Demystifying Arch-hints for Model Extraction: An Attack in Unified Memory System

Zhendong Wang
University of Texas at Dallas
Richardson, Texas, USA
Zhendong.Wang@utdallas.edu

Xiaoming Zeng
University of Texas at Dallas
Richardson, Texas, USA
Xiaoming.Zeng@utdallas.edu

Xulong Tang
University of Pittsburgh
Pittsburgh, Pennsylvania, USA
tax6@pitt.edu

Danfeng Zhang
The Pennsylvania State University
University Park, Pennsylvania, USA
zhang@cse.psu.edu

Xing Hu
Institute of Computing Technology
Beijing, China
huxing@ict.ac.cn

Yang Hu
Tsinghua University
Beijing, China
hu_yang@tsinghua.edu.cn

ABSTRACT

The deep neural network (DNN) models are deemed confidential due to their unique value in expensive training efforts, privacy-sensitive training data, and proprietary network characteristics. Consequently, the model value raises incentive for adversary to steal the model for profits, such as the representative model extraction attack. Emerging attack can leverage timing-sensitive architecture-level events (i.e., Arch-hints) disclosed in hardware platforms to extract DNN model layer information accurately. In this paper, we take the first step to uncover the root cause of such Arch-hints and summarize the principles to identify them. We then apply these principles to emerging Unified Memory (UM) management system and identify three new Arch-hints caused by UM's unique data movement patterns. We then develop a new extraction attack, UMProbe. We also create the first DNN benchmark suite in UM and utilize the benchmark suite to evaluate UMProbe. Our evaluation shows that UMProbe can extract the layer sequence with an accuracy of 95% for almost all victim test models, which thus calls for more attention to the DNN security in UM system.

1. INTRODUCTION

The ever-increasing employment of machine learning (ML) technology, especially deep neural networks (DNNs), in various applications has tremendously improved these application domains' efficiency, such as computer vision [18, 20, 48], speech recognition [37, 73], natural language processing [8, 42], and etc. These ML and DNN models possess unique value, which is reflected in the expensive training efforts, privacy-sensitive training data, and proprietary network architectures and parameters. Thus, these models are usually deemed confidential as protected IPs. Consequently, these confidential models make them appealing targets to the adversary who intends to steal the models for profits [6]. The adversary can explore the model execution and infer the non-disclosed model architecture and parameters through extraction attacks. Such attack is known as ***model extraction attacks*** [1, 2, 5, 24, 40, 57], which not only destroys the confidentiality of a model and damages the IP of the owner but also benefits further attacks [36, 41].

ML and DNN models are mainly deployed in cloud with publicly accessible query interfaces/APIs, known as ML-as-a-service (MLaaS), allowing users to obtain service (e.g., predictive analytics) without accessing the black-box models. Thus, the adversary can duplicate the model functionality by exploring such attack surfaces as querying APIs [49]. Recently, with the proliferation of ML techniques in edge devices, such as autonomous driving [30, 69], model extraction attack has such effective approaches as hardware side-channel attacks (SCAs) to pry into the model's internal architecture. The prevalence of edge-deployed ML models and the pursuit of the extraction accuracy drive the adversaries to explore new attack surfaces instead of the superficial querying mode.

Prior works [1, 24, 40] demonstrate the SCAs can capture certain architectural events or hardware behaviors (e.g., bus traffic through bus snooping) during model execution. These timing-sensitive architectural events or hardware behaviors can be leveraged by the adversary to infer the DNN layer architectures and perform accurate DNN model extraction attacks. We argue that such *effective* architectural events, dubbed ***Architecture hints (Arch-hints)*** in this paper, present a new ***Hardware/Architecture-level attack surface*** for the model extraction in the edge/local deployment. Though existing work shed some light in utilizing architecture-level events and behaviors in GPU-based model extraction [24, 40, 56], these events are used in an ad-hoc manner. It still lacks a systematic exploration and formal definition of such architectural behaviors, which conceals the universality of such threat on different platforms.

In this paper, we set out to investigate prior identified Arch-hints, uncover the root cause of Arch-hints, and clearly define the Arch-hints. The key insight is that we identify that these Arch-hints essentially result from ***data movement in hardware platforms during model execution***. Nevertheless, simply caused by tractable data movement during model execution doesn't entitle an architectural event to an Arch-hint. The data movement during model execution must also exhibit distinguishable and stable patterns across the DNN model layers to make it a qualified Arch-hint.

Nowadays, the Graphics Processing Unit (GPU) has become the dominant hardware to deploy DNN applications, both in cloud and edge scenarios [11, 16, 33, 45, 59]. Also, the considerable memory footprint of DNN-based workloads and ever-increasing requirements of programming flexibility has pushed the GPU memory management on the verge of a major shift from the traditional copy-then-execute (CoE) model to the unified memory (UM) model [4, 32, 34, 43, 54, 67, 72], especially on these memory-limited edge platforms [4, 9, 54]. Based on the principles to define Arch-hints, we identify three Primary Arch-hints that are specifically caused by the data movement patterns of UM, namely *page fault latency (PFLat), page migration latency (MigLat)*, and *page migration size (MigSize)*, which exhibits distinguishable patterns on layer features and model architecture during model execution (Sec. 3.1).

We propose a metric *effectiveness_score* to validate the effectiveness of these Primary Arch-hints (Sec. 3.2) by evaluating their distributions across the DNN model layers. Then, we propose a new model extraction attack, **UMProbe**, which thoroughly explores the new Arch-hints in UM system to perform model extraction accurately (Sec. 4). We also evaluate how existing Arch-hints and their combinations with Primary Arch-hints can affect the model extraction accuracy.

Lastly, we substantially modify the Darknet framework and develop the UM implementation for a portfolio of representative DNN benchmarks. To the best of our knowledge, no UM implementations of DNN models has been published. We evaluate UMProbe performance on these benchmarks and demonstrate that UMProbe can extract victim model layer sequence with the accuracy of 95% for almost all victim models (Sec. 5). In summary, this paper makes following contributions:

- We investigate prior identified Arch-hints, uncover the root cause of these Arch-hints, and formally define the Arch-hints. Based on the definition, we identify three primary Arch-hints cause by the unique data movement patterns of GPU UM system.
- We characterize multiple Arch-hints candidates in UM system and propose a metric to quantify their effectiveness.
- The newly explored Arch-hints expose a new attack surface in UM which has not been explored before. We develop an extraction attack UMProbe based on that.
- We create the first DNN benchmark suite under UM to facilitate the related researches in the community.
- We evaluate UMProbe performance using the benchmark suite and demonstrate UMProbe's high accuracy, calling for attention to the DNN security in UM system.

2. EXTRACTING MODEL USING HARDWARE ARCHITECTURAL HINTS

2.1 Model Extraction Essential

Model extraction attacks target the ML models deployed in the cloud with publicly accessible query interfaces/APIs, which known as ML-as-a-Service (MLaaS) since its inception. The adversary tries to duplicate a functionally equivalent model by frequently querying APIs for cloud-based models. Recently, model extraction attack also extends to the ML models served on the edge and local devices with the proliferation of ML techniques in modern applications such as autonomous driving [10, 33, 44, 46, 51, 58, 59]. In this paper, we focus on this emerging trend and set out to explore the model extraction attack targeting the ML deployment in edge scenarios, which present higher threats to the models.

Attack Target: What to Steal? As a DNN model consists of network architecture, parameters and hyper-parameters, the adversary can target extracting architecture [24, 41], parameters [49], hyper-parameters [53]. Specifically, architecture indicates layer types and connections. Parameters are weights and biases that are learned during training process. Hyper-parameters are the configuration variables used to control the training process, such as learning rate, batch size, etc.

Among all these targets, the *network architecture is the most fundamental and valuable targets for a DNN model extraction* as both model parameters and hyper-parameters can be inferred with the knowledge of the model architecture [49, 53]. The adversary can even launch adversarial attack based on the extracted network architecture [24, 36]. The desired network architecture usually consists of layer number, layer types/dimension, and layer connections. The layer connection can include the sequential (e.g., VGG [48]) and the non-sequential (e.g., shortcut in ResNet [18]).

2.2 Attack Surface: How to Steal?

Application/API-Level Attack Surface: Conventionally, the adversary performs extraction attack at *application/API-level*. In this attack surface, the adversary accesses the API by querying the victim model and receives the replies. It then leverages the input-output pairs of the victim model to detect the decision boundary (i.e., the classification boundary between different classes) of the model and subsequently duplicates the model [41, 49]. However, such attack needs tons of queries and consumes significant computation resources and time [41]. Moreover, the attack can only duplicate the functionality of the model instead of being able to probe the accurate internal architecture of the model [28, 41] due to its limited access to the cloud-deployed black-box model, which can hardly satisfy adversary's appetite. Thus, new attack surface revealing accurate information on model internal architecture besides model functionality is needed.

Hardware/Architecture-Level Attack Surface: The pursuit of the extraction accuracy and the prevalence of edge-deployed ML models drive the adversaries to explore new attack surfaces instead of the superficial querying mode [21, 22, 57]. The hardware side-channel attacks (SCAs) have recently drawn attention since they can provide effective approaches to pry into the model's internal architecture. For example, [1, 24, 40] demonstrate that SCAs can obtain information that is closely correlated to the model internal archi-

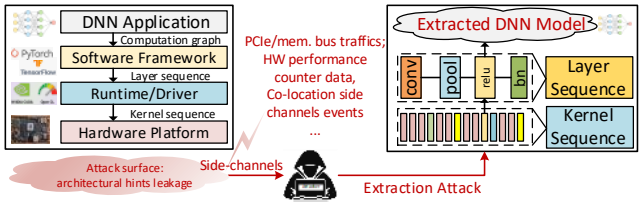


Figure 1: Extracting DNN models by exploring the attack surface provided by Architectural-hints.

tecture by capturing certain architectural events and hardware behaviors during model execution. With further data analysis, the internal model architecture could be accurately inferred based on these critical hardware architectural behaviors. We observe that such *hardware/architecture-level behavior leakage* provides a new attack surface for model extraction attack. We name this hardware/architecture-level visible information as *Architecture-hints (Arch-hints)*.

