EXTRACTING NEURAL NETWORK MODELS VIA CONTENTION-BASED SIDE CHANNEL ATTACKS ON SHARED MEMORY SYSTEM-ON-CHIPS

by

Alexander W. Cieslewicz

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Master of Science (Computer Science).

Golden, Colorado

Date ____________________________

Signed: ____________________________

Alexander W. Cieslewicz

Signed: ____________________________

Dr. Mehmet E. Belviranli
Thesis Advisor

Golden, Colorado

Date ____________________________

Signed: ____________________________

Dr. Iris Bahar
Department Head
Department of Computer Science

ABSTRACT

Shared Memory System-on-Chip (SM-SoC) devices are used in a multitude of environments in order to execute sensitive and critical operations. Some of these operations include the execution of deep neural networks (DNN). Several side-channel attacks that extract neural network information have previously been proposed. However the side-channel vector used by these attacks assumes a high level of access to the target system.

In this work, we propose a novel side-channel attack for SM-SoCs used in mobile platforms. Our attack relies on a unique memory contention leakage detection (MCLD) mechanism that minimizes the level of privilege an attacker requires to execute a DNN extraction attack. MCLD generates an artificial memory traffic on the CPU and observes the contention exerted on the shared memory bus in order to gather information about a target process. MCLD's implementation requires no physical access or elevated permissions on the target system. Using MCLD, the paper further implements and end-to-end DNN model used to extract the information from the victim DNN. Our experimental results performed on a state-of-the-art mobile/edge SM-SoC and popular neural networks showed that our proposed scheme can predict the neural network topology of critical workloads with average layer error rate, i.e. percentage of mispredicted layers, of 5%.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Bandwidth	BW
External Memory Controller	EMC
Memory Contention Leakage Detection	MCLD
Shared Memory System on a Chip	SM-SoC
System on a Chip	SoC

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CCF-2124010. Any opinions, findings, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

I would like to thank and acknowledge Dr. Mehmet Belviranli and Dr. Selçuk Köse for their guidance and support. My thesis committee for their time and making my defense enjoyable. Lastly, I would like to thank my family for their support and patience.

CHAPTER 1

INTRODUCTION

System on a Chip (SoC) architectures represent a significant development for many computational devices. Unlike traditional designs, SoCs provide a cheaper, more energy efficient, and smaller footprint that make it suitable for many domains [1]. In addition to their adoption, several SoC innovations, have lead to significant improvements in their performance. One example of such an improvement, is the introduction of a shared or unified memory architecture.

A shared memory architecture allows the central processing unit (CPU) and any other domain specific processors, such as a graphics processing unit (GPU), to directly access and utilize a single shared memory unit. Unlike discrete memory architectures, where each different type of processing unit operates on its own private memory, a shared memory system can reduce chip area dedicated to private memory units. Apart from conserving space a shared memory system can provide performance improvements by eliminating data transfer overhead between system processors.

These benefits have resulted in a significant adoption of shared memory SoCs (SM-SoCs). Several manufacturers produce SM-SoCs including: Apple's M1, which is found in many mobile devices and NVIDIA's Xavier Tegra, which has found adoption in edge computing environments [2, 3]. The sizable adoption of this architecture, means that SM-SoCs are handling many resource heavy and system-critical operations such as object recognition, bio-metric authentication, and simultaneous localization and mapping (SLAM). However, these critical tasks are not the only functionality that these devices need to provide. For instance an M1 device, such as an iPhone, not only needs to run IOS but also any user apps that may have been downloaded through the Apple App Store. Concurrent execution of critical and non-critical applications, results in indirect interaction between different processes [4]. The interactions can leak information about a process that would normally not be known. An extra information vector produced by a process is often referred to as a side-channel.

Side-channel attacks have been used in many scenarios to exploit target applications or systems. Spectre and Meltdown attacks utilized a cache based side channel in order to leak memory contents of other processes [5, 6]. A more specific domain, targeted by side-channel attacks, is that of Deep Neural Networks (DNN). DNNs are machine learning techniques designed for recognition and classification tasks. These capabilities have made DNNs extremely useful in computer vision and natural language processing. More generally, DNNs are used in many critical applications such as biometric authentication, autonomous driving, and speech recognition. The architecture of a DNN, specifically the layer types, weights and topology, are key in determining how a DNN will operate. As such, extracting this information enables an attacker to reveal information on intellectual property (IP) and/or affect the operation of critical workload execution.

Several prior works have utilized side-channel information in order to extract a DNN's architecture during inference. Attacks such as Cache Telepathy [7], utilize a cache based side channel during the CPU execution of a DNN in order to estimate layer dimensions. The Reverse Engineering CNNs attack utilizes a memory bus side channel in order to determine layer dimensions [8]. Recently, DeepSniffer proposed utilizing several side channel features in order to predict a DNN's layer execution sequence [9].

While these previous works provide numerous improvements to the extraction of DNN model architectures, they contain several limitations. These limitations pertain to the assumed level of access the attacker has to a system, which can be enumerated as follows. 1) Assumed access to system performance counters: it is assumed that system performance counters are available, allowing the attacker to gain exact information about cache and memory. 2) Physical Access to the System: it is assumed that the attacker can physically probe or snoop the system. 3) Non-SoC system: it is assumed that certain buses are accessible for analysis. These assumptions constrain the possible utility and applicability of the proposed extraction attacks. Reducing these constraints increases the effectiveness of DNN information extraction attacks.

This work proposes a side-channel vector that is capable of targeting SM-SoCs. It is a non-privileged side-channel attack that makes limited assumptions about potential accesses to the system, and utilizes memory contention on a shared memory system, to extract information about a running victim process. This information can be utilized in order to identify processes, as well as gain insight into the memory characteristics of a target process. In order to verify the utility of this side-channel, a DNN extraction attack is performed on traces collected from the side-channel. In summary, this work makes the following contributions:

- We demonstrate that critical workloads leave a unique memory access trace, which can be detected with no elevated privileges or hardware access on shared memory SoCs.
- We propose a novel memory-contention-based side channel attack, which extracts the memory access signature of the victim application by measuring the shared memory contention.
- We build an end-to-end recurrent neural network based extraction model that utilizes the signatures gathered from the side-channel attack to find the complete layer-topology of a victim neural network.
- We demonstrate that our proposed scheme can predict the neural network topology of critical workloads with average layer error rate, i.e., percentage of miss-predicted layers, of 5% .

CHAPTER 2

BACKGROUND

The goal of this chapter is to provide background, so that the motivations and objectives of this work can be understood. As previously discussed the objective of this work is to create a SM-SoC side-channel vector and use it to conduct an end-to-end DNN model extraction attack. In the upcoming sections the following is addressed: the target architecture, the side channel vector and the targeted information. This information, provides the necessary knowledge in order to understand how and why this work was conducted.

2.1 Shared Memory System on a Chip (SM-SoC)

System on a Chip (SoC) devices have become an increasingly common platforms for many existing computing devices. They have numerous benefits over traditional systems including: smaller size, lower production cost, and increased power efficiency. This makes such devices suitable for a wide assortment of applications with a focus on mobile and edge computing applications. With their commonality in many computational environments, several improvements to SoC designs have been developed. One significant improvement is that of a shared memory (SM) architectures. Unlike traditional architectures, where each device has its own private memory, SM architectures allow any number of processors or accelerators on a system to utilize a single shared memory unit. A SM architecture has many hardware benefits over the non-SM counterpart, including a smaller footprint size and lower power usage. Additionally, unlike non-SM architectures where data needs to be copied to an accelerator's private memory before computations, in a SM architecture the already allocated memory can simply be shared with another accelerator. This provides a significant reduction in the memory copy overhead that is seen normally.

The benefits of SM-SoCs has resulted in their significant adoption by many hardware manufacturers. Platforms such as Apple's M1 and NVIDIA's Xavier Tegra all utilize an SM-SoC architecture [2, 3]. This wide adoption means that SM-SoCs are being utilized in order to execute a wide range of both critical and non-critical applications. Concurrent execution of both critical and non-critical applications can allow a non-critical application to gain information about a critical process. Due to their ubiquity, such an attack on SM-SoCs would enable an individual to target a wide range of platforms and devices, including smartphones.

2.2 Memory Footprint Leakage in SM-SoCs

In order for a process to gain information about another concurrent process, the system needs to have some information leakage vector. Any extra information that is produced during the execution of a program is referred to as side-channel information. Side channel information can take on a wide range of forms from cache hit rates to power draw. The following sections discuss: a memory bandwidth based side channel information vector and how this information can be measured on SM-SoCs.

2.2.1 Memory Bandwidth Footprint

A running process will need to access memory to load both data and instructions. During the execution of a program the amount of data that the process needs to load will vary. For instance, the compute intensive portion of a program would access less data than the memory intensive portion. Therefore, the memory utilization of a process will vary throughout its execution. As such, measuring the bandwidth utilization of a process produces a unique process signature. Figure 2.1 shows the bandwidth utilization of VGG-16, a popular convolutional neural network for image recognition. As can be seen in the figure, each layer type results in a detectable unique memory bandwidth utilization. If the memory bandwidth usage of the entire network can be measured, information about each layer can be extracted. This information at a minimum includes which portions of a process are memory/compute intensive as well as a method of identifying which process is running. Thus the memory bandwidth utilization of a program is a side channel that leaks information.

