DNN Model Architecture Fingerprinting Attack on CPU-GPU Edge Devices

Kartik Patwari, Syed Mahbub Hafiz, Han Wang, Houman Homayoun, Zubair Shafiq, and Chen-Nee Chuah
University of California, Davis, CA, USA
{*kpatwari, shafiz, hjlwang, hhomayoun, zshafiq, chuah*}*@ucdavis.edu*

Abstract—**Embedded systems for edge computing are getting more powerful, and some are equipped with a *GPU* to enable on-device deep neural network (DNN) learning tasks such as image classification and object detection. Such DNN-based applications frequently deal with sensitive user data, and their architectures are considered *intellectual property* to be protected. We investigate a potential avenue of *fingerprinting attack* to identify the (running) DNN model *architecture* family (out of state-of-the-art DNN categories) on CPU-GPU edge devices. We exploit a *stealthy* analysis of aggregate system-level *side-channel* information such as *memory*, *CPU*, and *GPU* usage available at the *user-space level*. To the best of our knowledge, this is the first attack of its kind that does not require physical access and/or `sudo` access to the victim device and only collects the system traces passively, as opposed to most of the existing reverse-engineering-based DNN model architecture extraction attacks. We perform feature selection analysis and *supervised* machine learning-based classification to detect the model architecture. With a combination of RAM, CPU, and GPU features and a Random Forest-based classifier, our proposed attack classifies a *known DNN model* into its model architecture family with 99% accuracy. Also, the introduced attack is so *transferable* that it can detect an *unknown DNN model* into the right DNN architecture category with 87.2% accuracy. Our rigorous feature analysis illustrates that memory usage (RAM) is a critical feature for such fingerprinting. Furthermore, we successfully replicate this attack on two different CPU-GPU platforms and observe similar experimental results that exhibit the capability of *platform portability* of the attack. Also, we investigate the *robustness* of the proposed attack to varying background noises and a modified DNN pipeline. Besides, we exhibit that the leakage of model architecture family information from this stealthy attack can *strengthen* an adversarial attack against a victim DNN model by 2×.**

Index Terms—**DNN Model Architecture Fingerprinting, Side-Channel Attack, GPU-enabled Embedded System**

1. Introduction

Modern edge devices [1] have significant on-device computing that allows them to employ powerful deep neural network (DNN) models [2], [3]. DNN models employed on edge devices process sensitive user information and are themselves considered confidential intellectual property. Thus, the research community is actively investigating direct and side-channel attacks that leak information about on-device DNN models employed by smart edge applications. [4]–[6]. Furthermore, unauthorized knowledge of the DNN model can also enable downstream attacks, including model inversion attacks [7], membership inference attacks [8], and adversarial attacks [9], [10].

This paper investigates a DNN model architecture fingerprinting attack on GPU-enabled edge devices via side-channel leakage. Specifically, we propose a novel DNN model fingerprinting attack on CPU-GPU-based edge devices through passive analysis of system-level side-channel information such as global memory, GPU, and CPU usages available at the user-space level. System statistics and performance counters provide handy information for users to monitor application performance or behavior problems [11]. Our black-box attack involves training a supervised classifier using system-level traces for a diverse set of popular DNN model architectures used in deep learning (DL) applications. Rather than reverse-engineering the victim DNN model architecture, parameters (i.e., more fine-grained properties) by exploiting fine-grained side-channel leakage, our attack focuses on classifying a victim DNN into a category/family of architectures (i.e., less fine-grained property) utilizing only coarse-grained side-channel knowledge. Prior research [9], [12] has shown that an attacker's knowledge of DNN model architecture—even though the acquired information is less fine-grained—allows it to improve the effectiveness of adversarial attacks.

While prior literature has investigated model extraction through memory access pattern-based side-channel venues [9], [13], [14], they are limited in the following ways. First, some require physical access to the victim device (e.g., by probing electromagnetic (EM) emissions) [9], [13]. EM emanations enable more fine-grained memory statistics and can reconstruct model network architecture without prior knowledge. Second, some utilize popular cache-based side-channel attacks like *Flush+Reload* or *Prime+Probe* [14], [15]. These cache-based methods solve the issue of requiring physical access but require active cache probing. This is undesirable as it involves directly probing the system cache, and due to the significant emergence of cache-based side-channel attacks, researchers are developing detection techniques and algorithms [16], [17]. Our attack overcomes these limitations as it does not require physical access to collect passive system information or any assumptions of code sharing the same memory space, e.g., L3 cache or attacker having to either access or manipulate cache or such. Furthermore, utility functions that provide fine-grained side-channel leakage (e.g., per-process stats) are disabled on many systems. We investigate a new vulnerability to snitch model architecture family using only coarse-grained side-channel stats.