2.3 Arch-hints for Model Extraction

In this work, we take the first step to present an in-depth exploration of how Arch-hints can contribute to the extraction attacks of the edge-deployed DNN models. Specifically, we illustrate a threat model with a GPU-based DNN inference setup. We investigate prior identified Arch-hints, analyze the root cause of these Arch-hints, and define the Arch-hints. Then, we use this definition to identify two critical Arch-hints in existing unified memory management systems, which can lead to new model extraction attacks towards edge-deployed DNN models.

Fig.1 shows an abstract view of how DNN model information translates to Arch-hints during the model execution. When the DNN application is executed in the DL framework (e.g., Pytorch), the framework formalizes the DNN model into a framework-level computational graph and then transforms the computational graph into the runtime layer *execution sequence*, which is then issued to the runtime/hardware driver (e.g., CUDA, GPU driver). The runtime/hardware driver will launch a series of operational *kernel sequence* accordingly. These kernel sequences could be revealed by carefully chosen Arch-hints while executing in the hardware platform (e.g., CPU-GPU heterogeneous platforms).

The adversary typically has physical access to the victim platforms and can co-locate its spy application in the same platforms. Thus, the adversary can capture these Arch-hints by leveraging hardware SCAs. Prior works leverage architectural behaviors in model extraction attacks based on different platforms. Though these architectural behaviors share the similar functionalities as Arch-hints, they are used in an ad-hoc manner. It still lacks a systematic exploration and formal definition of such architectural behaviors.

We summarize these Arch-hints and explore the hidden principles behind them. We categorize these Arch-hints into three types: a) cache-based, b) DRAM-based, c) GPU kernel-based, as shown in Table 1. We illustrate the Arch-hints caught in GPU platforms. For instance, [40] collects the GPU memory write transactions and GPU unified cache throughput as the Arch-hints. [56] utilizes the Arch-hints including the number of GPU DRAM read/write requests and the number of GPU texture cache requests. [24] leverages Arch-hints

Table 1: Commonly-used Arch-hints in prior works.

Attack	Arch-hints Used			Platform & Mem. Model	Scenario	Approach
	Cache	Memory	Kernel			
RenderedInsecure [40]	✓	✓		GPU, CoE	Cloud/Edge	Predict
LeakyDNN [56]	✓	✓		GPU, CoE	Cloud	Predict
DeepSniffer [24]		✓	✓	GPU, CoE	Edge	Predict
DeepRecon [22]	✓			CPU, N/A	Cloud	Predict
0wnNAS [21]	✓			CPU, N/A	Cloud	Infer
StealNN [2]			✓	CPU, N/A	Cloud	Predict
Cachetelepathy [57]	✓			CPU, N/A	Cloud	Search
ReverseCNN [1]		✓	✓	FPGA, N/A	Cloud	Search

such as memory bus traffics and kernel execution latency.

We delve into these Arch-hints and observe that these Arch-hints are essentially resulted from the *data movement that exhibits in the hardware platforms* during model execution. For example, it is the data movement between GPU memory and GPU cache that causes the Arch-hint of memory bus traffic. The data movement between the GPU memory hierarchy system and GPU SMs that significantly contributes to the kernel execution latency.

While data movement is crucial to model execution exhibiting Arch-hints, we further identify that not all architectural behaviors caused by data movement can serve as effective "Arch-hints" of being leveraged in the attack. In fact, valid Arch-hints should be architectural events and hardware behaviors that present certain recognizable information for the adversary.

Based on the analysis above, Arch-hints is defined as effective architectural events and hardware behaviors that*1) being caused by tractable data movement during model execution, and 2) being able to exhibit recognizable information in extraction attack.* We will utilize the definition to explore new Arch-hints in GPU unified memory system.

3. DEMYSTIFYING ARCH-HINTS IN UNIFIED MEMORY

Unified memory (UM) has gained widely adoption today due to its efficient memory footprint and programmability. In this section, we identify two unique Arch-hints in UM management system based on the definition proposed in Sec.2.3 and validate their effectiveness.

3.1 GPU Execution and Unified Memory

We first introduce background of GPU execution model and unified memory management. As a representative vendor, NVIDIA GPU consists of several Streaming Multiprocessors (SMs), on-chip L2 cache, and high-bandwidth DRAM. All SMs share the unified L2 cache and the device memory through an on-chip interconnection network. In a typical discrete GPU setup, the GPUs are connected to the host CPU through PCIe interconnect. Note that, the discrete GPU has its own on-board physical memory which is physically separated from the CPU host memory. Since the GPU memory usually has less capacity compared to the CPU memory (e.g., 32 GB in NVIDIA V100 [65] v.s. hundreds GB of host CPU memory), the conventional GPU workload execution pattern is to copy the data from CPU memory to the GPU memory when needed, and copy the data back to CPU memory after the computation finishes. This execution model is referred to as *copy-then-execute (CoE)*.

Unified Memory Model: However, CoE execution mecha-

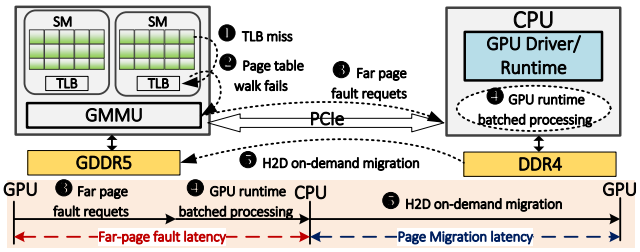


Figure 2: Far page fault and page migration in UM system.

nism suffers from i) frequent data copy between CPU and GPU [4] and ii) out-of-memory problem due to limited GPU memory capacity [13]. To address these disadvantages, *Unified Memory (UM)* model is introduced to ease GPU programming by removing the necessity of explicit data copy between the CPU and GPU [67, 68]. Specifically, UM provides the illusion of unified virtual memory space for both CPU and GPU, and allows applications to access data on both CPU memory and GPU memory through a single shared pointer in the program. In CUDA programming, the API *cudaMallocManaged*() is used to allocate UM space. Unlike CoE model which transfers the data by chunks, UM employs an on-demand paging method and transfers/migrates the data at page-level granularity. Fig.2 shows the data processing flow under UM model. At system level, the GPU's page table walk will fail if SMs try to access a physical memory page that is not currently available in GPU local memory (i.e., the address translation lookup misses in both TLB and page table, steps ① ∼ ②) and a *far page fault* exception is raised (step ③) by the GPU memory management unit (MMU) [61]. These exceptions are sent to the host CPU and handled by the driver (step ④). In particular, the driver first interrupts the CPU to handle the page faults, and then *migrates* the requested pages to the GPU memory (step ⑤) [13]. We claim the system with CoE management model as CoE system. Accordingly, UM system indicates the system with UM management model.

3.2 Arch-hints for UM System

As discussed in Sec. 2.3, the ever-explored Arch-hints for GPU platform target copy-then-execute (CoE) system. Since the memory management and data movement in UM system differ from that in CoE, we explore how this difference impacts the Arch-hints patterns and effectiveness in model extraction.

What Should be Effective Arch-hints: Extraction attack essentially explores the relation between the observed Arch-hints and the internal architectures of the victim model, and typically utilizes a training-based approach to learn the exhibited patterns and leaked information from the Arch-hints to predict the architecture, as shown in Table 1. For example, [40] utilizes the Arch-hints of memory write transactions and unified cache throughput as inputs to train classification models (e.g., KNN) to predict the victim model neuron number. [56] utilizes the Arch-hints, including the number of DRAM read/write requests and the number of texture cache requests, as inputs to train the LSTM model to predict different DNN layer types.

Since the Arch-hints work as the input feature vector of the adversary learning model, the distribution property of Arch-hints across different layers significantly impacts the model extraction performance (e.g., extraction accuracy, Sec. 5.3). It is expected that the Arch-hints distribution across different layers during model execution exhibits clear and accurate patterns. Unfortunately, some Arch-hints distribution can become blurred and inaccurate in UM system.

Existing Arch-hints May be Ineffective in UM: In fact, we observe that *the CoE platform-targeted Arch-hints can get blurred in UM system during model execution*, which potentially undermines the extraction attack. Typical cases are Arch-hints based on kernel activity or memory traffic.

For example, such a common Arch-hint as kernel latency is closely associted with kernel size. It only consists of kernel execution latency in CoE platform. Since the data has been copied into GPU memory, the kernel execution can access the data in local memory and the execution latency is stable. During model execution, each layer shows stable latency. Due to the different size, different layers show different but stable latencies. The Arch-hint shows clear and accurate distribution across different layers. In comparison, the kernel latency includes far fault latency and migration latency besides the execution latency in UM [67], and the execution latency can overlap with the other two. Thus, each layer shows in-stable and variant latency during model execution, and the Arch-hint shows blurred and inaccurate distribution across different layers. Consequently, the Arch-hint can become ineffective for distinguishing different layers.

Regarding the memory bus traffic, in CoE platform, as the data has been in GPU memory, the memory read transaction can reveal the input size of a kernel [24]. Thus, the Arch-hint of memory transaction is accurate. However, in UM, besides reading data from DRAM, a kernel can migrate large amount of data from CPU memory on demand, which will be directly loaded into GPU cache. Thus, the Arch-hint of memory transaction is inaccurate to reveal the kernel size.