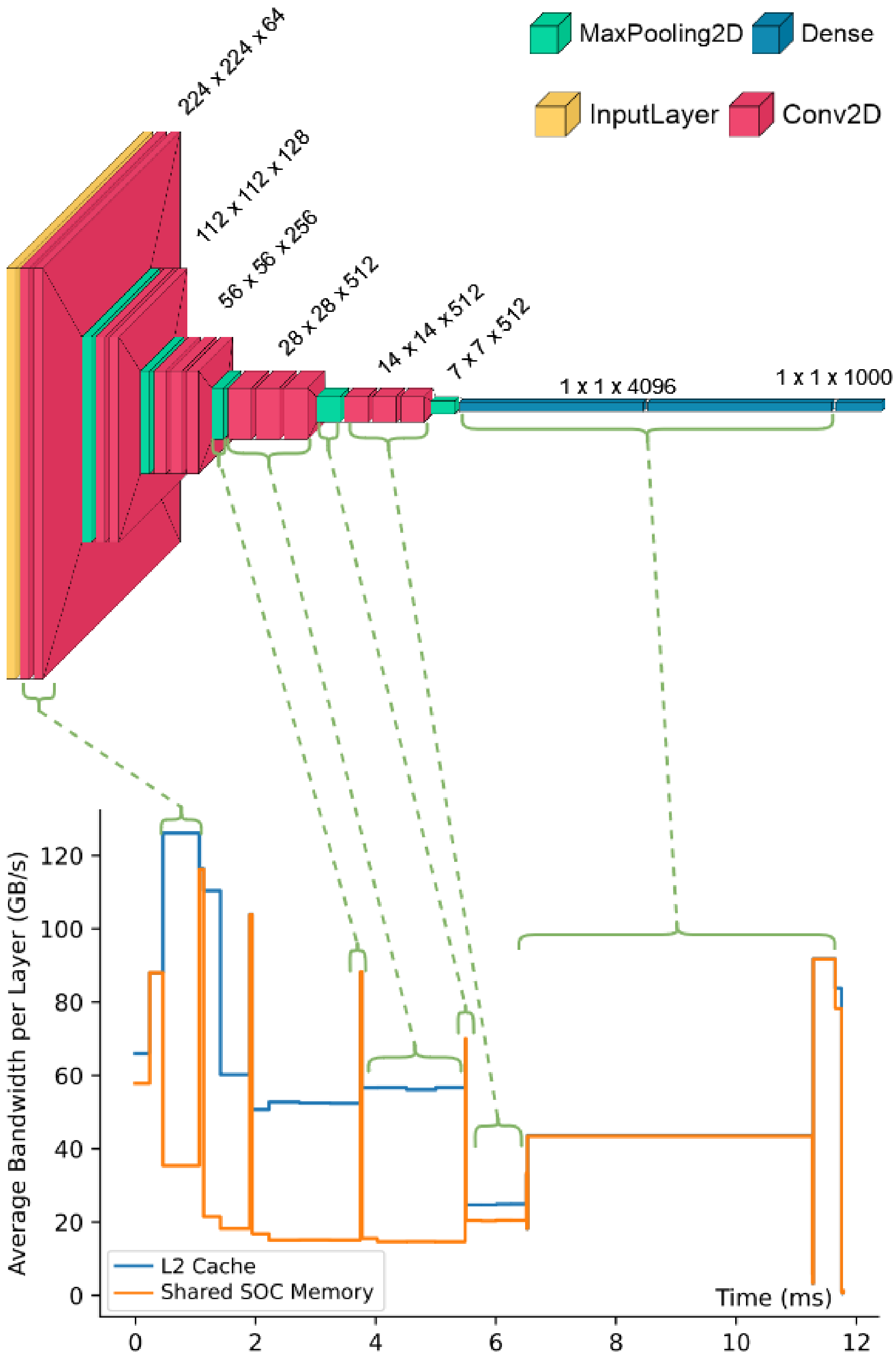

Figure 2.1 In the top figure, layer topology and input sizes are shown for VGG-16 (Image generated using VisualKeras [10]). In the bottom figure, the memory accesses bandwidth of the layers in this networkm when executed on NVIDIA Xavier AGX's GPU, is given. X-axis represents the execution timeline of each layer and y-axis represents the average bandwidth of the data retrieved from GPU L2 cache (blue line) and shared SOC-memory (orange line) throughout the execution of the GPU kernel corresponding to each layer. Dashed green arrows are sampled mappings between the results and the corresponding layers of the network being executed. The measurement is performed via NVIDIA NSight Compute [11]

2.2.2 Memory Contention Based Bandwidth Detection

As discussed in Section 2.2.1, the memory bandwidth signature produced by a process during its execution leaks information about the running process. While the memory behavior can be trivially measured with a profiler, such a program may not be available or accessible on a system. There is a need for a low privilege memory bandwidth measurement method on such devices. In modern SM-SoCs, such as NVIDIA's Xavier AGX, every CPU and accelerator has access to a single shared system memory. While these type of architectures provides numerous benefits as outlined in Section 2.1, they also come with a set of trade-offs. Most significantly, the shared memory unit means the data traffic from all processors and accelerators needs to pass through a single memory controller. This memory controller serves as a bottleneck in the system. Prior work has shown that simultaneous access from a GPU and CPUs in a shared memory system will result in contention that is noticeable from the perspective of both processors [12]. By measuring the effective memory bandwidth any computational unit on the shared memory bus has access to, it is possible to measure the effective memory bandwidth of another process. A primitive example of this behavior is shown in Figure 2.2. By measuring the perceived memory bandwidth of a program, the contention effects produced by another processing unit can be measured. This enables the indirect measurement of the memory bandwidth utilized by another application, which can be used to acquire the footprints discussed in Section 2.2.1.

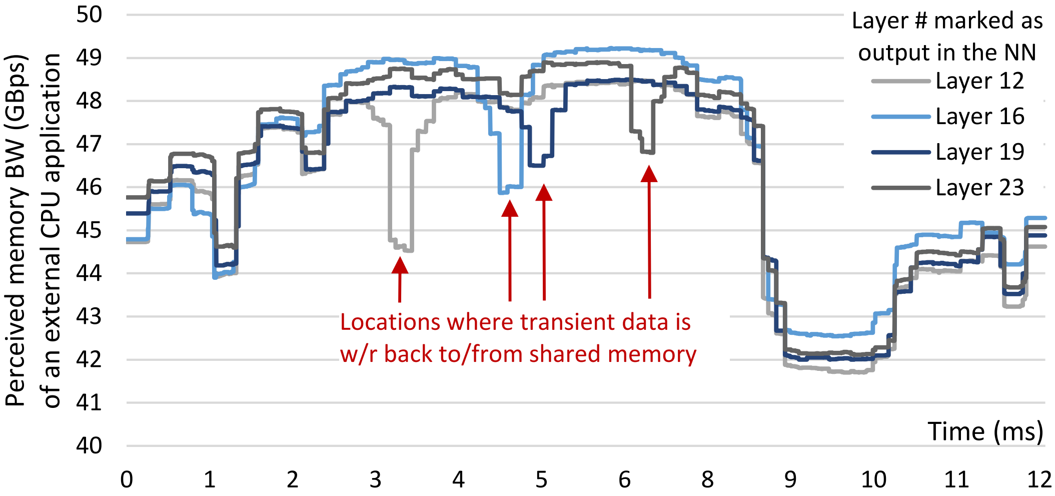

Figure 2.2 The measured effective bandwidth from a CPU during the execution a VGG network on the GPU. The Y-axis shows the perceived memory bandwidth on a CPU application that demands around 52 GB/s of constant memory traffic. When VGG-19 is run on the GPU, the perceived bandwidth of the application drops depending on the executing layer's memory demand. Dips marked on graph, are extreme cases where a layer was forced to flush data to the shared memory.

2.3 DNN Model Extraction

The information provided by the memory bandwidth side-channel enables identification of running processes in a system. Deep Neural Networks (DNN)s are a common machine learning technique used in a variety of critical operations. Tasks such as facial and speech recognition can all be accomplished through the use of DNNs. Gaining information about an executing DNN exposes the critical operations that are taking place. This opens the possibility of intellectual property theft as well as targeted attacks against a DNN that require prior knowledge about the executing model. The following sections outline the type of DNN model information that can be extracted as well as prior extraction techniques that have been used to extract model information.

2.3.1 Model Characteristics

When targeting a DNN model, there are several characteristics that can be extracted. Potential characteristics that may be extracted fall into a few key categories: network architecture, parameters, and hyper-parameters. The network architecture refers to the layer types, layer configuration, and network topologies. Parameters include the weights, biases and other values that are updated during a networks training. Hyper-parameters refer to the parameters that are utilized during the training of a network. Extracting a neural network model's characteristics, enables other attacks on the victim network. Just knowing a model's network architecture enables an attacker to further explore a model's parameters and hyper-parameters, as well as implement adversarial attacks on the network which can negatively affect performance [13].

2.3.2 Model Architecture Extraction Techniques

Due to the significance of knowing a DNN model architecture, several prior works have explored potential extraction techniques. Initial works such as Rendered Insecure [4] determined neuron numbers utilizing GPU performance counters with prior assumption about the model type. Other works such as Reverse Engineering CNNs [8] and Cache Telepathy [7] produce a more general approach that estimates layer dimension sizes based on their respective side-channels. They utilize the estimated dimension sizes in order to determine a potential layer type and topology. Deepsniffer[9] expands on this further, directly transforming per layer architectural hints into a network topology using a neural network and then utilizes the predicted layer sizes based on the type. This approach further improves the generality of the attack and shows effectiveness in the development of adversarial attacks.

Despite their benefits, these attacks have a common weakness in their assumed level of access. Works such as Rendered Insecure assume prior knowledge about the execution model as well as access to performance counters. Cache Telepathy makes assumptions about elevated to utilized resources, which is not possible on modern mobile operating systems without root privileges. Deepsniffer, despite being more general, still proposes attack vectors that rely on physical access to a system which is not always possible or feasible. The proposed bus snooping method, for examples, is not feasible on SM-SoC devices. Additionally DeepSniffer assumes that there are multiple side channel information vectors which is not realistic in the case of limited access. As such there is a need for a DNN model extraction attack with no privileged or physical access.

CHAPTER 3

METHODOLOGY

As discussed in Chapter 2 DNN model extraction attacks provide a method of acquiring information about critical operations. This information enables IP theft as well as the development of adversarial attacks that can negatively affect a models performance. The goal of this work is to provide a non-privileged side-channel vector and verify its functionality by using it to perform a DNN model extraction attack. The following sections describe the threat model, side-channel implementation, as well as the model extraction attack itself.

