragments, and it infers only leak upper bound. CACHEQL precisely computes PMI/MI via PD and localizes flaws. We now introduce estimating PD.

6.4.3 Estimating PD $c(k, o)$ via CP

Because PD makes *no* assumption on the secret's distribution, our approach can infer different types of secrets (e.g., key or images). We denote k as a general representation of one secret, and for simplicity, we write $p(k{=}k^*)$ as $p(k^*)$ in followings. The same applies for o and o^*. o^* is one side channel observation produced by k^*. However, o^* may not be the only one, given randomness like blinding can also induce different observations even with a fixed k^*. We now recast computing PD over deterministic side channels as estimating conditional probability (CP) via binary classification [238].

Transforming PD to CP

Let T depict that a pair $\langle k, o\rangle$ co-occurs (i.e., positive pair $\langle k, o\rangle \sim P_{K\times O}$). Let F denote that k and o in $\langle k, o\rangle$ occur independently (i.e., negative pair $\langle k, o\rangle \sim P_K P_O$). Therefore, $p(k^*, o^*)$

[6] "Uniform distribution" does *not* hold for image pixel values [278].

and $p(k^*)p(o^*)$ can be represented as the posterior PDF $p(k^*, o^*|T)$ and $p(k^*, o^*|F)$, respectively. According to Bayes' Theorem, PD $c(k^*, o^*)$ is re-expressed as

$$\text{PD} = \frac{p(k^*, o^*)}{p(k^*)p(o^*)} = \frac{p(k^*, o^*|T)}{p(k^*, o^*|F)} = \frac{p(F)}{p(T)} \frac{p(T|k^*, o^*)}{p(F|k^*, o^*)}, \tag{6.3}$$

where $p(T)$ and $p(F)$ are constants (decided by the analyzed software). Given $P_K P_O$ is produced by separating each pair in $P_{K\times O}$ and collecting random combinations of k and o, $\frac{p(F)}{p(T)}$ equals to $|K|$. In practice, $P_{K\times O}$ is prepared by running the analyzed software with each k^* and collecting the corresponding o^*. For the program in Fig. 3(a), Fig. 3(b) colors $P_{K\times O}$ and $P_K P_O$ in green and yellow. Since $\frac{p(F)}{p(T)}$ is unrelated to k^* or o^*, $c(k^*, o^*)$ — representing leaked k^* from o^* — is only decided by CP $p(T|k^*, o^*)$. A larger CP indicates that more information is leaked.

• *Example:* We demonstrate this transformation using Fig. 6.3: for k="0" and o=a[0], fetching a block of $\langle 0, a\rangle$ from Fig. 3(b) has a 50% chance of selecting the green one (in the upper-left corner). That is, CP=$p(T|\text{“0”}, a[0])$=0.5, and therefore, we can have the equality $p(T|\text{“0”}, a[0])$= $p(F|\text{“0”}, a[0])$. Since $\frac{p(F)}{p(T)}$=4, Eq. 6.3 yields $4 \times \frac{0.5}{0.5} = 4$, and therefore, $\log c(k, o)$ in Eq. 6.2 yields $\log 4 = 2$ bits, equaling the leakage result computed in Sec. 6.4.1.

Advantages of CP vs. Marginal Probability (MP)

CP captures what factors make o^*, which corresponds to k^*, distinguishable from other o. By observing both dependent and independent $\langle k, o\rangle$ pairs, CACHEQL measures the leakage via describing how the distinguishability between different o is introduced by the corresponding k. It is principally distinct with existing quantitative analysis [26, 73, 255]. Abacus and MicroWalk approximate MP $p(o^*)$ via sampling, which has the following three limitations compared with CP.

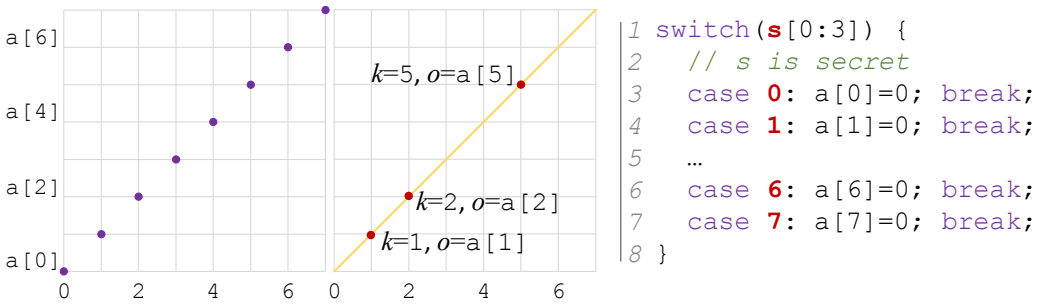
Computing Cost. Estimating CP is an one-time effort over a collection of $\langle k, o\rangle$ pairs. Estimating MP, however, has to *re-perform* sampling for each $\langle k, o\rangle$. Note that the cost for CP to estimate over the collection of $\langle k, o\rangle$ and each re-sampling of MP is comparable. Thus, MP is much more costly.

Estimation Error. Recall that for a leakage program point l, Abacus finds all $k \in K'$ that cover l via constraint solving and denotes the leakage as $-\log(\frac{|K'|}{|K|})$. Suppose it observes the first `for` loop in Fig. 1(a) has 128 consecutive accesses to $x[0]$. To quantify the leakage, Abacus constructs the symbolic constraint $(s[0{:}4]\%4 == 0) \wedge \ldots \wedge (s[508{:}512]\%4 == 0)$. Nevertheless, sampling one key that satisfies this constraint has only an extremely low probability of $(\frac{1}{4})^{128}$. That is, the MP can be presumably underestimated when $|K'|$ is small. Thus, the leaked information can be largely overestimated via $-\log(\frac{|K'|}{|K|})$. MicroWalk observes o^*'s frequency by sampling different k; it thus has similar issues. Worse, once o^* is influenced by randomness like blinding, no o^* would be identical. Thus, it will incorrectly regard $p(o^*)$ as $\frac{1}{|K|}$ and report $\log|K|$ leaked bits (i.e., equals to the key length). In contrast, Eq. 6.3 is free from this issue: even o^* is only produced by processing one or a few k, CACHEQL directly characterizes PD via CP $p(\cdot|k^*, o^*)$.

Overall, CP reflects: 1) the portion of records in o^* affected by its k^* [238], and 2) to what extent k^* affects each record in o^* (see *Example* below). Further, since any difference on o^*, whether due to explicit or implicit information flows, contributes to *distinguishing* o^* from the rest $o \in O$, CACHEQL takes both explicit and implicit flows into consideration.

• *Example:* Consider the memory accesses at `L10` and `L12` of the program in Fig. 1(b) and suppose o^* is "$y[0], z[0]$". To estimate the leakage over o^* via MP, it requires sampling keys where $s[0{:}256]$ constitutes either 1 or 0, so do the second 256 bits, which results in a total of 4 cases for $s[0{:}512]$. $s[0{:}512]$ has 2^{512} cases, denoting a large search space. Nevertheless, CP can infer the leakage by only observing that, the first record in o increases "1" (i.e., distinguishable from other o) whenever $s[0{:}256]$ increases 2 (with no need to simultaneously consider $s[256{:}512]$). The same applies to the second 256 bits.

Coverage Issue. CACHEQL, by using CP, principally alleviates the coverage issue of conventional dynamic methods. Consider the program in Fig. 4(c), CACHEQL can quantify the 8 SDA *without* covering all paths, since CP captures how o changes with k. As shown in Fig. 4(b), covering a few cases is sufficient to know that o increases "one" (e.g., $a[0] \rightarrow a[1]$) when k increases one (e.g., $0 \rightarrow 1$), thus inferring program behavior in Fig. 4(a). Prior dynamic methods need to fully cover all paths to infer the program behavior and quantify the leaks, which is hardly achievable in practice.



(a) program behaviors. (b) inferred behaviors by CP. (c) vulnerable program.

Figure 6.4: A schematic view of how the coverage issue of dynamic methods is alleviated via CP when *quantifying* leaks.

Obtaining CP $p(T|k,o)$ via Binary Classification

We show that performing probabilistic classification can yield CP. In particular, we employ neural networks $\mathcal{F}_\theta$ (parameterized by θ) to classify a pair of $\langle k, o\rangle$, whose associated confidence score is $p(T|k,o)$. Details of $\mathcal{F}_\theta$ are in Sec. 6.5.

Using neural networks (NN) to estimate MI is *not* our novelty [238, 30]. However, we deem NN as particularly suitable for our research context for three reasons: 1) non-parametric approaches, as in [85], suffer from "curse of dimensionality" [32, 33]. They are thus infeasible, as even the AES-128 key is 128-dimensional. NN shows encouraging capability of handling high-dimension data (e.g., images with thousands of dimensions). 2) Recent works show that NN can effectively process lengthy but sparse data, including side channel traces where only a few records out of millions are informative and leaking program secrets [278, 137, 261]. 3) It's generally vague to define "information" in media data. For instance, a 64×64 image may retain the same information as a 32×32 version from human perspective since the content is unchanged. Recent research [278] shows that high-dimensional media data have perceptual constraints which implicitly encode image "information." NNs are currently widely used to process media data and extract critical information for comprehension.

6.4.4 Handling Non-determinism

In practice, due to hardening schemes like RSA blinding and ORAM, side channel observations can be non-deterministic, where memory access traces may vary during different runs despite the same key is used. As discussed in Table 6.1 (i.e., ❷), however, non-determinism is not properly handled in previous (quantitative) analysis.

Generalizability. For deterministic side channels, only k induces changes of o. Fitting $\mathcal{F}_\theta$ on enough $\langle k, o\rangle$ pairs from $P_{K\times O}$ and $P_K P_O$ can capture distinguishability for quantification. In contrast, for non-deterministic side channels, the differences between o may be due to random factors, not only k. Therefore, in addition to *distinguishability* between $\langle k, o\rangle$ pairs, we also need to consider *generalizability* to alleviate over-estimation caused by random differences.

In statistics, *cross-validation* is used to test generalizability. Here, we propose a simple yet effective method by using a *de-bias* term with cross-validation to prune non-determinism in the estimated PD. We first mix $\langle k, o\rangle$ from $P_{K\times O}$ and $P_K P_O$ and split them into non-overlapping groups. Then, we assess if the distinguishability over one group applies to the others.

PD Estimation via De-biasing. We first extend the PD computation in Eq. 6.3 to handle non-determinism. In Eq. 6.3, F and T are finite sets, and $p(F)/p(T)$ equals to $|K|$. Here, we conservatively assume that there exist infinite non-deterministic side channels. That is, F is a set with infinite elements. We first require m positive pairs $\langle k, o\rangle \sim P_{K\times O}$, dubbed as $T^{(m)}$. We also construct $m' \leq m^2$ negative pairs (i.e., $\langle k, o\rangle \sim P_K P_O$), denoted as $F^{(m')}$, by replacing o (or k) of a pair from $P_{K\times O}$ with that of other random pairs. This way, PD defined in Eq. 6.3 is extended in the following form:

$$\text{PD} = \frac{\log|K|}{\log(m'/m)} \log \frac{p(F^{(m')})}{p(T^{(m)})} \frac{p(T|k^*, o^*)}{p(F|k^*, o^*)}, \tag{6.4}$$

where the $p(F^{(m')})/p(T^{(m)})$ works as a *de-bias* term to assess the generalizability for non-deterministic side channels. We denote $p(T|k^*, o^*)/p(F|k^*, o^*)$ as the *leakage ratio*: a 100% ratio implies that all bits of the key are leaked whereas 0% ratio implies no leakage. Consider the following two cases:

• *Case*$_1$: In case the differences between samples from $T^{(m)}$ and $F^{(m')}$ are all introduced by random noise (i.e., each o^* is independent of its k^*), the distinguishable factors should not be generalizable, and the above formula yields a zero leakage. To understand this, let our neural networks $\mathcal{F}_\theta$ identify each pair based on random differences, which is indeed equivalent to memorizing all pairs. This way, when it predicts the label of $\langle k, o\rangle$, the output simply follows the frequency of $T^{(m)}$ and $F^{(m')}$. Therefore, given an unseen pair $\langle k^*, o^*\rangle$, $\mathcal{F}_\theta$ has $p(T|k^*, o^*)/p(F|k^*, o^*) = p(T^{(m)})/p(F^{(m')})$, and the estimated leakage is thus $\log \frac{p(F^{(m')})}{p(T^{(m)})} \frac{p(T^{(m)})}{p(F^{(m')})} = \log 1 = 0$.

• *Case*$_2$: If o^* depends on k^*, $p(T|k^*, o^*)$ would not merely follow the distribution of $T^{(m)}$ and $F^{(m')}$, indicating a non-zero leakage. More importantly, de-biased by $p(F^{(m)})/p(T^{(m')})$, quantifying leakage using Eq. 6.4 *only* retains differences related to k. This way, we precisely quantify leakage for non-deterministic side channels.

Implementation Consideration. To alleviate randomness in each o^*, we collect four observations o_i^* by running software using k^* for four times. By classifying all $\langle k^*, o_i^*\rangle$ as positive pairs, $\mathcal{F}_\theta$ is guided to extract common characters shared by o_i^* while neglecting randomness in each o_i^*. Also, considering un-optimized neural networks generally make prediction by chance (i.e., $p(T|k, o) = p(F|k, o)$), we let $m = m'$.

6.5 Framework Design

Fig. 6.5 shows the pipeline of CACHEQL, including three components: 1) a sparse encoder $\mathcal{S}$ for converting side channel traces o^* into latent vectors, 2) a compressor $\mathcal{R}$ to shrink information in o^*, and 3) a classifier $\mathcal{C}$ that fits the CP in Eq. 6.3 via binary classification. We compute CP $p(T|k^*, o^*)$ using the following pipeline:

$$p(T|k^*, o^*) = \mathcal{F}_\theta(k^*, o^*) = \mathcal{C}(k^*, \mathcal{R}(\mathcal{S}(o^*))), \tag{6.5}$$

where parameters of these three components are jointly optimized, i.e., $\theta = \theta_\mathcal{S} \cup \theta_\mathcal{R} \cup \theta_\mathcal{C}$.

The framework takes a tuple $\langle k, o\rangle$ as input. As introduced in Sec. 6.4.3, we label a tuple $\langle k, o\rangle$ as positive if o is produced when the software is processing k. A tuple is otherwise

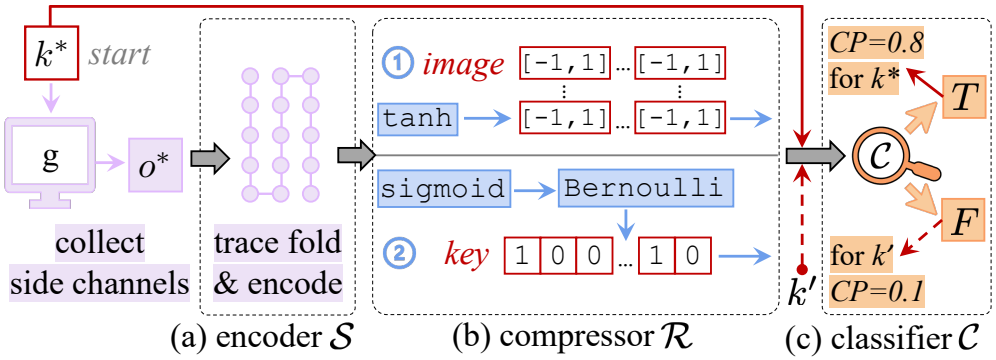


Figure 6.5: The framework of CACHEQL. $\langle k^*, o^* \rangle \in P_{K \times O}$; it is labeled as T. In contrast, $\langle k', o^* \rangle \in P_K P_O$ and is labeled as F.

negative. In Fig. 6.5, $\langle k^*, o^* \rangle$ is positive and $\langle k', o^* \rangle$ is negative.

$\mathcal{S}$: Encoding Lengthy and Sparse Side Channel Traces. According to our tentative experiments, naive neural networks perform poorly when analyzing real-world software due to the *highly lengthy* side channel traces. An o, obtained via Pin or cache attacks, typically contains millions of records, exceeding the capability of typical neural networks. ORAM can add dummy memory accesses, often resulting in a tenfold increase of trace length. Yuan et. al [278] found that side channel traces are generally *sparse*, with few secret-dependent records. It also has spatial locality: adjacent records on a trace often come from the same or related functions. Encoder $\mathcal{S}$ is inspired by [278]: to approximate the locality, we fold the trace into a matrix (see configurations in Sec. 6.8). We employ the design in [278] to construct $\mathcal{S}$ as a stack of convolutional NN layers. We find that our pipeline effectively extracts informative features from o.