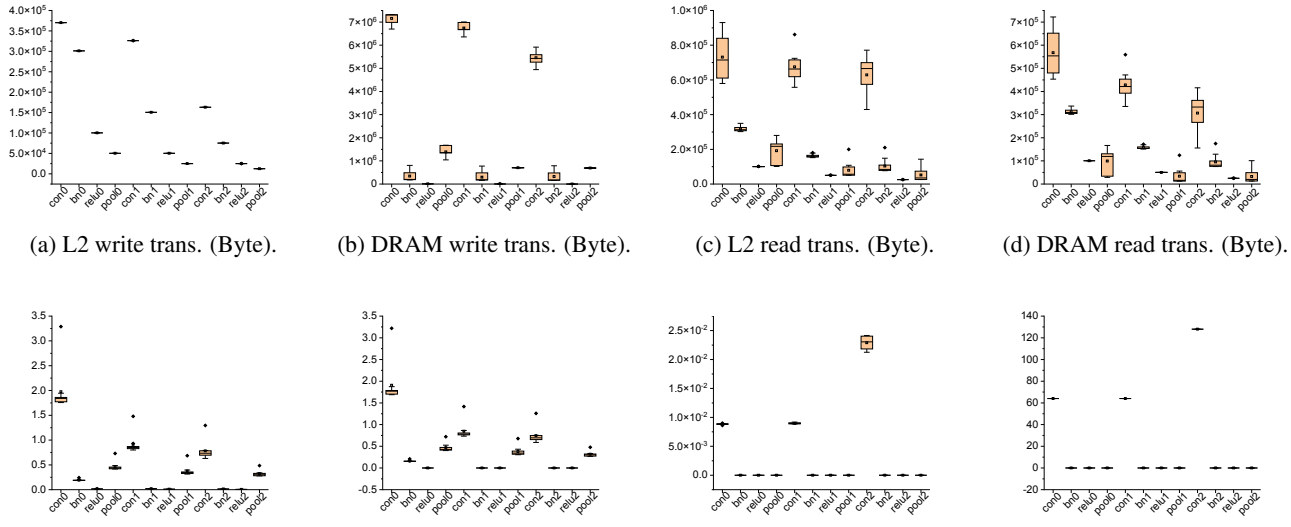
Here, we utilize the Darknet reference model as an example to illustrate our observation. We choose Darknet framework in this paper for three reasons: 1) it is open source; 2) it is written in C and CUDA, and well supported with the CUDA UM model APIs (e.g., *cudaMallocManaged*(), *cudaFree*()); 3) it provides a variety of standard pre-trained models for objective classification and detection applications. We execute the model and collect the Arch-hints on GPU platforms with UM system. Besides the commonly-used Arch-hints, we explore three candidate Arch-hints specified in UM model: *far fault latency, data migration latency, and data migration size*, as shown in Table 2. For each Arch-hint, we execute the model 10 times and collect 10 samples, and then draw the box-plot, as shown in Fig.3.

Quantifying the Effectiveness of Arch-hints: To evaluate how the Arch-hints effectiveness are undermined and the extraction attack performance is impacted, we propose metrics to quantify the effectiveness of each Arch-hint. Note that all Arch-hints can be used in extraction attack theoretically, however, an effective Arch-hint can faithfully mirror the patterns of the model execution (e.g., layer features). If the arch-hint is ineffective, it will be more difficult for the adversary to extract the victim model, for example, the adversary has to pay more observations to obtain accurate results.

We mainly evaluate an Arch-hint's effectiveness in terms

Table 2: The collected and characterized candidate Arch-hints during DNN execution in UM system.

Platform	Memory Hierarchy	Network Model	Collected candidate Arch-hints (8)
Titan RTX (GPU)	Unified Memory (UM)	Darknet Reference [71]	L2 write transaction, L2 read transaction, DRAM read transaction, DRAM write transaction, kernel latency, far fault latency, data migration latency, migration size



(a) L2 write trans. (Byte). (b) DRAM write trans. (Byte). (c) L2 read trans. (Byte). (d) DRAM read trans. (Byte).

(e) Kernel latency (ms). (f) Far fault latency (ms). (g) Migration latency (ms). (h) Migration size (KB).

Figure 3: Comprehensive characterization of distributions of different Arch-hints in UM system.

of its distribution across the NN layers. The distribution can be measured using *coefficient of variation (CoV)* from two aspects: 1) *distinguishability*, 2) *consistency*. *CoV* is a statistical measure of the dispersion of a series of data independent of the measurement unit used for the data. As different Arch-hints have different measurement units, *CoV* is useful for comparing the different Arch-hints distributions. *CoV* is calculated as the ratio of the standard deviation (σ) to the mean (μ), as shown in Equation 1. Fig.3 shows the box-plot of each Arch-hint, where the x-axis indicates the layers in Reference model and the y-axis indicates the 10-samples distribution of the Arch-hint on each layer. We detail below how the *CoV* is used to measure the *distinguishability* and *consistency* of each Arch-hint. Note that we only show early layers of the model to save space, however, the calculation is applied to all layers.

$$CoV = \frac{\sigma}{\mu} \tag{1}$$

a) Distinguishability (dis) indicates variability of the Arch-hint value among different layers during model execution. As different layers of a model (e.g., Conv, BN, Pool, etc.) have different computation complexity and dimension size, it is supposed that one Arch-hint behave differently on different layers. *distinguishability* is defined as the variability of the Arch-hint among all layers of a model and is calculated as the *CoV* of the Arch-hint values of these layers, as shown in Equation 2, where the n is total layer number. Intuitively, the larger the CoV_{dis}, the better the *distinguishability* is (i.e., the *distinguishability* positively correlates to CoV_{dis}). Then, the larger the Arch-hints difference on different layers is, the easier the adversary can distinguish different layers and explore the model internal architecture, and thus, the more effective the Arch-hint is.

$$dis = variability_{Arch-hint}(layer_1, layer_2, ..., layer_n) \tag{2}$$

b) Consistency (con) indicates the variability of each-layer Arch-hint values among the multiple samples/executions. It is expected that the Arch-hint shows consistent behaviors among the multiple samples (i.e., low variability) to provide accurate information about the model architecture, otherwise, the Arch-hint value contain great noises, increasing the difficulty for adversary to accurately extract the model architecture. As Equation 3 shows, the *consistency* is calculated as the *CoV* of each-layer Arch-hint values of the multiple samples, where i $\leq$ n. Accordingly, the lower the CoV_{con} is, the larger the Arch-hint *consistency* is (i.e., the *consistency* negatively correlates to CoV_{con}). That is, the more accurate and less-noisy information about the model architecture that Arch-hint can provide, the more effective the Arch-hint is.

$$con = variability^{layer_i}_{Arch-hint}(sample_1, sample_2, ..., sample_m) \tag{3}$$

Integratively Evaluate the Effectiveness of Arch-hints: We analyze above that an Arch-hint distribution property among different layers during model execution matters to the Arch-hint effectiveness and the Arch-hint distribution can be measured from both *distinsuishability* and *consistency*. Here, we integrate the *distinsuishability* and *consistency* of each Arch-hint, and define the *Arch-hints Effectiveness Score (ArchES)* to evaluate the overall effectiveness of an Arch-hint.

The *ArchES* is defined as *the ratio of distinsuishability to consistency*, that is, CoV_{dis}/CoV_{con} (Sec. 5.2). On one hand, the CoV_{dis} is expected to be large such that the Arch-hint behaves significantly differently on different layers, providing recognizable information on the model architecture. On the other hand, the CoV_{con} is expected to be low such that the

Arch-hint behaves consistently among multiple model executions, providing accurate and noise-less information of the model architecture. The higher *ArchES* is, the more effective the Arch-hint is. By utilizing the *ArchES*, We identify that UM system exhibits several unique and effective Arch-hints, which provides a new attack surface for adversary to extract DNN model (Sec. 4).

In fact, when we discuss the effectiveness of Arch-hints, we essentially explore whether the data movement during model execution can exhibit distinguishable and accurate patterns, which can be regarded as the leak of the model architecture information, for being learned by adversary in a given system. In conventional Copy-then-Execute (CoE) system, such commonly-used Arch-hints as memory bus traffic, kernel latency can represent the input/output data size and computation complexity of a layer accurately and distinguish different layers. However, they become blurred in UM system. Instead, some new Arch-hints can reveal the data movement pattern clearly and accurately during model execution.

4. EXTRACTING MODELS WITH ARCH-HINTS IN UM

In this section, we show how the identified Arch-hints based on page fault handling and on-demand data migration in UM system exhibit patterned information, revealing layer features and DNN characteristics. We then leverage these Arch-hints to launch an extraction attack, termed as **UMProbe**. To the best of our knowledge, this is the first model extraction attack targets unified memory system.

4.1 Threat Model and UM Arch-hints

The threat model focuses on edge security where the adversary is able to physically access the victim platform. Also, with GPU multi-instances technology [70] and GPU's support for the concurrency of multi-tenant inference applications in edge scenarios [35], the adversary can share the physical GPU platform with the victim and co-locate its application with the victim model in the GPU.