3.1 Threat Model

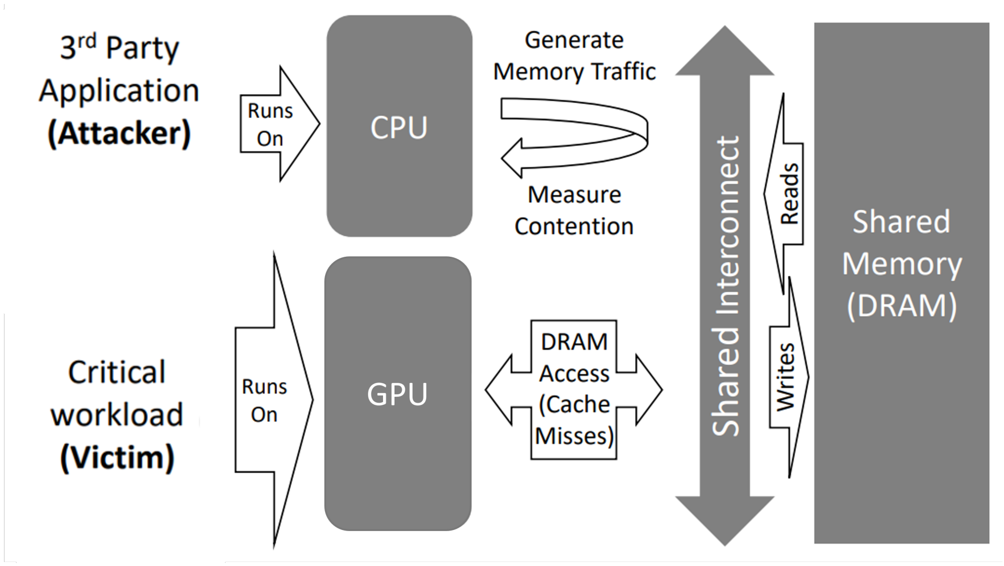

Figure 3.1 The threat model assumes a shared memory architecture (SM-SoC) where the attacker, disguised as a third party application, runs on the CPU and the victim uses the GPU.

A single GPU system with a shared memory architecture, similar to the one depicted in Figure 3.1, is assumed for the computational device under attack. In these systems the GPU features dedicated private caches and share their higher level memory with the system's CPUs. Such systems often employ a DRAM-based architecture such as DDR4 for the shared memory and lack a shared last level cache (LLC) due to the inherent architectural heterogeneity [12]. Each computational unit utilizes a shared memory bus to access the shared memory via a centralized memory controller.

In this study, it is assumed that the target architecture is utilized concurrently by multiple applications. The victim is a critical workload, such as face detection on a mobile phone or object detection in an autonomous system, carried out through the use of a DNN. The secret of the victim workload is the DNN model's architecture, specifically the executing layer types as well as layer topology. The victim is assumed to be running on a the system's GPU, in addition to the CPU.

The adversary is a third party application with an objective of stealing secret information from critical workloads. The adversary application is assumed to be installed via an application store that the underlying vendor platform provides [14, 15]. The adversary is not allowed to run their applications on the GPU that the critical workloads execute on. Additionally, the adversary is assumed to be run in user-mode and accesses only the CPU cores without any access hardware performance counters. The architecture of the target platform is assumed to be known by the adversary. The adversary does not have a physical access to the device and therefore cannot directly measure physical leakage such as power consumption and electromagnetic emanations.

The number of load/store (L/S) instructions which miss the GPU-private cache is dependent upon the application characteristics and memory subsystem. For memory intensive operations, the combination of data access pattern, memory footprint size, and cache size of the target GPU leave a unique DRAM access footprint that can be used to identify characteristics of the underlying operation. The adversary application generates an artificial memory traffic and monitors the slowdown in its own memory accesses which is caused by the memory accesses issued by the critical workloads running on the GPU.

3.2 Memory Contention Based Leakage Detection (MCLD)

Based on the described threat model, we design and implement a memory-contention-based leakage detection (MCLD) mechanism that will serve as the foundation of a DNN model architecture extraction attack. MCLD extracts a signature by observing memory-access behavior that leaks as critical operations are run on the GPU. MCLD is designed to run as a standalone application (*e.g.*, a Linux user-level process) on the CPU and does not need access to the GPU, as long as the CPU shares the same memory bus and module (*e.g.*, DRAM). Unlike state-of-the-art attacks exploiting shared memory accesses, this approach neither assumes that physical and virtual memory address traces are available [16] nor requires proximity to the hardware [17]. MCLD is composed of two components:

1. *Contention generation:* This component generates synthetic L/S instructions to create additional traffic on the shared memory subsystem. To minimize the number of cycles required for each memory access, this component utilizes architecture-specific vector instructions. Since many state-of-the-art shared memory systems also require simultaneous threads to fully exploit the available bandwidth, the contention generator is capable of being configured to utilize multiple CPU cores.

2. *Contention sensing:* MCLD relies on the idea that simultaneous accesses from system accelerators and CPUs in a shared memory system will result in contention that is noticeable from the perspective of both accelerator and CPU, which has been demonstrated in [12]. The sensing is quantified by the memory throughput of the MCLD by measuring the time it takes to execute the issued L/S instructions. At a given time, if the victim is issuing a high amount of shared memory accesses, the effective memory throughput experienced by the MCLD will be lower.

3.2.1 MCLD Design

In order for MCLD to sense contention it must first generate traffic on the shared memory system. The generated traffic must have the following characteristics: be constant, repeating and sufficient to sense contention. In practice, this is achieved by repeatedly executing data L/S operations on memory, where each operation counts as a unique access to the memory unit. When run without contention, these memory accesses will complete in approximately the same amount of time. However as the victim begins to contend with MCLD for memory access the average latency of this instructions will increase. Thus, in order to sense contention, MCLD simply needs to measure the time it takes to write a fixed amount of data. Under low contention conditions this duration will be shorter and thus a higher effective bandwidth will be measured, with the opposite holding true in the case of high memory contention.

The concept of generating a fixed amount of memory traffic and recording performance has been explored by several preexisting benchmarks such as the stream benchmark [18]. Thus by adapting the fundamental concepts of a memory benchmark, it is possible to create the MCLD algorithm which is shown in Algorithm 1. Overall, the algorithm creates two fixed sized arrays, henceforth referred to as contention arrays, copies the data from one array to the other, and measures the total time the copy operation takes. Using the size of the contention arrays as well as the time it takes to copy the array, enables MCLD to calculate the perceived bandwidth. The total data is twice the array size as each element is read and written (R/W) from/to memory. The recorded bandwidth is then used to create a trace for the victim's execution.

Within Algorithm 1 there are two variable terms: the contention array size as well as the contention array copy function. This is due to both being dependant on the target platform. The size of the contention arrays is determined by system hardware features, specifically the system cache size. Determining the applicable array size is discussed in more detail in Section 3.2.2. As for the array copy function, while it is possible to simply sequentially copy over individual elements, this may not be sufficient to reach a minimum bandwidth requirement such that contention is detectable. Section 3.2.3 covers optimizing the copy operation through several system options in order to increase throughput and reach a target bandwidth.

Algorithm 1 MCLD Algorithm

```
$a \leftarrow Array[ARRAY\_SIZE]$
$b \leftarrow Array[ARRAY\_SIZE]$
$start_t \leftarrow 0$
while recording contention do
    $start_t \leftarrow get\_system\_time()$
    copy_a_to_b()                                        ▷ Function that copies data from a to b
    $total_t \leftarrow get\_system\_time() - start_t$
    $BW_{memory} \leftarrow (ARRAY\_SIZE * 2)/total_t$     ▷ 1 L/S per element
    record $BW_{memory}$
end while
```

3.2.2 MCLD Contention Array Size

Determining an appropriate contention array size for MCLD is largely dependent on the CPU's cache size. While SM-SoCs share a unified memory unit, each CPU and GPU contains its own private caches that are not accessible by other computational units. Thus if MCLD is running only on the CPU and accesses data that is primarily within a private cache, these accesses will not face any memory contention and thus will not be able to sense the bandwidth utilization of a victim process. This means that the array size needs to be sufficiently large so that enough of memory access go to the SM unit as opposed to a private cache. However determining the array size is not trivial, as creating an unnecessarily large array will reduce the sampling interval at which bandwidth measurements can be taken. If the victim process has a short execution this can result in insufficient sample collection for a process trace. Additionally, operating systems often limit the amount of memory a process can utilize. For instance, Apple IOS restricts how much memory an application can use and will stop it if the limits are exceeded [19]. These restrictions mean that the array size needs to be carefully chosen so that it is the smallest possible, while still being capable of applying a sufficient amount of traffic to the memory controller.

As MCLD assumes that prior knowledge of the target architecture, the contention array size can be carefully tuned before the attack is implemented. By pre-testing various possible array sizes, it is possible to find an optimal array size for MCLD. An example of such a selection process was conducted on the NVIDIA Xavier AGX. In this experiment two fixed sized arrays were created and data was repeatedly copied between them. The average reported external memory controller (EMC) utilization was then recorded for each array size. As shown in Figure 3.2, a small contention array results in almost no EMC utilization. Initially, increasing the contention array size results in a linear increase in EMC utilization. However after reaching an array size of about 12 MB, this increase begins to taper. Based on this experiment, it can be concluded that for MCLD the array size needs to be around 8MB minimum in order to see an impact on the EMC and that after the array size exceeds 12MB, further increases have diminishing returns. In this scenario choosing an 12MB contention array, would result in adequate memory traffic with a reduced of detection and increasing the sampling rate. The aforementioned analysis could be conducted on any target platform in order to improve efficiency.