$\mathcal{R}$: Shrinking Maximal Information. A side channel trace o^* frequently contains information unrelated to secret k^*. Our preliminary study shows that directly bridging the latent vectors (outputs of $\mathcal{S}$) to $\mathcal{C}$ is difficult to train. This stage thereby compresses $\mathcal{S}$'s output in an information-dense manner. We propose that *information*[7] *in* o^*, *namely* $H(o^*)$, *should never exceed* $H(k^*)$. Accordingly, we apply mathematical transformations $\mathcal{R}$ to confine the value range of $\mathcal{S}$'s outputs. We propose various transformations for media data and secret keys; as discussed below.

[7]Only secret-related information. Non-secret variables (e.g., public inputs) that affect o are regarded as randomness and handled as in Sec. 6.4.4.

Media Data. We use image to demonstrate the cases for media data; the conclusion can be extended to other media data straightforward [278]. It is generally obscure to measure the amount of information encoded in high-dimensional media data like images [281]. To extract information, an image, before being fed into modern neural networks (e.g., classifier $\mathcal{C}$ in CACHEQL), is generally normalized into $[-1, 1]$ and represented as a *channel* $\times$ *width* $\times$ *height* matrix [135]. Let a private image be k_m and the associated side channel log be o_m, we set the output of $\mathcal{S}(o_m)$ as a matrix of the same size. Accordingly, our compressor $\mathcal{R}_m$ is implemented using the $tanh : [-\infty, \infty] \rightarrow [-1, 1]$ function, facilitating $H(o_m) \leq H(k_m)$. *tanh* is parameter free (i.e., $\theta_{\mathcal{R}} = \emptyset$), eliminating extra training cost. Also, if certain properties in an image are particularly desirable by attackers, e.g., the gender of portrait images, k_m and the matrix from $\mathcal{S}(o_m)$ can be replaced with a vectorized representation for image properties. We refer readers to [202] for vectorizing image attributes.

Cryptographic Keys. For a cryptographic key k_c of length L, there are total 2^L uniformly distributed key instances. The information in one key instance is thus $H(k) = \log 2^L = L$ bits. Since a key only contains binary values and neural networks are hard to be deployed with binary parameters, for the associated side channel log o_c, we need to transform the floating-point encoding output $\mathcal{S}(o_c)$, which is accordingly a vector of length L, into binary bits. Existing profiling-based side channel attacks map side channels to keys via L bit-wise classification tasks [107, 124, 98]. In each task, a floating point is transformed to 1 if it is greater than a threshold. Nevertheless, this transformation is not applicable in $\mathcal{R}$, given that it is not differentiable, which impedes the optimization of θ. Inspired by the optimization for binary variables [61, 62], we design $\mathcal{R}_c$ by joining two components: 1) a non-parametric $sigmoid : [-\infty, \infty] \rightarrow [0, 1]$ function which generates L independent (as bits in k_c are independent) probabilities, and 2) a parametric Bernoulli distribution $Bern : \Pr(Bern(\cdot) = 1) + \Pr(Bern(\cdot) = 0) = 1$ which takes its inputs, i.e., $sigmoid(\mathcal{S}(o_c))$, as parameters for optimization. Thus, we have $\mathcal{R}_c = Bern \circ sigmoid$ and $\theta_{\mathcal{R}_c} = sigmoid(\mathcal{S}(o_c))$. See our codebase [6] for details.

$\mathcal{C}$: Optimizing Parameters via Classification. Let the parameter space be Θ and $\theta \in \Theta$. To train a neural network, we search for parameter $\theta^\dagger \in \Theta$ to maximize a pre-defined objective. As shown in Eq. 6.3, we recast leakage estimation as approximating CP $p(T|k, o)$,

which is further formed as a classification task using $\mathcal{F}_\theta$. θ is updated by gradient-guided search in Θ to maximize the following objective:

$$\mathbb{E}_{P_{K\times O}}[\log \mathcal{F}_\theta(k,o)] + \mathbb{E}_{P_K P_O}[\log(1-\mathcal{F}_\theta(k,o))], \tag{6.6}$$

which is a standard binary cross-entropy loss over $P_{K\times O}$ and $P_K P_O$. Overall, this loss function compares the output of $\mathcal{F}_\theta$ to the ground truth, and it calculates the score that penalizes $\mathcal{F}_\theta$ based on its output distance from the expected value.

• *Example:* Consider the program in Fig. 6.3, in which we have $\langle \texttt{0}, \texttt{a} \rangle$ labeled as T. To prepare $P_K P_O$, there is one $\langle \texttt{0}, \texttt{a} \rangle$ marked as F when randomly combining k and o separated from pairs in $P_{K\times O}$. Thus, $\mathcal{F}_\theta(\text{“}\texttt{0}\text{”}, a[0])$ is simultaneously guided to output 1 and 0 with equal penalty. As expected, it eventually yields 0.5 to minimize the global penalty, which outputs a leakage of 4 bits (since $|K| = 4$) following Eq. 6.7.

Computing PD. Let the optimized parameter be $\theta^\dagger$, our definition of PD in Eq. 6.3 is re-expressed in the following way to compute point-wise information leak of k^* in its derived o^*:

$$c_{\theta^\dagger}(k^*;o^*) = |K| \frac{\mathcal{F}_{\theta^\dagger}(k^*,o^*)}{1-\mathcal{F}_{\theta^\dagger}(k^*,o^*)} \tag{6.7}$$

Furthermore, we have the following program-level information leak assessment over K and O.

$$I(K;O) = \mathbb{E}_{P_{K\times O}}\left[\log c_{\theta^\dagger}(k,o)\right] \tag{6.8}$$

Approximation and Correctness. Having access to all samples from a distribution P is difficult, if not impossible. As a common approximation, the objective in Eq. 6.6 is instead optimized over the *empirical* distribution $P^{(n)}$ produced by n samples drawn from P. Thus, the estimated leakage becomes:

$$\hat{I}^{(n)}_{\theta^\dagger}(K;O) = \mathbb{E}_{P^{(n)}_{K\times O}}[\log \hat{c}_{\theta^\dagger}(k,o)]. \tag{6.9}$$

Despite we estimate MI for side channels, the skeleton for analyzing *correctness* can be adopted from prior works [238, 30], since all approaches involve optimizing parameterized neural networks. In particular, we prove that $\exists\theta^{\dagger} \in \Theta$,

$$|\hat{I}^{(n)}_{\theta^{\dagger}}(K;O) - I(K;O)| \leq \mathcal{O}(\sqrt{\log(1/\delta)/n}), \tag{6.10}$$

with probability at least $1-\delta$ where $0 < \delta < 1$. We present detailed proofs in Sec. 6.11.1.

6.6 Apportioning Information Leakage

We analyze how leakage over $\langle o^*, k^* \rangle$ is apportioned among program points. This section models information leakage as a *cooperative game* among players (i.e., program points). Accordingly, we use Shapley value [220], a well-developed game theory approach, to apportion player contributions.

Overview. We use Shapley value (described below) to *automatically* flag certain records on a trace that contribute to leakage. Those flagged records are *automatically* mapped to assembly instructions using Intel Pin, since Pin records the memory address of each executed instruction. We then *manually* identify corresponding vulnerable source code. We report identified vulnerable source code to developers and have received timely confirmation. To clarify, this step is not specifically designed for Pin; users may replace Pin with other dynamic instrumentors like Qemu [31] or Valgrind [187].

Shapley value decides the contribution (i.e., leaked bits) of each program point covered on *one* trace o. To compute the average leakage (as reported in Sec. 6.8.3), users can analyze multiple traces and average the leaked bits at each program point. We now formulate information leakage as a cooperative game and define leakage apportionment as follows.

Definition 3 (Leakage Apportionment)**.** *Given total n bits of leaked information and m program points covered on the Pin-logged trace, an apportionment scheme allocates each program point a_i bits such that $\sum_{i=1}^{m} a_i = n$.*

Shapley Value. We address the leakage apportionment via Shapley value. Recall that each observation o denotes a trace of logged side channel records when target software is

processing a secret k. Let $\phi(o)$ be the leaked bits over one observation o, and let R^o be the set of indexes of records in o, i.e., $R^o = \{1, 2, \cdots, |o|\}$. For *all* $S \subseteq R^o \setminus \{i\}$, the Shapley value for the i-th side channel record is formally defined as

$$\pi_i(\phi) = \sum_S \frac{|S|!(|R^o| - |S| - 1)!}{|R^o|!}[\phi(o_{S\cup\{i\}}) - \phi(o_S)], \tag{6.11}$$

where $\pi_i(\phi)$ represents the information leakage contributed by the i-th record in o. o_S denotes that only records whose indexes in S serve as players in this cooperative game, and accordingly $o_{R^o} = o$. Eq. 6.11 is based on the intuition that contribution of a player (i.e., its Shapley value) should be decided by its marginal contribution to all $2^{|o|-1}$ coalitions over the remaining players. All players cooperatively form the overall leakage $\phi(o)$.

6.6.1 Computation and Optimization

The conventional procedure of deciding each player's contribution $\pi_i(\phi)$ for ϕ requires to generate a collection of variants V over o, where in each variant $o_v \in V$, some players `involved` and others `removed` [220]. In our scenario, it is infeasible to however remove a player when estimating leakage—removing a side channel record requires a new ϕ. Similar to [171], we propose to involve or remove a side channel record from o as follows:

Definition 4 (`Involved`)**.** *The i-th record of o gets involved in $\phi(o)$ if $o[i]$ is retained.*

Definition 5 (`Removed`)**.** *The i-th record of o is removed from $\phi(o)$ if $o[i]$ is reset to a constant, namely "base".*

The intuition is that, given k, if all records in its derived side channel observation o, are set to the same constant, i.e., $o_\emptyset = [base, \ldots, base]$, it's obvious that $o_\emptyset$ leaks no information of k, namely $\phi(o_\emptyset) = 0$. Conversely, by gradually setting $o_\emptyset[i] = o[i]$, which turns into $o_{\{i\}}$, we finally have $\phi(o_{R^o}) = \phi(o)$. The *base* is 0 in our setting for simplicity.

As stated in Sec. 6.6, computing Shapley value is costly, with complexity $\mathcal{O}(2^{|o|})$. This is particularly challenging, since a side channel trace o frequently contains millions of

records. We now propose several simple yet highly effective optimizations which successfully reduce the time complexity to (nearly) constant. These optimizations are based on domain knowledge and observations about side channel traces.

Approximating All and Tuning Later. Shapley value given in Eq. 6.11 can be equivalently expressed as [45]:

$$\begin{aligned}\pi_i(\phi) &= \sum_{u \in \text{Perm}(R^o)} \frac{1}{|R^o|!}[\phi(o_{u^i \cup \{i\}}) - \phi(o_{u^i})] \\ &= \mathbb{E}_{\text{Perm}(R^o)}[\phi(o_{u^i \cup \{i\}}) - \phi(o_{u^i})],\end{aligned} \tag{6.12}$$

where u is a permutation[8] that assigns each position t a player indexed with $u(t)$ and $\text{Perm}(R^o)$ is the set of all permutations over side channel records with indexes in R^o. u^i is the set of all predecessors of the i-th participant in permutation u, e.g., if $i = u(t)$, then $u^i = \{u(1), \cdots, u(t-1)\}$.

This equation transforms the computation of Shapley values into calculating the expectation over the distribution of u. Each time for a randomly selected u, we can calculate $\phi(o_{\{u(1),\cdots,u(j)\}})$ and $\pi_{u(j)}(\phi)$ for all j by incrementally setting $o[u(j)]$ as `involved`. Each $\pi_{u(j)}(\phi)$ is further updated as more permutations u are sampled. From the implementation side, Eq. 6.11 iteratively calculates accurate Shapley values for each record (but too slow), whereas Eq. 6.12 approximates Shapley values for all side channel records and tunes the values in later iterations of updates. We point out that Eq. 6.12 is more desirable for side channels, because *not all side channel records are correlated*. That is, updating Shapley value for one record may not affect the results of other records (i.e., "Dummy Players"; see Theorem 3 in Sec. 6.11.2). Given that sample mean converges to the true expectation when #samples increases, $\pi_{u(j)}(\phi)$ reaches its true value when it gets convergent. As a result, the calculation can be terminated early to reduce overhead, once the Shapley values stay unchanged. Our empirical results show that the Shapley values have negligible changes (i.e., the maximal difference of adjacent updates is less than 0.5) after only tens of updates.

[8] A set of permuted indexes. Note that the permutation does not exchange side channel records; it provides an order for records to get `involved`.

Pruning Non-Leaking Records Using Gradients. As discussed in Sec. 6.5, real-world software often generates *lengthy* and *sparse* side channel records [26, 278, 37]. That is, usually only a few records in a trace o really contribute to inferring secrets, and most records are "Null Players" (has no leak; see Theorem 3.1 in Sec. 6.11.2) in this cooperative game. Recall that the ϕ is formed by neural networks, whose gradients are typically informative. Here, we use gradients to prune Null Players before computing the standard Shapley values. In general, neural networks characterize the influence of one input element (i.e., one record on o) via gradients, and the volumes of gradients over inputs reflect how sensitive the output is to local perturbations on these input elements: higher volumes suggest more important elements.

We first rank all records by gradient volumes. Then, starting with the top one, we gradually set each record as `removed`. This way, we expose Null Players, as they are the remaining ones left when the leakage is zero. We find that, by setting at most a few hundred records as `removed` (which is far less than $|o|$), the leakage can be reduced to zero.

Batch Computations. The above optimizations reduce cost from $\mathcal{O}(2^{|o|})$ to hundreds of calls to ϕ. Further, modern hardware offers powerful parallel computing, allowing neural networks to accept a batch of data as inputs. Therefore, we batch the computations formed in previous steps; eventually, with *one or two batched calls* to ϕ, whose cost is (nearly) constant and negligible, we obtain accurate Shapley values.

Error Analysis. Our above approximation of Shapley value is 1) *unbiased*: it arrives the ground truth value with enough iterations. It is also 2) *convergent*: such that we can finish iterating whenever the approximated value unchanged.

Let the estimated and ground truth Shapley value be $\hat{\pi}$ and π. Previous studies have pointed out that $\hat{\pi} \sim \mathcal{N}(\pi, \frac{\sigma^2}{m})$ where $\mathcal{N}$ is the normal distribution and m is #iterations. It is also proved that $\sigma^2 < \frac{(\pi_{\max} - \pi_{min})^2}{4}$ where $\pi_{\max}$ and $\pi_{\min}$ are the maximum and minimal $\hat{\pi}$ during all iterations [45, 44].

Accuracy. Shapley value is based on several important properties that ensure the accuracy of localization [220]. In short:

Enabled by Shapley value, leakage localization, as a cooperative game, is precise with nearly no false negatives.

The standard Shapley value ensures no false negatives (see Theorem 3.1 in Sec. 6.11.2). Nevertheless, since we trade accuracy for scalability to handle lengthy o, our optimized Shapley value may have a few false negatives. Empirically, we find that it is rare to miss a vulnerable program point, when cross-comparing with findings of previous works [243].

6.7 Implementation

We implement CACHEQL in PyTorch (ver. 1.4.0) with about 2,000 LOC. The $\mathcal{C}$ of CACHEQL uses convolutional neural networks for images and fully-connected layers for keys; see details in [6]. We use Adam optimizer with learning rate 0.0002 for all models. We find that the learning rate does not largely affect the training process (unless it is unreasonably large or small). Batch size is 256. We ran experiments on Intel Xeon CPU E5-2683 with 256GB RAM and a Nvidia GeForce RTX 2080 GPU. For experiments based on Pin-logged traces, Sec. 6.8.4 presents the training time: CACHEQL is generally comparable or faster than prior tools. Experiments for `Prime+Probe`-logged traces take 1–2 hours.

6.8 Evaluation

We evaluate CACHEQL by answering the following research questions. **RQ1:** What are the quantification results of CACHEQL on production cryptosystems and are they correct? **RQ2:** How does CACHEQL perform on localizing side channel vulnerabilities? What are the characteristics of these localized sites? **RQ3:** What are the impact of CACHEQL's optimizations, and how does CACHEQL outperform existing tools? **RQ4:** How is the extendability of CACHEQL to different types of software and forms of side channels? We first introduce evaluation setups below.

