DeepCache: Revisiting Cache Side-Channel Attacks in Deep Neural Networks Executables

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

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

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

Sihang Hu
Huawei Technologies
Shenzhen, China
husihang@huawei.com

Tianxiang Li
Huawei Technologies
Shenzhen, China
litianxiang4@huawei.com

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

ABSTRACT

Deep neural networks (DNN) are increasingly deployed in heterogeneous hardware, including high-performance devices like GPUs and low-power devices like mobile/IoT CPUs, FPGAs, and accelerators. In order to unlock the full performance potential of various hardware, deep learning (DL) compilers automatically optimize DNN inference computations and compile DNN models into DNN executables for efficient computations across hardware backends.

As valuable intellectual properties, DNN architectures are one primary attack target. Since previous works already demonstrate the abuse of cache side channels to steal DNN architectures from DL frameworks (e.g., PyTorch and TensorFlow), we first study using those known side-channel attacks against DNN executables. We find that attacking DNN executables presents unique challenges, and existing works can hardly apply. Particularly, DNN executables exhibit a standalone paradigm that largely reduces cache side channel attack surfaces. Meanwhile, cache side channels capture only limited behaviors of the whole DNN execution while facing daunting technical challenges (e.g., noise and low time resolution).

However, we unveil a unique attack vector in DNN executables, such that the cache-aware optimizations, which are extensively employed by contemporary DL compilers to harvest the full potentials of hardware, would result in distinguishable DNN operator cache access patterns, making model architecture recovery possible. We propose DeepCache, an end-to-end side channel attack framework, to infer DNN model architectures from DNN executables. DeepCache leverages cache side channels as the attacking primitives and combines contrastive learning and anomaly detection to enable precise inference. Our evaluation using the standard Prime+Probe shows that DeepCache yields a high accuracy in exploiting complex DNN executables under both the basic L1 cache attack and the more practical but challenging last level cache (LLC) attack settings.

*Corresponding Author

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690241

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**;

KEYWORDS

Cache Side Channel; DNN Executable; DNN Stealing

ACM Reference Format:
Zhibo Liu, Yuanyuan Yuan, Yanzuo Chen, Sihang Hu, Tianxiang Li, and Shuai Wang. 2024. DeepCache: Revisiting Cache Side-Channel Attacks in Deep Neural Networks Executables . In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3658644.3690241

1 INTRODUCTION

Despite the prosperous development of deep learning and its extensive usage in diverse real-world applications, adopting and optimizing deep neural networks (DNNs) for various hardware backends takes time and requires expensive expertise. To ease the deployment of DNNs on heterogeneous devices, DL compilers are proposed to automatically compile and optimize DNN models into DNN executables running on multiple platforms [14, 49, 60]. Typically, a DL compiler takes high-level DNN model specifications as inputs and yields efficient executables optimized specifically for target hardware backends. During compilation, a series of hardware-aware optimizations and learning-assisted autotuning [14] are applied to harvest the full computation power of the underlying hardware.

To date, we have seen DL compilers being adopted by major cloud service providers like Google and Amazon [4, 21]. Remarkably, Amazon spends considerable efforts to contribute the compiler code to TVM, one of the most popular DL compiler projects supported by Apache [6], and has already deployed TVM to accelerate its Machine Learning as a Service (MLaaS) [32, 44]. As more and more cloud service providers (e.g., Meta, AWS, and Google) [5, 15, 24] are providing DNN inference services in a resource-sharing environment for cost and profit reasons, DNN executables are expected to be increasingly applied to resource-sharing clouds to boost DNN inference services. However, with that comes an emerging research question that haunts the research community:

Is there a risk of valuable properties being leaked via remote side channel(s) when using DNN executables?

Recent years have seen continuous works proposed to attack DNNs with different side channels, including cache side channels [27, 28, 43, 79], electromagnetic (EM) emanation side channels [43, 85], power side channels [18, 20], bus snooping [30, 80, 92], and rowhammer-based leakage [58]. These attacks steal valuable intellectual properties (IP) like DNN model architectures and pre-trained weights. While they hold different threat models and assume distinct attack scenarios, all previous works mainly focus on DNNs deployed with popular DL frameworks like PyTorch and TensorFlow. Meanwhile, although DL compilers are developing rapidly and receiving increasing attention from industry and academia, analyses and interpretations of side channels in their output (i.e., DNN executables) have long received little attention.

This paper aims to fill the research blank in side channel security regarding DNN executables. Specifically, following the research of prior works [11, 20, 30, 79, 85, 92], we explore the feasibility of remotely stealing DNN model architectures via side channels. First, given a standard, resource-sharing cloud environment, we revisit existing side channel attacks in the new task of exploiting DNNs executables. With a comprehensive review, we report that all existing attacks are infeasible to attack DNN executables. We attribute such infeasibility to three major causes: 1) no shared memory regions, 2) no shared third-party libraries, and 3) DL compiler optimizations; details are in Sec. 3. Second, we reverse engineer DNN executables and accordingly provide a thorough discussion on the cache-aware optimizations employed by state-of-the-art DL compilers. We find that cache-aware optimizations make the cache access patterns of different model components (i.e., DNN operators) distinct and distinguishable, thus unveiling a unique, novel, and critical attack vector of cache side channels toward DNN executables.

Following the above study, we propose DeepCache, a novel automated, end-to-end cache side channel attack framework against DNN executables. DeepCache is designed to be independent of underlying cache side channels and compatible with different attacking techniques (e.g., Prime+Probe [40]). Despite the difficulty in distinguishing real world cache side channel observations derived from L1 cache/LLC,[1] we propose a series of specifically tailored learning techniques and optimizations to process noisy cache access traces. As a result, DeepCache can precisely infer DNN model architectures (including DNN operator types and hyperparameters) with an accuracy higher than 80% for many complex cases, regardless of the cache level at which the attack is launched. Moreover, we, for the first time, justify how the layout optimizations employed by DL compilers will further impede DNN model weight stealing attacks, and show that DeepCache can precisely identify optimized weight layouts and likely enhance weight stealing attacks. In sum, this paper makes the following main contributions:

[1]LLC attack is deemed to be more difficult to exploit than the L1 cache attack. For readers unfamiliar with the relevant background, we recommend referring to Sec III in the pioneer paper [40] for a good explanation of the concepts. We also briefly list the differences between the L1 cache and LLC attacks in Sec. 2.4

- We deliver the first study investigating the possibility of attacking DNN executables deployed on resource-sharing environments with cache side channels. While none of the existing attacks are applicable to this scenario, we provide a detailed analysis of DNN executables and pinpoint a new attack opportunity caused by cache-aware optimizations.
- We propose a novel end-to-end DNN architecture stealing framework, DeepCache, to remotely infer model architectures of DNN executables with cache side channels as the attacking primitive.
- We evaluate DeepCache against DNN executables compiled from real-world DNN models using two cutting-edge DL compilers. With the L1 and LLC Prime+Probe as attacking primitives, respectively, DeepCache achieves high accuracy in recovering model architectures across all settings.

DeepCache presents a general framework to digest the traces recorded by the cache side channels and infer the DNN architectures. There are no restrictions for underlying side channel attacks, and different cache side channels can be exploited to generate input for DeepCache. We release our research prototype at [2] to facilitate future research.

2 BACKGROUND

2.1 Deep Neural Network

DNN models consist of multiple connected DNN operators (e.g., convolutional layers, fully-connected layers, pooling layers), and each operator involves massive matrix computations. A DNN model is typically trained to learn weights that approximate a non-linear map between input data and output for a specific task. While the training process is computation-intensive and usually equipped with GPUs, the trained model can be distributed to and deployed on different types of hardware devices and even cloud environments [66]. The design and training of a DNN model involve a considerable investment of expertise and computing power. Thus, DNN models become an important IP concerned by attackers.

Nevertheless, we clarify that due to the complexity of DNNs, no existing work aims to steal complete DNN IP remotely. Instead, prior works attack different components separately (see Sec. 3). Similarly, this work focuses on partial components (as illustrated in Fig. 1), and is expected to collaborate with other works to steal the full DNN. To ease the presentation, we summarize the DNN IP in the following three major components.

DNN Architecture. The architecture denotes all operators in a DNN, and each operator is specified by its type (e.g., Conv vs. Pooling) and the hyperparameters (e.g., kernel size). In general, the architecture decides how a DNN computes and serves as the base of the remaining IPs. Nevertheless, the search space of operator types and their hyperparameters is vast; exhausting possible architectures via brute force is impractical. Thus, most previous studies aim to recover the victim DNN's architecture from various side channels [20, 27, 28, 30, 43, 79, 80]. This work, accordingly, also focuses on recovering architecture. Consistent with existing works, the following three types of operators are considered.

- Fully-connected (FC) layers (and cascaded activation functions[2]), whose hyperparameters include the number of input neurons and output neurons.

[2]For example, ReLU, Sigmoid, etc. The same applies to Conv.

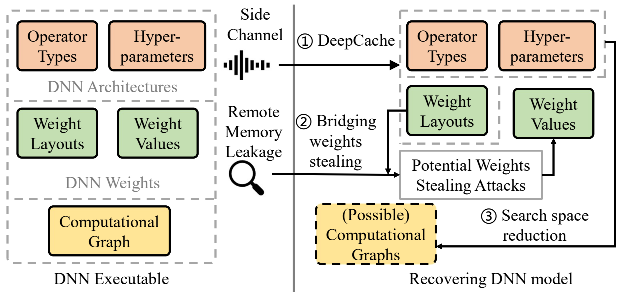


Figure 1: Attacking objectives and general pipeline. DeepCache, as the starting step, mainly focuses on DNN architectures (①). Yet, DeepCache can bridge existing weights stealing attacks (②) by recovering weight layouts, and help reduce the search space of computational graphs (③, as discussed in Sec. 6.2) with recovered architectures.

- Convolutional (Conv) layers (and cascaded activation functions), whose hyperparameters include the number of filters (i.e., #input channels), the filter size, the output feature map size (i.e., #output channels), and the stride.
- Pooling layers (e.g., max-/average-pooling), whose hyperparameters include kernel size, stride, and padding size.

While the above three types subsume operators adopted in mainstream DNNs, DeepCache is not limited to them. DeepCache supports an open set of operators, and can be smoothly extended to new operators (see Sec. 5 and Sec. 4.4).

DNN Computational Graph. It is worth noting that not all DNN models exhibit a sequential structure. Some models (e.g., ResNet [25]) use shortcuts to improve training efficiency. In general, DNN operators are chained into a computational graph, which, as part of the full model specification, indicates how the information flow is passed from input to output. However, as discussed in existing research [30, 79], connecting operators requires accurate data flow tracking, which is usually unavailable with remote side channels. DeepSniffer [30] and Hermes Attack [92] rebuild computational graphs by monitoring the memory bus (PCIe) traffic, which requires physical access and cannot be extended to the remote scenario.