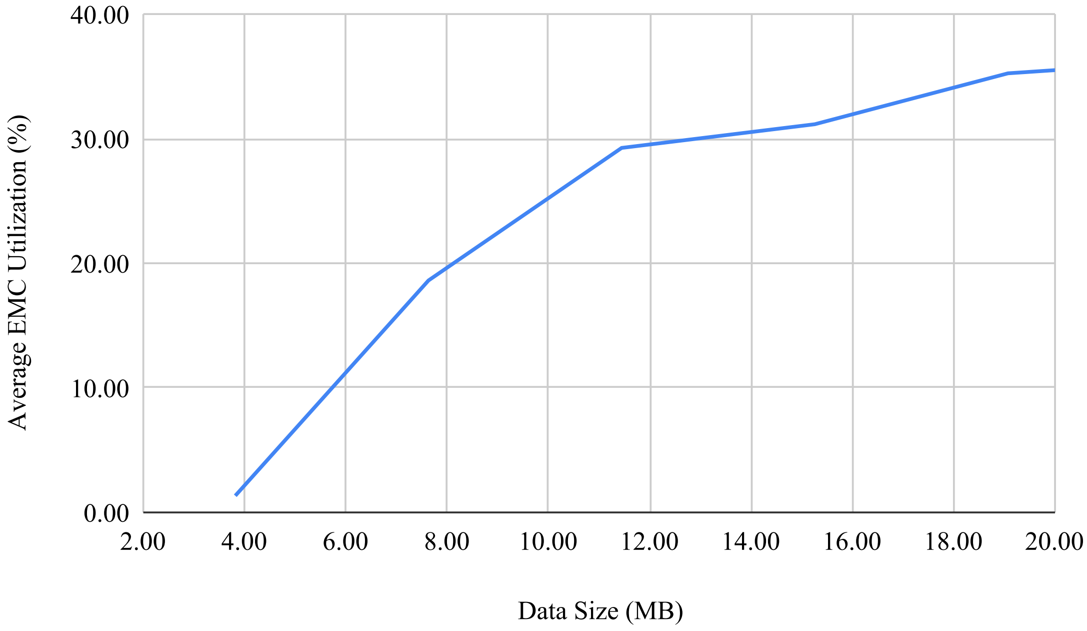

Figure 3.2 Effects of increasing array size in MCLD on the EMC utilization on the hardware device NVIDIA Xavier AGX.

3.2.3 MCLD Copy Optimization

While tuning array size is necessary for MCLD to meet its operational requirements, further tuning of the copy operation can greatly improve sample rate and increase EMC utilization. In a trivial implementation of a copy operation, the data copy would be executed element by element. There are two improvements that can greatly optimize this operation if available on a target platform: vector instructions and multi-core execution.

Vector instructions provide the an avenue to increasing MCLD's throughput. Rather than sequentially copying the elements within the MCLD array, utilizing an architecture's vector operations allows multiple data elements to be copied simultaneously. This reduces instruction overhead and enables MCLD to achieve its target memory bandwidth more easily. A simple experiment on the Xavier AGX, shows that moving from element-wise to vector instructions results in an up to 1.13 times speed up. However pure vector instructions are often not enough to achieve a sufficient bandwidth utilization on a device. As such MCLD also utilizes multiple cores in order to achieve higher throughput.

Multi-core execution is used in conjunction with *vector instructions* in order to improve data throughput. This is necessary in many systems to maximize the bandwidth pressure MCLD can apply. In MCLD parallel execution is handled through OpenMP. Specifically the *for* construct is used when copying the MCLD contention array [20]. While converting MCLD to a parallel implementation, there are certain considerations that need to be made with regards to core selection when running the parallel version. This specifically relates to cache thrashing. If any of the cores that are used to run MCLD share a low level cache, the resulting measurements can be noisy and impact the measured bandwidth. Figure 3.3 compares the effects of running a simple array copy on the two different configurations. As can be seen in the figure, it is important to take into account the hardware architecture of the target device and properly select which cores are utilized.

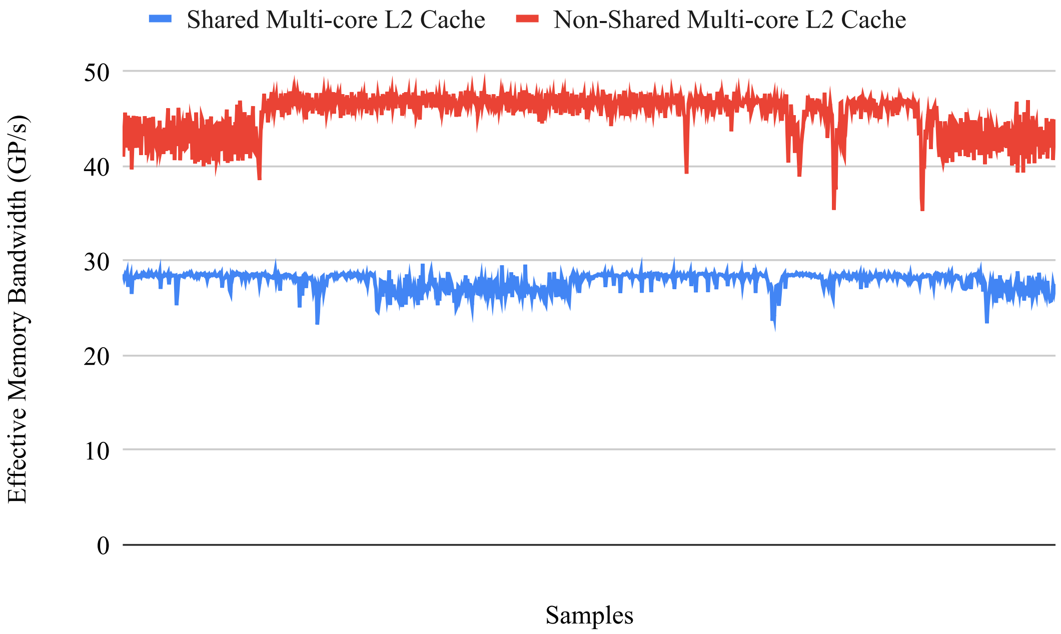

Figure 3.3 Effects of parallel copies on two cores that share an L2 cache compared against two cores that do not share an L2 cache.

3.2.4 MCLD Noise Reduction

While the optimizations presented in Section 3.2.2 and Section 3.2.3 enable MCLD to achieve its goals efficiently, we have employed further improvements to reduce signal noise and improve sampling accuracy. While it is possible to run MCLD without these improvements, the benefits they provide reduce data processing complexity and were used in order to simplify the DNN extraction attack, which is discussed in Section 3.3. In our work we explore following optimizations: core binding, core isolation, and clock count based timings.

Core binding involves binding MCLD to specific cores on the target system. This can be achieved through the use of utilities such as `taskset`. Once bound, the program execution will not be moved between cores that are available on a system. Binding cores reduces noise through two methods. First as discussed in Section 3.2.3, by ensuring utilized cores do not share a low level cache noise produced by a cache thrashing can be eliminated. Second by binding cores core switching overheads can be avoided, eliminating another potential source of measurement noise.

Core isolation excludes cores from being used by the system scheduler as well as for kernel level tasks. This helps ensure that MCLD is the only process running on a core and help reduce noise due to context switching between different processes.

Clock count timing utilizes provided CPU clock cycle counters in order to determine the measurement timings, instead of system calls. By utilizing clock counts to perform timing, the timing precision is significantly increased. Additionally clock counts provide a lower overhead as compared to other timing methods based on system calls.

3.2.5 Resulting Traces

The result of the initial design and further optimizations, have produced the final implementation of MCLD that will be used in the further section as a side channel vector. In order to verify whether MCLD can collect information about memory contention a simple experiment was run on the Xavier AGX [21]. In this experiment traces were collected when the system was idle, as well as during contention applied by repeated executions of VGG-19 [22] network. Figure 3.4 shows the results of this experiment. As can be seen from the results, MCLD measures a constant signature when idle and a repeating pattern for each execution is observed for consecutive executions of VGG-19. We will use similar traces to those described in Section 3.3 to perform a DNN model extraction attack.

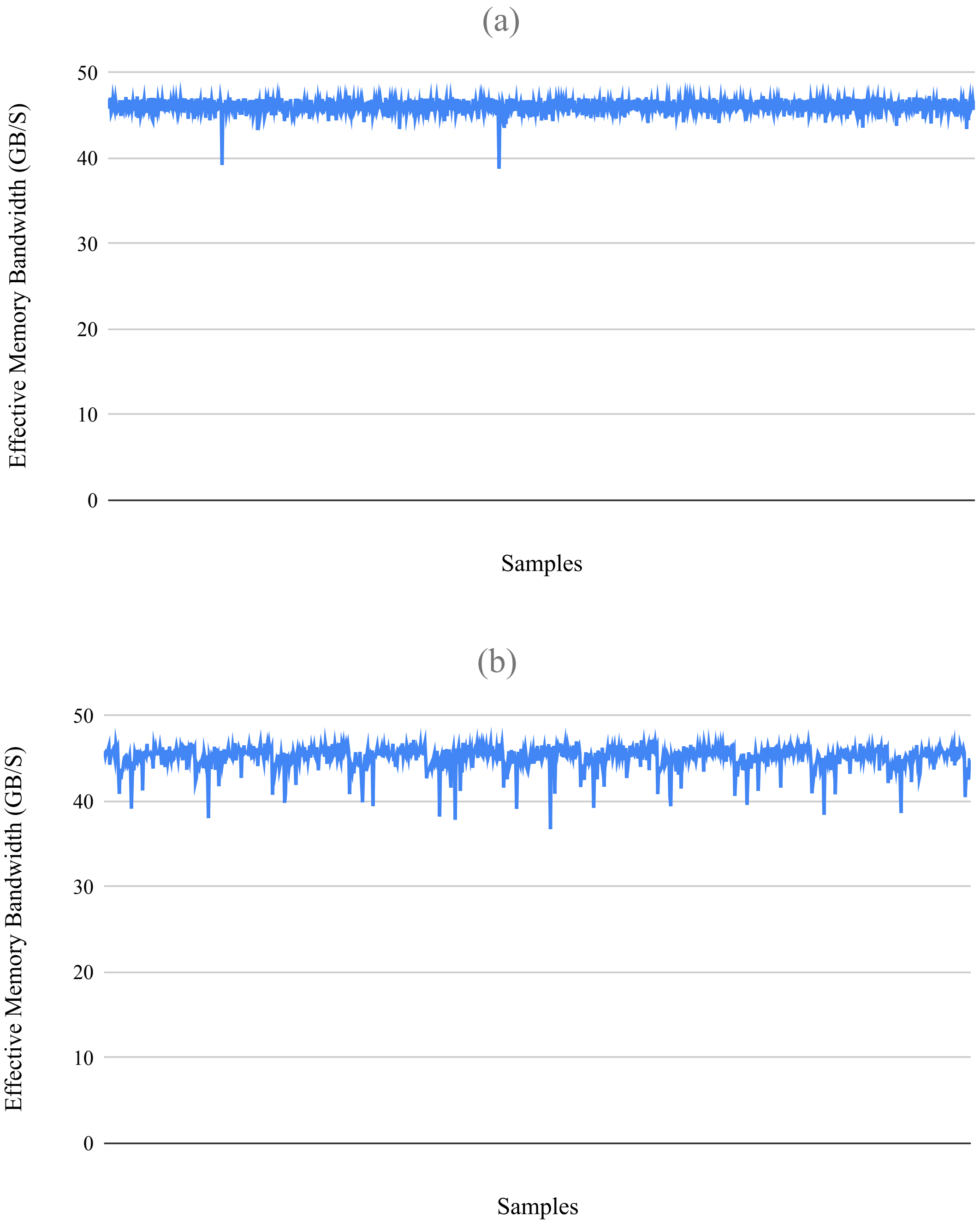

Figure 3.4 Comparison of two traces collected by MCLD with and without contention. Figure a was measured while the system was idle. Figure b was measured during repeated inference using VGG-19.