First, the adversary can utilize the PCIe bus snooping method to obtain the GPU kernel and data migration activities, which has been proven with an accuracy of ~98% in practice [62]. Specifically, the GPU activities are initialized and terminated from the host commands, which are transferred through the PCIe connection between GPU and host. Accordingly, the far-fault handling requests and on-demand page migration both cause PCIe traffic in UM system, as shown in Fig.2. By capturing these critical traffic and events related to far-fault requests and data migration [66], the adversary can obtain the Arch-hints of *Page Fault Latency (PFLat), Page Migration Latency (MigLat) and Migration Size (MigSize)* of each kernel and layer. We consider these Arch-hints to be **Primary Arch-hints** (*PriArchs*) in UM system.

We show that the primary Arch-hints can show strong patterns and are effective in leaking model information for adversary and that UMProbe is able to extract the victim DNN architecture accurately by merely leveraging these *PriArchs*.

Additionally, since the adversary and victim share the same GPU platform (e.g., the hardware cache/memory, the open deep learning library (e.g., Darknet)), as demonstrated in [39, 40], the adversary can co-locate its spy application

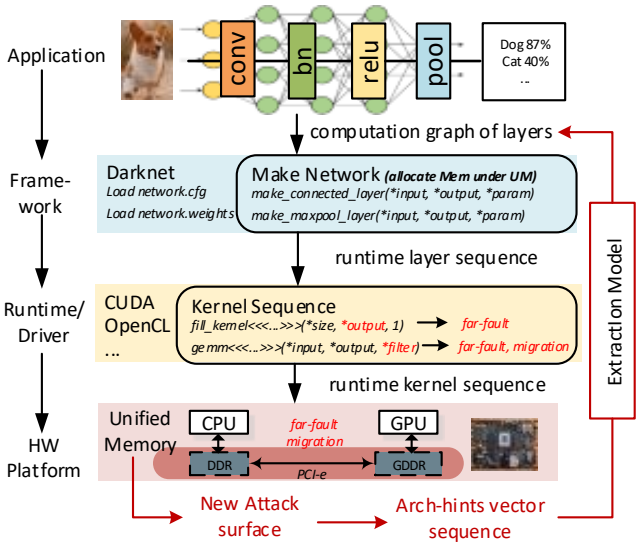


Figure 4: Overview of UMProbe.

with the victim model to obtain the victim cache properties (e.g., L2 transactions) by leveraging cache-side channels, which can achieve ~ 90% accuracy. UMProbe can collect the *common Arch-hints (ComArchs), such as L2 read and write transactions* to further enhance its extraction performance.

UMProbe Overview: After obtaining the Arch-hints, we overview how *PriArchs* exhibit patterned information to reveal different layer features and manifest model architecture.

As show in Fig.4, a DNN application issues its computation graph to the Darknet framework, and Darknet forms the graph as the connected layer sequence of the DNN architecture. Then, Darknet transforms the layer sequence into the GPU commands of runtime kernel execution sequence corresponding to the layer sequence. Finally, the kernel sequence is executed in GPU platforms, which exhibits the Arch-hints of being learned by the adversary.

Regarding Darknet framework, it mainly loads the network configuration file and parameters file (i.e., weights) after receiving the DNN execution request. Essentially, Darknet constitutes the network architecture and allocates memory space for each layer (i.e., the IFM, OFM, Filter) in UM system by calling the API *cudaMallocManaged*(). In UM system, it is lazy allocation, indicating that the physical page of the data populates in the host memory and the virtual page is invalid in the GPU side (i.e., the virtual-to-physical mapping does not exist or page valid flag is not set) after allocation completes. Thus, when the GPU SMs execute the layer and kennel sequentially, the SMs will encounter far-page fault exception if the SMs access the data region for the first time and may cause on-demand page migration.

As different layer types utilize different GPU kernels, the *PriArchs* can exhibit patterned information in leaking the different kernels characteristics and layers features. Considering the kernel sequences of different layers have a static execution order related to the original computational graph of a DNN model [24], the kernels characteristics and layers features revealed by *PriArchs* can be learned by adversary to predict the model layer sequence accurately.

Table 3: Associated kernels and Arch-hints of the typical layers in Darknet.

Layer	Kernels	PFLat	MigLat	MigSize
Conv	fill_kernel, im2col_kernel, gemmSN_kernel_nn, sgemm_xx_nn	✓	✓	✓
FC	fill_kernel, gemmSN_kernel_tn, sgemm_xx_tn, axpy_kernel,	✓	✓	✓
BN	normalize_kernel, scale_bias_kernel, add_bias_kernel			
ACT	activate_kernel (ReLu)			
Pool	forward_maxpool_kernel, forward_avgpool_kernel	✓		
Shortcut	copy_kernel, shortcut_kernel	✓		

4.2 New Attack Surface in UM

Essentially, UMProbe utilizes a learning-based model to explore the relationship between the extracted Arch-hints and victim model's internal architectures, the input Arch-hints containing the victim architecture information can reveal the victim layer features. In this section, we show how the primary Arch-hints under UM represent the different kernels' characteristics and reveal different layer features, which thus exposes a new attack surface in unified memory system for the adversary to infer victim DNN architecture.

4.2.1 Primary Arch-hints Reveal Layer Features

Primary Arch-hints Vary with Layer OFM/Filter Characteristics: As a DNN layer can be specified by its feature map (i.e., IFM, OFM) and its parameters (e.g., the Filter of a Conv layer), we observe that the primary Arch-hints of *PFLat, MigLat and MigSize* are closely associated with the feature map and parameter characteristics of a runtime layer. Table 3 shows the most-commonly-used layer types in a DNN model. By analyzing Darknet code, we identify the associated kernels of each layer. We will analyze how these kernels behave during DNN execution and how the Arch-hints can eventually reveal the kernel characteristics and layer features.

a) Conv and FC layers. The execution of a Conv layer involves multiple kernels, such as *fill_kernel*, *gemm_kernel*. Here, the kernel *fill_kernel* works to initialize the OFM region of the layer (i.e., filling value 1 in the region), which is allocated in GPU memory before the convolution operation (i.e., the kernel *gemm_kernel*) begins. To initialize the region, the GPU SMs access it for the first time after the memory being allocated, causing far-page fault handling and PFLat.

After the OFM region is initialized, the kernels *im2col_kernel* and *gemm_kernel* are launched to execute convolution operation. During *gemm_kernel* execution, the GPU SMs has to access the Filter region (i.e., storing the weights) for the first time, causing far-page fault handling and PFLat as well. Moreover, since the physical pages of Filter data populate in the remote system memory at this moment, data migration is required after the far-page fault is processed, which results in MigLat and MigSize.

For the FC layer, it almost follows the same pattern of Conv layer as they are both linear operation layers in a model. Specifically, FC layer starts with the kernel *fill_kernel* that initializes the OFM region and can cause far-page fault, and then executes the computation kernel *gemmSN_kernel_tn* that accesses the Filter region and causes both far-page fault and data migration, and ends with the kernel *axpy_kernel* that again accesses OFM region and does not cause far-page fault or migration.

Besides, although the layer implementations may vary in runtime, like a Conv layer can be implemented with a *gemmSN_kernel_nn* or a *sgemm_xx_nn* kernel (xx indicates different dimension, such as 32×32, 64×32), this makes no difference to our analysis above. These kernels always access the OFM and Filter region and result in both far-page fault handling and data migration.

b) BN and ACT layers. A Conv layer is typically followed by a BN layer and then a ACT layer. As Table 3 shows, BN layer consists of three kernels, including *normalize_kernel*, *scale_bias_kernel*, and *add_bias_kernel*, and ACT layer consists of *activate_kernel* (e.g., the most-commonly-used ReLU). When SMs execute a BN layer, the layer takes the OFM of the previous layer as its own IFM, indicating the OFM region has been accessed before and the data pages have populated in the local GPU memory. Thus, the SMs accessing the region does not cause far-page fault or data migration. Analogously, the ACT layer execution causes no far-page fault or migration.

Besides, some earlier DNN models, such as Alexnet, do not include BN layer, the ACT layer directly takes the previous Conv/FC layer OFM as its IFM. Also, modern models include BN and ACT layers in Conv layer, known as Conv-BN-ReLu block [27]. However, our analysis above is also applicable to these different models variants. Namely, both BN and ACT layers do not cause far-page fault or data migration.

c) Pooling and Shortcut layers. Pooling layer mainly involves the kernel *maxpool_kernel* or *avgpool_kernel*, which outputs the down-sampling result of the previous layer. When SMs execute the kernel, the SMs access the OFM region of the Pooling layer for the first time, causing far-page fault handling and PFLat. During the down-sampling operation, the SMs does not need other parameters [50] and does not cause data migrate. Thus, the Pooling layer is featured with far-page fault handling but no data migration.

Modern DNN models can be configured with more complex non-sequential architecture, such as the popular ResNet using shortcut connection. During runtime execution, the shortcut and the main branch are actually executed in sequence in GPU platforms [24].

In Darknet, the shortcut layer is composed of three kernels, including *copy_kernel*, *shortcut_kernel*, *activate_kernel*. The kernel *copy_kernel* is first executed to copy the identity of the IFM of the divergence point to the OFM region of shortcut layer, then the kernel *shortcut_kernel* is executed to perform addition operation in OFM region again, and finally the kernel *activate_kernel* is executed. As Pooling layer does, the shortcut layer accesses its OFM region for the first time during kernel *copy_kernel* execution and causes far-fault handling, however, it does not require additional parameters in layer execution, thus causing no data migration.

Observation-1: During DNN model execution, different types of layers perform differently and cause different patterns on page fault handling and on-demand data migration, as shown in Table 3. The Conv/FC layer is featured with both *PFLat, MigLat and MigSize*, while the layers of BN and ACT do not cause far-page fault or data migration. Both the Pooling and Shortcut layers only cause far-page fault and *PFLat*. Fig.5 shows examples of one Reference block and one ResNet18 residual block, we observe their primary Arch-hints behave as we discussed above.