This work does not take computational graph recovery as the direct goal. Nevertheless, given the architecture of a DNN model (i.e., a sequence of operators and corresponding hyperparameters), it is much easier for the attacker to infer possible computational graphs [28, 79]. Specifically, two operators can be connected only if one operator's output shape matches another's input shape [41, 45]. Consistent with prior work [79], our evaluation shows that recovering model architectures can largely reduce the search space of possible computational graphs from 10^7 to less than 10^2 (see Sec. 6.2 for details). Fig. 1 demonstrates the status of DeepCache in a complete DNN stealing attack pipeline.

DNN Weights. Given the architecture and computational graph, the weights (parameters) learned during training specify the functionality of a DNN model, making weights one of the high-profile attacking targets. Existing works discuss leaking weights via hardware defects (i.e., RamBleed [37]) or monitoring and reversing weights from PCIe traffic. Nonetheless, we discuss how optimized weight layout (i.e., the shape of the weight stored in memory) will prevent existing attacks (see Sec. 3.3). While DeepCache does not aim to steal weights, we identify the optimized layouts along with the architectures via the cache side channel. As demonstrated in Fig. 1, identifying layouts fills the gap between architecture stealing and weights stealing attacks to recover the complete DNN model.

2.1.1 Importance of DNN Architectures. Except for DNN weights stealing attacks discussed in Fig. 1, many other DNN privacy attacks rely on the complete knowledge of DNN architectures. Specifically, the membership inference attacks [47, 64] aim to infer whether a data sample was present in the training data. Such attacks usually assume that knowledge about the DNN architectures is known in order to prepare multiple models as inputs for inference algorithms. Knowing the DNN architectures is also necessary for the DNN knowledge stealing attacks [26, 84], which synthesize representative images from the image distribution in the training dataset. The model extraction attacks [70], which seek to build a substitute model that is close to the target model, also assume the attacker knows the DNN architectures for substitute model training.

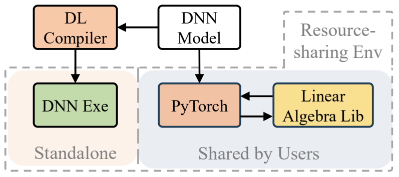


Figure 2: Comparing DL frameworks and DNN executables.

2.2 DL Compiler and DNN Executable

DL Compiler. Given a DNN model pre-trained with DL frameworks like PyTorch, DL compilers take its high-level model specification (e.g., in the ONNX format [1]) as the input, and produce standalone, efficient binary code directly running on hardware backends. Usually, platform-agnostic and hardware-aware optimizations will be successively applied to the input model [14, 60, 71]. Platform-agnostic optimizations, often accompanied by high-level graph-based model intermediate representations (IRs), conduct transformations like operator fusion and constant folding. In contrast, hardware-aware optimizations handle low-level memory-related operations and parallelization with platform-specific IRs.

DNN Executables. Typical DL frameworks like PyTorch leverage third-party linear algebra libraries crafted by experts to speed up DNN inference. Differently, DNN executables exhibit a distinct execution paradigm. As shown in Fig. 2, instead of interpreting the DNN model and calling linear algebra library APIs, DL compilers generate *standalone* executables in which each DNN operator has been specifically compiled and optimized for optimal efficiency [14].

While existing side channel attacks may target different devices, including CPUs, FPGAs, and various hardware accelerators [11, 18, 79, 85], they primarily focus on DNN models deployed with DL frameworks (e.g., PyTorch and TensorFlow). This paper presents the first attack targeting DNN executables. Since GPUs have well-optimized drivers and libraries provided by vendors, DL compilers are primarily used for lower-power but cost-efficient hardware. To be aligned with prior works on DNN executables [45, 77], we present our work on CPUs, although our approach can be adapted to various devices that are vulnerable to cache side channels.

Real-world Significance. DL compilation is a promising technology that is growingly applied in real-world scenarios. For example, Amazon and Meta use DL compilers to compile DNN models on Intel x86 CPUs [32, 44]. A startup raised millions of dollars to accelerate ML services in the cloud using DL compilers [52]. More importantly, PyTorch 2.0 also started to use compilation techniques to speed up model execution [57]. Overall, DL compilation and DNN executables are increasingly essential to boost DNNs.

2.3 Cache Side Channels

Holistically speaking, the sharing and competition between different processes for cache resources result in cache side channels [40, 83]. Cache denotes a storage unit used to reduce the time costs of accessing data from the main memory. On typical cloud computing platforms, virtual machines owned by different users may share the same cache, leading to potential secret information leakage to unprivileged, co-located users. Specifically, an attacker who shares the same cache with the victim can secretly and passively observe the cache access patterns of the victim process.

Flush+Reload [83] and Prime+Probe [40, 55] are the two most well-studied cache side channels. Flush+Reload assumes sensitive code or data is shared. 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. Prime+Probe does not require any shared user-space memory pages and is more potent and widely applicable. However, Prime+Probe 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, resulting in a low sampling rate and noise when measuring access latency. Due to its higher generality and practicality, DeepCache launches Prime+Probe attacks toward DNN executables. Nevertheless, recent advances in cache side channels provide more precise and high-resolution attacks [17, 56]. Our method can similarly use advanced attacks for more precise results.

2.4 L1 Cache Attack vs. LLC Attack

The modern cache is multi-level designed and uses a hierarchy of memory stores based on varying access speeds to cache data. L1 (Level-one) cache is closest to the processor with a small size and fast accessing time, and LLC, on the contrary, is orders of magnitude larger and slower. L1 cache is private to a specific processor core, posing a practical challenge to the attacker, who has to co-locate on the same core as the victim. In contrast, the LLC side channel is deemed a more realistic attack vector, giving LLC is typically shared between cores. While Prime+Probe is proven adaptable to LLC [40], several unique challenges exist, making exploiting LLC side channel observations difficult. Precisely, LLC is sliced, and memory traffic is uniformly distributed to the per-core LLC slices with (usually undocumented) hash functions. Constructing the eviction set requires more specific designs to fill target sets over all slices. Secondly, LLC has higher associativity and longer access latency. Consequently, probing an LLC set takes a longer time, resulting in lower probing resolution and results that are harder to analyze. Nevertheless, we reuse existing side channel attack frameworks and consider both L1 cache and LLC attacks in the evaluation to demonstrate the effectiveness DeepCache.

3 MOTIVATION

This section reviews existing DNN stealing attacks and discusses the obstacles to attack DNN executables.

3.1 DNN Stealing

Table 1 compares DeepCache with relevant side channel-based DNN stealing attacks. In general, existing works follow two different threat models based on assumptions of the attacker's capability. We mark them as the *physical* and *remote* access models in Table 1. Physical access-based attacks mainly focus on edge/IoT devices, where the adversary can access the victim's hardware devices. Differently, remote access-based approaches present more powerful attacks to extract model information from cloud services like MLaaS. The adversary is assumed to be able to run *unprivileged* processes that co-locate on the same processor as the victim process.

Physical Access. Earlier works [11, 85] use EM side channels to infer architecture information of neural networks from specific IoT/edge devices and accelerators. Among them, CSI NN can also recover model weights with a customized variant of differential power analysis. Such EM-based side channel attacks usually assume physical access to the victim device and require special EM probe equipment (e.g., an oscilloscope). Besides, they only focus on simple small models or Binarized Neural Networks (BNNs). Whether they are scalable on modern DNN models is uncertain (marked as partial support ◐).

Differently, DeepSniffer [30] directly snoops the DRAM bus (or uses EM side channels) to record memory access addresses and volumes and then leverages ML techniques to infer model architectures. Hermes Attack [92] captures and reverses PCIe traffic to recover DNN models. Moreover, since model weights are transferred with GPU PCIe without encryption, Hermes Attack can also directly obtain DNN model weights. HuffDuff [80] extends the bus snooping attack on sparse accelerators to deduce structures of pruned DNNs. Since bus snooping provides accurate and noise-free memory access addresses, bus snooping attacks could easily chain DNN operators into the computational graph. However, such attacks hold an assumption stronger than physical access that the attacker can directly monitor the memory bus and PCIe events. We refer to this assumption as *hardware privilege* in Table 1.

MaskedNet [18], on the other hand, assumes the architecture is known and recovers model parameters with the power side channel of FPGAs. Nevertheless, it can only attack BNNs (whose weights are binary values), and extending it to DNNs is still an open question.

Remote Access. One recent power side channel-based model stealing work, DeepTheft [20], leverages the Running Average Power Limit (RAPL) interface provided by modern Intel and AMD processors. Notably, the RAPL delivers energy consumption information of underlying hardware at a fine granularity for software power management. DeepTheft takes such information and uses a learning-based framework to predict the model architecture. While this attack achieves high accuracy, RAPL requires user privileges, and such an attack can be mitigated by disabling the RAPL interface.

Table 1: Comparison with existing works. ●, ◐, ○ denote full support, partial support, and no support.

	Side Channels	Access	Target Device	Target Model	Requirement	Target Model Objectives		
						Archit.	Weights	Layouts
CSI NN [11]	Electromagnetic	Physical	ARM Microcontroller	Small MLP&CNN	EM Probe	◐	◐	○
DeepEM [85]	Electromagnetic	Physical	Accelerator	Binarized NN	EM Probe	◐	○	○
DeepSniffer [30]	Bus Snooping or EM	Physical	GPU	DNN	Hardware Privilege	●	○	○
Hermes Attack [92]	Bus Snooping	Physical	GPU	DNN	Hardware Privilege	●	●	NA
HuffDuff [80]	Bus Snooping	Physical	Sparse Accelerator	Pruned DNN	Hardware Privilege	●	○	○
MaskedNet [18]	Power	Physical	FPGA	Binarized NN	Power Probe	○	◐	○
DeepTheft [20]	Power	Remote	CPU	DNN	RAPL Interface	●	○	○
Cache Telepathy [79]	FR or PP*	Remote	CPU	DNN	Shared Cache&Mem	●	○	○
DeeepRecon [28]	FR	Remote	CPU	DNN	Shared Cache&Mem	●	○	○
Hong et al. [27]	FR	Remote	CPU	DNN	Shared Cache&Mem	●	○	○
GANRED [43]	PP	Remote	CPU	DNN	Shared Cache	●	○	○
DeepSteal [58]	Rowhammer	Remote	CPU+DRAM	Quantized DNN	Vulnerable RAM	○	◐	○
DeepCache	PP	Remote	CPU	**DNN Executable**	Shared Core or Cache	●	○	●

* FR denotes Flush+Reload and PP denotes Prime+Probe.

Other remote access works mainly use cache side channels to recover model architectures. DeepRecon [28] and Cache Telepathy [79], targeting ML frameworks running on CPUs with third-party linear algebra libraries, leverage Flush+Reload or Prime+Probe to detect matrix multiplications to infer hyperparameters of fully-connected and convolutional layers. Specifically, Cache Telepathy pre-analyzes linear algebra libraries to locate sensitive code in Generalized Matrix Multiply (GEMM) functions and demonstrates the feasibility of inferring DNN architectures by monitoring such code.