3.3 DNN Model Extraction

In this section we detail how we extract the DNN model architecture using the traces (i.e., side channel vector) gathered through MCLD. For the remainder of this work, we will use the term *DNN model architecture* to refer to the layer types and topology of a victim neural network. Specifically, the attack we propose targets the convolution and pooling layers, which are the most common layers in convolutional neural networks (CNN) such as VGG [22]. In order to convert a memory bandwidth trace collected by MCLD, there are two key steps that need to take place. (i) First, features need to be extracted from the bandwidth trace and (ii) subsequently, these features need to be classified into their corresponding layer types. The following sub-sections detail the techniques we use to accomplish this task.

3.3.1 Bandwidth Traces and Automatic Speech Recognition (ASR)

In automatic speech recognition (ASR), the objective is to transform the audio waveform created by speech into a corresponding textual representation. Th left hand side of Figure 3.5 depicts an example of such a conversion from waveform to text form. In this work, we make an analogy between the ASR process and the process for extracting layers from a bandwidth trace, where the audio waveform is similar to the bandwidth waveform and the extracted letters correspond to the extracted layer types. The right hand side of Figure 3.5 presents this analogy for bandwidth-based layer extraction for Convolution (C) and Pooling (P) layers. The similarity depicted makes it possible to adopt techniques used for ASR for the purpose of DNN layer sequence extraction.

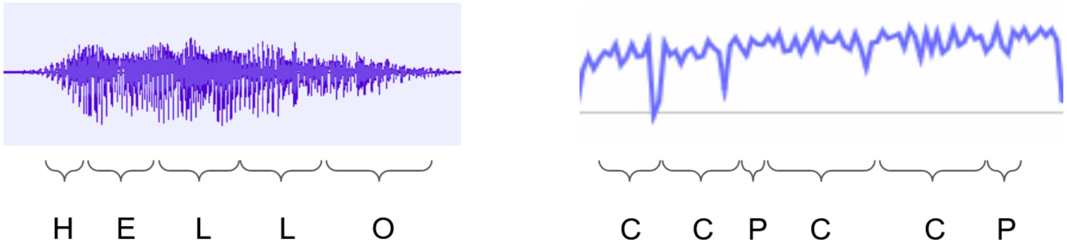

Figure 3.5 Comparison of audio waveform to a bandwidth waveform. On the left is a waveform and the corresponding letters for the waveform. On the right is an MCLD trace and the corresponding layers Convolution (C) and Pooling (P).

Several techniques have been designed and implemented in the field of ASR. DeepSpeech2 [23] utilizes a deep recurrent neural network in order to perform ASR on both English and Mandarin speech. The model for this work, is composed of a few key components. The convolution layers extract features from the input audio spectrogram. The recurrent and fully connected layers in combination with a CTC decoder predict the character sequence. The recurrent network and CTC decoder are necessary in order to model the relationship between certain characters. For instance in English the letter 'h' is more likely to come after the letter 't' than the letter 'z' [24]. This is again similar to the characteristics of neural networks where a pooling layer is more likely to be followed by a convolution layer than another pooling layer [9].

Due to the similarities in problem type and characteristics, the DeepSpeech2 model was adapted in order to implement our proposed DNN model architecture extraction attack. The potential of such an approach has been shown in DeepSniffer [9]. Deepsniffer specifically a recurrent network structure as well as a CTC decoder, in order to convert discrete execution features into layer sequences without utilizing convolution for feature extraction. Our work adopts the whole DeepSpeech2 model using convolution to extract features from the memory bandwidth trace, and then utilizes the recurrent portion along with a CTC decoder in order to predict the layer types and sequence.

3.3.2 Feature Extraction

The first portion of our proposed network is composed of extracting features from an MCLD trace collected during the execution of a targeted DNN. In order to structure the feature extraction portion of the neural network, the input to the network first has to be defined. Unlike in the case of DeepSpeech2, a spectrogram is not utilized for the input. Instead our work relies on stacking traces in order to form a two dimensional array. While we initially tried using a spectrogram of the bandwidth trace as the input, a lack of information in the frequency domain, made this approach unsuccessful. Our stacked trace input takes N zero padded traces that were collected for a single neural network and layers them on top of each other. A visual representation of this input structure can be found in Figure 3.6. N is a value that can be chosen based on the target system's features. For example if the target is executing many iterations of inference, using a larger trace count may improve accuracy.

Stacking the traces mainly increases the available amount of information to the network. By stacking traces it is more likely that certain features that may be difficult to record due to the sampling rate, will be present within the input. Furthermore this setup enables the use of operations such as 2D convolution. Using the described input, feature extraction is conducted with two layers of 3x3 convolution. The resulting feature vectors are then passed to the recurrent network portion for classification.

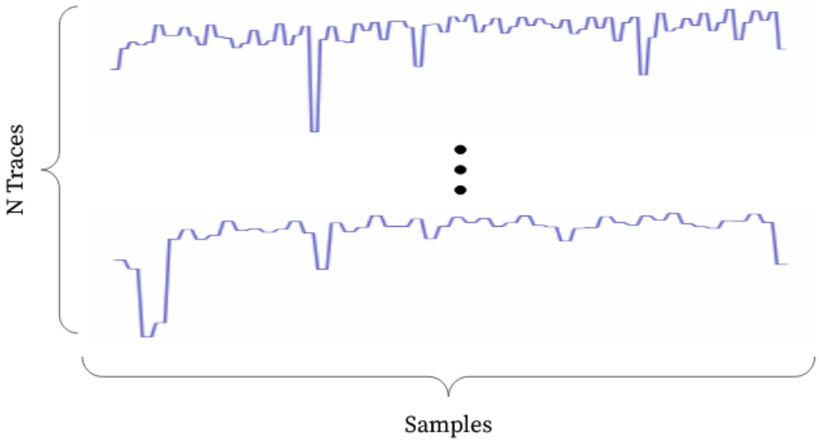

Figure 3.6 Example of the input structure passed to the DNN network, directly utilizing the traces collected by MCLD.

3.3.3 Layer Classification

Converting the generated feature vectors to a predicted layer type was accomplished through the use of a recurrent structure. The recurrent structure is composed of 128 LSTM units which output the entire prediction sequence. Each sequence is passed through a fully connected layer that produces a vector of probabilities for each potential layer type. The potential layer types are as follows: convolution, pooling, load, and save. This output sequence is then passed through a CTC decoder in order to find the most probably sequence of layers. The final output is a sequence of predicted layers for the victim DNN such as: {load, convolution, convolution, save}. Additional output representations of the output structure can be found in Table 4.3 and Table 4.4.

3.3.4 End to End Model

While it is possible to run feature extraction and layer classification separately, the goal of this work is to provide an end-to-end model. Thus the two stages are combined into a single continuous structure similar to DeepSpeech2 [23]. As a result, the stacked MCLD traces can be passed to the network producing the predicted layer sequence for the victim network. The final resulting model structure is shown in Figure 3.7.

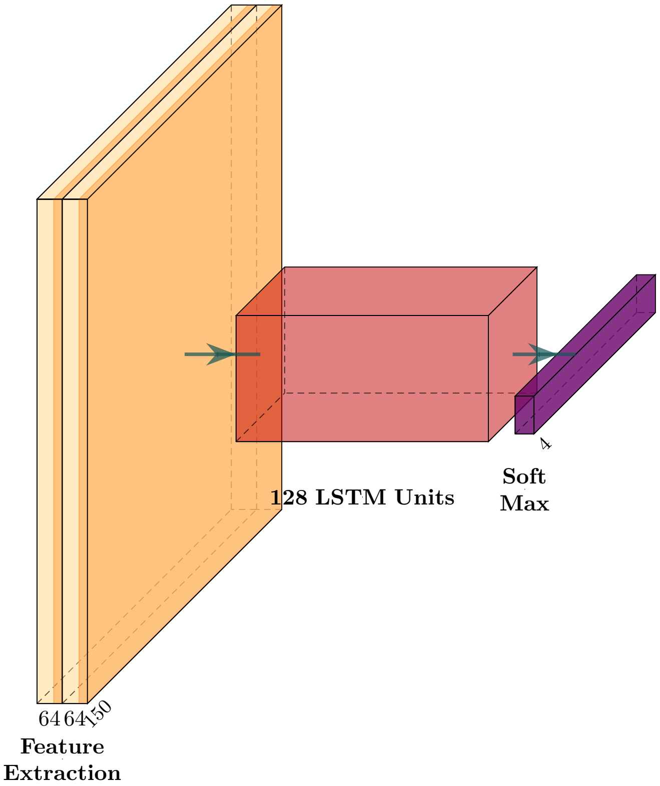

Figure 3.7 Representation of end-to-end model used for extracting the layer sequence from a set of MCLD traces of a DNN. Figure generated using PlotNeuralNet [25].

3.4 Extraction Network Training

In order to utilize the end-to-end model, the model has to be trained. While there are many different CNN model types, the total amount of existing networks would be insufficient to train our network with high accuracy. As such, to generate sufficient data, random networks are generated. The following sections outline how these random networks were generated, as well as how network data was processed.