6.8.1 Evaluation Setup

Software. We evaluate T-table-AES and RSA in OpenSSL 3.0.0, MbedTLS 3.0.0, and Libgcrypt 1.9.4. We consider an end-to-end pipeline where cryptographic libraries load the private key from files and decrypt ciphertext encrypted from "hello world." We quantify input image leaks for Libjpeg-turbo 2.1.2. We use the *latest* versions (by the time of writing) of all these software. We also assess old versions, OpenSSL 0.9.7, MbedTLS 2.15.0 and Libgcrypt 1.6.1, for a cross-version comparison. Some of them were also analyzed by existing works [244, 243, 26]. We compile software into 64-bit x86 executable using their default compilation settings. Supporting executables on other architectures is feasible, because CACHEQL's core technique is platform-independent.

Libjpeg & `Prime+Probe`. For the sake of presentation coherence, we focus on cryptosystems under the in-house setting (i.e., collecting execution traces via Pin) in the evaluation. Experiments of Libjpeg (including quantified leaks and localized vulnerabilities) and `Prime+Probe` are in Sec. 6.8.5.

Data Preparing & Training. When collecting the data for training/analyzing, we fix the public input and randomly sample keys to generate side-channel traces. For AES, we use the Linux `urandom` utility to generate 40K 128-bit keys for estimating CP using their corresponding side channel traces (collected via Pin or `Prime+Probe`). We also generate 10K extra keys and their side channel traces to de-bias non-determinism induced by ORAM (Sec. 6.8.2). The same keys are used for benchmarking AES of all cryptosystems. For RSA, we follow the same setting but generate 1024-bit private keys using OpenSSL. We have no particular requirements for training data (e.g. achieving certain coverage) — we observe that execution flows of cryptosystems are not largely altered by different keys, except that key values may influence certain loop iterations (e.g., due to zero bits). We find that the execution flows of cryptosystems are relatively more "regulated" than general-purpose software, which is also noted previously [244]. If secrets could notably alter the execution flow, it may indicate obvious issues like timing side channels, which should have been primarily eliminated in modern cryptosystems.

Trace Logging. Pin is configured to log program memory access traces to detect cache side channels due to SDA. We primarily consider cache side channels via cache lines and cache

Table 6.2: Leaked bits of AES/AES-NI in OpenSSL/MbedTLS.

	OpenSSL 3.0.0	OpenSSL 0.9.7	MbedTLS 3.0.0	MbedTLS 2.15.0
SCB	0	0	0	0
SDA	128.0	128.0	0	0

banks: for an accessed memory address `addr`, we compute its cache line and bank index as `addr >> 6` and `addr >> 2`, respectively. We also consider SCB, where Pin logs all control transfer destinations. Cache line/bank indexes are computed in the same way. We clarify that cache bank conflicts are inapplicable in recent Intel CPUs; we use this model for easier empirical comparison with prior works [244, 26, 243, 279, 278]. Trace statistics are presented in Table 6.4.

Ground Truth. To clarify, CACHEQL does *not* require the ground truth of leaked bits. Rather, as discussed in Sec. 6.4.3, CACHEQL is trained to *distinguish* traces produced when the software is processing different secrets. The ground truth is a one-bit variable denoting whether trace o is generated when the software processing secret k.

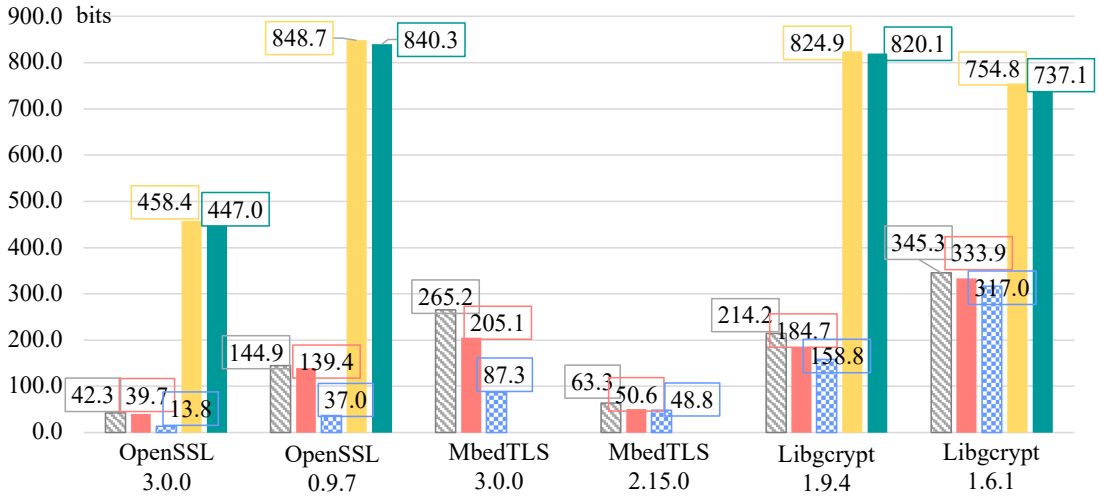
Non-Determinism. We quantify the leaks when enabling RSA blinding (Sec. 6.8.2). We also evaluate PathOHeap [225], a popular ORAM protocol, and consider real attack logs.

6.8.2 RQ1: Quantifying Side Channel Leakage

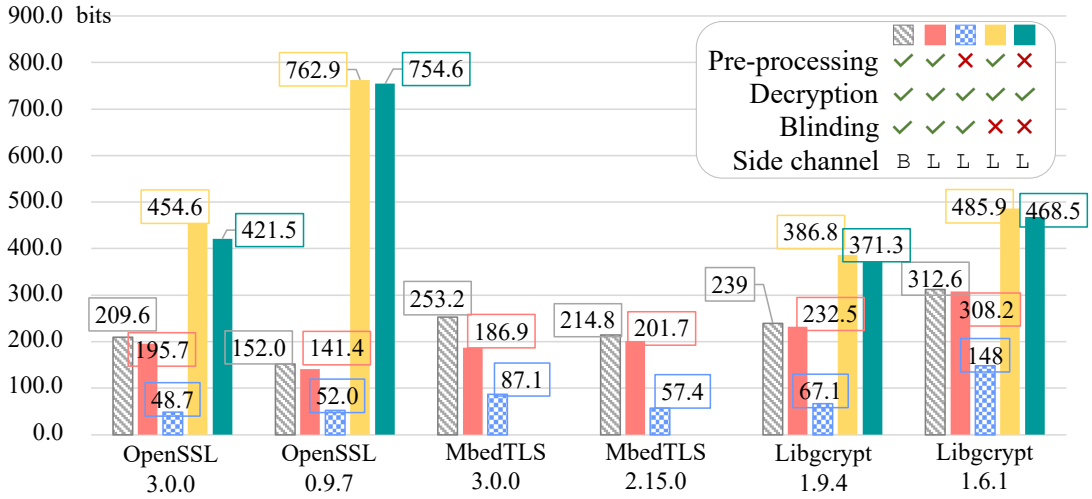
We report quantitative results over Pin-logged traces. Table 6.2 and Fig. 6.6 summarize the quantitative leakage results computed by CACHEQL regarding different software, where a large amount of secrets are leaked across all settings. We discuss each case in the rest of this section. Quantitative analyses of Libjpeg and `Prime+Probe` are presented in Sec. 6.8.5.

AES

The side channels of AES collected from the in-house settings are deterministic. The SDA of AES standard T-table version can leak all key bits, but this implementation has no SCB [26, 244]. These facts serve as the ground truth for verifying CACHEQL's quantification and localization. MbedTLS by default uses AES-NI, which has no SDA/SCB. As shown in Table 6.2, CACHEQL reports no leak in it.



(a) Leak bits via SDA vulnerabilities. `B/L` denote cache bank/line index.



(b) Leak bits via SCB vulnerabilities. `B/L` denote cache bank/line index.

Figure 6.6: Leaked bits of RSA in different settings. Blinding for Libgcrypt 1.6.1 refers to RSA optimized with CRT (see Sec. 6.8.2). For cache line side channels (`L` in the last row of legend), we present detailed breakdown: enabling (✓ in the 4th row of legend) vs. disabling (✗ in the 4th row of legend) blinding, and with (✓ in the 2nd row) vs. w/o (✗ in the 2nd row) considering Pre-processing.

CACHEQL reports 128 bits SDA leakage in AES-128 of OpenSSL whereas the SCB leakage in this implementation is zero. This shows that the quantification of CACHEQL is precise. We distribute the leaked bits to program points via Shapley value. All 128 bits are apportioned evenly toward 16 memory accesses in function `_x86_64_AES_encrypt_compact`. Manually, we find that these 16 memory accesses are all key-dependent table lookups.

Test Secure Implementations

CACHEQL also examines secure cryptographic implementations with no leakage. For these cases, the quantification derived from CACHEQL can also be seen as their correctness assessments. Given that said, as a dynamic method, CACHEQL is for bug detection, *not* for verification.

ORAM. PathOHeap yields non-deterministic side channels, by randomly inserting dummy memory accesses to produce highly lengthy traces. Since it takes several hours to process one logged trace of RSA, we apply PathOHeap on AES from OpenSSL 3.0.0. Overall, PathOHeap delivers provable mitigation: memory access traces, after being processed by PathOHeap, should not depend on secrets. CACHEQL reports consistent and accurate findings to empirically verify PathOHeap, as the leaked bit is quantified as *zero*.

Constant-Time Implementations. 13 constant-time utilities [10] from Binsec/Rel [66] are evaluated using CACHEQL. Side channel traces from these utilities are deterministic, whose quantified leaks are also *zero*. These results empirically show the correctness of CACHEQL's quantification.

RSA

RSA blinding is enabled by default in production cryptosystems. We quantify the information leakage of RSA with/without blinding, such that the logged traces are *non-deterministic* when blinding is on. As noted in Sec. 6.3 (❼), prior works mainly focus on the decryption fragment of RSA due to limited scalability [244, 74, 243, 26, 37]. As will be shown, this tradeoff neglects many vulnerabilities, primarily in the pre-processing modules of cryptographic libraries, e.g., key parsing and BIGNUM initialization.[9] CACHEQL efficiently analyzes the whole trace, covering Pre-processing and Decryption (see Fig. 6.6). As an ablation, CACHEQL also analyzes only Decryption, e.g., the green bar in Fig. 6.6.

Setup. Libgcrypt 1.9.4 uses blinding on both ciphertext and private keys. We enable/disable them together. Libgcrypt 1.6.1 lacks blinding but implements the standard RSA and another version using Chinese Remainder Theorem (CRT). Libgcrypt 1.9.4 uses blinding

[9]For simplicity, we refer to the pre-processing functions of cryptographic libraries as Pre-processing, whereas the following decryption functions as Decryption.

in the CRT version and disables blinding in the standard one. We evaluate these two RSA versions in Libgcrypt 1.6.1. MbedTLS does not allow disabling blinding.

Results Overview. Fig. 6.6 shows the quantitative results. Since cache bank only discards two least significant bits of the memory addresses, it leaks more information than using cache line which discards six bits. Blinding in modern cryptosystems notably reduces leakage: blinding influences secret-dependent memory accesses, introducing nondeterminism to prevent attackers from inferring secrets. Leakage varies across different software and variants of the same software. Secrets are leaked via SCB and SDA to varying degrees. If blinding is disabled, the total leak bits when considering only Decryption are close to the whole pipeline's leakage. This is reasonable as they leak information from the same source. With blinding enabled, leakage in Decryption is inhibited, and Preprocessing contributes the most leakage. Though blinding minimizes leakage in Decryption, Pre-processing remains highly vulnerable, and it is generally overlooked previously.

OpenSSL. OpenSSL 3.0.0 has higher SCB leakage in Pre-processing with blinding enabled. As will be discussed in Sec. 6.8.3, this leakage is primarily introduced by `BN_bin2bn` and `bn_expand2` functions, which convert key from string into BIGNUM. The issue persists with OpenSSL 0.9.7. Moreover, compared with ver. 3.0.0, OpenSSL 0.9.7 has more SDA (but less SCB) leakage with blinding enabled. These gaps are also primarily caused by the `BN_bin2bn` function in Pre-processing. We find that OpenSSL 3.0.0 skips leading zeros when converting key from string into BIGNUM, which introduces extra SCB leakage. In contrast, OpenSSL 0.9.7 first converts the key with leading zeros into BIGNUM and then uses `bn_fix_top` to remove those leading zeros, causing extra SDA leakage. Also, if blinding is disabled, OpenSSL 0.9.7 leaks approximately twice as many bits as OpenSSL 3.0.0. According to the localization results of CACHEQL, OpenSSL 0.9.7 has memory accesses and branch conditions that directly depend on keys, which are vulnerable and lead to over 800 bits of leakage. We manually check OpenSSL 3.0.0 and find that most of those vulnerable functions have been re-implemented in a constant-time way.

MbedTLS. CACHEQL finds many SDA in MbedTLS 3.0.0, which primarily occurs in the `mbedtls_mpi_read_binary` and `mbedtls_mpi_copy` functions during Pre-processing. The problem is not severe in ver. 2.15.0. We manually compare the two versions' Preprocessing and find that the CRT initialization routines differ. In short, MbedTLS 3.0.0

avoids computing DP, DQ and QP (parts of the RSA private key in CRT) and instead reads them from the PKCS1 structure, and therefore, `mbedtls_mpi_copy` function is called for several times. The 2.15.0 version calculates DP, DQ and QP from the private key via BIGNUM involved functions (e.g., `mbedtls_mpi_mul_mpi`). The `mbedtls_mpi_copy` function leaks information via SDA and SCB, whereas the BIGNUM computation in the 2.15.0 version mainly leaks via SCB. This difference also explains why both versions have many SCB, which are dominated by their Pre-processing.

Libgcrypt. Libcrypt 1.9.4 has most SCB leakage in Pre-processing with blinding enabled. Nearly all leaked bits are from the `do_vsexp_sscan` function, which parses the key from s-expression. Decryption only leaks negligible bits. Manual studies show that in Libgcrypt 1.9.4, most BIGNUM-involved functions in Decryption are constant time and safe. Nevertheless, CACHEQL identifies leaks in `do_vsexp_sscan`. This illustrates that CACHEQL comprehensively analyzes production cryptosystems, whereas developers neglect patching all sensitive functions in a constant-time manner, enabling subtle leakages. Also, both versions have SDA leakage primarily in the `_gcry_mpi_powm` function; this is also noted in prior works [244, 243, 37]. As aforementioned, Libgcrypt 1.9.4 uses the standard RSA without CRT when blinding is disabled. The 1.6.1 version does not offer blinding for both the standard RSA and the CRT version. It's obvious that the standard version leaks more than the CRT version.

Correctness. It is challenging to obtain ground truth in our evaluation. Aside from the AES cases and secure implementations in Sec. 6.8.2 who have the "ground truth" (either leaking 128 bits or zero bits) to compare with, there are several cases in RSA whose leaked bits can be calculated manually, facilitating to assess the correctness of CACHEQL's quantification.

Case$_1$: `BN_num_bits_word` function in OpenSSL 0.9.7, which is first identified by CacheD [244] and currently fixed in OpenSSL 3.0.0, has 256 different entries depending on secrets. It leaks $-\log \frac{1}{256} = 8.0$ bits, in case entries are accessed evenly (which should be true since key bits are generated independently and uniformly). CACHEQL reports the leakage as 7.4 bits, denoting a close quantification.

Case$_2$: `do_vsexp_sscan` function (see Fig. 6.7) in both versions of Libgcrypt has control branches depending on whether a secret is greater than 10. The SCB at L2 of Fig. 6.7, in

Table 6.3: Representative vulnerable func. and their categories.

OpenSSL 3.0.0	Type	**MbedTLS 3.0.0**	Type	**Libgcrypt 1.9.4**	Type
`bn_expand2`	Ⓐ, Ⓑ, Ⓒ	`mbedtls_mpi_copy`	Ⓐ, Ⓑ, Ⓒ	`mul_n_basecase`	Ⓑ, Ⓒ, Ⓓ
`BN_bin2bn`	Ⓐ, Ⓒ	`mbedtls_mpi_read_binary`	Ⓐ, Ⓒ	`do_vsexp_sscan`	Ⓐ, Ⓓ
`BN_mod_exp_mont`	Ⓑ, Ⓒ Ⓓ, Ⓔ	`mpi_montmul`	Ⓑ, Ⓒ, Ⓓ	`_gcry_mpih_mul`	Ⓑ, Ⓒ, Ⓓ, Ⓔ

theory, leaks $-\log\frac{1}{16}+\log\frac{1}{10}$ = 0.68 bits of information, as the possible key values are reduced from 16 to 10 when L2 is executed. Similarly, the SCB at L4 leaks $-\log\frac{1}{16}+\log\frac{1}{6}$ = 1.42 bits. When CACHEQL analyzes one trace, it apportions around 1 bit to each of the two records corresponding to the SCB at L2 and L4. We interpret that CACHEQL provides accurate quantification and apportionment for this case.