Due to the limitation of cache side channels, the attacker cannot accurately observe the memory access addresses. Thus, none of them can recover the computational graph of the victim model. Nevertheless, they managed to reduce the search space of possible model architectures. Similarly, Hong et al. [27] leverage Flush+Reload to narrow down the correct architectures of novel DNN models.

The above works observe the victim's behaviors by exploiting cache side channels, which demands shared memory regions between the attacker and the victim. Cache Telepathy leverages prior knowledge of the GEMM execution patterns to apply Prime+Probe in DNN stealing. However, it still requires shared GEMM functions. Since DL frameworks widely use third-party linear algebra libraries for matrix multiplications, such libraries are shared by different DL processes and can be leveraged for cache side channel attacks. However, shared memory is not always available, particularly in the context of DNN executables. GANRED [43] proposes a more practical side channel attack that does not require shared memory or pre-analyzing libraries in advance. It recovers the correct model architectures by repeatedly and incrementally generating an adversary model and comparing the memory access patterns leaked by the cache side channel. Nevertheless, as discussed in Sec. 3.3, DL compilers may generate largely distinct binary code (and consequently distinct cache access patterns) for similar models due to optimizations. Thus, GANRED cannot apply to DNN executables.

Besides, DeepSteal [58] proposes the first model weights stealing attack targeting quantized DNN models using rowhammer-based memory leakage [37]. It targets RAM devices that are vulnerable to the rowhammer attack [35] and repeatedly hammers neighboring memory pages to infer weights values. This work is, therefore, limited by the capability and efficiency of contemporary rowhammer attacks. Besides, the memory layout of model weights may be optimized to speed up memory access (see Sec. 4.1). This problem, which can hinder existing model weights stealing attacks from interpreting values leaked from memory into matrices, has been overlooked by previous studies. For the first time, this paper points out the necessity of identifying weight layout and proposes a practical solution accordingly.

3.2 Threat Model

This paper aims explicitly to attack DNN executables deployed in resource-sharing clouds. The threat model is described below.

Black-box Remote Access. White-box cache side-channel attacks have been proposed for decades, where attackers can pre-analyze target software to identify sensitive code. In contrast, we assume attackers can only arbitrarily invoke the DNN executable. Beyond that, other information, including architectures, weights, training data, and binary code, is unknown.

Unprivileged Attackers. We assume the attacker has no privileges other than those of a normal user. The attacker does not possess adversarial probing devices, and cannot leverage system-level interfaces to monitor the system states.

Hardware Resource Sharing. Consistent with previous cloud co-residency attacks [8, 9, 19, 59, 73, 78, 79, 89], we assume the attacker shares the hardware with the victim DNN executable. Depending on the underlying side channel attack method, a shared cache (for LLC attack) or a shared CPU core (for L1 cache attack) may be required. The difference is that we do not assume any shared memory or third-party linear algebra libraries accessible to attackers.

Besides, ML service providers tend to document and advertise their DL compilation technologies, which are usually open-source and maintained by the community, for propaganda [4]. We thus reasonably assume the DL compiler is open and available to the attacker. We also assume all DNN executables are optimized (the default configuration of DL compilers) for efficiency.

3.3 Difficulty of Attacking DNN Executables

Since we aim to attack DNN executables remotely, side channels requiring physical access or specialized devices (e.g., oscilloscope) are unsuitable. We mainly discuss the difficulties of applying existing remote cache side channels attacking DNN executables.

Standalone Executables. One of the major obstacles to reusing existing attacks is that compiled DNNs are standalone executables, which prevent existing attacks twofold. On the one hand, unlike

DL frameworks that leverage handcrafted linear algebra libraries to speed up inference, DL compilers generate computation-related code during compilation and *statically* link it into the output binary. Thus, attackers cannot expect to share memory regions with DNN executables. While previous Flush+Reload attacks use page deduplication [7, 23, 88] to create shared regions, it does not apply to DNN executables since no replicated data or code exists, which prevents Flush+Reload from being used to attack DNN executables.

On the other hand, existing attacks rely on pre-analysis to identify sensitive code in shared libraries [27, 79]; notably, Cache Telepathy conducts a thorough analysis of linear algebra libraries to infer operator hyperparameters by observing the invocation of specific sensitive functions. However, since the computation code in DNN executables is specifically generated by DL compilers and varies between different DNNs, such pre-analysis is not applicable. DNN executables are black boxes to attackers, and no sensitive code can be located, posing a new challenge for attacking DNN executables.

Model- and Hardware-aware Optimizations. Works relying on Prime+Probe do not require shared memory. However, the only existing solution[43] still does not apply to DNN executables. GANRED assumes the same operator will have similar cache access traces. However, during compilation, DL compilers first search for graph-based model-aware optimization opportunities (e.g., operator fusion), then apply unique (machine learning autotuned) hardware-aware optimizations to each operator. Therefore, a Conv belonging to different models may have largely distinct cache access patterns due to optimizations, and we cannot identify the victim model by incrementally constructing a similar model.

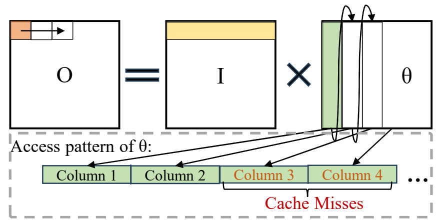
4 DEMYSTIFYING SIDE CHANNEL LEAKAGE

Although DNN executables cannot be attacked by prior side channel-based model stealing works, with an in-depth analysis and reverse engineering, we recognize that *hardware-aware optimizations* extensively applied by DL compilers (which intend to exploit the performance potential of hardware) introduce new cache side channel leakage. More concretely, each optimized DNN operator (e.g., a Conv or an FC) shows a distinguishable and unique memory access pattern, which can be observed using conventional cache side channels (e.g., Prime+Probe). In the following, we elaborate on how DL compilers apply optimizations to DNN operators, and accordingly uncover how an operator (including its type, hyperparameters, and optimized weight layout) can be recognized from its optimized memory access patterns.

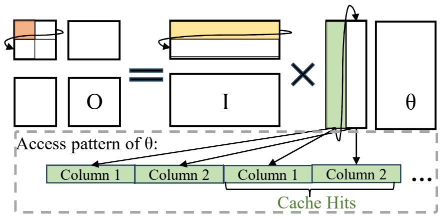
4.1 Optimizations for Matrix Computations

Since massive matrix computations (e.g., multiplication) are involved in DNN inference, DL compilers primarily leverage the following two optimization schemes to improve efficiency.

Blocking for Cache Locality. Fig. 3(a) illustrates a conventional matrix multiplication of $O = I \times \theta$. As marked by the black arrow, elements in O are sequentially calculated in every row. This way, θ is accessed by sequentially putting each column into the cache. Since these columns are usually mapped to different cache lines and the cache size is limited (e.g., 32 KB), cache misses frequently occur during the computation. DL compilers leverage blocking to improve the cache locality. As shown in Fig. 3(b), the output matrix O is chunked into blocks, such that when computing within one block, accesses to θ are limited to only several columns. Since these columns are iteratively accessed, the cache hit rate is improved.



(a) Sweeping θ results in low memory locality.



(b) Blocking results in high memory locality.

Figure 3: Memory access patterns before/after blocking.

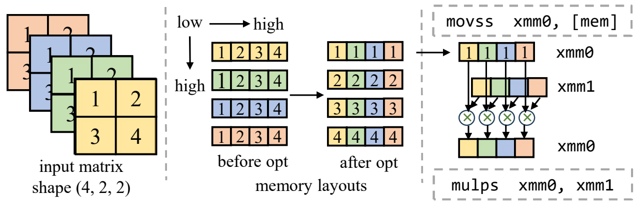


Figure 4: Example of vectorization optimization.

Vectorization for Parallel Computation. Single Instruction Multiple Data (SIMD) instructions provided by modern CPUs can read consecutive memory and perform the same operation on multiple data (which constitute a vector) in parallel. The right side of Fig. 4 shows two SIMD instructions, in which `movss` reads four floats sequentially, and `mulps` takes two vectors (`xmm0` and `xmm1` registers) and writes results in `xmm0`.

In order to speed up matrix computations with SIMD, DL compilers tend to rearrange the memory layouts of matrices to conform to the computation pattern of SIMD instructions. As shown in Fig. 4, given a matrix of shape $(N, \cdot)$, DL compilers optimize its memory layout as $(N/K, \cdot, K)$, where K is a multiple of 4. Thus, the original N matrix computations are optimized as K parallel groups of N/K computations. Since all previous attacks only assume architecture is known while ignoring potentially altered memory layout, this optimization may prevent existing weights stealing attacks [58, 77], i.e., attackers cannot recover the correct matrix from dumped bytes values without knowing the optimized memory layouts. This work potentially lessens this problem for previous stealing attacks by identifying the weight layouts.

4.2 Leakage in Optimizations

While the above optimizations improve the efficiency of DNN executables, they also implicitly give attackers an opportunity to peek at DNN architectures. Without loss of generality, we consider Conv a representative example to demonstrate the impact of optimizations. Below, we illustrate how distinct loop structures derived from optimizations can be treated as a unique identifier for an operator.

```
 1  def Conv(I, W, O):
 2    # output channels
 3    for oc in range(256):
 4      # output height
 5      for oh in range(14):
 6        # output width
 7        for ow in range(14):
 8          # lines 2-7: each output element
 9          # input channels
10          for ic in range(128):
11            # kernel height
12            for kh in range(3):
13              # kernel width
14              for kw in range(3):
15                v_1 = oh * stride + kh
16                v_2 = ow * stride + kw
17                O[1][oc][oh][ow] += \
18                  I[1][ic][v_1][v_2] * \
19                  W[oc][ic][kh][kw]
```

low memory locality

Figure 5: Example of a naive Conv without optimizations.

Computations in Conv. Fig. 5 shows a conventional Conv implementation without optimizations. In Fig. 5, the input size is $(128, 29, 29)$, and the output size is $(256, 14, 14)$. The size of weight W is $(256, 128, 3, 3)$, where $(3, 3)$ denotes the kernel size. Lines 2-7 walk through every element in the output, and lines 10-19 calculate the output value. Since the memory read at line 18 visits different elements in matrix I every time, this implementation manifests a low memory locality, resulting in a low cache utilization.

Optimized Conv Computation. As mentioned in Sec. 4.1, DL compilers use blocking to improve the cache locality. More specifically, DL compilers will optimize the layout of weight W as shape $(256/K_O, 128/K_I, 3, 3, K_I, K_O)$. The optimal factors K_I and K_O are decided by DL compilers. We now elaborate on how these optimization factors, together with the operator's hyperparameters, uniquely determine the loop structure in the generated binary code.