3.4.1 Network Generation and Trace Collection

Since the target network type for our proposed attack are CNNs, we analyzed two highly popular networks, Resnet and VGG [22, 26] to generate random networks capable of representing such class of neural networks. This analysis identified common layer configurations and topologies, and recorded them as potential choices for the network generator. More specifically, this information included common convolution filter sizes, pooling sizes, and typical layer groupings (*e.g.*, three convolution layers followed by a pooling layer). Using this information, the network generator picks random layer sizes, groupings and orderings and then builds a network using TensorFlow [27]. This network is then exported to an open neural network exchange (ONNX) format [28]. The ONNX model is then processed using TensorRT, so that an optimized network graph can be built [29]. These optimizations include layer fusions and a selection of lowest latency GPU kernels. This final optimized network graph, referred to as an engine, was run during MCLD trace collection.

Graphs are executed using the NVIDIA provided tool called Trtexec, which enables benchmarking networks by providing them with randomly generated inputs. To simplify data collection, Trtexec is modified so that it sends a signal to MCLD when inference is running. Several hundred inference traces are collected for each network which are then pre-processed before being utilized for training.

3.4.2 Data Pre-processing

Several preprocessing steps took place before utilization and training. First any outliers are removed from the data set. This was done by comparing the magnitudes of data points within a trace to the average values and eliminating any values that are significant outliers. Then, the reported average execution time of the network was compared to the number of samples that are collected. Any traces with an outlying amount of samples are removed. Of the remaining samples, N traces were randomly selected to be stacked as input to the network. These traces are then normalized before processing is complete.

In addition, to filtering and normalizing the data, traces are also partitioned based on reported execution percentage of each layer. This process produces many sub-layer groupings that were also used during training. This technique, increased the number of training data samples available and also helped the network learn to classify smaller features sets.

3.5 Overall Attack Methodology

The stages of this attack can be summarized as follows. To begin with, MCLD collects traces for the victim DNN model during inference. Then several techniques are utilized in order to prepare the samples. The processed samples are input to a trained DNN extraction model that produces the predicted layer sequence. A visual summary of this process is shown in Figure 3.8.

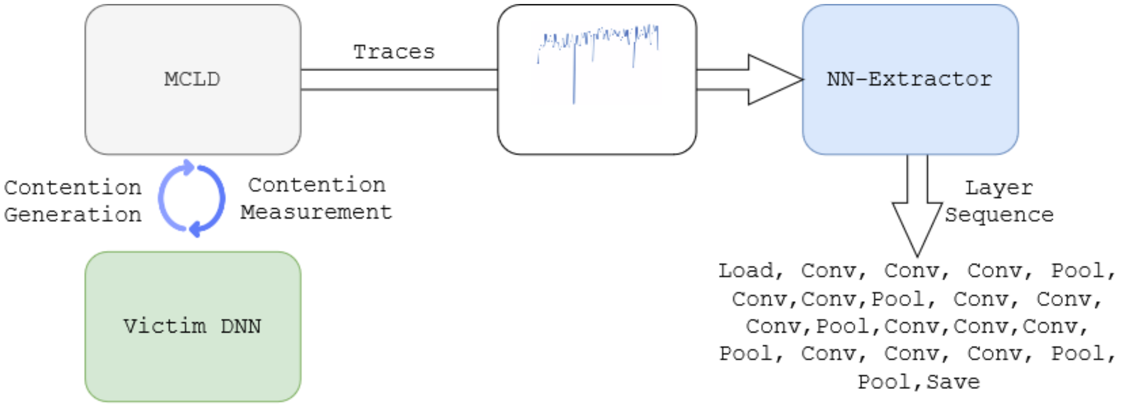
Traces
MCLD
NN-Extractor
Contention Generation
Contention Measurement
Victim DNN
Layer Sequence
Load, Conv, Conv, Conv, Pool, Conv,Conv,Pool, Conv, Conv, Conv,Pool,Conv,Conv,Conv, Pool, Conv, Conv, Conv, Pool, Pool,Save

Figure 3.8 A visual summary of the DNN model extraction methodology.

CHAPTER 4
RESULTS

In order to verify the functionality of the techniques proposed in Chapter 3, experiments are conducted to verify the accuracy and effectiveness of both MCLD as well as the layer extraction DNN. The following sections describe these experiments and the corresponding results.

4.1 Evaluation Methodology

To validate the proposed techniques, experiments are run on the NVIDA Xavier AGX hardware platform. This platform is a Tegra architecture SM-SoC with a Volta GPU and an 8 core NVIDIA Carmel ARM CPU [21]. MCLD is configured with 13 MB contention array and run on two isolated cores using available ARM vector instructions. The victim network is executed primarily on the GPU, with kernels being dispatched by the CPU. This setup follows the threat model described in Section 3.1. MCLD is used to collect traces for 400 random networks generated as described in Section 3.4.1, with each network utilizing a batch size of eight. All traces are optimized using methods outlined in Section 3.4.2, with 150 traces stacked per network. The collected traces are then partitioned into a training, validation and test sets each being respectively 90%, 5%, and 5% of the collected traces. The network outlined in Section 3.4.1 is then trained using this set of data. 1000 epochs are run and the best performing model on the validation set is saved as the final output. Testing is done utilizing both test set of randomly generated networks as well as on commonly used DNN models including VGG and ResNet [22, 26].

4.2 MCLD Analysis

In order to confirm whether MCLD traces can be utilized to extract a network architecture both the characteristics and behavior of MCLD need to be verified on the target system. This is done in two parts: analyzing the MCLD behavior and verify the MCLD trace.

4.2.1 MCLD Behavior

With the described configuration MCLD is able to generate a consistent, roughly 49 GB/s of memory traffic, on the shared memory unit. As will be shown in the following subsection, this is enough to measure unique traces for a running network. Each bandwidth measurement takes an average of 0.7 ms to complete. While this sampling interval is enough to measure multiple times during certain layer executions, it may be insufficient for processes or kernels that have a very quick latency. For example on the Xavier AGX, an analysis of VGG-19 given in Table 4.1, shows that while multiple samples may be collected for convolution layers, it is improbable for a pooling layer to have an independent measurement [22].

Table 4.1 Average execution times per layer type in VGG-19.

Layer Type	Convolution	Pooling
Execution Time (ms)	3.4744	0.2875

4.2.2 MCLD Trace Analysis

MCLD traces are verified in order to determine their accuracy, as well as consistency across different measurements for DNNs. To do this, several traces are gathered for two different neural network executions. Two example traces for these two unique generated networks are shown in Figure 4.1. A surface level analysis of the traces for a single network have similar visual characteristics as well as sample counts, and are unique for a network.

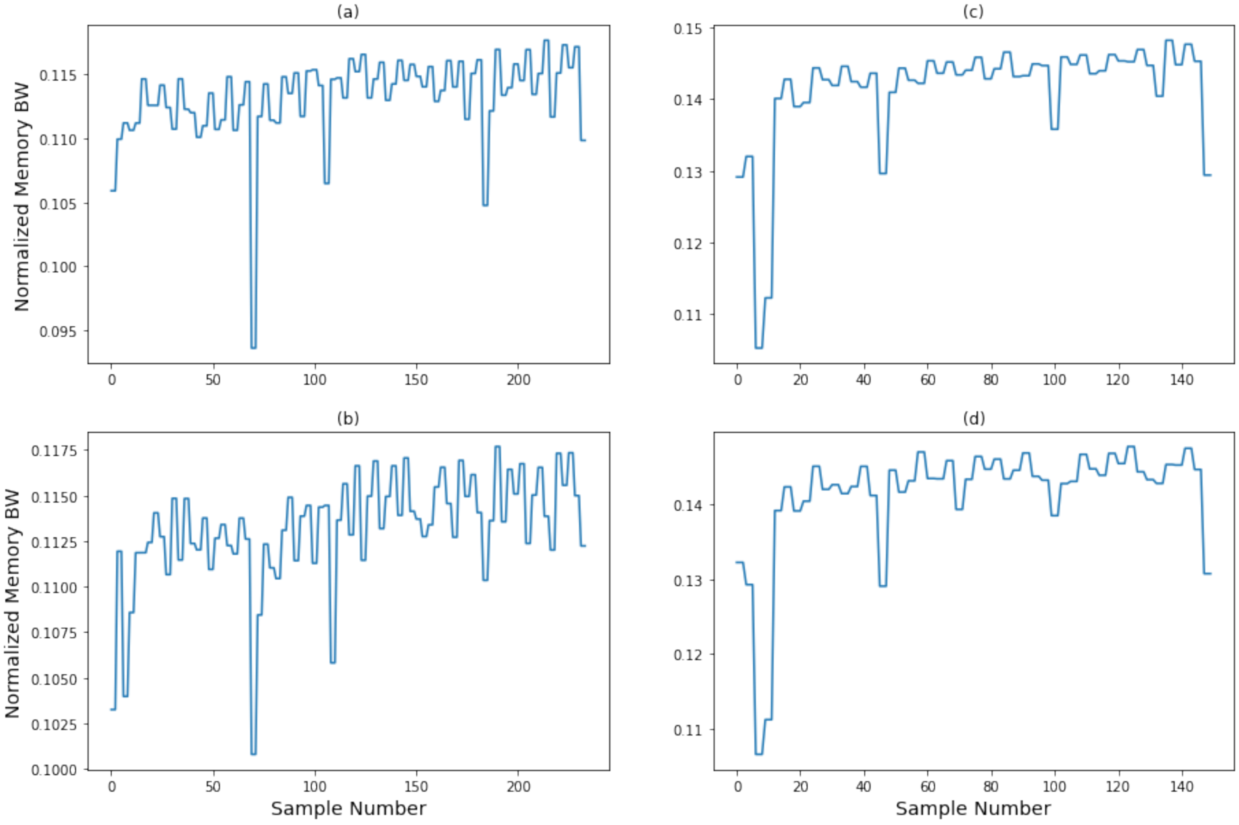

Figure 4.1 A visualization of two different traces for two different networks collected by MCLD on the Xavier AGX. Figures a and b are collected for a generated network with the following layer execution sequence: {Conv, Conv, Conv, Conv, Pool, Conv, Conv, Pool, Conv, Conv, Conv, Conv, Pool, Conv, Conv, Pool, Conv, Conv, Pool}. Figures c and d are collected for a different network with a layer sequence with the following layer sequence: {Conv, Conv, Pool, Conv, Conv, Conv, Pool, Conv, Conv, Conv, Pool, Conv, Conv, Pool, Conv, Conv, Pool}

In order to represent these similarities in a quantitative manner, two analyses are conducted. First NSight Compute is utilized in order to gather an exact measurement for the execution trace [11]. This trace is normalized and compared to the normalized MCLD traces. The characteristics of both these traces are consistent. Second the traces from each network are compared to one another using dynamic time warping (DTW). These results, shown in Table 4.2 show that traces gathered for a single network are similar to one another and different from traces collected for another network. This means that MCLD signatures do indeed create a unique footprint for a network's execution. Having verified the trace accuracy and uniqueness, the following step is to verify the accuracy of the DNN model extraction network.