We implement our attack against popular and state-of-the-art DNN model families, such as ResNet, VGG, DenseNet, SqueezeNet, AlexNet, MobileNet, ShuffleNet, and Inception, running on NVIDIA Jetson family edge devices. Our attack pipeline consists of two steps. First, we collect global system traces using tegrastats on Nvidia Jetson Nano (4GB) when a DNN model is running and select suitable features from the collected time-series data. Second, we train a machine learning classifier on the labeled data to fingerprint victim DNN architecture and analyze the impact of different features—RAM, CPU, and GPU usage. Many model families include variants of models that share a similar backbone architecture. While it is common to use the popular models, they can also be slightly modified—with more layers by smart application developers—for specific tasks. Hence, we inspect further the *transferability*[1] property of our attack with variants of model families that the attackers classifier was not trained on. In addition, we successfully clone our attack on two different CPU-GPU edge platforms — Jetson Xavier NX and TX2 – to show the attack can be generalized to a different platform. We investigate that the common property of the platforms is that they are shared-memory systems. Furthermore, we inspect the *robustness*[2] of the trained attacker classifier when there is a secondary application running with varying memory loads and when DNN models are modified via transfer learning. Finally, with the fingerprinted knowledge of the model family, we illustrate an improved adversarial attack on the victim DNN.

Our results evidence that we can use a combination of GPU and CPU loads and memory usage (of RAM) to achieve an accuracy of 99% on trained models. We perform *feature ablation* to notice that feature combinations with RAM attain an accuracy of 98%-99%. Furthermore, our *transferability* experiment also indicates RAM usage as a critical characteristic to distinguish between DNN architectures. The attacker's classifier obtains a success rate of about 87%-90% for *unseen model variants* with RAM usage. For our *platform portability*[3] tests, we find that the combination of RAM and GPU statistics yield an accuracy of 98.9% on seen and 86.6% on unseen model variants. Besides, we demonstrate the accuracy of the attack falls short from 86.4% to 16.9% with a concurrent AES encryption/decryption application on 10 to 100 megabytes, respectively. Also, we showcase that the adversarial attack, DeepFool [18], on a DenseNet DNN model degrades the benign accuracy from 83% to 51% when the attacker has incorrect knowledge of the ensemble of the DNN models and to 28% when it has the correct knowledge of the family of the DNN model. The DNN model fingerprinting open-source library has been released in GitHub [19].

We summarize our key contributions as follows:

- For the first time, our fully experimental work shows the use of global system-level statistics at the user-space level to fingerprint DNN model architecture. This is very valuable as no admin access or sudo access, or jailbreaking of the device is required.
- We illustrate that global side-channel statistics can be exploited for accurate DNN model architecture fingerprinting on CPU-GPU-based Nvidia Jetson devices.
- We conduct our DNN architecture fingerprinting in a more realistic scenario, i.e., remote host setting without prior knowledge of the victim model.
- We show that global memory (total RAM) usage is the most crucial feature for transferability.
- We demonstrate that our attack pipeline can be replicated to different GPU-enabled platforms.
- We execute an exhaustive feasibility analysis of our proposed attack in the lens of its robustness to a memory-bound secondary application with different stresses and modified DNN models with transfer learning. We also evaluate the attack performance on the TensorFlow DL framework in addition to our original pipeline on PyTorch.
- Finally, we illustrate that the advanced knowledge of the DNN model family doubles the effectiveness of a semi-black-box adversarial attack.

2. Related Work

Prior related works exploited different side-channel information for different attack targets such as model architecture, inputs, or parameters. Most of the works focused on extracting/re-engineering model network architectures. Knowledge of the network architecture constitutes two main concerns: (1) network architectures are considered intellectual property, and (2) studies showed knowledge of network architecture could improve adversarial attacks and membership inference attacks [8], [9], [13], [14]. The common sources of side-channel leakage include cache [15], [20], memory-access, [9], [13], [14], electromagnetic (EM) emission & power [7], [21]–[23], timing [24], [25], and GPU [26], [27] statistics. While most of the works focused on network architectures, a few also aimed to extract model parameters [7], [13], [28] and inputs [21], [22], [25].

Usually, the collection of *EM/power* traces requires physical access to the target device. Batina et al. [21] leveraged power/EM side-channel leakages and queried a target model to reverse engineer network architectures. Xiang et al. [7] extracted parameters (with known inputs) and the model architecture via power traces. Wei et al. [22] showed that power traces could be utilized to recover an input image from a physically accessible FPGA-based CNN accelerator with known parameters. Note that recovering parameters or input images require one to infer the other. In one of the most recent works, Chmielewski, Lukasz, and Weissbart [23] performed EM emanation and timing analysis on NN layers execution on a GPU implementation (Jetson Nano). They were able to recover the number of layers and number of neurons per layer, and identify different types of activation functions with power analysis.