Answer to RQ1: By quantifying leakage with CACHEQL, we find that information leaks are prevalent in cryptosystems, even when hardening methods (e.g., blinding) are enabled. Most leaks reside in the pre-processing stage neglected by existing research. For some cases, the development of cryptosystems may increase the amount of leakage. The correctness of CACHEQL is empirically validated using a total of 24 instances of known bit leakages.

6.8.3 RQ2: Localizing Leakage Sites

This section reports the leakage program points localized in RSA by CACHEQL using Shapley value. We report representative functions in Table 6.3. See [6] for detailed reports.

When blinding is enabled, CACHEQL localizes all previously-found leak sites and hundreds of new ones.

Clarification. Some leak sites localized by CACHEQL are dependent, e.g., several memory accesses within a loop where only the loop condition depends on secrets. To clarify, CACHEQL does not distinguish dependent/independent leak sites, because from the game theory perspective, those dependent leak sites (i.e., players) collaboratively contribute to the leakage (i.e., the game). Also, reporting all dependent/independent leak sites may be likely more desirable, as it paints the complete picture of the software attack surface. Overall, identifying *independent* leak sites is challenging, and to our best knowledge, prior works also do not consider this. This would be an interesting future work to

explore. On the other hand, vulnerabilities identified by CACHEQL are from *hundreds of functions* that are not reported by prior works, showing that the localized vulnerabilities spread across the entire codebase, whose fixing may take considerable effort.

Categorization of Vulnerabilities

We list all identified vulnerabilities in [6]. Nevertheless, given the large number of (newly) identified vulnerabilities, it is obviously infeasible to analyze each case in this chapter. To ease the comparison with existing tools that feature localization, we categorize leak sites from different aspects. We first categorize the leak sites according to their locations in the codebase (Ⓐ and Ⓑ). We then use Ⓓ and Ⓔ to describe how secrets are propagated. Moreover, since leaking-leading-zeros is less considered by previous work, we specifically present such cases in Ⓒ.

Ⓐ Leaking secrets in Pre-processing: Leak sites belonging to Ⓐ occur when program parses the key and initializes relevant data structures like BIGNUM. Note that this stage is rarely assessed by previous static (trace-based) tools due to limited scalability; empirical results are given in Table 6.5.

Ⓑ Leaking secrets in Decryption: While Ⓑ is primarily analyzed by prior static tools, in practice, they have to trade precision for speed, omitting analysis of full implicit information flow (❽ in Table 6.1). Therefore, their findings related to Ⓑ compose only a small subset of CACHEQL's findings. Also, prior dynamic tools, including DATA and MicroWalk, are less capable of detecting Ⓑ. This is because blinding is applied at Decryption (❷ in Table 6.1). DATA likely neglects leak sites when blinding is enabled since it merely differentiates logged side channel traces with key fixed/varied. MicroWalk incorrectly regards data accesses/control branches influenced by blinding as vulnerable. Blinding can introduce a great number of records (see Table 6.4 for increased trace length), and MicroWalk fails to correctly analyze all these cases.

Ⓒ Leaking leading zeros: Besides CACHEQL, findings belonging to Ⓒ were only partially reported by DATA. Particularly, given DATA is less applicable when facing blinding (noted in Sec. 6.3), it finds Ⓒ only in Pre-processing, where blinding is not enabled yet. Since CACHEQL can precisely quantify (❹ in Table 6.1) and apportion (❺) leaked bits,

it is capable of identifying Ⓒ in Decryption; the same reason also holds for Ⓑ. At this step, we manually inspected prior static tools and found they only "taint" the content of a BIGNUM, which is an array, if BIGNUM stores secrets. The number of leading zeros, which has enabled exploitations (CVE-2018-0734 and CVE-2018-0735 [252]) and is typically stored in a separate variable (e.g., `top` in OpenSSL), is neglected.

Ⓓ Leaking secrets via explicit information flow: Most findings belonging to Ⓓ have been reported by existing static tools. CACHEQL re-discovers **all** of them despite it's dynamic. We attribute the success to CACHEQL's precise quantification, which recasts MI as CP (Sec. 6.4.3), and localization, where leaks are re-formulated as a cooperative game (Sec. 6.6).

Ⓔ Leaking secrets via implicit information flow: As discussed above, prior static tools are incapable of fully detecting Ⓔ. Also, many findings of CACHEQL in Ⓔ overlap with that in Ⓑ. Since DATA cannot handle blinding well (blinding is extensively used in Decryption), only a small portion of Ⓔ were correctly identified by DATA. DATA also has the same issue to neglect CACHEQL's findings in Ⓓ.

In sum, static-/trace-based tools (CacheD, CacheS, Abacus) can detect Ⓑ ∩ Ⓓ but cannot identify Ⓐ ∪ Ⓒ ∪ Ⓔ. As noted in Sec. 6.3, MicroWalk cannot properly differ randomness induced by blinding vs. keys, and is inaccurate for the RSA case with blinding enabled. DATA pinpoints Ⓐ (accordingly include Ⓐ ∩ Ⓒ) and is less applicable for Ⓑ. CACHEQL, due to its precise quantification, localization, and scalability, can identify Ⓐ ∪ Ⓑ ∪ Ⓒ ∪ Ⓓ ∪ Ⓔ.

Characteristics of Leakage Sites

The leakage sites exist in all stages of cryptosystems. Below, we use case studies and the distribution of leaked bits to illustrate their characteristics. In short, the leaks start when parsing keys from files and initializing secret-related BIGNUM, and persist during the whole life cycle of RSA execution.

Case Study$_1$: Fig. 6.7 presents a case newly disclosed by CACHEQL, which is the key parsing implemented in Libgcrypt 1.6.1 and 1.9.4. As discussed in Sec. 6.8.2 (see Case$_2$), this function has SCB explicitly depending on the key read from files. It therefore contains Ⓐ and Ⓓ. Similar leaks exist in other software. For instance, as localized by CACHEQL

```
1 int hextonibble(char s) {
2   if(s >= '0' && s <= '9')
3     return s - '0';
4   if(s >= 'A' && s <= 'F')
5     return 10 + s - 'A';
6   if(s >= 'a' && s <= 'f')
7     return 10 + s - 'a';
8 }
```

```
9  static gpg_err_code_t
10 do_vsexp_sscan(gcry_sexp_t *ret,
11         char *buf, size_t len) {
12   struct make_space_ctx c;
13   for(char *s=buf; len; len--) {
14     *c.p++ = hextonibble(*(s++));
15   }
16 }
```

Figure 6.7: Simplified vulnerable program points localized in Libgcrypt 1.9.4. This function has SCB directly depending on bits of the key.

and DATA, the `EVP_DecodeUpdate` function in two versions of OpenSSL have SDA via the lookup table `data_ascii2bin` when decoding keys read from files.

Case Study$_2$: Fig. 6.8 depicts the life-cycle of BIGNUM in OpenSSL 3.0.0, including initialization and computations. We show how secrets are leaked along the usage of BIGNUM.

1 `BN_bin2bn`@L39: A BIGNUM is initialized using `s` at L40, which is parsed from the key file in the `.pem` format. A `for` loop at L42 *skips leading zeros*, propagating `s` to `len` via implicit information flow. Then, `len` is propagated to `top` (L49 or L53). Thus, future usage of `top` clearly leaks secret.

2 `BN_mod_exp_mont`@L1: `BN_num_bits` is called to calculate #bits (after excluding leading zeros) of BIGNUM `p`. `BN_num_bits` further calls `BN_num_bits_word` which we have discussed in Sec. 6.8.2. #bits is stored in `bits` at L5. Later, `bits` is propagated to `wstart` at L7.

3 `BN_window_bits_for_exponent_size`@L20: `w` is propagated from `bits` at L6, given control branches from L21 to L24 directly depend on `b`.

4 `BN_is_bit_set`@L33: `top` of BIGNUM `p` directly decides the return value at L36. Its content, namely array `d`, also sets the return value at L37. Given `wvalue` and `wend` at L13 and L15 are updated according to the return value of `BN_is_bit_set`, they are thus implicitly propagated.

5 `bn_mul_mont_fixed_top`@L26: The access to array `val` at L17 is indexed with `wvalue`, and therefore, it induces SDA. Variable `b` at L27 is also propagated via `wvalue`, and the `if` branch at L28 thus introduces SCB.

```
1  int BN_mod_exp_mont(BIGNUM *rr, BIGNUM *a,
2                      BIGNUM *p, BIGNUM *m, ) {
3    // table of variables obtained from 'ctx'
4    BIGNUM *val[TABLE_SIZE];
5    int bits = BN_num_bits(p);
6    int w = BN_window_bits_for_exponent_size(bits);
7    int wstart = bits - 1;
8    for(;;) {
9      int wvalue = 1;
10     int wend = 0;
11     for(int i = 1; i < w; i++)
12       if(BN_is_bit_set(p, wstart - i)) {
13         wvalue <<= (i - wend);
14         wvalue |= 1;
15         wend = i;
16       }
17     bn_mul_mont_fixed_top(r, r, val[wvalue >> 1]);
18   }
19 }
```

implicit information flow "taints" **wvalue** and **wend**

```
20 #define BN_window_bits_for_exponent_size(b)\
21                 ((b) > 671 ? 6 : \
22                  (b) > 239 ? 5 : \
23                  (b) >  79 ? 4 : \
24                  (b) >  23 ? 3 : 1)
25
26 int bn_mul_mont_fixed_top(BIGNUM *r,
27                    BIGNUM *a, BIGNUM *b) {
28   if(a == b)
29     bn_sqr_fixed_top(tmp, a)
30   else
31     bn_mul_fixed_top(tmp, a, b)
32 }
33 int BN_is_bit_set(BIGNUM *a, int n) {
34   int i = n / BN_BITS2;
35   int j = n % BN_BITS2;
36   if(a->top <= i) return 0;
37   return (int)(((a->d[i]) >> j) & 1;
38 }
```

```
39 BIGNUM *BN_bin2bn(int len,
40   char *s, BIGNUM *ret) {
41   // s is secret
42   for ( ; len && *s == 0; s++) {
43     // skip leading zeros
44     len --;
45   }
46
47   n = len;
48   if (n == 0) {
49     ret->top = 0;
50     return ret;
51   }
52   i = ((n - 1) / BN_BYTES) + 1;
53   ret->top = i;
54   /* top is the "size" of a
55   BIGNUM in later computing */
56   return ret;
57 }
```

len is tainted via implicit information flow here.

Figure 6.8: Vulnerable program points localized in OpenSSL 3.0.0. We mark the line numbers of SDA vulnerabilities and SCB vulnerabilities found by CACHEQL. Secrets are propagated from p to other variables via explicit or implicit information flow, which confirm each SDA/SCB vulnerability found by CACHEQL.

Overall, 1 executes at Pre-processing and is only detected by DATA and CACHEQL. It has both explicit (L47) and implicit (L42) information flow. Thus, it has Ⓐ Ⓒ Ⓓ Ⓔ. Similarly, 2 contains Ⓑ Ⓒ Ⓓ Ⓔ. Both 3 and 4 have Ⓑ Ⓒ Ⓓ. 5 only has Ⓑ Ⓓ. Among the leak sites discussed above, only five SCB at L21-L24 and L36 are detected by previous static tools; remaining ones are newly reported by CACHEQL.

Case Study$_3$: Fig. 6.9 shows leaking sites disclosed by CACHEQL in MbedTLS 3.0.0. In

```
1  int mbedtls_mpi_div_mpi(Q, R, A, B) {
2    // Q, R, A, B are mbedtls_mpi*
3    mbedtls_mpi_copy(&X, A) ;
4    mbedtls_mpi_copy(&Y, B) ;
5    int n = X.n - 1;
6    int t = Y.n - 1;
7    while(mbedtls_mpi_cmp_mpi(&X, &Y)) {
8      Z.p[n - t]++; // local variable Z
9      mbedtls_mpi_sub_mpi(&X, &X, &Y);
10   }
11 }
12 int mbedtls_mpi_copy(mbedtls_mpi *X,
13                      mbedtls_mpi *Y) {
14   for(size_t i = Y->n-1; i > 0; i--)
15     if(Y->p[i] != 0) break;
16 }
```

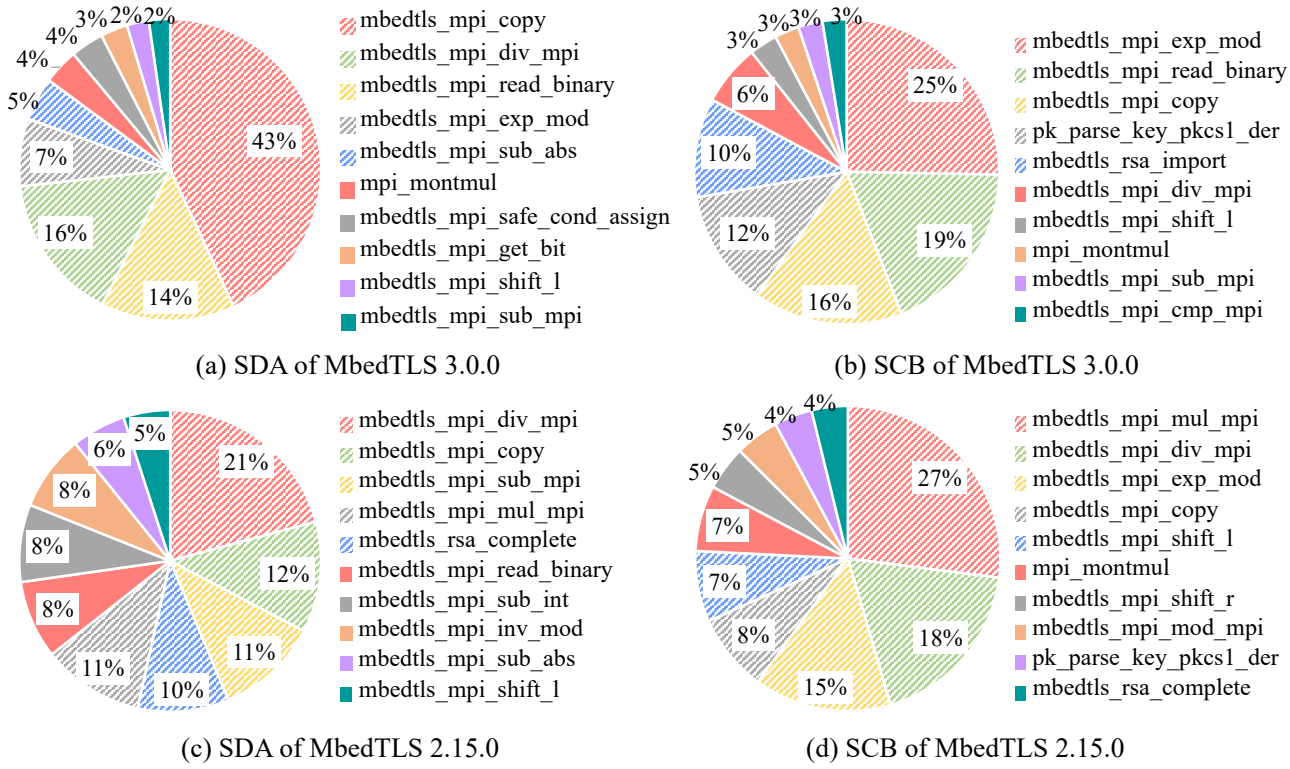
```
17 int mbedtls_mpi_cmp_mpi(X, Y) {
18  // X and Y are mbedtls_mpi*
19   for(i = X->n; i > 0; i--)
20     if(X->p[i - 1] != 0)
21       break;
22   for(j = Y->n; j > 0; j--)
23     if(Y->p[j - 1] != 0)
24       break;
25
26   for( ; i > 0; i-- ) {
27     if(X->p[i-1] > Y->p[i-1])
28       return(X->s);
29     if(X->p[i-1] < Y->p[i-1])
30       return(-X->s);
31   }
32 }
```

both SDA and SCB here.