Table 4.2 Average DTW similarity scores of traces collected for two different networks visualized in Figure 4.1. Lower scores represent more similarity.

	Network 1	Network 2
Network 1	0.0267	0.3237
Network 2	0.3237	0.0277

4.3 Layer Sequence Results

The following sections analyse the accuracy of the network that is utilized to extract a network layer sequence from the traces collected by MCLD. First the accuracy metric is defined, followed by the results on previously described test sets.

4.3.1 Layer Error Rate (LER)

In order to verify the accuracy of the DNN extraction network, it is necessary to define a metric that can represent error. Previous works for DNN model architecture extraction have utilized a layer error rate (LER) in order to measure the error between the predicted and true layer sequence [9]. LER is calculated as the normalized edit distance between the original layer sequence and the predicted one. The LER calculation is shown in equation 4.1 where L is the ground truth sequence, L* is the predicted sequence, ED(L, L*) is the edit distance between the two sequences, and $|L*|$ is the length of the ground truth sequence. In summary this calculation gives the length normalized amount of layer insertions, deletions and substitutions that would need to be made in order to convert the prediction to the ground-truth sequence.

$$LER = \frac{ED(L, L*)}{|L*|} \tag{4.1}$$

4.3.2 Layer Sequence Accuracy

The DNN extraction is first evaluated on a test set of twenty randomly generated networks. Subsequently the LER is determined for a set of real world networks including: VGG and Resnet. A comparison of the predicted and ground-truth sequences is shown for VGG-16 in Table 4.3 and VGG-19 in Table 4.4. As can be seen, errors are more likely to occur at the later layers. This is likely a result of the later layers having a lower execution time, which prevents MCLD from being able to collect as many samples as it could for earlier layers. The overall LER for each of the test network groups is shown in Table 4.5. While the results for the randomly generated and VGG networks is fully representative of prediction accuracy, there is a caveat to the accuracy of branching and residual networks. As these networks feature branching and concatenation, the extraction technique can not fully capture these layer topologies. In reality, during execution these layers are simply executed in some sequential order as shown in Figure 4.2 on the following page. This sequential execution is what is detected as opposed to the original computational graph.

Table 4.3 Comparison of predicted and ground truth sequence for VGG-16. Red layers are missing int he prediction.

Predicted Seq.	True Seq.
Load, Conv, Conv, Conv, Pool, Conv, Conv, Pool, Conv, Conv, Conv, Pool, Conv, Conv, Conv, Pool, Conv, Conv, Conv, Pool, Pool, Save	Load, Conv, Conv, Conv, Pool, Conv, Conv, Pool, Conv, Conv, Conv, Pool, Conv, Conv, Conv, Pool, Conv, Conv, Conv, Pool, Pool, Save

Table 4.4 Comparison of predicted and ground truth sequence for VGG-19. Red layers are missing in the prediction. Crossed out layers are extra predictions.

Predicted Seq.	True Seq.
Load, Conv, Conv, Conv, Pool, Conv, Conv, Pool, Conv, Conv, Conv, Conv, Pool, Conv, Conv, Conv, Conv, Pool, Conv, Conv, Conv, Conv, Pool, ~~Conv~~, Pool, Save	Load, Conv, Conv, Conv, Pool, Conv, Conv, Pool, Conv, Conv, Conv, Conv, Pool, Conv, Conv, Conv, Conv, Pool, Conv, Conv, Conv, Conv, Pool, Pool, Save

Table 4.5 LER for test networks utilizing trained extraction model.

	Random Networks	VGG-16	VGG-19	Resnet-18	Resnet-34
LER	0.025	0.136	0.080	0.170	0.100

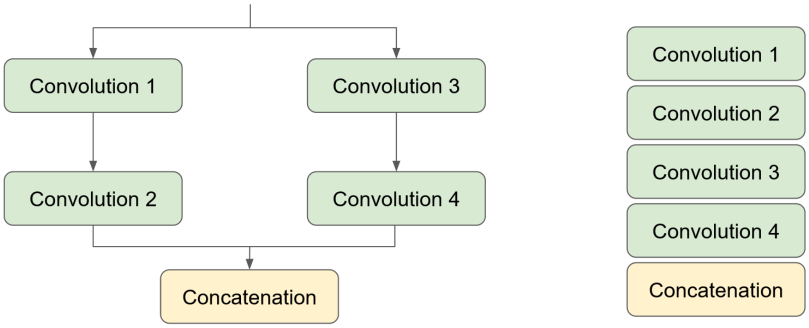

Figure 4.2 An example of how a branching DNN may be executed. On the left is an original computational graph and on the right is the actual layer execution order.

4.3.3 Robustness to Noise

While initial results show a low error rate, it is important to verify how well the extraction network functions when it encounters noise. In order to test how DNN extraction model responds to MCLD signal noise, the network functionality is tested on several traces that adds varying levels of Gaussian noise. The noise level and corresponding LER are shown in Figure 4.3 on the next page. As can be seen in the figure, the extraction network's LER increases up to 7.7 times at a 20% noise level. Regardless, the LER remains at a level or reasonable accuracy for the predicted trace reaching a maximum of 0.108.

4.3.4 Stacked Traces Count

Apart from noise it is also possible to vary the number of stacked traces passed to the extraction network. A larger amount of traces, accounts for a greater amount of potential timing offsets that can occur during MCLD data collection and thus can carry additional information that normally could be missed. In situations where access to the system is limited, the amount of traces that can be collected by MCLD may be small. As such it is important to account for how well the extraction network is able to handle a smaller number of traces. In order to avoid having to retrain the network, this is done by choosing a fixed number of traces and repeating them until the dimensional requirements of the network are met. The LER is then measured for each of these sample counts and the results are shown in Figure 4.4 on the following page. These results shows that as the network has access to more traces, the LER decreases. With a single digit count of traces, the LER is noticeably higher. However the LER improvements diminish as the trace counts increase past around 25 traces.

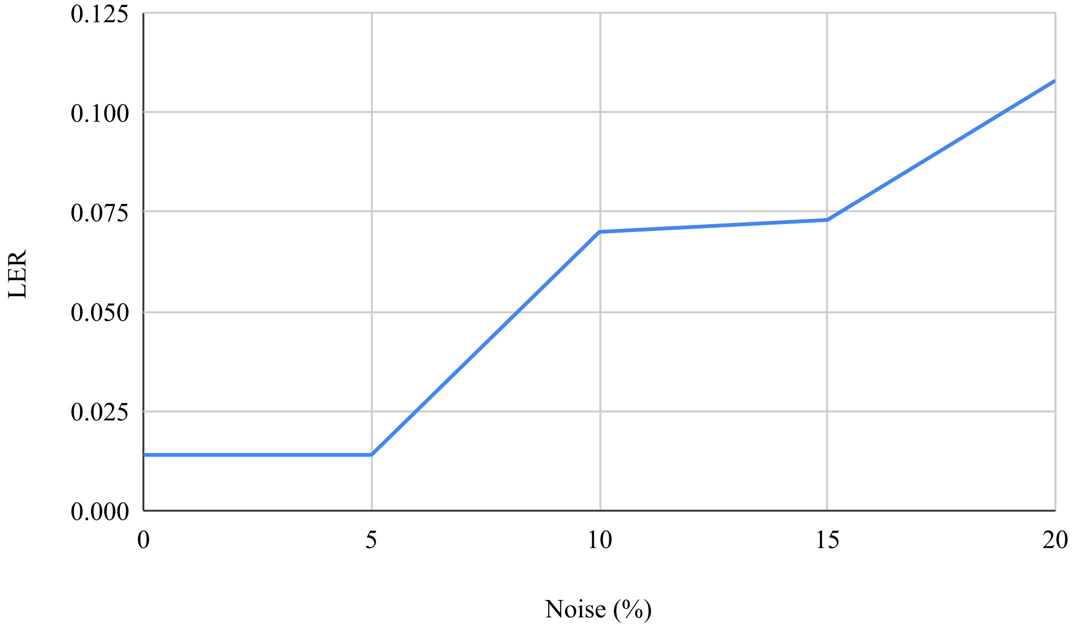

Figure 4.3 Graph of the noise level and its impact on the LER.