Fig. 6 shows the optimized Conv implementation, where $K_I = 8$, $K_O = 32$. Accordingly, input and output matrices are reshaped into $(128/K_I, 29, 29, K_I)$ and $(256/K_O, 14, 14, K_O)$. The highest dimension (i.e., channels) of the output (input) matrix is split into two dimensions, i.e., `oc_outer` (`ic_outer`) and `oc_inner` (`ic_inner`). The loops are then permutated to calculate the results block by block. Optimized code in Fig. 6(a) shows better memory locality as accesses to the same elements are clustered. Furthermore, the inner loops can be unrolled. As a result, DL compilers may use SIMD instructions (e.g., `mulps`) to speed up multiplication operations with parallelism, as shown in Fig. 6(b).

Unique Identifier. DL compilers decide how to optimize a DNN operator (e.g., choosing block size, permuting, and unrolling the loops) according to its hyperparameters and model architecture. For example, as in Fig. 6(b), Conv with different numbers of output channels will result in different `oc_outer` (`oc_inner`) values. Consequently, distinct loop structures are exhibited in binary code. In short, we observed that:

Each operator will be specifically optimized according to its hyperparameters by the DL compioler, leading to different loop structures in compiled low-level code.

Specifically, given the model architectures, including operators and hyperparameters, the DL compiler uniquely determines optimization factors that can achieve optimal performance, i.e., the optimized loop structures only depend on the model architectures regardless of model weights. Hence, we take the consistent and deterministic loop structures as an operator's unique identifier.

4.3 Observable Cache Access Patterns

Intuitively, DNN inference is a long sequence of matrix computations. We reasonably hypothesize that the optimized loop structure can result in distinguishable cache activities due to the massive memory accesses. Since real cache side channel observations are extremely hard for humans to recognize, to ease readers' understanding and to validate our assumption, we use Intel Pin [48], a dynamic binary instrumentation tool, to mimic noise-free Prime+Probe side channel observations [40, 55]. Nevertheless, our evaluation uses real world LLC/L1 cache side channels in practical settings (details in Sec. 6). Fig. 7 shows examples of cache access traces from three Conv[3], whose statistics are listed in Table 2. Each row in a trace represents a cache state observed in one round of probing, where dark and light pixels denote cache hits and misses, respectively.

Table 2: The statistics of operators in Fig. 7.

	kernel size	#input channels	#output channels	model	compiler
Conv A	3	128	128	ResNet18	TVM
Conv B	3	128	256	VGG16	TVM
Conv C	1	256	512	ResNet18	Glow

Loop_I and Loop_O. In Fig. 7, the pattern boundaries are marked by the red dashed box. For both compilers, we can observe certain patterns periodically appearing. Also, the patterns and their period vary with operators.[4] In particular, we note that:

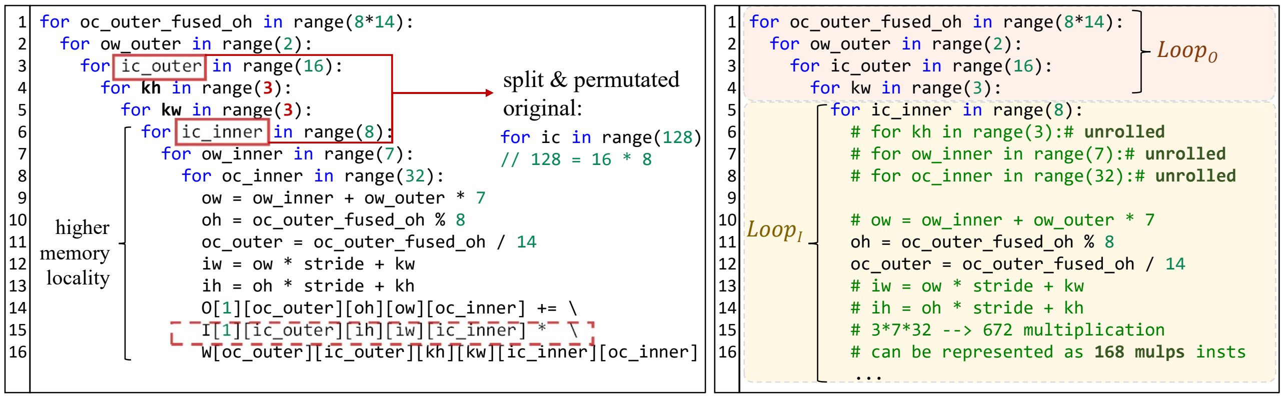
Each operator's cache activities can be uniquely depicted with $Loop_I$ and $Loop_O$, which are defined below.

We define $Loop_I$ and $Loop_O$ from the attacker's perspective. Since binary is deployed remotely, the attacker cannot directly infer loop structures from observed traces due to the limitation of cache side channels. Instead, we can approximate a mapping from cache side channel observations to loop structures. Here, we use $Loop_I$ (inner loop) to denote the pattern itself (corresponds to the optimized loops of "high memory locality" in Fig. 6(a)) and $Loop_O$ (outer loop) to represent the frequency of a pattern's occurrence (which depends on the outer loops that traverse different blocks in the output, as shown in Fig. 6(b)). The $Loop_I$ and $Loop_O$, combined together, can be viewed as the representation of optimized loop structures observed via cache side channels.

Ideally, recognizing $Loop_I$ and $Loop_O$ from side channel observations enables identifying a DNN operator (i.e., the operator type and hyperparameters). Nevertheless, side channel observations are highly noisy; for example, one iteration of Prime+Probe could take

[3]Since a complete trace is too long to be presented in the paper, the traces shown in Fig. 7 are short snippets that are long enough to demonstrate the patterns.

[4]Note that although Fig. 7 only shows L1 cache access traces, the discussion and conclusions in Sec. 4 are general and robust across different cache hierarchy levels. As demonstrated in Sec. 6, DeepCache can leverage different levels of cache access traces to infer model architectures.



```
for oc_outer_fused_oh in range(8*14):
  for ow_outer in range(2):
    for ic_outer in range(16):
      for kh in range(3):
        for kw in range(3):
          for ic_inner in range(8):
            for ow_inner in range(7):
              for oc_inner in range(32):
                ow = ow_inner + ow_outer * 7
                oh = oc_outer_fused_oh % 8
                oc_outer = oc_outer_fused_oh / 14
                iw = ow * stride + kw
                ih = oh * stride + kh
                O[1][oc_outer][oh][ow][oc_inner] += \
                I[1][ic_outer][ih][iw][ic_inner] * \
                W[oc_outer][ic_outer][kh][kw][ic_inner][oc_inner]
```

```
for ic in range(128)
// 128 = 16 * 8
```

(a) Example of Conv with blocking.

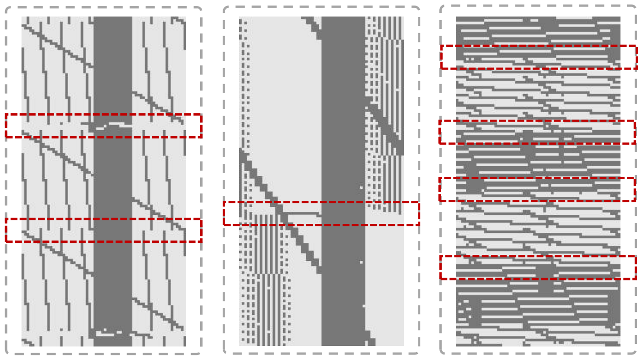
```
for oc_outer_fused_oh in range(8*14):
  for ow_outer in range(2):
    for ic_outer in range(16):
      for kw in range(3):
        for ic_inner in range(8):
          # for kh in range(3):# unrolled
          # for ow_inner in range(7):# unrolled
          # for oc_inner in range(32):# unrolled

          # ow = ow_inner + ow_outer * 7
          oh = oc_outer_fused_oh % 8
          oc_outer = oc_outer_fused_oh / 14
          # iw = ow * stride + kw
          # ih = oh * stride + kh
          # 3*7*32 --> 672 multiplication
          # can be represented as 168 mulps insts
          ...
```

(b) Example of Conv with vectorization.

Figure 6: Example of a Conv with optimizations.

thousands of CPU cycles, during which later cache activities overwrite many previous activities. In Sec. 5, we tackle the challenge of recognizing $Loop_I$ and $Loop_O$ from noisy side channels.



(a) Trace of Conv A. (b) Trace of Conv B. (c) Trace of Conv C.

Figure 7: Examples of cache access traces logged by Intel Pin. The borders of patterns are marked with red dashed boxes.

4.4 Generalization

This paper focuses on three main types of operators that form the cornerstone of mainstream DNNs. However, DeepCache is not limited to specific operator types. Since operators used to constitute DNNs almost always contain extensive memory accesses and floating-point arithmetics, we reasonably assume they will present observable cache access patterns under typical cache side channels. Therefore, they are analyzable, following the same strategy used in this section. Meanwhile, according to our observation, DL compilers generate optimized loop structures to handle floating-point values regardless of operator type, reinforcing our attack's feasibility on other operators. From another perspective, our attack framework makes no assumptions about the function and behavior of target operators; it thus shall be smoothly extended to other operators.

Moreover, as discussed in Sec. 5, with several careful design considerations, DeepCache supports an open set of operators; extending DeepCache for new *unknown* operators is technically straightforward without human expertise. Besides, operators considered in this study (i.e., Conv, FC, and Pooling) are representative, subsuming most operators in modern DNNs. For example, typical NLP operators, including RNN, GRU, and LSTM, are made up of basic FC operators. We confirm that during compilation by contemporary DL compilers, those NLP operators are broken down into FCs, and thus can naturally be handled by DeepCache.

5 DESIGN OF DEEPCACHE

As mentioned in Sec. 4.3, obtaining precise cache activities is challenging due to noise, making recovering loop structures challenging. However, we do not need to identify full cache activities — vague patterns embedded in cache traces are enough to decide $Loop_I$ and $Loop_O$. Accordingly, DeepCache is designed as two main stages: 1) *feature extraction* and 2) *trace segmentation*, as illustrated in Fig. 8.

Overview. DeepCache takes side channel observations as input. Specifically, we use Prime+Probe to observe cache states when the victim DNN executable is running. Each state is a binary vector of size S (S denotes the number of monitored cache sets) consisting of only 0s and 1s, where 0 denotes a cache miss, and 1 denotes a cache hit. After N times of probing, we have a matrix of size (N, S) as the cache access trace. Then, for an operator, the feature extraction stage extracts a abstracted feature vector that has much lower dimensions than the original side channel trace. Simultaneously, the trace segmentation phase outputs an integer as $Loop_O$. To do so, we formulate it as an "anomaly detection" problem [3, 10, 38, 61], where we view the segment point as an "abnormal" state (Sec. 5.2).