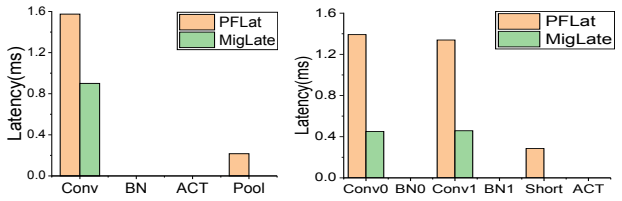


Figure 5: Layer feature from one block of a) Reference, b) ResNet18. For Conv layer, the migration causes MigSize 4608B for Reference and 2304B, 2304B for ResNet18, respectively.

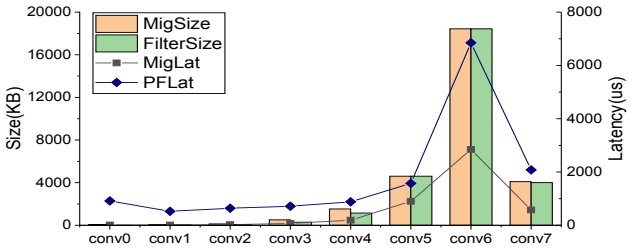


Figure 6: Arch-hints reveal Filter size in Reference model.

Primary Arch-hints Reveal Filter Size: As the Conv/FC layer is featured with the Arch-hints of far-page fault and data migration, and the data migration is mainly caused by the Filter data of the layer, we explore how the primary Arch-hints of *PFLat, MigLat, MigSize* can reveal the Filter Size characteristic. As Conv layer is the dominant layer in DNN architecture, we characterize all the Conv layers of Reference model to show the analysis.

The Filter size of a Conv layer can be calculated as $Channel_{IFM} \times Width_{Filter} \times Height_{Filter} \times Channel_{OFM} \times 4$ bytes (i.e., each data is a $float$ type in memory). As shown in Fig.6, the x-axis indicates the different Conv layers with the network going deeper, and the y-axis on left-hand side indicates the migration and Filter data size, and the y-axis on the right-hand side indicates page fault and migration latency. We observe that the MigSize is almost equal to the Filter Size, indicating that the migration is mainly caused by the Filter data. Also, with network going deeper, the Filter Size increases, and MigSize increases accordingly.

Meanwhile, the MigLat and PFLat increases as well, following the trend of Filter Size and MigSize. Intuitively, increasing MigSize causes increasing MigLat. Also, with Filter Size increasing, the SMs have to access a larger Filter data region in the memory during layer execution, which can trigger a larger amount of far page fault latency.

Observation-2: During Conv/FC layer execution, the data migration mainly result from the Filter data. Thus, the migration data size well reveals the Filter data size, and the far fault latency and migration latency both positively correlates to Filter data size.

Primary Arch-hints Reveal Layer Features: As the primary Arch-hints of PFLat, MigLat, MigSize can reveal OFM size and Filter size, we show how these primary Arch-hints can leak layer features and manifest model architecture during model execution.

We characterize the Arch-hints *PFLat, MigLat, MigSize* of each layer (e.g., Conv, Pool, FC) during Reference model execution, as shown in Fig.7. The x-axis in Fig.7 indicates

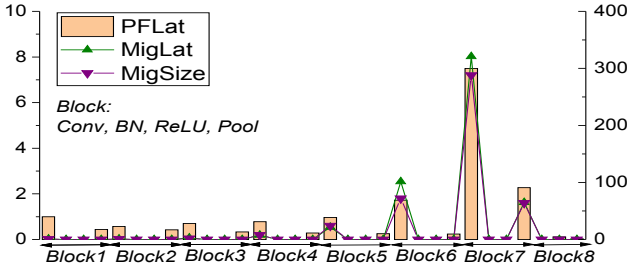


Figure 7: The Arch-hints of different blocks in Reference.

different blocks as the network going deeper, and the y-axis on the left hand indicates the latency while the right hand indicates the data size.

We observe that as the network goes deeper, the scale of *PFLat, MigLat, MigSize* increases significantly, especially for the Conv layer. The different scales can identify the different blocks. Second, the *PFLat, MigLat, MigSize* of a Conv layer are usually much larger than other types of layers (e.g., Pool, the last layer) within the same block. This is because the large Filter size and feature map size in a Conv layer cause large amounts of far page fault and data migration. Third, the BN layer and ACT layer (i.e., ReLU) exhibit quite similar execution features as they both do not cause far page fault or migration, resulting in difficulties for the adversary to accurately distinguish them.

Observation-3: The primary Arch-hints of *PFLat, MigLat, MigSize* can reveal different types of layers and blocks during model execution by identifying the layer's feature map and Filter characteristics and leak information on the model internal architecture. Thus, these Arch-hints exposes a new attack surface in UM system for extraction attack, which has not been explored before.

Common Arch-hints Further Helps: We analyze above the primary Arch-hints of *PFLat, MigLat, MigSize* can leak the layer information during model execution. However, some adjacent layers, like BN and ACT, do not cause *PFLat, MigLat, MigSize*, and thus exhibit similar execution features, causing difficulties to UMProbe. Although UMProbe can utilize DNN model design philosophy (i.e., the empiric to follow a BN and ACT layer after Conv layer) to infer these layers, we consider UMProbe exploring other Arch-hints in UM system to conquer the difficulties.

As we analyzed in Sec. 3, the L2 read/write transaction obtains a high *ArES* and is considered effective in extraction attack. Thus, UMProbe utilizes the common Arch-hints of L2 read/write transaction in the attack besides the primary Arch-hints of *PFLat, MigLat, MigSize*. Fig.3a and 3c shows the Arch-hint of L2 write/read transaction shows noticeable difference on the BN and ACT layers, indicating UMProbe can utilize the Arch-hints to further improve its extraction accuracy (Sec. 5.3).

4.2.2 *Learning-based Extraction Attack*

As We learned that the Arch-hints in UM system, especially the primary Arch-hints, can reveal layer features and leak model information during model execution, we will show how UMProbe can extract and identify victim model by learning these Arch-hints. Since model architecture, especially model layer sequence, is the most fundamental one among DNN model's properties and can be used to infer other

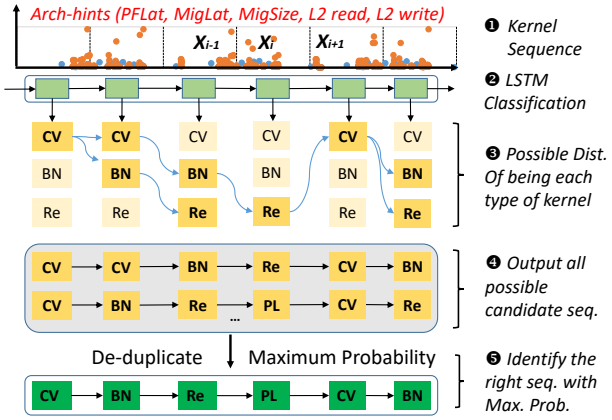


Figure 8: Scheme of layer sequence prediction in UMProbe.

parameters [23, 24, 24, 36, 41, 49, 53], UMProbe is designed to identify the model architecture as the first step to extract the model.

Attack Methodology: UMProbe adopts the Connectionist Temporal Classification (CTC) [14] model to predict victim layer sequence including layer number, types and connection, which has been proven effective in [24]. CTC is a sequence-to-sequence model, and it can be trained by minimizing the difference between the ground-truth layer sequence L^* and predicted layer sequence L, and outputs a layer sequence which is as close to the ground-truth as possible.

Fig.8 shows the specific attack methodology that includes 5 steps. ① The kernel sequence is composed of multiple kernels featured with their own Arch-hints Vectors X_i (i.e., <*PFLat, MigLat, MigSize, L2 read, L2 write*>). ② For the i_{th} kernel, its Arch-hints X_i can reveal the characteristics of the kernel. Then, UMProbe conducts the i_{th} kernel classification based on X_i by using a LSTM-classification model [24] and ③ will output a probability distribution K_i of which type of layer(i.e., Conv, ReLU, BN, Pool, etc.) those kernels belong to. ④ UMProbe utilizes the CTC model to estimate the conditional probability with the distribution of prior kernels(i.e., K_1, K_2, ... K_i). Then UMProbe outputs all of the kernel sequence candidates here, such as (CV-CV-BN-Re), (CV-BN-Re-PL), etc. ⑤ The CTC decoder eventually recognize the kernel sequence with the largest conditional possibility as the output L by utilizing greedy search and de-duplication techniques [60]. Table 3 shows each layer is associated with their own specific kernels, and Fig.4 shows a DNN layer sequence that can be mapped to the kernel sequence in runtime, thus, UMProbe can successfully predict the layer sequence by extracting and identifying the kernel sequence.

With layer sequence predicted, we then show how the layer dimension is estimated, though [24] demonstrates that the layer dimension is less important than layer sequence in extraction attack. Specifically, [24] provides a method utilizing the DRAM read transaction to estimate the input and output size of ReLU layer and other layers. Similarly, UMProbe utilizes L2 read transaction to estimate the input and output size of different layers by following the same method. Regarding GPU memory hierarchy, L2 cache read transaction can provide more accurate information to estimate the input and output size during kernel execution as the L2 cache cannot be bypassed in kernel transaction. Moreover,

Table 4: Effectiveness Evaluation of Arch-hints in UM.