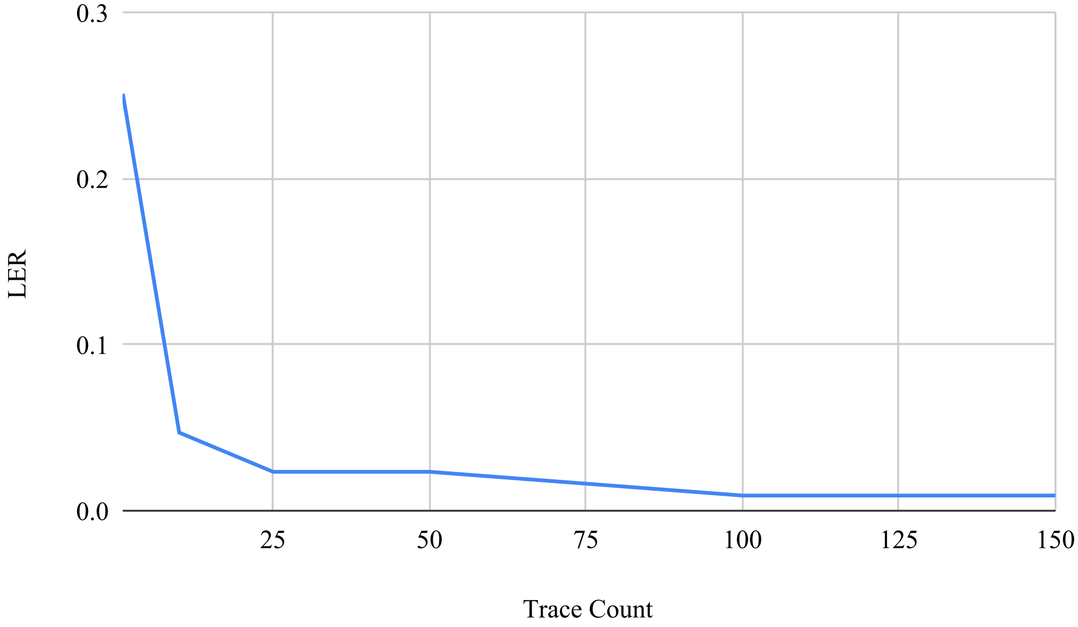

Figure 4.4 Comparison on the number of traces used and its relation to the LER.

CHAPTER 5

ANALYSIS, FUTURE DIRECTIONS AND CONCLUSION

5.1 Discussion

The methods displayed in this attack provide both a general side channel vector as well as the transformation of leaked information into usable information. The utility, applicability, and protection against these techniques is outlined in the following sections.

5.1.1 Utility

Gaining information about the critical DNN operations taking place on a target system, can provide several valuable capabilities from the attackers perspective. First of, the attacker is able to reverse engineer the networks structure, which may be undisclosed private IP property. More significantly knowing the architecture of a network can aid in the development of adversarial attacks on a target network. Attacks such as adversarial image generation can greatly reduce the accuracy of a running network [13]. These and other attacks are enabled by knowing the execution properties of a network. As such the MCLD vector presented in this work, enables several attacks that are capable of significantly impacting the effectiveness of secure processes.

5.1.2 Applicability

While MCLD's potential was verified on a single system, with a limited layer subset, this does not mean that the attack is limited in applicability. The attack does not work solely on a single target platform, rather the only requirement is that the target system needs to have shared memory architecture. As for layer types, as long as an operation has a unique memory bandwidth pattern it is possible to identify it. Thus it is possible to not only target more layer types, but also extract information from other critical workloads such as graphics processing.

5.1.3 Attack Prevention

In order to protect against the MCLD side channel, certain techniques can be implemented in order to reduce MCLD effectiveness. As this attack vector simply relies on memory traffic, it can not be protected by encrypting memory access or randomizing access. Other techniques must be implemented. A trivial workaround would be to remove the shared memory unit. This would significantly reduce the accuracy of the measured memory traffic as rather than continuously occurring during execution, data transfers would be grouped into a single transfer operation. Adding obfuscating memory traffic, could significantly affect accuracy of the attack. However in order to significantly impact performance, the magnitude of the generated traffic would have to be exceedingly large, potentially significantly reducing system latency and causing a significant overhead in the process.

Additionally hardware level modification could be made to the memory access controller in order to prevent MCLD and similar contention attack vectors from being able to acquire a trace. MCLD requires a monotonic, consecutive, and uniform memory access pattern in order to detect the point of contention that another processor creates. By prioritizing accesses of the victim process, the accuracy of the contention trace could be significantly impacted. A more direct memory controller modification, could involve adding randomized delays to memory traffic coming from MCLD in order to cause MCLD to report incorrect bandwidth spikes in the collected memory trace.

5.2 Related Work

Further expanding this attack vector, can target several potential avenues. Extending potential information that can be extracted using MCLD. While the demonstrated attack is capable of extracting the model architecture, more information could potentially be gleaned from the side channel. Extracting layer dimension sizes may be possible given the bandwidth utilization of the identified layers. Additional information about the weights could also be ascertained by analysing the access patterns in order to determine what type of weights were used. Gaining more information about given a network would allow for a more accurate reverse engineering attempt as well as enable more effective attacks [13].

Exploring how this attack would work with other accelerators outside of GPUs could be another avenue of research. The Xavier AGX platform that was utilized for experimentation and validation, also has a deep learning accelerator (DLA) that is connected to the shared memory bus. This means that such the attack proposed in this work may be able to target this hardware unit as well. As the DLA has a different private cache setup as compared to a DLA it may be possible to identify which units are executing a program in order to gain more information about a critical process execution. These improvements would expand the utility and value of the proposed attack method described in our work.

5.3 Conclusion

As the utilization of SM-SoC becomes more common, architectural vulnerabilities that allow an attacker insight into the operation of secure operations become increasingly useful. This paper proposes a side channel attack that works on any SM-SoC. This vector is able to extract information about the memory intensity of a running program through a low privileged memory contention measurement utility called MCLD. The value of this attack vector is verified by using collected traces in order to decode a DNN model architecture. This vector and methodology pose a great risk to a multitude of mobile and edge computing environments.

REFERENCES

[1] Linear Micro Author. System-on-a-chip vs. single board computers — linear microsystems, 10 2019. URL https://linearmicrosystems.com/system-on-a-chip-single-board-computers/.

[2] Introducing m1 pro and m1 max: the most powerful chips apple has ever built, 10 2021. URL https://www.apple.com/newsroom/2021/10/introducing-m1-pro-and-m1-max-the-most-powerful-chips-apple-has-ever-built/.

[3] Tegra xavier - nvidia - wikichip, 12 2019. URL https://en.wikichip.org/wiki/nvidia/tegra/xavier.

[4] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 01 2018. doi: 10.1145/3243734.3243831.

[5] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *2019 IEEE Symposium on Security and Privacy (SP)*, 05 2019. doi: 10.1109/sp.2019.00002.

[6] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. Meltdown. *Communications of the ACM*, 63:46–56, 05 2020. doi: 10.1145/3357033.

[7] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2003–2020. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/yan.

[8] Weizhe Hua, Zhiru Zhang, and G. Edward Suh. Reverse engineering convolutional neural networks through side-channel information leaks. *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 06 2018. doi: 10.1109/dac.2018.8465773.

[9] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, and Yuan Xie. Deepsniffer. *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 03 2020. doi: 10.1145/3373376.3378460.

[10] Paul Gavrikov. visualkeras. https://github.com/paulgavrikov/visualkeras, 2020.

[11] NVIDIA. Nsight compute, 04 2022. URL https://docs.nvidia.com/nsight-compute/NsightCompute/index.html.

[12] Yuanchao Xu, Mehmet Esat Belviranli, Xipeng Shen, and Jeffrey Vetter. Pccs: Processor-centric contention-aware slowdown model for heterogeneous system-on-chips. *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 10 2021. doi: 10.1145/3466752.3480101.

[13] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks, 2016. URL https://arxiv.org/abs/1611.02770.

[14] Android apps on google play. URL https://play.google.com/store/games?hl=en_US&gl=US.

[15] App store. URL https://www.apple.com/app-store/.

[16] Yongbing Huang, Licheng Chen, Zehan Cui, Yuan Ruan, Yungang Bao, Mingyu Chen, and Ninghui Sun. Hmtt. *ACM Transactions on Architecture and Code Optimization*, 11:1–25, 02 2014. doi: 10.1145/2579668.

[17] Moumita Dey, Alireza Nazari, Alenka Zajic, and Milos Prvulovic. Emprof: Memory profiling via em-emanation in iot and hand-held devices. *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 10 2018. doi: 10.1109/micro.2018.00076.

[18] John McCalpin. Memory bandwidth: Stream benchmark performance results. URL https://www.cs.virginia.edu/stream/.

[19] Apple. Apple developer documentation. URL https://developer.apple.com/documentation/metrickit/improving_your_app_s_performance/reducing_your_app_s_memory_use.

[20] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5:46–55, 1998. doi: 10.1109/99.660313. URL https://www.cs.swarthmore.edu/~newhall/readings/openmp.pdf.

[21] NVIDIA. Deploy ai-powered autonomous machines at scale. URL https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/.

[22] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014. URL https://arxiv.org/abs/1409.1556.

[23] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, Jie Chen, Jingdong Chen, Zhijie Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Ke Ding, Niandong Du, Erich Elsen, Jesse Engel, Weiwei Fang, Linxi Fan, Christopher Fougner, Liang Gao, Caixia Gong, Awni Hannun, Tony Han, Lappi Johannes, Bing Jiang, Cai Ju, Billy Jun, Patrick LeGresley, Libby Lin, Junjie Liu, Yang Liu, Weigao Li, Xiangang Li, Dongpeng Ma, Sharan Narang, Andrew Ng, Sherjil Ozair, Yiping Peng, Ryan Prenger, Sheng Qian, Zongfeng Quan, Jonathan Raiman, Vinay Rao, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Kavya Srinet, Anuroop Sriram, Haiyuan Tang, Liliang Tang, Chong Wang, Jidong Wang, Kaifu Wang, Yi Wang, Zhijian Wang, Zhiqian Wang, Shuang Wu, Likai Wei, Bo Xiao, Wen Xie, Yan Xie, Dani Yogatama, Bin Yuan, Jun Zhan, and Zhenyao Zhu. Deep speech 2 : End-to-end speech recognition in english and mandarin. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 173–182, New York, New York, USA, 20–22 Jun 2016. PMLR. URL https://proceedings.mlr.press/v48/amodei16.html.

[24] Conditional frequency of letter pairs in english — heatmap made by prooffreader — plotly. URL https://chart-studio.plotly.com/~prooffreader/17/conditional-frequency-of-letter-pairs-in-english.embed.

[25] Haris Iqbal. Harisiqbal88/plotneuralnet v1.0.0, December 2018. URL https://doi.org/10.5281/zenodo.2526396.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. URL https://arxiv.org/abs/1512.03385.

[27] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

[28] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2019.

[29] EunJin Jeong, Jangryul Kim, and Soonhoi Ha. Tensorrt-based framework and optimization methodology for deep learning inference on jetson boards. *ACM Transactions on Embedded Computing Systems*, 01 2022. doi: 10.1145/3508391.