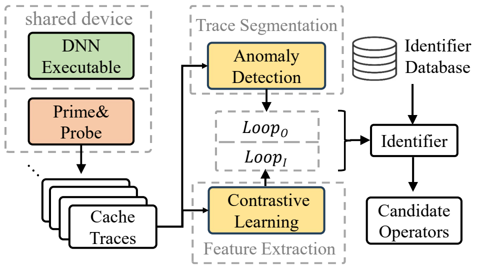


Figure 8: Overview of DeepCache.

Regression. Unlike previous works that treat operator recovery as a classification task [20, 30], we formulate this problem as a regression task (see **Benefits** below for comparison). Specifically, we first prepare a collection of operators and build a database using their $Loop_I$ and $Loop_O$ (which constitute a pair). Once we get the $(Loop_I, Loop_O)$ from a side channel trace, we compute their similarity with those in the database. Each operator record in the database is accompanied by its *type*, *hyperparameters*, and *optimized layout*. Our recovery result will be deemed as the operator whose $(Loop_I, Loop_O)$ has the highest similarity.

Benefits. Our formulated regression task enables the following benefits. First, the number of possible operators is huge since the hyperparameters often consist of multiple variables. Forming a classification of massive output classes is fundamentally challenging and data-intensive. As will be introduced in Sec. 5.1, our re-formed regression task can alleviate this hurdle by leveraging the contrastive learning paradigm. Second, with the fast development of the DNN community, new operators are continuously designed. Adding support for new operators requires rebuilding a classification model. We, in contrast, only require adding the new operators and their $Loop_I$ and $Loop_O$ in the database.

5.1 Feature Extraction

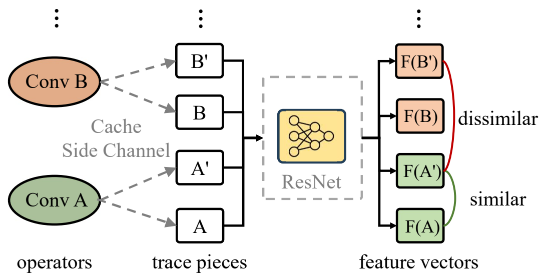


Figure 9: The overview of contrastive learning.

Contrastive Learning. As illustrated in Fig. 9, the feature extraction module $\mathcal{F}$ aims to yield closely similar feature vectors (i.e., $Loop_I$) for side channel traces logged from the same operator. That is, given two side channel traces A and A' logged from the same operator, despite the noise in A, A', the feature extraction module is expected to be resilient and outputs $\mathcal{F}(A) \approx \mathcal{F}(A')$. However, if two traces A and B are logged from different operators, we require the extracted features $\mathcal{F}(A)$ and $\mathcal{F}(B)$ to be sufficiently distinct. Intuitively, since both the operator itself and noise can affect the resulting side channel observations, the aforementioned feature-contrast paradigm can force the feature extraction module to neglect noise and focus on valuable information.

Specifically, we adopt machine learning techniques to implement the feature contrast. $\mathcal{F}$ is implemented as a typical image classification model (i.e., ResNet [25]), except outputs a low-dimensional vector instead of confidence scores. The input trace is fed into F as a single-channel image. By training $\mathcal{F}$ with a rich set of locally collected operators and their side channel traces, $\mathcal{F}$ can learn to recognize visual patterns discussed in Sec. 4.3, and implicitly rule out noise. To further make $\mathcal{F}$ resilient to the unalignment mentioned above, during training, $\mathcal{F}$ takes a snippet randomly cut from a trace as input, rather than the full trace. It is worth noting that we let $\mathcal{F}$'s output have much lower dimensions (e.g., 128 dimensions) than its input; this way, we can further help $\mathcal{F}$ rule out irrelevant information and focus on informative records.

Feature Similarity. During training, we require an effective and efficient metric to measure the similarity between extracted features. Since the extracted features are vectors, we compute their similarity via dot product. The dot product $u \cdot v$ increases when u and v become more similar. Compared with distance metrics of vectors, similarity derived from dot product is more precise and computation-efficient.

Training Loss Function. To train $\mathcal{F}$, we randomly sample $2n$ side channel traces $\{x_1, x_1', x_2, x_2', ..., x_n, x_n'\}$ collected from operators $\{1, 2, ..., n\}$ in each training iteration. x_i and x_i' are two different traces (due to noise) collected from operator i. Let $P(i|x_j)$ denote the probability (from $\mathcal{F}$'s view) that x_j is produced by operator i, computed as

$$P(i|x_j) = \frac{exp(\mathcal{F}(x_i)^\intercal \mathcal{F}(x_j))}{\sum_{k=1}^{n} exp(\mathcal{F}(x_k)^\intercal \mathcal{F}(x_j))}. \tag{1}$$

Then, we compute the joint probability P_i depicting that, x_i' is produced by operator i whereas other x_j $(j \neq i)$ are not, as

$$P_i = P(i|x_i') \prod_{j \neq i} (1 - P(i|x_j)). \tag{2}$$

Therefore, $\mathcal{F}$ will be optimized to maximize all P_i. To void the numerical issues induced by the factorial operation, we further compute the logarithm of P_i. The training is driven by minimizing the loss function, which is $-\sum_i \log P_i$.

5.2 Trace Segmentation

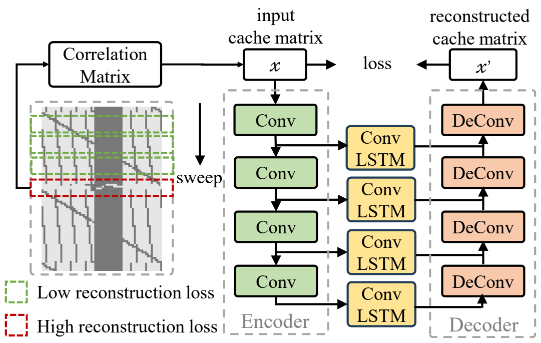


Figure 10: The structure of encoder-decoder network.

Side Channel Traces as Time Series. The trace segmentation module aims to identify $Loop_O$. We treat each input trace as a time series of multiple variates, i.e., each cache set is viewed as a binary variate with the value of 0 or 1. Each row in the input trace denotes one element in the time series and the status of each variate changes over time. Our manual analysis shows that most adjacent rows change smoothly (Fig. 7). In most cases, each row is similar to the result of slightly shifting preceding rows.

To interpret, recall that each cache access pattern corresponds to computations within inner loops (in Sec. 4.3), where the accessed memory addresses often change continuously. Thus, the segment point (the interval between two patterns) can be deemed an abnormal sequence element. This observation is general and widely applicable because it captures the commonality of DNN operators.

Anomaly Detection via Reconstruction. Based on the above observations, we formulate the trace segmentation problem as an anomaly detection task over time series. Intuitively, anomaly denotes the minority trace elements that are distinct from others. Since traces of different operators are distinct, anomaly detection should be performed among records from the same operator. Also, given that new (unknown) operators may be designed in the future and DeepCache is expected to support them, the anomaly detection should be agnostic to an operator's implementation. Therefore, the following procedure is proposed.

As illustrated in Fig. 10, we first prepare an encoder $\mathcal{E}$ and a decoder $\mathcal{D}$. For each sequence element x, the encoder $\mathcal{E}$ converts it into an intermediate variable $z = \mathcal{E}(x)$ and the decoder $\mathcal{D}$ reconstructs $x' = \mathcal{D}(z)$. To avoid $\mathcal{D}(\mathcal{E}(x))$ simply copying x, we let z have much lower dimensions than x. To train $\mathcal{E}$ and $\mathcal{D}$, we use all sequence elements as the training data. Since only a few elements are segment points, the (implicit) reconstruction rules learned by $\mathcal{E}$ and $\mathcal{D}$ are dominated by normal elements within patterns and likely do not apply to the segment points. As a result, $\mathcal{E}$ and $\mathcal{D}$ will fail to reconstruct segment points, i.e., $x \neq \mathcal{D}(\mathcal{E}(x))$. Note that the above procedure does not rely on specific implementations and can apply to any (unknown) new operators.

5.2.1 Convolutional-based Encoder-Decoder Network. As shown in Fig. 10, the encoder-decoder network consists of 4 Conv-based encoders and decoders, which are initially proposed for the image segmentation task [46, 81]. Besides, we employ the ConvLSTM [39, 63, 67] to enhance the encoding with temporal information.

Spatial Information. The encoder part, similar to CNN models that try to extract distinguishing features from the input image, is composed of several sequential Conv layers and is used to encode input into feature maps with smaller sizes. For the l-th Conv layer, assuming the input tensor is $\mathcal{X}^l \in \mathbb{R}^{n\times n\times d}$, the output is defined as

$$\mathcal{X}^{l+1} = f(W^l * \mathcal{X}^l + b^l), \tag{3}$$

where $*$ denotes the convolutional operation, $f(\cdot)$ denotes the activation function, W^l and b^l denote the weights and biases.,

The decoder part consisting of a sequence of deconvolutional layers (DeConv) [46] takes the encoded feature maps as input and tries to recover the original input. Each DeConv layer is used to uncompress the output of previous layer. The output of the last DeConv layer represents the reconstructed matrices and will be compared with the input matrices. The training objective is defined as the reconstruction errors between input and recovered matrices.

Frequency and Correlation of Cache Activities. As shown in Fig. 7, certain cache sets are frequently accessed (the dark vertical "lines" in Fig. 7). Also, some cache sets are correlated, i.e., they may be accessed simultaneously or never accessed at the same time. The frequency and correlation of cache activities are also necessary to identify patterns in the side channel trace. Therefore, we encode such correlations into a correlation matrix M as the segmentation module's input. The (i, j)-th entry in M is calculated as:

$$m_{i,j} = \frac{\sum_{t=0}^{n}(v_i^t + v_j^t)}{2n}, \tag{4}$$

where v_i^t is the i-th cache set's status at position t in the sequence (i.e., the t-th row). To interpret, for diagonal elements $m_{i,i}$, a higher value indicates that the i-th cache set is more frequently accessed. For non-diagonal elements $m_{i,j}$ $(i \neq j)$, a higher value implies that they are often accessed together. M is later fed into $\mathcal{E}$ for encoding.

Temporal Information. Given that a cache status is usually temporally correlated with previous states, we intuitively guide the encoder and decoder to leverage such temporal information. This is implemented by stacking multiple $\mathcal{E}$ and $\mathcal{D}$ and connecting them with ConvLSTM [63], an effective sequential neural network component aiming at capturing temporal information from 2D sequences. As shown in Fig. 10, each $\mathcal{E}_i$ takes the output from $\mathcal{E}_{i-1}$ and passes its output to $\mathcal{E}_{i+1}$ and the i-th ConvLSTM cell. Then, each $\mathcal{D}_i$ takes outputs from both $\mathcal{D}_{i+1}$ and the i-th ConvLSTM cell. This way, the temporal information is encoded.