Arch-hints	Distinguishability/CoV_{dis}	Consistency/CoV_{con}	ArES
L2 write trans.	1.55	0.0017	873.41
DRAM write trans.	1.56	0.22	6.84
L2 read trans.	1.71	0.46	3.72
DRAM read trans.	1.34	0.51	2.62
Kernel latency	1.81	0.11	16.59
Far fault latency	2.24	0.095	23.38
Migration latency	3.58	0.012	293.84
Migration size	3.55	0.0081	437.14

as we analyzed in Sec. 4.2 that the MigSize can reveal the Filter size of a layer (i.e., Conv/FC). That is, in UM system, the new attack surface, especially the Arch-hint of MigSize, has a advantage in estimating the Filter size of a layer.

5. EVALUATION

5.1 Experimental Setup

Platform: All sample collection, model training and validation, and attack evaluation are conducted on NVIDIA Titan RTX GPU platform. The DNN models are implemented in Darknet framework, with CUDA 10.0. We use the GPU performance counter [64] to emulate bus snooping for page fault latency, page migration latency, page migration size and L2 cache read/write transaction information collection.

Benchmarks: We use multiple pre-trained DNNs on Darknet framework [71]. The benchmark includes Sequential models (Alexnet, VGG-16, Reference, Tiny Darknet, and Extraction [71]) and Non-Sequential models (Resnet18, Resnet50, and Resnet101 [18]). It is important to emphasize that, all of the aforementioned models do *not* have specific corresponding UM implementations in public domain. We substantially modify the Darknet framework to support its execution in UM system using CUDA APIs *cudaMallocManaged*() and *cudaFree*().

Model training and deployment: Essentially, UMProbe contains a LSTM+CTC learning model to extract the victim DNN architecture. To train UMProbe, we randomly generate enough numbers of DNN models (i.e., random layer number, types, connections and dimensions) with both sequential and non-sequential connections, and utilize them as white-box models. We then execute the DNN models and collect the kernel execution samples (i.e., the Arch-hints of DNN kernel sequence) as the input to the model. After model being trained, we test UMProbe on the representative DNN benchmarks. These DNNs work as black-box models to UMProbe, and UMProbe predicts their model architectures by analyzing their Arch-hints exposed.

We collect five types of samples to train/test UMProbe, as shown in Table 5, that is, samples using Arch-hints of 1) PFLat, 2) MigSize, 3) PFLat, MigLat, MigSize (PriArchs), 4) L2 read transaction, L2 write transaction (ComArchs), 5) PFLat, MigLat, MigSize, L2 read transaction, L2 write transaction (AllArchs). As different Arch-hints represent different DNN model characteristics, we evaluate the UMProbe performance by using different Arch-hints (Sec. 5.3).

5.2 Effectiveness of Different Arch-hints

Metric: As characterized in Sec. 3, we define *Arch-hints*

Effectiveness Score (ArchES) to quantify the effectiveness of each Arch-hint in UM system in terms of *distinguishability* (CoV_{dis}) and *consistency* (CoV_{con}), as shown in Equation 4. Regarding the Arch-hint, the higher the *distinguishability* is (i.e., higher CoV_{dis}) and the stronger the *consistency* is (i.e., lower CoV_{con}), the more effective the Arch-hint is.

$$effectiveness\_score = \frac{CoV_{distinsuishability}}{CoV_{consistency}} = \frac{CoV_{dis}}{CoV_{con}} \quad (4)$$

Evaluation: We then calculate the *ArchES* of each Arch-hint as well as their *dis* and *con* factors, as shown in Table 4. We observe that the Arch-hints of L2 write trans, migration latency and size gain much higher *ArchES* than the other Arch-hints due to the high *dis* and strong *con*. When comparing the L2 transaction to the DRAM transaction, we find that their *dis* is competitive, however, the *con* of L2 write transaction is significantly lower than that of DRAM write transaction. Because of the limited capacity of L2 cache, the data is eventually written back to the DRAM. Thus, the total amount of L2 write transaction is typically capped by the L2 cache capacity and shows strong consistency. In comparison, the DRAM capacity is much larger, and there is little limit to the DRAM write transaction, thus, DRAM write transaction shows much more inconsistency. Accordingly, the L2 read transaction shows more consistency than DRAM read transaction.

Then, taking the remaining Arch-hints into consideration, the migration latency and migration size obviously outperform the other two in terms of *ArchES*. The kernel latency shows the lowest *ArchES* compared to far fault latency due to its low *dis* (i.e., low CoV_{con}). This is because, in UM system, the kernel latency is composed of execution latency, far fault latency and migration latency, and the execution latency can overlap wit far fault latency, causing overall kernel latency variable [67]. Thus, the kernel latency can get blurred in multiple execution and the Arch-hint of far fault latency can be more effective.

To summarize, we learned that in UM system the Arch-hints of L2 write transaction, L2 read transaction, far fault latency, and migration latency and size show greater effectiveness in terms of *ArchES* compared to the other Arch-hints. Essentially, *ArchES* measures the information leakage from an Arch-hint in UM system by examining the relation between the Arch-hints pattern (i.e., *distinguishability*, *consistency*) and the victim model internal architectures. As we analyze in Sec. 4.2.1, the far page fault latency is closely associated with the OFM size of almost all layers, the migration latency and size can reveal the Filter size of a layer (i.e., Conv/FC). As the three primary Arch-hints in UM provide an ever explored attack surface for adversary, we will show below that they exhibit sufficient information for UMProbe to extract the model architecture. Then, the common Arch-hints of L2 read and write transaction can further enhance UMProbe performance by providing additional information to identify such blurring layers as BN and ACT layer.

5.3 UMProbe Performance

Metric: As UMProbe targets extracting the victim DNN model layer sequence (i.e., layer number, layer types and layer connection), we measure the performance of UMProbe's DNN extraction ability by quantifying the extracted layer sequence accuracy. We define the extracted *Layer Sequence Accuracy (LSA)* as follows,

$$LSA = 1 - \frac{ED(L, L^*)}{|L^*|} \quad (5)$$

where $ED(L, L^*)$ is the edited distance between extracted layer sequence L and ground-truth layer sequence L^* (i.e. the minimum number of insertions, substitutions, or deletions required to change L into L^*) [3], while $ED(L, L^*)/|L^*|$ indicates the extracted layer sequence error rate. $|L^*|$ is the length of L^*, thus, the larger *LSA* is, the less the difference between L and L^*, and the more accurate UMProbe extraction.

Table 5: Samples using different Arch-hints.

Sample	s_1	s_2	s_3	s_4	s_5
Arch-hints	PFLat	MigSize	PriArchs	ComArchs	AllArchs

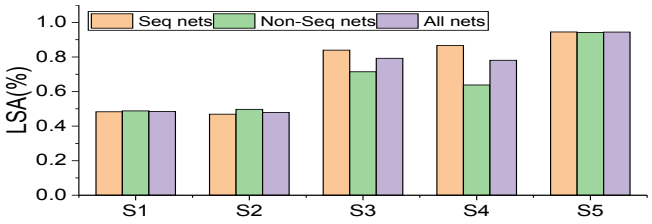


Figure 9: Avg LSA of UMProbe on different models.

Evaluation: As UMProbe works by leveraging different Arch-hints samples (see Table 5), different Arch-hints are able to reveal different DNN layer features and model characteristics, and make a great difference to UMProbe performance in terms of LSA. We measure UMProbe performance on DNN benchmarks, and further validate the importance and effectiveness of the Arch-hints in UM system.

First, we calculate the the average LSA of UMProbe using different Arch-hints on three kinds of networks (i.e., Seq nets, Non-Seq nets and all nets), as shown in Fig.9. We observe that the LSA of UMProbe by using either s_1 or s_1 is around 50%, indicating that UMProbe can effectively extract partial DNN layer sequence, though its performance is low. As we analyzed in Sec. 4.2.1, all layers except BN/ACT can cause far page fault and exhibit PFLat, which is closely associated with the OFM size of the layer, while the MigSize closely correlates to the Filter size of a layer (i.e., the dominant Conv/FC). Thus, both Arch-hints can provide effective information for adversary to infer the DNN layer sequence, but the amount of information is limited.

Then, we learned that, by using s_3 of PriArchs (i.e., PFLat, MigLat, MigSize), the UMProbe performance is obviously improved. As we analyzed above, PFLat is closely associated with a layer OFM size while MigLat/MigSize can reveal the Filter size of a layer, indicating the Arch-hints can provide complementary information about the DNN architecture. By using this three primary Arch-hints, UMProbe can effectively extract most layer sequence information.

Meanwhile, by using s_4 of ComArchs, the UMProbe performance is also improved, and is comparative to UMProbe using s_3. Regarding GPU memory hierarchy, the L2 cache is shared by all GPU SMs, while a kernel can be dispatched to multiple SMs and the kernel typically cannot bypass L2

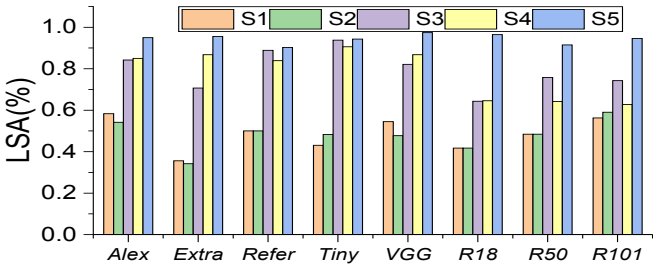


Figure 10: LSA of benchmarks using different Arch-hints.

cache to read/write data from DRAM. Thus, the L2 transaction provides relatively complete and highly distinguishable trace of data activities from different layer (i.e., input and output), as Table 4 shows the high *ArchES* of the Arch-hints. Thus, UMProbe performance using ComArchs is high as well. Based on the analysis above, we can say that as the new attack surface in UM system, the PriArchs provide sufficient information for UMProbe to effectively extract most of victim layer sequence, though UMProbe performance is not high enough.