Figure 6.9: Vulnerable program points localized in MbedTLS 3.0.0. We mark the line numbers of SDA and SCB.

short, MbedTLS has similar implementation of BIGNUM with OpenSSL, where the variable `n` in BIGNUM stores the number of leading zeros. Later computations rely on `n` for optimization, for instance, the SDA at L8 in `mbedtls_mpi_div_mpi`@L1. It's worth noting that `mbedtls_mpi_copy`@L12 is extensively called within the life cycles of all involved BIGNUM, contributing to notable leaks in the whole pipeline. Similar leaks also exist in MbedTLS 2.15.0. See our website [6] for more details.

Distribution. Fig. 6.10 reports the distribution of leaked bits among top-10 vulnerable functions localized in MbedTLS. The two versions of MbedTLS primarily leak bits in Preprocessing and have different strategies when initializing BIGNUMs for CRT optimization. Thus, the distributions of most vulnerable functions vary. For instance, the most vulnerable functions in ver. 2.15.0 are for multiplication and division; they are involved in calculating BIGNUMs for CRT. Notably, `mbedtls_mpi_copy` is among the top-5 vulnerable functions on all four charts in Fig. 6.10. This function leaks the leading zeros of the input BIGNUMs via both SDA and SCB. The `mbedtls_mpi_copy` function, as a memory copy routine function, is frequently called (e.g., more than 1,000 times in ver. 2.15.0). Though this function only leaks the leading zeros, given that its input can be the private key or key-dependent intermediate value, the accumulated leakage is substantial.



(a) SDA of MbedTLS 3.0.0

(b) SCB of MbedTLS 3.0.0

(c) SDA of MbedTLS 2.15.0

(d) SCB of MbedTLS 2.15.0

Figure 6.10: Distribution of top-10 functions in MbedTLS leaking most bits via either SDA or SCB vulnerabilities. Legend are in descending order.

Answer to RQ2: CACHEQL confirms all known flaws and identifies many new leakage sites, which span over the life cycle of cryptographic algorithms and exhibit diverse patterns. Distributions of leaked bits among vulnerable functions varies notably between software versions.

6.8.4 RQ3: Performance Comparison

To assess CACHEQL's optimizations and re-formulations, we compare CACHEQL with previous tools on the speed, scalability, and capability of quantification and localization.

Trace Statistics. We report the lengths (after padding) of traces collected using Pin in Table 6.4. In short, all traces collected from real-world cryptosystems are lengthy, imposing high challenge for analysis. Nevertheless, CACHEQL employs encoding module $\mathcal{S}$ and compressing module $\mathcal{R}$ to effectively process lengthy and sparse traces, as noted in Sec. 6.5.

Impact of Re-Formulations/Optimizations. CACHEQL casts MI as CP when quantifying

Table 6.4: Padded length of side channel traces collected using Intel Pin. The above/below five rows are for SDA/SCB.

	Pre-processing Decryption Blinding	- Decryption Blinding	Pre-processing Decryption -	- Decryption -
OpenSSL 3.0.0	$256 \times 256 \times 64$	$256 \times 256 \times 15$	$256 \times 256 \times 40$	$256 \times 256 \times 9$
OpenSSL 0.9.7	$256 \times 256 \times 20$	$256 \times 256 \times 14$	$256 \times 256 \times 14$	$256 \times 256 \times 13$
MbedTLS 3.0.0	$256 \times 256 \times 16$	$256 \times 256 \times 2$	N/A	N/A
MbedTLS 2.15.0	$256 \times 256 \times 14$	$256 \times 256 \times 2$	N/A	N/A
Libgcrypt 1.9.4	$256 \times 256 \times 36$	$256 \times 256 \times 22$	$256 \times 256 \times 26$	$256 \times 256 \times 25$
Libgcrypt 1.6.1	$256 \times 256 \times 5$	$128 \times 128 \times 19$	$256 \times 256 \times 11$	$256 \times 256 \times 10$
OpenSSL 3.0.0	$256 \times 256 \times 40$	$256 \times 256 \times 10$	$256 \times 256 \times 24$	$256 \times 256 \times 4$
OpenSSL 0.9.7	$256 \times 256 \times 8$	$256 \times 256 \times 5$	$256 \times 256 \times 5$	$256 \times 256 \times 4$
MbedTLS 3.0.0	$256 \times 256 \times 6$	$256 \times 256 \times 1$	N/A	N/A
MbedTLS 2.15.0	$256 \times 256 \times 6$	$256 \times 256 \times 1$	N//A	N/A
Libgcrypt 1.9.4	$256 \times 256 \times 12$	$256 \times 256 \times 8$	$256 \times 256 \times 10$	$256 \times 256 \times 9$
Libgcrypt 1.6.1	$256 \times 256 \times 3$	$128 \times 128 \times 8$	$256 \times 256 \times 6$	$256 \times 256 \times 5$

Table 6.5: Scalability comparison of static- or trace-based tools.

	CacheD	Abacus	CacheS	CacheAudit
Technique	symbolic execution		abstract interpretation	
Libgcrypt	fail ($>$ 48h)	fail ($>$ 48h)	fail	fail
Libjpeg	fail	fail	fail	fail

the leaks. This re-formulation is faster (see comparison below) and more precise, because calculating MI via MP (as done in MicroWalk) cannot distinguish blinding in traces. As reported in Table 6.4, a great number of records are related to blinding, and they lead to false positives of MicroWalk. For localization, the unoptimized Shapley value has $\mathcal{O}(2^N)$ computing cost. Given the trace length N is often extremely large (Table 6.4), computing Shapley value is infeasible. With our domain-specific optimizations, the cost is reduced as nearly constant.

Time Cost and Scalability

Scalability Issue of Static-/Trace-Based Tools. As noted in ❼ in Sec. 6.3, prior static- or trace-based analyses rely on expensive and less scalable techniques. They, by default, primarily analyze a program/trace cut and neglect those pre-processing functions in cryptographic libraries. To faithfully assess their capabilities, we configure them to analyze the entire trace/software (which needs some tweaks on their codebase). We benchmark them

Table 6.6: Training time of 50 epochs for the RSA cases.

Configuration	SDA				SCB			
	Pre. Dec. Blind.	- Dec. Blind.	Pre. Dec. -	- Dec. -	Pre. Dec. Blind.	- Dec. Blind.	Pre. Dec. -	- Dec. -
OpenSSL 3.0.0	22h	5h	3h	50min	13h	3h	2.5h	20min
OpenSSL 0.9.7	6.5h	5h	1h	1h	2.5h	1.5h	25min	20min
MbedTLS 3.0.0	5h	40min	N/A	N/A	2.5h	20min	N/A	N/A
MbedTLS 2.15.0	5h	40min	N/A	N/A	2.5h	20min	N/A	N/A
Libgcrypt 1.9.4	12.5h	7.5h	1.5h	1.5h	4h	2.5h	50min	45min
Libgcrypt 1.6.1	1.5h	1.5h	55min	50min	1h	40min	30min	25min

1. Due to the limited space, we use Pre., Dec., and Blind. to denote Pre-processing, Decryption, and Blinding, respectively.
2. Blind. has $\times 4$ training samples.

on Libjpeg and RSA of Libgcrypt 1.9.4. Abacus/CacheD/CacheS/CacheAudit can only analyze 32-bit x86 executable. We thus compile 32-bit Libgcrypt and Libjpeg. Results are in Table 6.5. CacheS and CacheAudit throw exceptions of unhandled x86 instruction. Both tools, using rigorous albeit expensive abstraction interpretation, appear to handle a subset of x86 instructions. Fixing each unhandled instruction would presumably require defining a new abstract operator [64], which is challenging on our end. Abacus and CacheD can be configured to analyze the full trace of Libgcrypt. Nevertheless, both of them fail (in front of unhandled x86 instructions) after about 48h of processing. In contrast, CACHEQL takes less than 17h to finish the training and analysis of the Libcrypt case; see Table 6.6.

Training/Analyzing Time of CACHEQL. Table 6.6 presents the RSA case training time, which is calculated over 50 epochs (the maximal epochs required) on one Nvidia GeForce RTX 2080 GPU. In practice, most cases can finish in less than 50 epochs. For AES-128, training 50 epochs takes about 2 mins. Training 50 epochs for Libjpeg/PathOHeap takes 2-3 hours. As discussed in Sec. 6.4.3, since we transform computing MI as estimating CP, CACHEQL only needs to be trained (for estimating CP) once. Once trained, it can analyze 256 traces in **1-2 seconds** on one Nvidia GeForce RTX 2080 GPU, and less than **20 seconds** on Intel Xeon CPU E5-2683 of **4** cores.

In sum, CACHEQL is much faster than existing trace-based/static tools. By using CP, it principally reduces computing cost comparing with conventional dynamic tools (see Sec. 6.4). We also note that it is hard to make a fully fair comparison: training CACHEQL can use GPU while existing tools *only* support to use CPUs. Though CACHEQL

has smaller time cost on the GPU (Nvidia 2080 is *not* very powerful), we do not claim CACHEQL is faster than prior dynamic tools. In contrast, we only aim to justify that CACHEQL is *not* as heavyweight as audiences may expect. Enabled by our theoretical and implementation-wise optimizations, CACHEQL efficiently analyzes complex production software.

Capability of Quantification and Localization

Small Programs and Trace cuts. As evaluated in Sec. 6.8.4, previous static-/trace-based tools are incapable of analyzing the full side channel traces. Therefore, we compare them with CACHEQL using small program (e.g., AES) and trace cuts.

Overall, the speed of CACHEQL (i.e., training + analyzing) still largely outperforms static/trace-based methods. For instance, CacheD [244], a qualitative tool using symbolic execution, takes about 3.2 hours to analyze only the decryption routine of RSA in Libgcrypt 1.6.1 without considering blinding. CACHEQL takes under one hour for this setting. In addition, Abacus [26], which performs quantitative analysis with symbolic execution, requires 109 hours to process one trace of Libgcrypt 1.8. Note that it only analyzes the decryption module (several caller/callee functions) without considering the blinding, pre-rocessing functions, etc. In contrast, CACHEQL can finish the training within 2 hours (the trace length of Libgcrypt 1.9 is about the same as ver. 1.8) in this setting. It's worth noting that, CACHEQL only needs to be trained for once, and it takes only several seconds to analyze one trace. That is, when analyzing multiple traces, previous tools has fold increase on the time cost whereas CACHEQL only adds several seconds.

The quantification/localization precision of CACHEQL is also much higher. Abacus reports 413.6 bits of leakage for AES-128 (it neglects dependency among leakage sites, such that the same secret bits can be repetitively counted at different leakage sites), which is an overestimation since the key has only 128 bits. For RSA trace cuts, Abacus underquantifies the leaked bits because it misses many vulnerabilities due to implicit information-flow. When localizing vulnerabilities in AES-128, we note that all static/trace-based have correct results. For localization results of RSA trace cuts, see Sec. 6.8.3. In short, none of the previous tools can identify all the categories of vulnerabilities.

Dynamic Tools. Previous dynamic tools do not suffer from the scalability issue and have comparable speed with CACHEQL. Nevertheless, they require re-launching their whole pipeline (e.g., sampling + analyzing) for each trace.

For quantification, MicroWalk over-quantifies the leaked bits of RSA as 1024 when blinding is enabled, since it regards random records as vulnerable. Similarly, it reports that ORAM cases have all bits leaked despite they are indeed secure. MicroWalk can correctly quantify the leaks of whole traces for AES cases and constant-time implementations, because no randomness exists. Nevertheless, since the same key bits are repeatedly reused on the trace, its quantification results for single record, when summed up, are incorrectly inconsistent with the result of whole trace. For localization, as summarized in Sec. 6.8.3, previous dynamic tools are also either incapable of identifying all categories of vulnerabilities or yields many false positive. For instance, MicroWalk can regards all records related to blinding (over 1M in OpenSSL 3.0.0; see trace statistics in Table 6.4) as "vulnerable".

Answer to RQ3: With domain-specific transformations and optimizations applied, CACHEQL addresses inherent challenges like non-determinism, and features fast, scalable, and precise quantification/localization. Evaluations show its advantage over previous tools.

6.8.5 RQ4: Extending CACHEQL for Other Side Channels and Software

Data Preparing. We choose the CelebA dataset [168] as the image inputs of Libjpeg. The CelebA consists of 160,000 training and 10,000 validation images. We use the training images and their corresponding side channel traces to estimate CP via $\mathcal{F}_\theta$. The validation images and their induced side channels are adopted for de-biasing (in case there exists non-determinism).

Trace Logging. Following the same configuration of Sec. 6.8, we use Pin to log execution traces of Libjpeg.

Logging via `Prime+Probe`. Besides using Pin, we collect cache set access traces (for both cryptosystems and Libjpeg) via `Prime+Probe` [236], in userspace-only scenarios. Following [278], we use Mastik [269], a micro-architectural side channel toolkit, to perform `Prime+Probe` and log victim's access toward L1D and L1I cache. We use Linux

Table 6.7: Leakage ratios (see Eq. 6.4) of Libjpeg without $\rightarrow$ with considering generalizability.

	SDA	SCB
cache line	100% $\rightarrow$ 93%	100% $\rightarrow$ 49%

`taskset` to pin victim software and the spy process on the same CPU core. Scripts of `Prime+Probe` experiments are at [9].

Libjpeg

Quantification. Libjpeg does not contain mitigations like blinding. Thus, side channels collected by Pin are deterministic. Nevertheless, we argue that merely considering the difference among SDA/SCB, which is a conventional setup for cryptosystems, will *overestimate* the leakage. The reasons are two-fold: 1) Traces can be largely divergent by tweaking trivial pixels in the input images, e.g., background color pixels. 2) Even if all pixels are equally sensitive, the leakage may be insufficient to recover input images. For instance, attackers only infer whether each pixel value is larger than a threshold (as how keys are usually recovered) and obtain a binary image — it's still infeasible to recognize humans in such images.

To handle these, we extend the *generalizability* consideration, which is proposed to handle non-deterministic side channels derived from cryptosystems, to quantify deterministic side channels made by Libjpeg. Therefore, our quantification is the same as for processing non-deterministic side channels of cryptosystems. Due to the aforementioned issue, we deem trivial pixels contain little information. As introduced in Sec. 6.4.4, CACHEQL should accordingly report a zero leakage since these non-sensitive factors are not generalizable.

Given that the #images is infinite, it's infeasible to decide the value of $p(F)/p(T)$ in Eq. 6.3 which should be a constant. We thus report the leakage ratio for Libjpeg. As shown in Table 6.7, leakage ratios reach 100% for SDA and SCB if we only consider the *distinguishability*, which indicates side channels can differ images. Nevertheless, the leakage ratios are reduced to 93% for SDA and 49% for SCB when we faithfully take *generalizability* into account.

Table 6.8: Representative vulnerable functions localized in Libjpeg and their types.

Function	Type
`decode_mcu`	SDA, SCB
`jsimd_ycc_extbgrx_convert_avx2`	SDA
`jsimd_idct_islow_avx2`	SDA, SCB

Table 6.9: Leaks of side channels collected via `Prime+Probe`.

	RSA D	RSA D	RSA D	RSA D
#Repeating	1	2	4	8
Leakage	19.3 (1.8%)	29.4 (2.8%)	34.9 (3.4%)	35.5 (3.4%)
	RSA D	RSA I	Libjpeg D	Libjpeg I
#Repeating	16	8	8	8
Leakage	35.6 (3.4%)	21.6 (2.1%)	20.8%	12.9%

1. "D" denotes L1 D cache whereas "I" denotes L1 I cache.
2. We also report leakage ratios for RSA cases to ease comparison.

Localization. Representative vulnerable functions of Libjpeg are given in Table 6.8. The leaked bits are spread across hundreds of program points, mostly from the IDCT, YCC encoding, and MCU modules. Libjpeg converts JPEG images into bitmaps, whose procedure has many SDA and SCB. We manually checked our findings, which are *aligned* with [278]. Nevertheless, the YCC encoding-related functions are newly found by us. CACHEQL also shows that SDA in IDCT leaks the most bits, whereas the MCU modules leak more bits via SCB. [278] flags those issues without quantification.