6 EVALUATION

6.1 Setup

Platform. We evaluate DeepCache on a victim platform equipped with Intel Core i7-9700K. The processor has a 32 KB L1 data cache for each core and a 12 MB last level cache (LLC). Our attack is conducted on Ubuntu 22.04. The models used in DeepCache are trained on another platform with Nvidia GPUs.

DNN Models. We collect all image classification models from the ONNX Zoo [54] as our dataset. In total, our dataset includes 372 operators from 30 commonly used real-world CNN models (e.g., AlexNet [36], SqueezeNet [31], and MobileNet [29]) trained on the ImageNet dataset. Although real-world attackers can take advantage by including as many as possible common operators and models in the database, for a fair evaluation, we ensure that target victim models (i.e., VGG16 [65] and ResNet18 [25]) are not used as training data. All models are compiled and fully optimized with two state-of-the-art DL compilers, TVM [14] and Glow [60].

Implementation. The DeepCache framework is implemented in Python, with about 3K lines of code. Besides, we leverage the cache side channel toolkit Mastik [82] to implement the L1 data cache Prime+Probe attack. As for the LLC attack, we reuse the framework released by a recent work, Prime+Scope [56]. Following prior works, we leverage huge pages to build eviction sets during the LLC attack [40, 56]. Due to the large amount of LLC sets (1024 in our device), monitoring all sets leads to an unacceptable low time resolution. Thus, we monitor random consecutive 64 LLC sets.

To collect noise-free cache access traces (as shown in Fig. 7) for understandable presentations and an easier case study, we implement an Intel Pin [48] tool to mimic the Prime+Probe attack. Specifically, we instrument all memory access instructions and maintain a cache state describing which cache sets are visited. All memory accesses are converted into L1 cache set indexes by extracting the lower 6-12 bits from addresses during runtime. We output and reset the cache state each time after an appropriate time interval (the time of 100 memory accesses in our implementation).

Below, Sec. 6.2 and 6.3 demonstrate the capability of DeepCache when attacking real-world DNN executables. Sec. 6.4 presents a case study to explain differences in results across attacking primitives and DL compilers.

6.2 DeepCache with L1 Cache Attack

To evaluate the accuracy of DeepCache, we take ResNet18 and VGG16 as our test data, the statistics of which are shown in Table 3. Both models are deep models widely used in real-world applications, while VGG16 is a typical sequential model, and ResNet18 is a representative non-sequential model with shortcuts. Note that we have no requirement for the input of victim DNN, i.e., we observe the cache access patterns instead of the exact numerical values during DNN inference. Thus, attackers do not need to scope valid inputs of the victim DNN, which may be private (e.g., medical images), and any values (e.g., random noises) can be used as the inputs.

Table 3: The statistics of models in the test dataset.

	#Operators	#Hyper-parameters	#Dimensions of Mem Layouts
VGG16	21	68	93
ResNet18	23	86	126

Following the discussion in Sec. 4, this section first shows the results of equipping DeepCache with the L1 cache side channel attack. Then, Sec. 6.3 extends the evaluation of DeepCache to the more practical LLC attack. Table 4 summarizes the results regarding operator types, hyperparameters, and optimized memory layout recovery. DeepCache achieves high accuracy when recovering operator types for TVM-emitted DNN executables. While DeepCache shows relatively low accuracy when recovering hyperparameters and memory layout for Glow-emitted DNN executables, we attribute it to Glow's less intensive optimization strategies (see Sec. 6.4 for the explanation). Besides, the results shown in Table 4 do not reveal the upper-bound capability of DeepCache. Sec. 6.3 reports that DeepCache's performance can be largely enhanced with the LLC attack, presenting much higher accuracies threatening potential real-world applications. This section mainly aims to validate the conclusions in Sec. 4 and demonstrates the usefulness of DeepCache in stealing valuable DNN information. Below, we provide a more comprehensive discussion of the results in Table 4.

Table 4: The performance of DeepCache with L1 Prime+Probe attack in recovering DNN architectures, and memory layouts.

	TVM		Glow	
	ResNet	VGG	ResNet	VGG
Operator Types	95.2%	88.2%	94.4%	81.3%
Hyperparameters	96.2%	89.5%	71.9%	87.5%
Mem Layouts	100%	100%	71.0%	100%

Technically, recovering hyperparameters and optimized memory layouts is more challenging than recovering operator types due to a larger search space. Nevertheless, Table 4 reports an even higher accuracy of recovering hyperparameters/memory layouts than recovering operator types for some cases. With manual inspection, we find that DeepCache fails to infer some operators that do not have parameters (e.g., pooling) while still correctly recovering all optimized parameter layouts for other operators (100% in Table 4).

Conceptually, DNN executables exhibit a standalone paradigm that largely reduces cache side channel attack surfaces. Even more, cache side channels naturally face technical challenges, including noise and low time resolution, and consequently, capture only limited behaviors of DNN execution. This explains the challenge and the less than 100% accuracy of DeepCache. However, we interpret that the achieved accuracy is sufficiently high to pose a real-world threat to the security of DNN services, as discussed below.

Computational Graph. As mentioned in Sec. 2, DeepCache does not aim to recover the computational graph directly. Nevertheless, recovered operators can significantly reduce the search space. In particular, if we conservatively assume that the attacker knows

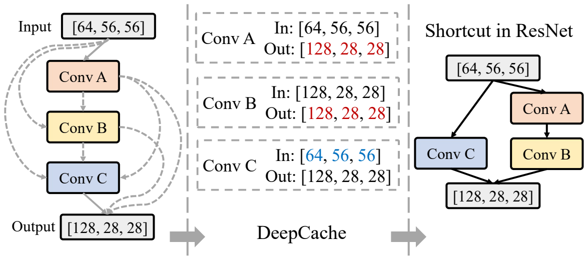

Input
[64, 56, 56]
Conv A
Conv B
Conv C
Output
[128, 28, 28]
Conv A In: [64, 56, 56] Out: [128, 28, 28]
Conv B In: [128, 28, 28] Out: [128, 28, 28]
Conv C In: [64, 56, 56] Out: [128, 28, 28]
DeepCache
Shortcut in ResNet
[64, 56, 56]
Conv A
Conv C
Conv B
[128, 28, 28]

Figure 11: Reducing computational graph search space.

the number of operators and operator types, considering the existence of non-sequential connections, there are $N \times 2^{N-1}$ possible computational graphs for a model with N operators. For example, models like VGG16 and ResNet18 have more than 10^7 possible computational graphs. While such a huge search space is intractable, DeepCache can significantly reduce it.

Specifically, two operators can be connected only if one's output size equals another's input size. Fig. 11 illustrates a typical shortcut connection in ResNet. While recovering hyperparameters, DeepCache simultaneously recovers each operator's input and output size, which helps filter out erroneous computational graphs, e.g., the outputs of `Conv A` and `Conv B` cannot be forwarded to `Conv C`. As shown in Fig. 11, the search space for the shortcut structure is reduced from 2^7 (there are seven dashed connections) to 1. Consequently, in our evaluation, the reduced search space for VGG16 and ResNet18 is 16 and 80, respectively.

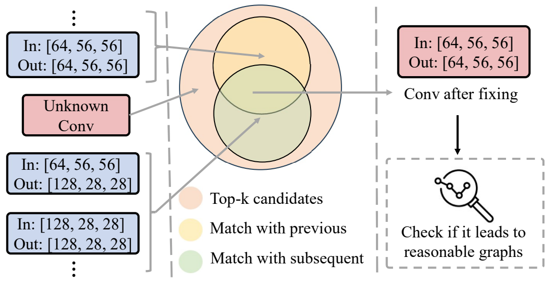

In: [64, 56, 56] Out: [64, 56, 56]
Unknown Conv
In: [64, 56, 56] Out: [128, 28, 28]
In: [128, 28, 28] Out: [128, 28, 28]
Top-k candidates
Match with previous
Match with subsequent
In: [64, 56, 56] Out: [64, 56, 56]
Conv after fixing
Check if it leads to reasonable graphs

Figure 12: Error fixing illustration.

Prediction Error Fixing. Our investigation shows that operators with wrong labels often exhibit low similarity with the candidate. Therefore, it can be assumed that those few wrong labels can be identified by experienced DNN developers semi-automatically. Fig. 12 illustrates the error-fixing process with manual intervention. Specifically, to fix one operator that is identified as potentially wrong due to low similarity, we identify among the top-K candidates that can match the previous and the subsequent operators, i.e., we choose candidates whose input (output) size matches the majority of the previous (subsequent) three operators' output (input). The candidates will later be manually verified to check if they lead to reasonable computational graphs. Our manual investigation shows that 8 out of 11 inference errors in Table 4 can be fixed.

6.3 DeepCache with LLC Attack

As mentioned at the end of Sec. 1, DeepCache is designed as a general framework to digest traces logged by different cache side channels. Compared with the L1 cache attack, the LLC attack poses a more severe and practical security challenge by relaxing the requirement of the shared physical resources. To demonstrate the generality and realism of DeepCache, Table 5 presents the evaluation results of DeepCache with traces logged by the typical LLC Prime+Probe attack. This section serves as a proof-of-concept demonstration of DeepCache's potential in real world scenarios.

Given the LLC traces, DeepCache performs better than L1 traces when inferring model architectures of TVM-emitted executables; it can successfully identify almost all operators compiled by TVM. Nevertheless, we observed that DeepCache equipped with LLC attack shows superior performance and correctly infers all operators compiled by Glow. Although the LLC attack is usually deemed more difficult than the L1 cache attack and its results are more challenging to analyze due to noise and lower resolution, we interpret the results as that LLC has a larger number of cache sets and, consequently, can embed more information of executed low-level code in the cache side-channel observations. DeepCache, as exhibited, effectively extracts such information from cache side channel observations and outputs distinguishable features. Besides, DL compiler optimizations also significantly affect side channel observations. Sec. 6.4 provides a detailed case study to illustrate the difficulty of inferring operators when less information is embedded in L1 cache traces due to non-optimal optimizations.

Overall, the results demonstrate DeepCache's ability to extract distinguishable features regardless of the specific underlying cache side channel attacks used, and also validate the reliability and generality of our observations in Sec. 4. In sum, we interpret the results as promising. Given that recently advanced cache side channel attacks [33, 56, 86] (see more discussion in Sec. 8) could provide a remarkably high time resolution of less than 70 CPU cycles (in contrast, the standard Prime+Probe attack leveraged in this paper requires thousands of CPU cycles), we envision that DeepCache could be benefited from advanced high-resolution cache attack methods working in a pluggable manner.

Table 5: The performance of DeepCache with LLC attack.