Finally, by using s_5, the average LSA of UMProbe on Seq, Non-Seq, and all networks can reach around 95%, indicating that UMProbe can effectively extract almost all layer sequence. As we analyzed above, the PriArchs provide sufficient information for UMProbe to successfully extract layer sequence. Now, given that the ComArchs can provide additional information to further identify such blurring layers as BN/ACT, which hardly causes page fault and data migration, UMProbe performance can be further improved with the help of ComArchs.

Besides, we calculate UMProbe LSA on each DNN benchmark, as shown in Fig.10. We observe UMProbe performance on each DNN model that follows the same track of analysis above. Basically, PFLat reveals the OFM characteristics and MigSize reveals the Filter characteristics, either of them provides limited information for UMProbe ($\sim$ 50% accuracy). Then, the three primary Arch-hints together can reveal a layer's features more completely, and UMProbe performance can be significantly improved. Especially, for the small and neat networks (e.g., Alexnet, Reference, Tiny and VGG), UMProbe performance is high ($\geq$ 80%), indicating that the primary Arch-hints are able to reveal such model architecture thoroughly. Later, by using all Arch-hints, UMProbe can achieve very high performance on all models ($\geq$ 90%).

To summarize, we conclude that the new attack surface provided by the Arch-hints based on far page fault and data migration provides sufficiently effective clues for the adversary to extract most victim model architecture information in UM system and the extraction attack can achieve a high performance. Also, with Arch-hints providing additional information, the attack surface can be extended and the attack performance can be enhanced. In fact, such an attack surface has never been explored before and is worth attention.

6. RELATED WORK

Unified Memory: GPU Unified Memory (UM) arises as it effectively eliminates the need for manual data migration, reducing programmer effort and enabling GPU memory oversubscription compared to the Copy-then-Execute system. However, the far fault handling and on demand migration can significantly impact the application performance, and many prior works focusing on performance optimization [12, 13, 17, 31, 47, 54, 55, 61] have been proposed. [61] proposes a software page prefetcher to further utilize PCIe bus bandwidth and hide page migration overheads. [31] comprehensively characterizes the inefficiency of far fault handling under UM model and proposes batch-aware UM management. [13] investigates the prefeching and eviction policies under UM model and proposes new locality-aware pre-eviction policies to reduce the performance overhead. However, this paper first explores the insecure communication pattern exposed by the far fault handling and on-demand migration under UM model and exploits this attack surface in UM system for stealing DNN models.

Model Extraction Attack: The extraction attack mainly targets the ML models deployed in cloud with publicly accessible query inter-faces/APIs, and the adversary can duplicate the functionality of the model by frequently querying APIs [41, 49]. Then, some works consider utilizing side-channel information to benefit the attacks [1, 21, 22, 57], such as cache-side channel. Recently, with ML models increasingly deployed in edge/local devices [33,51,59], the adversary utilizes physical or local side-channels to obtain architecture-level information leakage to accurately extract the model architecture. [40] utilizes hardware counters to predict the NN neuron number. [56] monitors the CUPTI events in GPU platforms to infer different DNN layer operations. [1] observe the memory access patterns to search for the possible DNN structures in FPGAs. [24] collects the kernel latency, DRAM read and write volume, etc., to extract the DNN model architectures. However, none of them explores the meanings and patterns behind architecture information or proposes new architecture hints and attack surface for extraction attack in UM system.

Mitigation Countermeasures: As the new attack surface relies on insecure communication pattern between GPU and CPU on PCIe bus, one potential defense approach is to obfuscate the communication pattern on PCIe bus. As GPU runtime process the far page fault first and then migrate data on demand, the runtime can dynamically obfuscate the requests, like postpone or even reorder some far fault requests. Also, the runtime/system can support transmitting dummy data to cover the real traffic, for example, the migrated data can be split/padded into a fixed size and be sent at fixed rate [25]. This way, the PCIe transmission and leaky communication pattern in UM system can obfuscated and interfered. However, such approaches will unavoidably incur significant PCIe bandwidth overhead and performance degradation.

Besides, GPU trust execution environment (TEE) can be considered to mitigate or eliminate co-location side channel [26, 38, 52]. These TEE disallows different tenants to share the underlying hardware or execute concurrently, which can prevent the adversary to observe the victim activities through the performance counter, etc. Similarly, this method can negatively impact the GPU performance and is non-trivial to be deployed in practice.

7. CONCLUSION

Emerging extraction attack can leverage architecture-level events (i.e., Arch-hints) in hardware platforms to extract DNN model layer information accurately. In this paper, we uncover the root cause of such Arch-hints and summarize the principles to identify them. We then apply these principles to emerging Unified Memory (UM) management system, identify three new Arch-hints, and develop a new extraction attack, UMProbe. We also create the first DNN benchmark suite in UM and utilize the benchmark suite to evaluate UMProbe. Evaluation shows UMProbe can extract the layer sequence with an accuracy of 95% for almost all victim test models, calling for more attention to the DNN security in UM system.

REFERENCES

[1] Hua, W., Zhang, Z., & Suh, G. E. (2018, June). Reverse engineering convolutional neural networks through side-channel information leaks. In 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC) (pp. 1-6). IEEE.

[2] Duddu, V., Samanta, D., Rao, D. V., & Balas, V. E. (2018). Stealing neural networks via timing side channels. arXiv preprint arXiv:1812.11720.

[3] Abu-Aisheh, Z., Raveaux, R., Ramel, J. Y., & Martineau, P. (2015, January). An exact graph edit distance algorithm for solving pattern recognition problems. In 4th International Conference on Pattern Recognition Applications and Methods 2015.

[4] Bateni, S., Wang, Z., Zhu, Y., Hu, Y., & Liu, C. (2020, April). Co-optimizing performance and memory footprint via integrated cpu/gpu memory management, an implementation on autonomous driving platform. In 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) (pp. 310-323). IEEE.

[5] Batina, L., Bhasin, S., Jap, D., & Picek, S. (2019). CSINN: Reverse engineering of neural network architectures through electromagnetic side channel. In 28th USENIX Security Symposium (USENIX Security 19) (pp. 515-532).

[6] Chen, K., Guo, S., Zhang, T., Xie, X., & Liu, Y. (2021, May). Stealing deep reinforcement learning models for fun and profit. In Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (pp. 307-319).

[7] Ciregan, D., Meier, U., & Schmidhuber, J. (2012, June). Multi-column deep neural networks for image classification. In 2012 IEEE conference on computer vision and pattern recognition (pp. 3642-3649). IEEE.

[8] Collobert, R., & Weston, J. (2008, July). A unified architecture for natural language processing: Deep neural networks with multitask learning. In Proceedings of the 25th international conference on Machine learning (pp. 160-167).

[9] Dashti, M., & Fedorova, A. (2017, June). Analyzing memory management methods on integrated CPU-GPU systems. In Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management (pp. 59-69).

[10] Feng, D., Haase-Schütz, C., Rosenbaum, L., Hertlein, H., Glaeser, C., Timm, F., ... & Dietmayer, K. (2020). Deep multi-modal object detection and semantic segmentation for autonomous driving: Datasets, methods, and challenges. IEEE Transactions on Intelligent Transportation Systems, 22(3), 1341-1360.

[11] Fowers, J., Ovtcharov, K., Papamichael, M., Massengill, T., Liu, M., Lo, D., ... & Burger, D. (2018, June). A configurable cloud-scale DNN processor for real-time AI. In 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA) (pp. 1-14). IEEE.

[12] Gandhi, J., Basu, A., Hill, M. D., & Swift, M. M. (2014, December). Efficient memory virtualization: Reducing dimensionality of nested page walks. In 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (pp. 178-189). IEEE.

[13] Ganguly, D., Zhang, Z., Yang, J., & Melhem, R. (2019, June). Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory. In Proceedings of the 46th International Symposium on Computer Architecture (pp. 224-235).

[14] Graves, A., Fernández, S., Gomez, F., & Schmidhuber, J. (2006, June). Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In Proceedings of the 23rd international conference on Machine learning (pp. 369-376).

[15] Graves, A., Mohamed, A. R., & Hinton, G. (2013, May). Speech recognition with deep recurrent neural networks. In 2013 IEEE international conference on acoustics, speech and signal processing (pp. 6645-6649). Ieee.

[16] Gupta, U., Hsia, S., Saraph, V., Wang, X., Reagen, B., Wei, G. Y., ... & Wu, C. J. (2020, May). Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA) (pp. 982-995). IEEE.

[17] Hao, Y., Fang, Z., Reinman, G., & Cong, J. (2017, February). Supporting address translation for accelerator-centric architectures. In 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA) (pp. 37-48). IEEE.

[18] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

[19] He, Y., Meng, G., Chen, K., Hu, X., & He, J. (2020). Towards security threats of deep learning systems: A survey. IEEE Transactions on Software Engineering.

[20] Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A. R., Jaitly, N., ... & Kingsbury, B. (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. IEEE Signal processing magazine, 29(6), 82-97.

[21] Hong, S., Davinroy, M., Kaya, Y., Dachman-Soled, D., & Dumitraş, T. (2020). How to 0wn NAS in your spare time. arXiv preprint arXiv:2002.06776.

[22] Hong, S., Davinroy, M., Kaya, Y., Locke, S. N., Rackow, I., Kulda, K., ... & Dumitraş, T. (2018). Security analysis of deep neural networks operating in the presence of cache side-channel attacks. arXiv preprint arXiv:1810.03487.

[23] Hu, X., Liang, L., Deng, L., Ji, Y., Ding, Y., Du, Z., ... & Xie, Y. (2021). A systematic view of leakage risks in deep neural network systems.