`Prime+Probe`

Following previous setups [278], a common `Prime+Probe` is launched on the same core with the victim software and periodically probes L1 cache when the victim software is executed. This mimics a practical and powerful attacker and is also the default setup of the `Prime+Probe` toolkit [269] leveraged in our evaluation and relevant research in this field. To prepare traces about Pre-processing, we halt the victim program after the pre-processing stage. Generally, launching `Prime+Probe` is costly. Without loss of generality, we use `Prime+Probe` to collect RSA side channels from OpenSSL 0.9.7 and Libjpeg. Attackers often repeat launching `Prime+Probe` [284]. As in Table 6.9, repeating an attack to collect more logs does improve information, but only marginal. Compared to merely doing `Prime+Probe` once, repeating $\times 4$ yields more information. However, repeating more times does not necessarily improve, since the information source is always the same. We

also note that Pre-processing has more leaks: for two (RSA, #Repeating=8) cases of L1 D and I cache (i.e., the 2nd and 3rd columns in the last row), Pre-processing has 22.5 (total 35.5) and 14.1 (total 21.6) leaked bits.

Libjpeg leaks more information than RSA. Not like recovering keys where each key bit needs to be analyzed, recovering every pixel is not necessary for inferring images. As previously stated [278], pixels conform to specific constraints to form meaningful contents (e.g., a human face), which typically have lower dimensions than pixel values. As a result, extracting these constraints can give rich information already.

Answer to RQ4: CACHEQL is highly generic: extending CACHEQL for other software and forms of side channels is straightforward and has no technical hurdles.

6.9 Discussion

Handling Real-World Attack Logs. Side channel observations (e.g., obtained in cross-virtual machine attacks [284]), are typically noisy. CACHEQL handles real attack logs by considering noise as non-determinism (see Sec. 6.4.4), thus quantifying leaked bits in those logs. Nevertheless, we do not recommend localizing vulnerabilities using real attack logs, since mapping these records back to program statements are challenging. Pin is sufficient for developers to "debug".

Analyzing Media Data. CACHEQL can smoothly quantify and localize information leaks for media software. Unlike previous static-/trace-based tools, which require re-implementing the pipeline to model floating-point instructions for symbolic execution or abstract interpretation, CACHEQL only needs the compressor $\mathcal{R}$ to be changed. In addition, CACHEQL is based on NN, which facilitates extracting "contents" of media data to quantify leaks, rather than simply comparing data byte differences.

Program Taking Public Inputs. We deem that different public inputs should not largely influence our analysis over cryptographic and media libraries, whose reasons are two-fold. First, for cryptosystems like OpenSSL, the public inputs (i.e., plaintext or ciphertext) has a relatively minor impact on the program execution flow. To our observation, public input values only influence a few loop iterations and `if` conditions. Media li-

braries mainly process private user inputs, which has no "public inputs". In practice, the influences of public inputs (including other non-secret local variables) are treated as non-determinism by CACHEQL. That is, they are handled consistently as how CACHEQL handles cryptographic blinding (Sec. 6.4.4), because neither is related to secret.

Given that said, configurations (e.g., cryptographic algorithm or image compression mode) may notably change the execution and the logged execution traces. We view that as an orthogonal factor. Moreover, modes of cryptographic algorithms and media processing procedures are limited. Users of CACHEQL are suggested to fix the mode before launching an analysis with CACHEQL, then use another mode, and so on.

Keystroke Templating Attacks. Quantifying and localizing the information leaks that enable keystroke templating attacks should be feasible to explore. With side channel traces logged by Intel Pin, machine learning is used to predict the user's key press [241]. Given sufficient data logged in this scenario, CACHEQL can be directly applied to quantify the leaked information and localize leakage sites.

Large Software Monoliths. For analyzing complex software like browsers and office products, our experience is that using Intel Pin to perform dynamic instrumentation for production browsers is difficult. With this regard, we anticipate adopting other dynamic instrumentors, if possible, to enable localizing leaks in these software. With correctly logged execution trace, CACHEQL can quantify the leaked bits and attribute the bits to side channel records.

Training Dataset Generalization. One may question to what degree traces obtained from one program can be used as training set for detecting leaks in another. In our current setup, we do not advocate such a "transfer" setting. Holistically, CACHEQL learns to compute PD by accurately distinguishing traces produced when the software is processing different secret inputs. By learning such distinguishability, CACHEQL eliminates the need for users to label the leakage bits of each training data sample. Nevertheless, knowledge learned for distinguishability may differ between programs. It is intriguing to explore training a "general" model that can quantify different side channel logs, particularly when collecting traces for the target program is costly. To do so, we expect to incorporate advanced training techniques (such as transfer learning [197]) into our pipeline.

Key Generation. In RSA evaluations, we feed cryptographic libraries with the key in a file. That is, the cryptographic library execution does not involve "key generation", which is *not* due to "limited coverage" of CACHEQL. Previous works [181] have exploited RSA key generation. With manual effort, we find that the key generation functions heavily use BIGNUM, involving vulnerable BIGNUM initialization and computation functions already localized by CACHEQL in Sec. 6.8.3, e.g., `BN_bin2bn` in Fig. 6.8 and `BN_sub` (see our full report [6]).

BIGNUM Implementation. We also investigate other cryptosystems. LibreSSL and BoringSSL are built on OpenSSL. Their BIGNUM implementations and OpenSSL share similar vulnerable coding patterns (i.e., the leading zero leak patterns; see ⓒ of Sec. 6.8.3). We also find similar BIGNUM vulnerable patterns in Botan (see [1]). In contrast, we find Intel IPP does not use an individual variable to record #leading zeros in BIGNUM (see [2]), hence it is likely free of ⓒ.

Extension for Other Side Channels. While this chapter focuses on cache side channels to present aligned comparisons with previous works, extending CACHEQL for other side channels should incur no technical challenges: users only need to feed the collected side channels and corresponding secrets to CACHEQL without any modification to CACHEQL. As shown in Sec. 6.5, CACHEQL's design is agnostic to side channel types and how side channel is collected. In Sec. 6.8, we also evaluate CACHEQL with various forms of cache side channels, including cache line and cache bank accesses collected in an in-house setting via Intel Pin, and noisy cache set observations recorded using Prime+Probe; all the experiments directly feed the side channels into CACHEQL without additional effort. Overall, we foresee the high potential of CACHEQL's applications to other side channels.

6.10 Conclusion

We present CACHEQL to quantify cache side channel leakages via MI. We also formulate secret leak as a cooperative game and enable localization via Shapley value. Our evaluation shows that CACHEQL overcomes typical hurdles (e.g., scalability, accuracy) of prior works, and computes information leaks in real-world cryptographic and media software.

6.11 Appendix for Chap. 6

6.11.1 Proof for Correctness of CACHEQL's quantification

This Appendix section proves Eq. 6.10 in a two-step approach; we refer the following proof skeleton to [30, 238]. Since $c(k,o)$ is only decided by $\mathcal{F}_\theta$, for simplicity, we also denote it as a parameterized function. In particular, Lemma 1 first shows that the empirically measured $\hat{I}^{(n)}_{\theta^\dagger}(K;O)$ is consistent with $\hat{I}_{\theta^\dagger}(K;O)$. Lemma 2 then proves that $\hat{I}_{\theta^\dagger}(K;O)$ stays close to $I(K;O)$. In all, Tsai et al. [238] has pointed out that the following two prepositions hold for neural networks:

Proposition 1 (Boundness). *$\forall \hat{c}_\theta$ and $\forall k, o$, there exist two constant bounds B_l and B_u such that $B_l \leq \log \hat{c}_\theta(k,o) \leq B_u$.*

Proposition 2 (log-smoothness). *$\forall k, o$ and $\forall \theta_1, \theta_2 \in \Theta$, $\exists \alpha > 0$ such that $|\log \hat{c}_{\theta_1}(k,o) - \log \hat{c}_{\theta_2}(k,o)| \leq \alpha \|\theta_1 - \theta_2\|$.*

Proposition 1 states that outputs of a neural network are bounded and Proposition 2 clarifies that the output of a neural network will not change too much if the parameter θ change slightly [238]. By incorporating the bounded rate of uniform convergence on parameterized functions [27], we have:

Lemma 1 (Estimation). *$\forall \epsilon > 0$,*

$$\Pr_{\{\langle k,o\rangle\}^{(n)}} \left(\sup_{\hat{c}_{\theta^\dagger} \in \mathcal{C}} \left| \hat{I}^{(n)}_{\theta^\dagger}(K;O) - \mathbb{E}_{P_{K\times O}}[\log \hat{c}_{\theta^\dagger}(k,o)] \right| \geq \epsilon \right)$$
$$\leq 2|\Theta| \exp\left(\frac{-n\epsilon^2}{2(B_u - B_l)^2} \right).$$

Lemma 1 applies the classical consistency theorem [82] for extremum estimators. Here, extremum estimators denote parametric functions optimized via maximizing or minimizing certain objectives; note that $\hat{c}_\theta$ is optimized to maximize the binary cross-entropy in Eq. 6.6. It illustrates that $\hat{I}^{(n)}_{\theta^\dagger}(K;O)$ convergent to $\hat{I}_{\theta^\dagger}(K;O)$ as n grows. Future, based the universal approximation theory of neural networks [106], we have:

Lemma 2 (Approximation). *$\forall \epsilon > 0$, there exists a family of neural networks $N = \{\hat{c}_\theta : \theta \in \Theta\}$ such that*

$$\inf_{\hat{c}_\theta \in N} |\mathbb{E}_{P_{K\times O}}[\log \hat{c}_\theta(k,o)] - I(K,O)| \leq \epsilon.$$

Lemma 2 states that $\hat{I}_{\theta^\dagger}(K;O)$ can approximate $I(K;O)$ with arbitrary accuracy. Therefore, Eq. 6.10 can be derived from Lemma 1 and Lemma 2 based on the triangular inequality.

6.11.2 Key Properties of Shapley Values

Following Sec. 6.6, this Appendix shows several key properties of Shapley value. We discuss why it is suitable for side channel analysis and how the properties help our localization.

Theorem 1 (Efficiency [220]). *The sum of Shapley value of all participants (i.e., side channel records) equals to the value of the grand coalition:*

$$\sum_{i \in R^o} \pi_i(\phi) = \phi(o) - \phi(o_\emptyset). \tag{6.13}$$

Theorem 1 states that the assigned Shapley value for each side channel record satisfies the apportionment defined in Definition 3. $\phi(o_\emptyset) = 0$ since an empty o leaks no secret.

Theorem 2 (Symmetry [220]). *If $\forall S \subseteq R^o \setminus \{i,j\}$, i- and j-th players are equivalent, i.e., $\phi(o_{S\cup\{i\}}) = \phi(o_{S\cup\{j\}})$, then $\pi_i(\phi) = \pi_j(\phi)$.*

Theorem 2 states records in o contributing equally to leakage have the same Shapley value. Divergent Shapley values suggest divergent leakage on the relevant program points. It ensures that all contributions are awarded Shapley values.

Theorem 3 (Dummy Player [220]). *If the i-th participant is a dummy player, i.e., $\forall S \subseteq R^o \setminus \{i\}, \phi(o_{S\cup\{i\}}) - \phi(o_S) = \phi(o_{\{i\}}) - \phi(o_\emptyset)$, then $\pi_i(\phi) = \phi(o_{\{i\}}) - \phi(o_\emptyset)$.*

Theorem 3, dubbed as "Dummy Player," states that the information leakage in one program point is not distributed to non-correlated points. In sum, Theorem 2 and Theorem 3 guarantee that the Shapley value apportionment is **accurate**. Further, if $\phi(o_{\{i\}}) = \phi(o_\emptyset)$, we have the following theorem.

Theorem 3.1 (Null player [220]). *If the i-th participant has no contribution to any grand coalition game ϕ, i.e., $\forall S \subseteq R^o \setminus \{i\}, \phi(o_{S\cup\{i\}}) = \phi(o_S)$, then $\pi_i(\phi) = 0$.*

Theorem 3.1 is one special case of Theorem 3, and it guarantees that the Shapley value has **no false negative**. That is, program points assigned with a zero Shapley value is guaranteed to not contribute to information leakage.

Theorem 4 (Linearity [220]). *If two coalition games, namely ϕ and ψ, are combined, then $\pi_i(\phi + \psi) = \pi_i(\phi) + \pi_i(\psi)$ for $\forall i \in R^o$.*

Theorem 4 implies that if a secret has several independent components, then the assigned Shapley value for each side channel record equals to the linear sum of secrets leaked on this record from all components. For instance, let $\pi(\phi_g)$ and $\pi(\phi_a)$ be the leaked information by recovering two privacy-related properties, "gender" and "age," over portrait photos. Since these two properties are independent, when considering both, the newly-computed leakage $\pi(\phi)$ must equal the sum of $\pi(\phi_g)$ and $\pi(\phi_a)$ according to Theorem 4. While this property allows for fine-grained leakage analysis, we currently do not separate a secret into independent components. We view this exploration as one future work.

Theorem 5 (Uniqueness [220]). *The apportionment derived from Shapley value is the only one that simultaneously satisfies Theorem 1, Theorem 2, Theorem 3 & 3.1, and Theorem 4.*

CHAPTER 7

CONCLUSION

With years of growing development, AI systems have gained widespread adoption across numerous security- and privacy-critical applications. Therefore, this thesis comprehensively studies the threat of different side channel leakages in infrastructures of modern AI systems. It identifies cache side channel leakage in data processing libraries, where a non-privileged malicious user can steal other user's inputs to the AI system. This thesis also finds that Trusted Execution Environments (TEEs) have severe secret leakage when protecting neural networks: by exploiting TEE's ciphertext side channels, user's inputs and the neural network's weight can be recovered by a malicious host (e.g., either the AI service provider or the host machine). Besides these attacks, this thesis also presents techniques for localizing sources of different side channel leakages for defensive purposes. Overall, the following works are presented in this thesis:

Our first work exploits cache side-channel leakage in data processing libraries of AI systems. Modern AI systems accept high-dimensional media data (e.g., images, audios) as inputs and adopt data processing libraries (e.g., Libjpeg, FFmpeg) to handle different input formats; recovering AI inputs is inherently challenging given the high dimension and complex format of such data. Unlike prior works that operate on data bytes (e.g., pixel values of an image), we focus on semantic information in AI inputs (e.g., what constitutes a face in an image) and significantly reduce the complexity of SCA. Our pipeline is unified for different formats of AI inputs and is fully automated. We also propose low-cost yet effective mitigation for the leakage.

Our second work further investigates data leakage induced by computations of neural networks and examines Trusted Execution Environments (TEEs). Despite that TEEs are widely employed to ensure secure computations on neural networks from untrusted providers, we show that this security belief is violated due to the recently disclosed ciphertext side channels in TEEs: a malicious neural network provider can precisely recover

user's inputs from the encrypted ciphertext in TEEs. We systematically examine the leakage in deep learning runtime interpreters including TensorFlow and PyTorch, and study how their different computation paradigms affect the leakage. Our results also show that optimizations in neural network compilers (e.g., TVM, Glow) can enlarge the leakage in neural network executables.

Our third work studies how neural networks can be extracted under TEE protection when deployed on an untrusted host machine. By exploiting ciphertext side channels in TEEs, this work for the first time demonstrates the feasibility and practicality of recovering neural networks. We implement a highly stealthy tool, HYPERTHEFT, that can recover weights by observing the TEE-shielded neural network's one execution without querying it. The recovered neural network weights constantly achieve 77%∼97% under different attack scenarios. Based on HYPERTHEFT's results, we further show how different downstream attacks can be enabled to leak training data and manipulate the neural network's outputs.

Moving to the defensive side, our last work addresses the quantification and localization problems, two fundamental challenges in detecting side channel leakage. We first review the (in-)adequacy of existing side channel detection methods and propose eight criteria for designing a full-fledged cache side channel detector. Accordingly, we propose CACHEQL that meets all these criteria. CACHEQL quantifies the leakage as the mutual information between the secret and its resulting side channel observation. To localize the leakage source, CACHEQL distributes the leaked information among vulnerable program modules via game theory. With meticulous optimizations, CACHEQL is highly scalable and applies to production software; it also largely improves the quantification precision and reduces the cost of localization from exponential to almost constant.

PUBLICATIONS

4. **Yuanyuan Yuan**, Zhibo Liu, Sen Deng, Yanzuo Chen, Shuai Wang, Yinqian Zhang, and Zhendong Su.

   Trusted Execution Brings False Trust: Disclosing User Data Leakage in TEE-Shielded Neural Networks.

   In 46th IEEE Symposium on Security and Privacy (**IEEE S&P**), 2025.

3. **Yuanyuan Yuan**, Zhibo Liu, Sen Deng, Yanzuo Chen, Shuai Wang, Yinqian Zhang, and Zhendong Su.

   HYPERTHEFT: Thieving Model Weights from TEE-Shielded Neural Networks via Ciphertext Side Channels.

   In 31st ACM Conference on Computer and Communications Security (**CCS**), 2024.