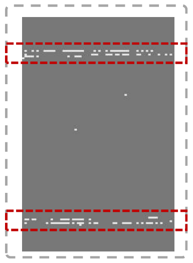
	TVM		Glow	
	ResNet	VGG	ResNet	VGG
Operator Types	95.2%	100%	100%	100%
Hyperparameters	92.6%	100%	100%	100%
Mem Layouts	91.9%	100%	100%	100%

6.4 Case Study

Comparing Table 4 with Table 5, DeepCache shows an unusually low accuracy with L1 traces. To explore the root causes and have a clear understanding of DeepCache's capability, we present a detailed case study in this section. Meanwhile, due to the difficulty in recognizing real cache access traces, we leverage noise-free traces described in Sec. 4.3 to speed up the manual investigation.

Featureless Traces. We first closely examine one operator's logged cache access trace that DeepCache fails to recognize. As shown in Fig. 13(a), even with noise-free traces, little meaningful information has leaked out, i.e., the trace is visually featureless, and almost all cache sets are visited (marked as dark pixels). Although the pattern boundaries are still noticeable (marked with red dash boxes), the featureless patterns prevent the feature extraction component of DeepCache from learning meaningful information.

After investigation, we find that 9 out of 12 errors in Table 4 are due to such cause. DeepCache is especially less functional when several Glow-compiled operators show similar, featureless cache access traces. However, when DNNs are compiled with TVM, such featureless traces are less frequent; most traces show repeated and visible patterns that DeepCache can distinguish. Below, we unveil that this distinction stems from Glow's less intensive optimizations.


(a) Example of featureless trace.

```
1  for ic in range(512):
2      addr1 = (W + w_idx + ic*8)
3      addr2 = (W + w_idx + ic*8 + 20h)
4      # read a ymmword from addr1
5      # read a ymmword from addr2
6      addr1 = (W + w_idx + ic*8 + 24000h)
7      addr2 = (W + w_idx + ic*8 + 24020h)
8      # read a ymmword from addr1
9      # read a ymmword from addr2
10     # floating-point multiplications
11     # ...
```

(b) Example of non-optimal code.

Figure 13: Case study.

Non-optimal Code. To explain the root reasons for the above "featureless" traces, we reverse-engineered the corresponding executables and manually analyzed the binary code. Our analysis shows that Glow's optimization strategies sometimes appear inflexible and less effective than TVM's. Fig. 13(b) shows a simplified code example generated by Glow. This code snippet is the inner loop part of a Conv operator, which conducts massive floating-point multiplications. As discussed in Sec. 4, DL compilers split and permutate loops for better memory locality. However, Glow failed to produce optimal, cache-friendly binary code. The loop iterates 512 times, and each time, four `ymmword` (32 bytes) will be read.

Note that lines 4–5 (and lines 8–9) read consecutive 64 bytes, precisely the size of a cache line. Thus, two cache lines will be updated each iteration. Since the DNN executable runs on a CPU with a 32 KB L1 data cache (per core), there are 512 cache lines, meaning the whole L1 data cache will be exhausted after 256 iterations. Therefore, such a self-competing memory access pattern can fill the L1 data cache twice, resulting in lots of cache misses.

Side Effects of Ineffective Optimizations. Ideally, the memory accessed during computation-intensive program execution should be restricted to a small region such that the cache hit rate could increase and the probability of competing with other programs for cache resources will be reduced. In this case, the computation does not reach its optimal performance due to less effective optimizations. However, from an attacker's perspective, such less effective optimizations instead compensate for cache side channel vulnerability that can leak DNN architectures. Specifically, the Prime+Probe attack is known to be slow; each probe iteration may take thousands of CPU cycles. The code snippet discussed above will update the whole cache in this period. The passive attacker can learn nothing else except all cache sets are visited.

Nevertheless, this seemingly secure feature is ineffective against attackers using LLC side channel attacks. Given the LLC's larger size, the non-optimal code cannot fill the LLC. Also, memory accesses are uniformly distributed to different cache sets, resulting in diffused but identifiable patterns. This explains the superior results in Table 5 and helps emphasize this work's practical significance and contribution. Beyond that, while the non-optimal code temporarily reduces the L1 cache side channel attack surfaces of Glow-emitted DNN executables, one may expect that the Glow community will gradually improve the optimization passes of its compiler, consequently "reviving" the attack in the future.

7 MITIGATION

By remotely recovering DNN architectures, DeepCache poses a significant risk to the security of DL services. Since DeepCache exploits well-studied cache side channel attacks, existing cache side channel mitigations can also defend against DeepCache. Prior works proposed system-level defenses, including partitioning cache resources [34, 51, 91], introducing noise in cache access timings [87], and hypervisor-level process scheduling [42, 68, 72, 90]. Such mitigations require system or even hardware-level modifications and cannot be easily applied from the customer side. On the other hand, software-level mitigations are transparent to users and require no system modifications that may impact other processes. We discuss potential solutions in more detail in the Appendix A.2.

8 RELATED WORK

Sec. 3.1 has reviewed relevant works launching side channels or other microarchitecture exploitations toward DNNs. This section reviews recent advances in cache side channel attacks. Recently, many advanced cache side channel attacks have been proposed. Reload+Refresh [13] detects accesses to a shared address by monitoring changes in the eviction candidate without forcing evictions on the victim's data. Prime+Scope [56] can infer fine-grained memory access patterns with a near-optimal time resolution of around 70 cycles. Prefetch+Reload [22] leverages the prefetch instruction provided by modern Intel processors to achieve almost zero error rates when monitoring access patterns. Prime+Store [33] uses transient execution to perform arbitrary manipulations of the cache state, presenting a timing-based cache attack that can sample the cache state at a rate higher than the clock rate. S^2C [86] presents a timer-less cross-core cache attack that exploits Load-Linked/Store-Conditional (LL/SC) instructions provided by Apple M1 processors, which enable direct observation of cache activities.

9 CONCLUSION

This paper reviews existing side channel attacks and illustrates the difficulties of applying them to attack DNN executables. However, we show that hardware-aware optimizations extensively employed by DL compilers result in information leakage in cache access patterns, and we design DeepCache which leverages contrastive learning and anomaly detection to precisely infer DNN model information from DNN executables. Thus, DeepCache poses a serious security challenge for the wide adoption of DNN executables.

ACKNOWLEDGEMENT

The HKUST authors were supported in part by a NSFC/RGC JRS grant under the contract N_HKUST605/23 and a RGC CRF grant under the contract C6015-23G.

REFERENCES

[1] 2023. ONNX. https://onnx.ai/.
[2] 2024. Research Artifact of DeepCache. https://github.com/monkbai/DeepCache.
[3] Ahmed Abdulaal, Zhuanghua Liu, and Tomer Lancewicki. 2021. Practical approach to asynchronous multivariate time series anomaly detection and localization. In *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*. 2485–2494.
[4] Amazon. 2023. Amazon SageMaker Neo. https://aws.amazon.com/sagemaker/neo/.
[5] Amazon. 2023. CPU Inference with Amazon EKS. https://docs.aws.amazon.com/deep-learning-containers/latest/devguide/deep-learning-containers-eks.html.
[6] Apache. 2023. TVM. https://tvm.apache.org/.
[7] Andrea Arcangeli, Izik Eidus, and Chris Wright. 2009. Increasing memory density by using KSM. In *Proceedings of the linux symposium*. Citeseer, 19–28.
[8] Ahmed Osama Fathy Atya, Zhiyun Qian, Srikanth V Krishnamurthy, Thomas La Porta, Patrick McDaniel, and Lisa Marvel. 2017. Malicious co-residency on the cloud: Attacks and defense. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 1–9.
[9] Ahmed Osama Fathy Atya, Zhiyun Qian, Srikanth V Krishnamurthy, Thomas La Porta, Patrick McDaniel, and Lisa M Marvel. 2019. Catch me if you can: A closer look at malicious co-residency on the cloud. *IEEE/ACM Transactions on Networking* 27, 2 (2019), 560–576.
[10] Julien Audibert, Pietro Michiardi, Frédéric Guyard, Sébastien Marti, and Maria A Zuluaga. 2020. Usad: Unsupervised anomaly detection on multivariate time series. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 3395–3404.
[11] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. 2019. CSI NN: reverse engineering of neural network architectures through electromagnetic side channel. In *Proceedings of the 28th USENIX Conference on Security Symposium*. 515–532.
[12] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics.. In *NDSS*.
[13] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. 2020. {RELOAD+ REFRESH}: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In *29th USENIX Security Symposium (USENIX Security 20)*. 1967–1984.
[14] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.
[15] Google Cloud. 2023. Deep Learning VM Images. https://cloud.google.com/deep-learning-vm/docs/images.
[16] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1497–1511.
[17] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. {Prime+ Abort}: A {Timer-Free}{High-Precision} L3 Cache Attack using Intel {TSX}. In *26th USENIX Security Symposium (USENIX Security 17)*. 51–67.
[18] Anuj Dubey, Rosario Cammarota, and Aydin Aysu. 2020. Maskednet: The first hardware inference engine aiming power side-channel protection. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 197–208.
[19] Paul Devadoss Ezhilchelvan and Isi Mitrani. 2015. Evaluating the probability of malicious co-residency in public clouds. *IEEE Transactions on Cloud Computing* 5, 3 (2015), 420–427.
[20] Yansong Gao, Huming Qiu, Zhi Zhang, Binghui Wang, Hua Ma, Alsharif Abuadbba, Minhui Xue, Anmin Fu, and Surya Nepal. 2024. DeepTheft: Stealing DNN Model Architectures through Power Side Channel. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE.
[21] Google. 2022. PyTorch on Google Cloud. https://opensource.googleblog.com/2022/09/accelerate-your-models-to-production-with-google-cloud-and-pytorch_0905763892.html.
[22] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. 2022. Adversarial prefetch: New cross-core cache side channel attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1458–1473.
[23] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C Snoeren, George Varghese, Geoffrey M Voelker, and Amin Vahdat. 2010. Difference engine: Harnessing memory redundancy in virtual machines. *Commun. ACM* 53, 10 (2010), 85–93.
[24] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 620–629.
[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *CVPR*. 770–778.
[26] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).
[27] Sanghyun Hong, Michael Davinroy, Yiğitcan Kaya, Dana Dachman-Soled, and Tudor Dumitraş. 2020. How to 0wn nas in your spare time. (2020).

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

[29] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[30] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, et al. 2020. Deepsniffer: A dnn model extraction framework based on learning architectural hints. In *ASPLOS*. 385–399.

[31] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[32] Animesh Jain, Shoubhik Bhattacharya, Masahiro Masuda, Vin Sharma, and Yida Wang. 2020. Efficient execution of quantized deep learning models: A compiler approach. *arXiv preprint arXiv:2006.10226* (2020).

[33] Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. 2023. The gates of time: Improving cache attacks with transient execution. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1955–1972.

[34] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. {STEALTHMEM}:{System-Level} Protection Against {Cache-Based} Side Channel Attacks in the Cloud. In *21st USENIX Security Symposium (USENIX Security 12)*. 189–204.