Duddu et al. [24] achieved model extraction by querying the target model and measuring the execution *time*. The execution time is used to predict the depth of the model and effectively reduce the search space to infer a target model. A reinforcement learning-based search finds an architecture closest to the target model. Another line of research used timing-based side-channel information to recover model inputs. For example, Dong et al. [25] was able to recover input images of DNNs deployed on micro-controllers using the running time of floating-point multiplications. Kernel execution times are also used

[1] *Transferability* is formally defined in Sec. 5.4 (see Definition 5.1).
[2] *Robustness* is formally defined in Sec. 5.6 (see Definition 5.3).
[3] *Portability* is formally defined in Sec. 5.5 (see Definition 5.2).

in conjunction with other side-channel information (e.g., memory-accesses) for model extraction [9].

In recent years, *GPU*s have become the dominant platform to train and deploy DNN models. There is a line of research attacking general-purpose GPUs. Wei et al. [26] exploited GPU context-switching penalty to extract model architecture. Their attack, Leaky DNN, requires the adversary to share the same GPU as the victim when training the model. Furthermore, this is an attack conducted on cloud-based GPUs and had to perform a denial of service (DoS) attack in order to slow down the transition between DNN layers to be able to recover them. Naghibijouybari et al. [27] utilized a CUDA spy application co-located with a victim CUDA DNN application and required a CUPTI NVIDIA profilers, to monitor hardware performance counters. The counter values are then used to infer the network architecture. Both two works required NVIDIA profilers such as CUPTI, which are not easily available on Jetson edge devices. Conversely, our focus is on GPU-powered edge devices, and `tegrastats` that comes installed. Besides, there is no "`nvprof`" profiler for the Jetson edge devices for fine-grained statistics.

Cache-based side-channel attacks exploit the shared cache between processes. Often, these methods require the attacker and victim to be co-located processes using the same DL framework. Most cache-based methods rely on having memory sharing and code access to monitor specific function invocations etc. Hong et al. [20] retrieved DL architecture using *Flush+Reload* side-channel attack to monitor specific function calls during inference. Yan, Fletcher, and Torrellas [15] monitored Generalized Matrix Multiply (GEMM) functions using *Flush+Reload* and *Prime+Probe* techniques to infer the model architecture.

Memory-access pattern-based attacks are closest to our work as we look at models' aggregate global memory usage to build the classifier for the fingerprinting task. Hua, Zhang, and Suh [13] were able to reverse engineer CNN models running on hardware accelerators by querying the victim model and monitoring the off-chip memory accesses. Hu et al. [9] work—*DeepSniffer* framework utilized memory access patterns (with physical access to the victim device) acquired through EM emissions and bus snooping methods to infer the complete victim NN architecture (including layer sequence, topology, and dimension sizes) without any prior knowledge of the model. For complex DNNs, their attack requires bus snooping attack to gain memory read/write statistics, which is more intrusive. These works require physical access to the device to probe for EM emanations. Liu and Ankur [14] developed a DNN architecture extraction framework, GANRED, under the remote host setting using cache timing to counter the physical access requirement of side-channel information. They compared the victim DNN architecture and the attacker DNN structure (created by a generative adversarial network (GAN) method [29]) via cache traces acquired from *Prime+Probe* side-channel attack technique. The traces reflect the memory-access patterns of the running DNNs and compare the generated and target models. However, using a cache-based attack such as *Prime+Probe* makes this attack less passive, having attack code to access and modify the shared cache.

In contrast to the above works, we classify deep learning applications deployed on edge devices into a set of targeted model architecture families—the fingerprinting-based model architecture leakage attack we propose is significantly novel and differs from other model architecture extraction attacks. Our attack is non-intrusive: *passively* and *remotely* collects only global memory-usage side-channel traces from a black-box victim DNN application. Our attack neither requires physical access nor need to modify shared resources (like cache). The traces are used in a supervised learning approach to classify the network architecture.

The *application/website fingerprinting* line of research also utilizes side-channel information for the fingerprinting task. Similar to model extraction, application/website fingerprinting has also been performed through the various side-channel venues such as EM [30], [31], power [32], [33], GPU [34] and network traffic [35] leaks. Chawla et al. [36] combined EM emissions and dynamic voltage frequency scaling (DVFS) states as features to fingerprint applications on android platform using supervised ML techniques. The majority of application fingerprinting attacks utilize per-process statistics (i.e., from `profcs`) predominantly targeting mobile (Android) devices [37]–[40]. Similar to our approach, Zhang et al. [41] employed global statistics (memory and network patterns) to explore application fingerprinting on iOS platforms without `procfs`—introducing new side-channel attack vectors (iOS APIs). We take inspiration from the technique to fingerprint model network architectures instead. Our focus is on edge devices, and there are a few works on fingerprinting on IoT/edge devices; most of them have focused only on fingerprinting the IoT devices themselves [42], [43].