2. **Yuanyuan Yuan**, Zhibo Liu, and Shuai Wang.

   CACHEQL: Quantifying and Localizing Cache Side-Channel Vulnerabilities in Production Software.

   In 32nd USENIX Security Symposium (**USENIX Security**), 2023.

1. **Yuanyuan Yuan**, Qi Pang, and Shuai Wang.

   Automated Side Channel Analysis of Media Software with Manifold Learning.

   In 31st USENIX Security Symposium (**USENIX Security**), 2022.

REFERENCES

[1] Bignum implementation in botan. https://github.com/randombit/botan/blob/3.0.0-alpha1/src/lib/math/bigint/bigint.cpp#L210.

[2] Bignum implementation in intel ipp. https://www.intel.com/content/www/us/en/develop/documentation/ipp-crypto-reference/top/appendix-a-support-functions-and-classes/classes-and-functions-used-in-examples/bignumber-class.html.

[3] FFMPEG. https://ffmpeg.org/.

[4] hunspell. http://hunspell.github.io/.

[5] Openssl security policy. https://www.openssl.org/policies/secpolicy.html.

[6] Research Artifact of CacheQL. https://github.com/Yuanyuan-Yuan/CacheQL.

[7] Research Artifact of CipherSteal. https://sites.google.com/view/cipher-steal.

[8] Research Artifact of HyperTheft. https://sites.google.com/view/hyper-theft.

[9] Research Artifact of Manifold-Based SCA. https://github.com/Yuanyuan-Yuan/Manifold-SCA.

[10] Test cases from binsec/rel. https://github.com/binsec/rel_bench/tree/82b9ad267a8a86c2cfd8560755aa0d9a0ef5a627/src/ct/openssl_utility.

[11] Face attributes analysis service. https://www.faceplusplus.com/attributes/, 2020.

[12] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael 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.

[13] Hervé Abdi and Lynne J Williams. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics*, 2(4):433–459, 2010.

[14] Johan Agat. Transforming out timing leaks. POPL, 2000.

[15] Nadhem J Al Fardan and Kenneth G Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. IEEE Security & Privacy, 2013.

[16] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Sec.*, 2016.

[17] Shun-ichi Amari. Backpropagation and stochastic gradient descent method. *Neurocomputing*, 1993.

[18] Amazon. Amazon SageMaker Neo uses Apache TVM for performance improvement on hardware target. https://aws.amazon.com/sagemaker/neo/, 2021.

[19] Maksym Andriushchenko, Francesco Croce, Nicolas Flammarion, and Matthias Hein. Square attack: a query-efficient black-box adversarial attack via random search. In *European conference on computer vision*, pages 484–501. Springer, 2020.

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

[21] Saeid Asgari Taghanaki, Kumar Abhishek, Joseph Paul Cohen, Julien Cohen-Adad, and Ghassan Hamarneh. Deep semantic segmentation of natural and medical images: a review. *Artificial Intelligence Review*, 2021.

[22] Aslan Askarov, Danfeng Zhang, and Andrew C Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 297–307, 2010.

[23] Anish Athalye, Nicholas Carlini, and David 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.

[24] Mukund Balasubramanian, Eric L Schwartz, Joshua B Tenenbaum, Vin de Silva, and John C Langford. The ISOMAP algorithm and topological stability. *Science*, 295(5552):7–7, 2002.

[25] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S Păsăreanu, and Tevfik Bultan. String analysis for side channels with segmented oracles. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 193–204, 2016.

[26] Qinkun Bao, Zihao Wang, Xiaoting Li, James R Larus, and Dinghao Wu. Abacus: Precise side-channel analysis. ICSE, 2021.

[27] Peter L Bartlett. The sample complexity of pattern classification with neural networks: the size of the weights is more important than the size of the network. *IEEE transactions on Information Theory*, 44(2), 1998.

[28] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel. In *USENIX Security*, 2019.

[29] David Bau, Jun-Yan Zhu, Hendrik Strobelt, Agata Lapedriza, Bolei Zhou, and Antonio Torralba. Understanding the role of individual units in a deep neural network. *Proceedings of the National Academy of Sciences*, 117(48):30071–30078, 2020.

[30] Mohamed Ishmael Belghazi, Aristide Baratin, Sai Rajeshwar, Sherjil Ozair, Yoshua Bengio, Aaron Courville, and Devon Hjelm. Mutual information neural estimation. ICML, 2018.

[31] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–5555. Califor-nia, USA, 2005.

[32] Richard Bellman. Dynamic programming. *Science*, 153(3731), 1966.

[33] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *TPAMI*, 2013.

[34] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT' 17)*, 2017.

[35] Benjamin A Braun, Suman Jana, and Dan Boneh. Robust and efficient elimination of cache and timing side channels. *arXiv preprint arXiv:1506.00189*, 2015.

[36] Jakub Breier, Dirmanto Jap, Xiaolu Hou, Shivam Bhasin, and Yang Liu. Sniff: reverse engineering of neural networks with fault attacks. *IEEE Transactions on Reliability*, 71(4):1527–1539, 2021.

[37] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. Casym: Cache aware symbolic execution for side channel detection and mitigation. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 505–521. IEEE, 2019.

[38] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[39] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. Convolutional neural networks with data augmentation against jitter-based countermeasures. In *CHES*, 2017.

[40] Nicholas Carlini, Steve Chien, Milad Nasr, Shuang Song, Andreas Terzis, and Florian Tramer. Membership inference attacks from first principles. In *IEEE S&P*, 2022.

[41] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. The secret sharer: Evaluating and testing unintended memorization in neural networks. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 267–284, 2019.

[42] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *IEEE S&P*, 2017.

[43] Pierre Carru. Attack arm trustzone using rowhammer. In *GreHack*, 2017.

[44] Javier Castro, Daniel Gómez, Elisenda Molina, and Juan Tejada. Improving polynomial estimation of the shapley value by stratified random sampling with optimum allocation. *Computers & Operations Research*, 82:180–188, 2017.

[45] Javier Castro, Daniel Gómez, and Juan Tejada. Polynomial calculation of the shapley value based on sampling. *Computers & Operations Research*, 2009.

[46] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *IEEE S&P*, 2020.

[47] Varun Chandrasekaran, Kamalika Chaudhuri, Irene Giacomelli, Somesh Jha, and Songbai Yan. Exploring connections between active learning and model extraction. In *USENIX Security*, 2020.

[48] Ya Chang, Changbo Hu, and Matthew Turk. Manifold of facial expression. In *AMFG*, pages 28–35, 2003.

[49] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. Quantifying information leakage in cache attacks via symbolic execution. *TECS*, 2019.

[50] Vinod Kumar Chauhan, Jiandong Zhou, Ping Lu, Soheila Molaei, and David A Clifton. A brief review of hypernetworks in deep learning. *arXiv preprint arXiv:2306.06955*, 2023.

[51] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX OSDI*, pages 578–594, 2018.

[52] Yu Chen, Fang Luo, Tong Li, Tao Xiang, Zheli Liu, and Jin Li. A training-integrity privacy-preserving federated learning scheme with trusted execution environment. *Information Sciences*, 522:69–79, 2020.

[53] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel tdx demystified: A top-down approach. *preprint arXiv:2303.15540*, 2023.

[54] Mikhail Chernov and Eric Ghysels. A study towards a unified approach to the joint estimation of objective and risk neutral measures for the purpose of options valuation. *Journal of financial economics*, 56(3):407–458, 2000.

[55] R. C. Chiang, S. Rajasekaran, N. Zhang, and H. H. Huang. Swiper: Exploiting virtual machine vulnerability in third-party clouds with competition for i/o resources. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1732–1742, June 2015.

[56] Duc-Hiep Chu, Joxan Jaffar, and Rasool Maghareh. Precise cache timing analysis via symbolic execution. RTAS, pages 1–12, 2016.

[57] TVM Community. Tvm deep learning compiler joins apache software foundation. https://tvm.apache.org/2019/03/18/tvm-apache-announcement, 2019.

[58] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *IEEE SP*, 2009.

[59] Jean-Sébastien Coron and Ilya Kizhvatov. An efficient method for random delay generation in embedded software. In *CHES*, 2009.

[60] Jose A Costa and Alfred O Hero. Geodesic entropic graphs for dimension and entropy estimation in manifold learning. *IEEE Transactions on Signal Processing*, 52(8):2210–2221, 2004.

[61] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *NIPS*, 2015.

[62] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[63] Patrick Cousot. Abstract interpretation. *ACM Computing Surveys (CSUR)*, 28(2):324–328, 1996.

[64] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of fixpoints. POPL, 1977.

[65] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting cache side-channel attacks through dynamic software diversity. NDSS, 2015.

[66] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level. IEEE S&P, 2020.

[67] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[68] Li 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.

[69] Sen Deng, Mengyuan Li, Yining Tang, Shuai Wang, Shoumeng Yan, and Yinqian Zhang. CipherH: Automated detection of ciphertext side-channel vulnerabilities in cryptographic implementations. In *USENIX Security*, 2023.

[70] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A timer-free high-precision L3 cache attack using Intel TSX. In *USENIX Sec.*, 2017.

[71] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L Cox, and Sandhya Dwarkadas. Shielding software from privileged side-channel attacks. In *27th USENIX Security Symposium*, pages 1441–1458, 2018.

[72] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2020.

[73] Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A tool for the static analysis of cache side channels. In *USENIX Sec.*, 2013.

[74] Goran Doychev and Boris Köpf. Rigorous analysis of software countermeasures against cache attacks. PLDI, 2017.

[75] Anuj Dubey, Rosario Cammarota, and Aydin Aysu. Maskednet: The first hardware inference engine aiming power side-channel protection. In *IEEE HOST*, 2020.

[76] Xing Fan, Wei Jiang, Hao Luo, and Mengjuan Fei. Spherereid: Deep hypersphere manifold embedding for person re-identification. *Journal of Visual Communication and Image Representation*, 60:51–58, 2019.

[77] Charles Fefferman, Sanjoy Mitter, and Hariharan Narayanan. Testing the manifold hypothesis. *Journal of the American Mathematical Society*, 29(4):983–1049, 2016.

[78] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *ICLR*, 2018.

[79] Zhenyong Fu, Tao Xiang, Elyor Kodirov, and Shaogang Gong. Zero-shot object recognition by semantic manifold distance. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2635–2644, 2015.

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

[81] Yansong Gao, Huming Qiu, Zhi Zhang, Binghui Wang, Hua Ma, Alsharif Abuadbba, Minhui Xue, Anmin Fu, and Surya Nepal. Deeptheft: Stealing dnn model architectures through power side channel. In *IEEE S&P*, 2024.

[82] Sara A Geer, Sara van de Geer, and D Williams. *Empirical Processes in M-estimation*. 2000.

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

[84] Robert Geirhos, Patricia Rubisch, Claudio Michaelis, Matthias Bethge, Felix A Wichmann, and Wieland Brendel. Imagenet-trained cnns are biased towards texture; increasing shape bias improves accuracy and robustness. In *International Conference on Learning Representations*, 2018.

[85] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual information analysis, a generic side-channel distinguisher. In *CHES'08*, 2008.

[86] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. STOC, 1987.

[87] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *JACM*, 1996.

[88] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[89] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

[90] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

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

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

[93] Gaël Guennebaud, Benoit Jacob, et al. Eigen. *URl: http://eigen. tuxfamily. org*, 3(1), 2010.

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

[95] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. USENIX ATC, 2017.

[96] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[97] Xiaofei He, Shuicheng Yan, Yuxiao Hu, Partha Niyogi, and Hong-Jiang Zhang. Face recognition using laplacianfaces. *IEEE transactions on pattern analysis and machine intelligence*, 27(3):328–340, 2005.

[98] Benjamin Hettwer, Stefan Gehrer, and Tim Güneysu. Profiled power analysis attacks using convolutional neural networks with domain knowledge. SAC, 2018.

[99] Benjamin Hettwer, Tobias Horn, Stefan Gehrer, and Tim Güneysu. Encoding power traces as images for efficient side-channel analysis. *arXiv preprint arXiv:2004.11015*, 2020.

[100] Annelie Heuser and Michael Zohner. Intelligent machine homicide. In *COSADE*, 2012.

[101] Geoffrey E Hinton and Richard S Zemel. Autoencoders, minimum description length, and helmholtz free energy. *NIPS*, 1994.

[102] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 1997.

[103] Daniel Holden, Jun Saito, Taku Komura, and Thomas Joyce. Learning motion manifolds with convolutional autoencoders. In *SIGGRAPH Asia 2015 Technical Briefs*, pages 1–4. 2015.

[104] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael 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.

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

[106] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[107] Gabriel Hospodar, Benedikt Gierlichs, Elke De Mulder, Ingrid Verbauwhede, and Joos Vandewalle. Machine learning in side-channel analysis: a first study. *Journal of Cryptographic Engineering*, 2011.

[108] Jiahui Hou, Huiqi Liu, Yunxin Liu, Yu Wang, Peng-Jun Wan, and Xiang-Yang 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. doi:10.1109/TDSC.2021.3126315.

[109] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[110] Bin Hu, Yan Wang, Jerry Cheng, Tianming Zhao, Yucheng Xie, Xiaonan Guo, and Yingying 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.

[111] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of computer security*, 1(3-4):233–254, 1992.

[112] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, et al. Deepsniffer: A dnn model extraction framework based on learning architectural hints. In *ASPLOS*, pages 385–399, 2020.

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

[114] 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.

[115] Gorka Irazoqui, Kai Cong, Xiaofei Guo, Hareesh Khattri, Arun K. Kanuparthi, Thomas Eisenbarth, and Berk Sunar. Did we learn from LLC side channel attacks? A cache leakage detection tool for crypto libraries. *CoRR*, 2017.

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

[117] Animesh Jain, Shoubhik Bhattacharya, Masahiro Masuda, Vin Sharma, and Yida Wang. Efficient execution of quantized deep learning models: A compiler approach. *arXiv preprint arXiv:2006.10226*, 2020.

[118] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. "they're not that hard to mitigate": What cryptographic library developers think about timing attacks. IEEE S&P, 2021.

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

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

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

[122] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, page 13, 2016.

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

[124] Jaehun Kim, Stjepan Picek, Annelie Heuser, Shivam Bhasin, and Alan Hanjalic. Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis. *TCHES*, 2019.

[125] Kyungtae Kim, Chung Hwan Kim, Junghwan John Rhee, Xiao Yu, Haifeng Chen, Dave (Jing) Tian, and Byoungyoung Lee. Vessels: efficient and scalable deep learning prediction on trusted processors. In Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi, editors, *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, pages 462–476. ACM, 2020. doi:10.1145/3419111.3421282.

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

[127] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[128] Diederik Kingma, Tim Salimans, Ben Poole, and Jonathan Ho. Variational diffusion models. *Advances in neural information processing systems*, 34:21696–21707, 2021.

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

[130] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. CRYPTO, 1996.

[131] Boris Köpf and Heiko Mantel. Transformational typing and unification for automatically correcting insecure programs. *International Journal of Information Security*, 6(2-3):107–131, 2007.

[132] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In *CAV*, 2012.

[133] Boris Köpf and Geoffrey Smith. Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks. In *2010 23rd IEEE Computer Security Foundations Symposium*, pages 44–56. IEEE, 2010.

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

[135] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. NIPS, 2012.

[136] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[137] Donggeun Kwon, HeeSeok Kim, and Seokhie Hong. Improving non-profiled side-channel attacks using autoencoder based preprocessing. *IACR Cryptol.*, 2020.

[138] Adam Langley. ctgrind. `https://github.com/agl/ctgrind`.

[139] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[140] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

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

[142] John A Lee and Michel Verleysen. *Nonlinear dimensionality reduction*. Springer Science & Business Media, 2007.

[143] Taegyeong Lee, Zhiqi Lin, Saumay Pushp, Caihua Li, Yunxin Liu, Youngki Lee, Fengyuan Xu, Chenren Xu, Lintao Zhang, and Junehwa Song. Occlumency: Privacy-preserving remote deep-learning inference using SGX. In *MobiCom*, 2019.

[144] Bochen Li, Xinzhao Liu, Karthik Dinesh, Zhiyao Duan, and Gaurav Sharma. Creating a multitrack classical music performance dataset for multimodal music analysis: Challenges, insights, and applications. *IEEE Transactions on Multimedia*, 21(2):522–535, 2018.