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

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

[37] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. 2020. Rambleed: Reading bits in memory without accessing them. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 695–711.

[38] Zhihan Li, Youjian Zhao, Jiaqi Han, Ya Su, Rui Jiao, Xidao Wen, and Dan Pei. 2021. Multivariate time series anomaly detection and interpretation using hierarchical inter-metric and temporal embedding. In *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*. 3220–3230.

[39] Zhihui Lin, Maomao Li, Zhuobin Zheng, Yangyang Cheng, and Chun Yuan. 2020. Self-attention convlstm for spatiotemporal prediction. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 11531–11538.

[40] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*. IEEE, 605–622.

[41] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 530–543.

[42] Li Liu, An Wang, WanYu Zang, Meng Yu, Menbai Xiao, and Songqing Chen. 2018. Shuffler: Mitigate cross-vm side-channel attacks via hypervisor scheduling. In *Security and Privacy in Communication Networks: 14th International Conference, SecureComm 2018, Singapore, Singapore, August 8-10, 2018, Proceedings, Part I*. Springer, 491–511.

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

[44] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019. Optimizing {CNN} model inference on {CPUs}. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 1025–1040.

[45] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, Xiaofei Xie, and Lei Ma. 2023. Decompiling x86 deep neural network executables. (2023).

[46] Jonathan Long, Evan Shelhamer, and Trevor Darrell. 2015. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3431–3440.

[47] Yunhui Long, Vincent Bindschaedler, Lei Wang, Diyue Bu, Xiaofeng Wang, Haixu Tang, Carl A Gunter, and Kai Chen. 2018. Understanding membership inferences on well-generalized learning models. *arXiv preprint arXiv:1802.04889* (2018).

[48] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[49] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling holistic deep learning compiler optimizations with {rTasks}. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 881–897.

[50] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 279–296.

[51] Soo-Jin Moon, Vyas Sekar, and Michael K Reiter. 2015. Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration. In *Proceedings of the 22nd acm sigsac conference on computer and communications security*. 1595–1606.

[52] OctoML. 2023. OctoML Raises $85M Series C to Accelerate ML Deployment for Enterprises Everywhere. https://octoml.ai/blog/octoml-raises-usd85m-series-c/.

[53] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious {Multi-Party} machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security 16)*. 619–636.

[54] ONNX. 2023. ONNX Zoo. https://github.com/onnx/models.

[55] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology–CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*. Springer, 1–20.

[56] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. 2021. Prime+ scope: Overcoming the observer effect for high-precision cache contention attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2906–2920.

[57] PyTorch. 2023. PyTorch 2.x: faster, more pythonic and as dynamic as ever. https://pytorch.org/get-started/pytorch-2.0/.

[58] Adnan Siraj Rakin, Md Hafizul Islam Chowdhuryy, Fan Yao, and Deliang Fan. 2022. Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1157–1174.

[59] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*. 199–212.

[60] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907* (2018).

[61] Lifeng Shen, Zhongzhong Yu, Qianli Ma, and James T Kwok. 2021. Time series anomaly detection with multiresolution ensemble decoding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 9567–9575.

[62] Elaine Shi. 2020. Path oblivious heap: Optimal and practical oblivious priority queue. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 842–858.

[63] Xingjian Shi, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-Kin Wong, and Wang-chun Woo. 2015. Convolutional LSTM network: A machine learning approach for precipitation nowcasting. *Advances in neural information processing systems* 28 (2015).

[64] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *2017 IEEE symposium on security and privacy (SP)*. IEEE, 3–18.

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

[66] M Smith. 2023. The case for running AI on CPUs isn't dead yet. *IEEE Spectrum* 1 (2023).

[67] Hongmei Song, Wenguan Wang, Sanyuan Zhao, Jianbing Shen, and Kin-Man Lam. 2018. Pyramid dilated deeper convlstm for video salient object detection. In *Proceedings of the European conference on computer vision (ECCV)*. 715–731.

[68] Deian Stefan, Pablo Buiras, Edward Z Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. 2013. Eliminating cache-based timing attacks with instruction-based scheduling. In *Computer Security–ESORICS 2013: 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings 18*. Springer, 718–735.

[69] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: an extremely simple oblivious RAM protocol. *Journal of the ACM (JACM)* 65, 4 (2018), 1–26.

[70] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2016. Stealing machine learning models via prediction {APIs}. In *25th USENIX security symposium (USENIX Security 16)*. 601–618.

[71] Apache TVM. 2023. Design and Architecture. https://tvm.apache.org/docs/arch/index.html.

[72] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. 2014. Scheduler-based defenses against {Cross-VM} side-channels. In *23rd USENIX security symposium (USENIX security 14)*. 687–702.

[73] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2015. A Placement Vulnerability Study in {Multi-Tenant} Public Clouds. In *24th USENIX Security Symposium (USENIX Security 15)*. 913–928.

[74] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again.. In *NDSS*.

[75] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*. 627–642.

[76] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. 2020. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 133–147.

[77] Ruoyu Wu, Taegyu Kim, Dave Jing Tian, Antonio Bianchi, and Dongyan Xu. 2022. {DnD}: A {Cross-Architecture} deep neural network decompiler. In *USENIX Security 22*. 2135–2152.

[78] Zhang Xu, Haining Wang, and Zhenyu Wu. 2015. A measurement study on co-residence threat inside the cloud. In *24th USENIX Security Symposium (USENIX Security 15)*. 929–944.

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

[80] Dingqing Yang, Prashant J Nair, and Mieszko Lis. 2023. HuffDuff: Stealing Pruned DNNs from Sparse Accelerators. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 385–399.

[81] Jimei Yang, Brian Price, Scott Cohen, Honglak Lee, and Ming-Hsuan Yang. 2016. Object contour detection with a fully convolutional encoder-decoder network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 193–202.

[82] Yuval Yarom. 2016. Mastik: A micro-architectural side-channel toolkit.

[83] Yuval Yarom and Katrina Falkner. 2014. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *USENIX security 14*. 719–732.

[84] Hongxu Yin, Pavlo Molchanov, Jose M Alvarez, Zhizhong Li, Arun Mallya, Derek Hoiem, Niraj K Jha, and Jan Kautz. 2020. Dreaming to distill: Data-free knowledge transfer via deepinversion. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 8715–8724.

[85] Honggang Yu, Haocheng Ma, Kaichen Yang, Yiqiang Zhao, and Yier Jin. 2020. Deepem: Deep neural networks model recovery through em side-channel information leakage. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 209–218.

[86] Jiyong Yu, Aishani Dutta, Trent Jaeger, David Kohlbrenner, and Christopher W Fletcher. 2023. Synchronization Storage Channels ({{{{{S2C)}}}}}: Timer-less Cache {Side-Channel} Attacks on the Apple M1 via Hardware Synchronization Instructions. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1973–1990.

[87] Rui Zhang, Xiaojun Su, Jianping Wang, Cong Wang, Wenyin Liu, and Rynson WH Lau. 2014. On mitigating the risk of cross-VM covert channels in a public cloud. *IEEE Transactions on Parallel and Distributed Systems* 26, 8 (2014), 2327–2339.

[88] Xiang Zhang, Zhigang Huo, Jie Ma, and Dan Meng. 2010. Exploiting data deduplication to accelerate live virtual machine migration. In *2010 IEEE international conference on cluster computing*. IEEE, 88–96.

[89] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2014. Cross-tenant side-channel attacks in PaaS clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 990–1003.

[90] Yinqian Zhang and Michael K Reiter. 2013. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 827–838.

[91] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. 2016. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 871–882.

[92] Yuankun Zhu, Yueqiang Cheng, Husheng Zhou, and Yantao Lu. 2021. Hermes Attack: Steal {DNN} Models with Lossless Inference Accuracy. In *USENIX Security*.

A APPENDIX

A.1 Operator Examples

Table 6 shows examples of DNN operators evaluated in this work, including their types, hyperparameters, and optimized layouts.

A.2 Software-level Mitigation

Below, we tentatively discuss using oblivious random access memory (ORAM) techniques to randomize memory accesses against side channel attacks [50, 53, 62, 69]. Moreover, we discuss potential domain-specific obfuscation techniques to randomize memory access patterns during DNN computation. The latter approach is more challenging (as it requires reverse engineering and binary code analysis) but has broader applicability. To mimic how third-party security vendors may protect (legacy) DNN executables in the wild, below, we assume the protection will be directly applied to DNN executables.

Table 6: Examples of operator types, hyperparameters, and optimized layouts.

Type	Hyperparameters	Optimized Layouts
Conv	kernel size: 3 #input channel: 512 #output channel: 512 stride: 1	[16, 1, 3, 3, 512, 32]
Conv+ReLU	kernel size: 1 #input channel: 128 #output channel: 256 stride: 2	[16, 2, 1, 1, 64, 16]
FC	#input neurons: 512 #output neurons: 1000	[125, 512, 8]
FC+ReLU	#input neurons: 25088 #output neurons: 4096	[512, 25088, 8]
Max Pool	kernel size: 3 padding size: 1 stride: 2	NA
Average Pool	kernel size: 1 padding size: 0 stride: 2	NA

Oblivious RAM. We view typical ORAM protocols, like PathORAM [69], as an out-of-box solution to shield DNN executables' memory access patterns. In general, each time the cache is accessed, PathORAM will randomly access several cache sets to deceive observers. To use PathORAM to protect DNN executables, we view each cache set as a "block" in the PathORAM protocol. PathORAM maintains a binary tree where each node denotes several blocks. Instead of directly accessing one block, it will randomly choose one path that contains the target block and access all blocks along the path from the root to the leaf. Thus, a single cache access is expended into several random accesses. However, it incurs a huge overhead. We leverage Intel Pin to simulate the PathORAM protocol. After rerunning our experiment with ORAM-protected cache access traces, we found that PathORAM can effectively protect DNN executables against cache side channel attacks. However, it incurs an overhead of at least six times more cache accesses.

Cache State Obfuscation. As a tradeoff between performance and security, we reconsider cache flushing defenses for DNN executable obfuscation. Each computation-intensive code block can be obfuscated to visit all cache sets, making its traces close to those exhibited in Fig. 13. To instrument and obfuscate DNN executables, for each computation-intensive block, extra obfuscation code could be inserted to randomly update cache sets not accessed by the original code during runtime. While the source code is unavailable, modern binary rewriting techniques [12, 16, 74–76] could be leveraged to modify the DNN executable, i.e., inserting obfuscation code. The obfuscation code in a block will randomly update cache sets that are not accessed by the original code. Therefore, similar to the case discussed in Sec. 6.4, after obfuscation, the attacker can only observe that all cache sets are accessed each time he tries to probe the cache state. Thus, the attack falls back to a random guess. Also, since we only update cache sets that are not accessed, extra cache accesses incurred by obfuscation code are limited in practice.