[24] Hu, X., Liang, L., Li, S., Deng, L., Zuo, P., Ji, Y., ... & Xie, Y. (2020, March). Deepsniffer: A dnn model extraction framework based on learning architectural hints. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 385-399).

[25] Hunt, T., Jia, Z., Miller, V., Rossbach, C. J., & Witchel, E. (2019, May). Isolation and beyond: Challenges for system security. In Proceedings of the Workshop on Hot Topics in Operating Systems (pp. 96-104).

[26] Hunt, T., Jia, Z., Miller, V., Szekely, A., Hu, Y., Rossbach, C. J., & Witchel, E. (2020). Telekine: Secure Computing with Cloud GPUs. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20) (pp. 817-833).

[27] Ioffe, S., & Szegedy, C. (2015, June). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In International conference on machine learning (pp. 448-456). PMLR.

[28] Jagielski, M., Carlini, N., Berthelot, D., Kurakin, A., & Papernot, N. (2020). High accuracy and high fidelity extraction of neural networks. In 29th USENIX security symposium (USENIX Security 20) (pp. 1345-1362).

[29] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., ... & Yoon, D. H. (2017, June). In-datacenter performance analysis of a tensor processing unit. In Proceedings of the 44th annual international symposium on computer architecture (pp. 1-12).

[30] Kato, S., Takeuchi, E., Ishiguro, Y., Ninomiya, Y., Takeda, K., & Hamada, T. (2015). An open approach to autonomous vehicles. IEEE Micro, 35(6), 60-68.

[31] Kim, H., Sim, J., Gera, P., Hadidi, R., & Kim, H. (2020, March). Batch-aware unified memory management in GPUs for irregular workloads. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 1357-1370).

[32] Landaverde, R., Zhang, T., Coskun, A. K., & Herbordt, M. (2014, September). An investigation of unified memory access performance in CUDA. In 2014 IEEE High Performance Extreme Computing

Conference (HPEC) (pp. 1-6). IEEE.

[33] Li, H., Ota, K., & Dong, M. (2018). Learning IoT in edge: Deep learning for the Internet of Things with edge computing. IEEE network, 32(1), 96-101.

[34] Li, W., Jin, G., Cui, X., & See, S. (2015, May). An evaluation of unified memory technology on nvidia gpus. In 2015 15th IEEE/ACM international symposium on cluster, cloud and grid computing (pp. 1092-1098). IEEE.

[35] Liang, Q., Shenoy, P., & Irwin, D. (2020). AI on the edge: Rethinking AI-based IoT applications using specialized edge architectures. arXiv preprint arXiv:2003.12488.

[36] Liu, Y., Chen, X., Liu, C., & Song, D. (2016). Delving into transferable adversarial examples and black-box attacks. arXiv preprint arXiv:1611.02770.

[37] Malik, M., Malik, M. K., Mehmood, K., & Makhdoom, I. (2021). Automatic speech recognition: a survey. Multimedia Tools and Applications, 80(6), 9411-9457.

[38] Wang, Z., Wang, R., Jiang, Z., Tang, X., Yin, S., & Hu, Y. (2021, November). Towards a Secure Integrated Heterogeneous Platform via Cooperative CPU/GPU Encryption. In 2021 IEEE 30th Asian Test Symposium (ATS) (pp. 115-120). IEEE.

[39] Naghibijouybari, H., Khasawneh, K. N., & Abu-Ghazaleh, N. (2017, October). Constructing and characterizing covert channels on gpgpus. In 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) (pp. 354-366). IEEE.

[40] Naghibijouybari, H., Neupane, A., Qian, Z., & Abu-Ghazaleh, N. (2018, October). Rendered insecure: Gpu side channel attacks are practical. In Proceedings of the 2018 ACM SIGSAC conference on computer and communications security (pp. 2139-2153).

[41] Oh, S. J., Schiele, B., & Fritz, M. (2019). Towards reverse-engineering black-box neural networks. In Explainable AI: Interpreting, Explaining and Visualizing Deep Learning (pp. 121-144). Springer, Cham.

[42] Otter, D. W., Medina, J. R., & Kalita, J. K. (2020). A survey of the usages of deep learning for natural language processing. IEEE transactions on neural networks and learning systems, 32(2), 604-624.

[43] Otterness, N., Yang, M., Rust, S., Park, E., Anderson, J. H., Smith, F. D., ... & Wang, S. (2017, April). An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In 2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) (pp. 353-364). IEEE.

[44] Ravindran, R., Santora, M. J., Faied, M., & Fanaei, M. (2019, December). Traffic sign identification using deep learning. In 2019 International Conference on Computational Science and Computational Intelligence (CSCI) (pp. 318-323). IEEE.

[45] Romero, F., Li, Q., Yadwadkar, N. J., & Kozyrakis, C. (2019). Infaas: Managed & model-less inference serving. arXiv preprint arXiv:1905.13348.

[46] Schoettle, B. (2017). Sensor fusion: A comparison of sensing capabilities of human drivers and highly automated vehicles. University of Michigan.

[47] Shin, S., Cox, G., Oskin, M., Loh, G. H., Solihin, Y., Bhattacharjee, A., & Basu, A. (2018, June). Scheduling page table walks for irregular GPU applications. In 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA) (pp. 180-192). IEEE.

[48] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

[49] Tramèr, F., Zhang, F., Juels, A., Reiter, M. K., & Ristenpart, T. (2016). Stealing machine learning models via prediction APIs. In 25th USENIX security symposium (USENIX Security 16) (pp. 601-618).

[50] Vasudev, R. (2019). Understanding and Calculating the number of Parameters in Convolution Neural Networks (CNNs).

[51] Verhelst, M., & Murmann, B. (2020). Machine learning at the edge. NANO-CHIPS 2030, 293-322.

[52] Volos, S., Vaswani, K., & Bruno, R. (2018). Graviton: Trusted Execution Environments on GPUs. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18) (pp. 681-696).

[53] Wang, B., & Gong, N. Z. (2018, May). Stealing hyperparameters in machine learning. In 2018 IEEE symposium on security and privacy (SP) (pp. 36-52). IEEE.

[54] Wang, Z., Jiang, Z., Wang, Z., Tang, X., Liu, C., Yin, S., & Hu, Y. (2020). Enabling Latency-Aware Data Initialization for Integrated CPU/GPU Heterogeneous Platform. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 39(11), 3433-3444.

[55] Wang, Z., Wang, Z., Liu, C., & Hu, Y. (2020). Understanding and tackling the hidden memory latency for edge-based heterogeneous platform. In 3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20).

[56] Wei, J., Zhang, Y., Zhou, Z., Li, Z., & Al Faruque, M. A. (2020, June). Leaky dnn: Stealing deep-learning model secret with gpu context-switching side-channel. In 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (pp. 125-137). IEEE.

[57] Yan, M., Fletcher, C. W., & Torrellas, J. (2020). Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In 29th USENIX Security Symposium (USENIX Security 20) (pp. 2003-2020).

[58] Yang, M., Wang, S., Bakita, J., Vu, T., Smith, F. D., Anderson, J. H., & Frahm, J. M. (2019, April). Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge. In 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) (pp. 305-317). IEEE.

[59] Yazici, M. T., Basurra, S., & Gaber, M. M. (2018). Edge machine learning: Enabling smart internet of things applications. Big data and cognitive computing, 2(3), 26.

[60] Zenkel, T., Sanabria, R., Metze, F., Niehues, J., Sperber, M., Stüker, S., & Waibel, A. (2017). Comparison of decoding strategies for ctc acoustic models. arXiv preprint arXiv:1708.04469.

[61] Zheng, T., Nellans, D., Zulfiqar, A., Stephenson, M., & Keckler, S. W. (2016, March). Towards high performance paged memory for GPUs. In 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA) (pp. 345-357). IEEE.

[62] Zhu, Y., Cheng, Y., Zhou, H., & Lu, Y. (2021). Hermes attack: Steal DNN models with lossless inference accuracy. In 30th USENIX Security Symposium (USENIX Security 21).

[63] NVIDIA. Profiler: CUDA Toolkit Documentation. https://developer.nvidia.com/cuda-toolkit

[64] Nvidia. CUDA Profiling Tools Interface. https://docs.nvidia.com/cupti/Cupti/index.html. 2020.

[65] NVIDIA V100 TENSOR CORE GPU. https://www.nvidia.com/en-us/data-center/v100/.

[66] Protocol Analyzer - PCI Express - Teledyne LeCroy, 2019. https://teledynelecroy:com/protocolanalyzer/pciexpress

[67] Mark Harris. Unified Memory for CUDA Beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[68] Radeons Next-generation Vega Architecture. https://www.techpowerup.com/gpu-specs/docs/amd-vega-architecture.pdf.

[69] Apollo, Baidu. Apollo open platform. https://developer.apollo.auto/developer.html.

[70] SHANKAR CHANDRASEKARAN. Ride the Fast Lane to AI Productivity with Multi-Instance GPUs. https://blogs.nvidia.com/blog/2020/05/14/multi-instance-gpus/.

[71] Joseph Redmon. ImageNet Classification. https://pjreddie.com/darknet/imagenet/.

[72] Wang, Z., Wang, Z., Liu, C., & Hu, Y. (2020). Understanding and tackling the hidden memory latency for edge-based heterogeneous platform. In 3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20).

[73] Graves, A., Mohamed, A. R., & Hinton, G. (2013, May). Speech recognition with deep recurrent neural networks. In 2013 IEEE international conference on acoustics, speech and signal processing (pp. 6645-6649). Ieee.