[145] Haoqi Li, Brian Baucom, and Panayiotis Georgiou. Unsupervised latent behavior manifold learning from acoustic features: Audio2behavior. In *2017 IEEE interna-*

tional conference on acoustics, speech and signal processing (ICASSP), pages 5620–5624. IEEE, 2017.

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

[147] Mengyuan Li, Yinqian Zhang, and Zhiqiang 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.

[148] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting unprotected {I/O} operations in {AMD's} secure encrypted virtualization. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1257–1272, 2019.

[149] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. CIPHERLEAKS: Breaking constant-time cryptography on AMD SEV via the ciphertext side channel. In *USENIX Security*, 2021.

[150] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. TLB Poisoning Attacks on AMD Secure Encrypted Virtualization. In *Annual Computer Security Applications Conference*, 2021.

[151] Shaofeng Li, Xinyu Wang, Minhui Xue, Haojin Zhu, Zhi Zhang, Yansong Gao, Wen Wu, and Xuemin Sherman Shen. Yes, one-bit-flip matters! universal dnn model inference depletion with runtime code fault injection. In *USENIX Security*, 2024.

[152] Xianfeng Li, Tulika Mitra, and Abhik Roychoudhury. Accurate timing analysis by modeling caches, speculation and their interaction. DAC, 2003.

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

[154] Yan Li, Vivy Suhendra, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. IEEE RTSS, 2009.

[155] Yanran Li, Hui Su, Xiaoyu Shen, Wenjie Li, Ziqiang Cao, and Shuzi Niu. Dailydialog: A manually labelled multi-turn dialogue dataset. *arXiv preprint arXiv:1710.03957*, 2017.

[156] Yuepeng Li, Deze Zeng, Lin Gu, Quan Chen, Song Guo, Albert Y. Zomaya, and Minyi Guo. Lasagna: Accelerating secure deep learning inference in sgx-enabled edge cloud. In Carlo Curino, Georgia Koutrika, and Ravi Netravali, editors, *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, pages 533–545. ACM, 2021. doi:10.1145/3472883.3486988.

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

[158] libjpeg. Main libjpeg-turbo repository, 2020. URL: https://github.com/libjpeg-turbo/libjpeg-turbo.

[159] Tong Lin and Hongbin Zha. Riemannian manifold learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(5):796–809, 2008.

[160] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft COCO: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[161] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. *ACM SIGPLAN Notices*, 50(4):87–101, 2015.

[162] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, 2016.

[163] Fangfei Liu and Ruby B. Lee. Random fill cache architecture. In *MICRO*, 2014.

[164] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE, 2015.

[165] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing {CNN} model inference on cpus. In *USENIX ATC*, pages 1025–1040, 2019.

[166] Yugeng Liu, Rui Wen, Xinlei He, Ahmed Salem, Zhikun Zhang, Michael Backes, Emiliano De Cristofaro, Mario Fritz, and Yang Zhang. ML-Doctor: Holistic risk assessment of inference attacks against machine learning models. In *USENIX Security*, 2022.

[167] Yuntao Liu and Ankur 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*, 2020.

[168] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. ICCV, 2015.

[169] Jake Longo, Elke De Mulder, Dan Page, and Michael Tunstall. Soc it to em: electromagnetic side-channel attacks on a complex system-on-chip. In *Cryptographic Hardware and Embedded Systems–CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings 17*, pages 620–640. Springer, 2015.

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

[171] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *NIPS*, 2017.

[172] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX OSDI*, pages 881–897, 2020.

[173] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *ICLR*, 2018.

[174] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. Breaking cryptographic implementations using deep learning techniques. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 3–26. Springer, 2016.

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

[176] Zoltán Ádám Mann, Christian Weinert, Daphnee Chabal, and Joppe W Bos. Towards practical secure neural network inference: the journey so far and the road ahead. *ACM Computing Surveys*, 56(5):1–37, 2023.

[177] Francisco J Martinez-Murcia, Andres Ortiz, Juan-Manuel Gorriz, Javier Ramirez, and Diego Castillo-Barnes. Studying the manifold structure of alzheimer's disease: A deep learning approach using convolutional autoencoders. *J-BHI*, 24(1):17–26, 2019.

[178] Tulika Mitra, Jürgen Teich, and Lothar Thiele. Time-critical systems design: A survey. *IEEE Design & Test*, 35(2):8–26, 2018.

[179] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Soteris Demetriou, Ilias Leontiadis, Andrea Cavallaro, and Hamed Haddadi. Darknetz: towards model privacy at the edge using trusted execution environments. In Eyal de Lara, Iqbal Mohomed, Jason Nieh, and Elizabeth 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. doi:10.1145/3386901.3388946.

[180] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How sgx amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES' 17)*, 2017.

[181] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. {CopyCat}: Controlled {Instruction-Level} attacks on enclaves. USENIX Security, 2020.

[182] Shayan Moini, Shanquan Tian, Daniel Holcomb, Jakub Szefer, and Russell 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.

[183] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *ICISC*, 2005.

[184] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. Severed: Subverting amd's virtual machine encryption. In *Proceedings of the 11th European Workshop on Systems Security*, pages 1–6, 2018.

[185] Timothy Prickett Morgan. INSIDE FACEBOOK'S FUTURE RACK AND MICROSERVER IRON. https://www.nextplatform.com/2020/05/14/inside-facebooks-future-rack-and-microserver-iron/, 2020.

[186] Andrew C Myers. JFlow: Practical mostly-static information flow control. PLDI, 1999.

[187] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

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

[189] Nvidia. NVVM IR. https://docs.nvidia.com/cuda/nvvm-ir-spec/index.html, 2021.

[190] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical Side-Channel attacks. In *USENIX ATC*, 2018.

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

[192] Openssl. https://www.openssl.org/.

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

[194] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1406–1418, 2015.

[195] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, 2006.

[196] D. Page. Partitioned cache architecture as a side-channel defence mechanism, 2005.

[197] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.

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

[199] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

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

[201] Stjepan Picek, Ioannis Petros Samiotis, Jaehun Kim, Annelie Heuser, Shivam Bhasin, and Axel Legay. On the performance of convolutional neural networks for side-channel analysis. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 157–176. Springer, 2018.

[202] Albert Pumarola, Antonio Agudo, Aleix M Martinez, Alberto Sanfeliu, and Francesc Moreno-Noguer. Ganimation: Anatomically-aware facial animation from a single image. ECCV, 2018.

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

[204] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource management for isolation enhanced cloud services. In *CCSW*, 2009.

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

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

[207] Mark Randolph and William Diehl. Power side-channel attack analysis: A review of 20 years of study for the layman. *Cryptography*, 4(2):15, 2020.

[208] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 431–446, 2015.

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

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

[211] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[212] Yuki Saito, Shinnosuke Takamichi, and Hiroshi Saruwatari. Statistical parametric speech synthesis incorporating generative adversarial networks. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 26(1):84–96, 2017.

[213] Florian Schroff, Dmitry Kalenichenko, and James 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.

[214] Christian Schuldt, Ivan Laptev, and Barbara 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.

[215] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). IEEE S&P, 2010.

[216] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. KeyDrown: Eliminating software-based keystroke timing side-channel attacks. In *NDSS*, 2018.

[217] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using sgx to conceal cache attacks. *arXiv preprint arXiv:1702.08719*, 2017.

[218] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*, pages 618–626, 2017.

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

[220] Lloyd S Shapley. *A value for n-person games*. Princeton University Press, 2016.

[221] Dongdong She, Yizheng Chen, Abhishek Shah, Baishakhi Ray, and Suman Jana. Neutaint: Efficient dynamic taint analysis with neural networks. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1527–1543. IEEE, 2020.

[222] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: Efficient fuzzing with neural program smoothing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P '19)*, 2019.

[223] Tianxiang Shen, Ji Qi, Jianyu Jiang, Xian Wang, Siyuan Wen, Xusheng Chen, Shixiong Zhao, Sen Wang, Li Chen, Xiapu Luo, Fengwei Zhang, and Heming Cui. SOTER: guarding black-box inference for general neural networks at the edge. In Jiri Schindler and Noa Zilberman, editors, *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 723–738. USENIX Association, 2022. URL: https://www.usenix.org/conference/atc22/presentation/shen.

[224] Yujun Shen, Jinjin Gu, Xiaoou Tang, and Bolei Zhou. Interpreting the latent space of gans for semantic face editing. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9243–9252, 2020.

[225] Elaine Shi. Path oblivious heap: Optimal and practical oblivious priority queue. IEEE SP, 2020.

[226] Xingjian Shi, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-Kin Wong, and Wang-chun Woo. Convolutional lstm network: A machine learning approach for precipitation nowcasting. *Advances in neural information processing systems*, 28, 2015.

[227] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE, 2017.

[228] Laurent Simon, David Chisnall, and Ross Anderson. What you get is what you C: Controlling side effects in mainstream C compilers. IEEE EuroS&P, 2018.

[229] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious ram protocol. In *CCS*, 2013.

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

[231] Zhichuang Sun, Ruimin Sun, Long Lu, and Somesh Jha. Shadownet: A secure and efficient system for on-device model inference. *CoRR*, abs/2011.05905, 2020.

[232] Chungha Sung, Brandon Paulsen, and Chao Wang. CANAL: a cache timing analysis framework via LLVM transformation. ASE, 2018.

[233] Nicolas Thorstensen. *Manifold learning and applications to shape and image processing*. PhD thesis, Ecole des Ponts ParisTech, 2009.

[234] Kris Tiri and Ingrid Verbauwhede. Securing encryption algorithms against dpa at the logic level: Next generation smart card technology. In *CHES*, 2003.

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

[236] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *J. Cryptol.*, 2010.

[237] Eran Tromer, DagArne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[238] Y-H Tsai, H Zhao, M Yamada, L-P Morency, and R Salakhutdinov. Neural methods for point-wise dependency estimation. NeurIPS, 2020.

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

[240] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[241] Daimeng Wang, Ajaya Neupane, Zhiyun Qian, Nael B Abu-Ghazaleh, Srikanth V Krishnamurthy, Edward JM Colbert, and Paul Yu. Unveiling your keystrokes: A cache-based side-channel attack on graphics libraries. In *NDSS*, 2019.

[242] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, Yajuan Wang, Endong Wang, Qing Zhang, Bo Shen, et al. Intel math kernel library. *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures*, pages 167–188, 2014.

[243] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. Identifying cache-based side channels through secret-augmented abstract interpretation. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 657–674, 2019.

[244] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. CacheD: Identifying cache-based timing channels in production software. In *USENIX Security*, 2017.

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

[246] Xiaosong Wang, Yifan Peng, Le Lu, Zhiyong Lu, Mohammadhadi Bagheri, and Ronald 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.

[247] Zhenghong Wang and Ruby B Lee. A novel cache architecture with enhanced performance and security. In *MICRO*, 2008.

[248] Sally Ward-Foxton. Google and Nvidia Tie in MLPerf; Graphcore and Habana Debut. https://www.eetimes.com/google-and-nvidia-tie-in-mlperf-graphcore-and-habana-debut/#, 2021.

[249] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209*, 2018.

[250] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. Osiris: Automatic Discovery of Microarchitectural Side Channels. In *USENIX Security Symposium*, 2021.

[251] Lingxiao Wei, Bo Luo, Yu Li, Yannan Liu, and Qiang Xu. I know what you see: Power side-channel attack on convolutional neural network accelerators. In *Pro-*

ceedings of the 34th Annual Computer Security Applications Conference, pages 393–406, 2018.

[252] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. Big numbers-big troubles: Systematically analyzing nonce leakage in (ec) dsa implementations. USENIX Security, 2020.

[253] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. DATA – differential address trace analysis: Finding address-based side-channels in binaries. In *USENIX Sec.*, 2018.

[254] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian 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.

[255] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MicroWalk: A framework for finding side channels in binaries. In *ACSAC*, 2018.

[256] Jan Wichelmann, Anna Pätschke, Luca Wilke, and Thomas Eisenbarth. Cipherfix: Mitigating ciphertext side-channel attacks in software. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6789–6806, 2023.

[257] Jan Wichelmann, Anja Rabich, Anna Pätschke, and Thomas Eisenbarth. Obelix: Mitigating side-channels through dynamic obfuscation. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 189–189. IEEE Computer Society, 2024.

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

[259] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. Sev-step: A single-stepping framework for amd-sev, 2023. arXiv:2307.14757.

[260] Sanghyun Woo, Jongchan Park, Joon-Young Lee, and In So Kweon. Cbam: Convolutional block attention module. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19, 2018.

[261] Lichao Wu and Stjepan Picek. Remove some noise: On pre-processing of side-channel measurements with autoencoders. *TCHES*, 2020.

[262] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. USENIX Security, 2012.

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

[264] Chaowei Xiao, Bo Li, Jun-Yan Zhu, Warren He, Mingyan Liu, and Dawn Song. Generating adversarial examples with adversarial networks. *arXiv preprint arXiv:1801.02610*, 2018.

[265] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of 2015 IEEE Symposium on Security and Privacy (S&P' 15)*, pages 640–656. IEEE, 2015.

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

[267] Dingqing Yang, Prashant J Nair, and Mieszko Lis. Huffduff: Stealing pruned dnns from sparse accelerators. In *ASPLOS*, 2023.

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

[269] Yuval Yarom. Mastik: A micro-architectural side-channel toolkit. *Retrieved from Adelaide: http://cs.adelaide.edu.au/yval/Mastik*, 16, 2016.

[270] Yuval Yarom and Katrina Falkner. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *23rd USENIX security symposium (USENIX security 14)*, pages 719–732, 2014.

[271] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

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

[273] Honggang Yu, Kaichen Yang, Teng Zhang, Yun-Yun Tsai, Tsung-Yi Ho, and Yier Jin. Cloudleak: Large-scale deep learning models stealing through adversarial examples. In *NDSS*, 2020.

[274] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE transactions on neural networks and learning systems*, 30(9):2805–2824, 2019.

[275] Yuanyuan Yuan, Zhibo Liu, Sen Deng, Yanzuo Chen, Shuai Wang, Yinqian Zhang, and Zhendong Su. CipherSteal: Ciphersteal: Stealing input data from tee-shielded neural networks with ciphertext side channels. IEEE S&P, 2024.

[276] Yuanyuan Yuan, Zhibo Liu, Sen Deng, Yanzuo Chen, Shuai Wang, Yinqian Zhang, and Zhendong Su. HyperTheft: Thieving model weights from tee-shielded neural networks via ciphertext side channels. ACM CCS, 2024.

[277] Yuanyuan Yuan, Zhibo Liu, and Shuai Wang. Cacheql: Quantifying and localizing cache side-channel vulnerabilities in production software. USENIX Security, 2023.

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

[279] Yuanyuan Yuan, Shuai Wang, and Junping Zhang. Private image reconstruction from system side channels using generative models. In *International Conference on Learning Representations*, 2021. URL: https://openreview.net/forum?id=y06VOYLcQXa.

[280] Matthew D Zeiler and Rob 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.

[281] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 586–595, 2018.

[282] Xianyi Zhang, Martin Kroeker, Werner Saar, Qian Wang, and Zaheer. Chothia. Openblas. `http://www.openblas.net/`, 2023.

[283] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. HomeAlone: Co-residency detection in the cloud via side-channel analysis. In *IEEE SP*, 2011.

[284] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. CCS, 2012.

[285] Ziqi Zhang, Chen Gong, Yifeng Cai, Yuanyuan Yuan, Bingyan Liu, Ding Li, Yao Guo, and Xiangqun Chen. No privacy left outside: On the (in-) security of tee-shielded dnn partition for on-device ml. In *IEEE S&P*, 2024.

[286] Ziqi Zhang, Yuanchun Li, Yao Guo, Xiangqun Chen, and Yunxin 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.

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

[288] Hattie Zhou, Janice Lan, Rosanne Liu, and Jason Yosinski. Deconstructing lottery tickets: Zeros, signs, and the supermask. *NeuIPS*, 2019.

[289] Bo Zhu, Jeremiah Z Liu, Stephen F Cauley, Bruce R Rosen, and Matthew S Rosen. Image reconstruction by domain-transform manifold learning. *Nature*, 555(7697):487–492, 2018.

[290] Jun-Yan Zhu, Philipp Krähenbühl, Eli Shechtman, and Alexei A Efros. Generative visual manipulation on the natural image manifold. In *European conference on computer vision*, pages 597–613. Springer, 2016.

[291] Yuankun Zhu, Yueqiang Cheng, Husheng Zhou, and Yantao Lu. Hermes attack: Steal dnn models with lossless inference accuracy. In *USENIX Security*, 2021.