Adversarial attacks generate adversarial examples with imperceptible perturbations, leading to miss-classification or erroneous results by machine learning systems. Transfer-based adversarial attacks [12], [44], [45] look into generating adversarial examples that can transfer across different model architectures. Liu et al. [12] used novel ensemble-based approaches to generate transferable adversarial examples from models trained over a large-scale (ImageNet) dataset. As a result, they can successfully generate targeted and non-targeted adversarial examples for a black-box image classification system (which need not be trained on the same dataset used to generate examples). A key observation is that the transferability of adversarial examples improves if the substitute[4] and victim models belong to the same network architecture family. DeepSniffer [9] follows similar techniques from Liu's work to demonstrate improved adversarial attack efficiency with network architecture knowledge.

To our best knowledge, this is the first work that aims to employ fingerprinting techniques to extract model architecture knowledge that further strengthens the adversarial attack.

3. Background

3.1. DNN Architecture Families

Deep Learning (DL) is a subset of machine learning, which involves building a sequence of layers that perform feature processing/extraction steps in a hierarchy for

[4] Substitute models are selected models that are trained by the adversary and are used to generate the adversarial examples.

Model family	Train/Test set 1	Test set 2
VGG	VGG11, 19	VGG13, 16
ResNet	ResNet18, 50, 152	ResNet34, 101
SqueezeNet	SqueezeNet 1.0	SqueezeNet 1.1
DenseNet	DenseNet121, 201	DenseNet161, 169
ShuffleNet	ShuffleNetv2 0.5	ShuffleNetv2 1.0
Inception	InceptionV3	N/A
MobileNet	MobileNetv2	N/A
AlexNet	AlexNet	N/A

TABLE 1: Explored Models. All models were obtained from torchvision models. Model variants are N/A if there exist no variants or if they were not easily available on the torchvision model suite.

pattern analysis and recognition. This hierarchy of layers is often referred to as the DNN architecture. Various researchers have developed strong performing DNNs, which can easily be re-trained for individual tasks while keeping the same backbone architecture. Furthermore, there can be multiple variants of a DNN stemming from a backbone architecture, e.g., ResNet18 and ResNet50 are based on the ResNet architecture with a different number of layers (as indicated by the number adjacent to the model name). Hence, we can say these variants belong to the same backbone architecture or family.

In our paper, we consider popular state-of-the-art DNN models which are available with `torchvision.models` [46] that goes in-hand with Pytorch [47]. We chose to study the image classification models, and all explored model families are shown in Table 1. For instance, *AlexNet* [48] enabled distributed CNN training between multiple GPU cores. *VGG* [49] initiated deep visual representations in computer vision for large-scale image classification tasks using small convolution filters (about twice as deep as AlexNet). Along this direction, *ResNet* [50] went significantly deeper in visual representation to improve accuracy. Furthermore, *DenseNet* [51] focused on shortening connections between layers that are closer to input and output and building connections with each layer to every other layer to reduce the number of parameters. Both ResNet and DenseNet are residual networks, which add 'shortcut' or skip links between layers. *ShuffleNet* [52] improved the computational complexity of CNN by evaluating the performance by memory-access overhead and operating platform. On the other hand, *SqueezeNet* [53], *Inception* [54], and *MobileNet* [55] have been proposed as alternate CNN models on lightweight devices (such as mobile, FPGA, edge devices) to facilitate fast inference with limited resources.

3.2. CPU-GPU Embedded Edge Devices

The NVIDIA Jetson devices are GPU-enabled embedded systems engineered for AI and edge computing [56]—all having a complete System on Module (SOM) including CPU, GPU, memory, high-speed interfaces, etc. The devices include the Jetson Nano [57], Jetson TX2 series [58], Jetson Xavier NX [59], and the Jetson AGX Xavier series [60], a variety with a different combination of performance, power-efficiency, and form factor. Being GPU-enabled embedded computers, they offer high compute capability, especially for DL applications. A vital component of these devices is that they have all shared memory systems—the CPU and GPU share the main memory. It is common for GPUs to have memory, and often for DL pipelines, the model (weights) and inputs are loaded from CPU to GPU for the inference task. On high-end servers, GPUs typically have their own memory for faster access. As the CPU and GPU share the main memory on these edge devices, the main memory also reflects the GPU memory utilization. Hence, RAM usage is an important feature for fingerprinting model architectures.

tegrastats is a utility tool available to Jetson-based devices that reports memory and processor usage at a selected sampling rate (in milliseconds). It can be run with and without `sudo` access. Using the utility with `sudo` access provides additional statistics. For instance, with `sudo` we can gain additional statistics about the video/audio processor engines. Furthermore, some statistics are augmented. With `sudo`, the frequency for the GPU engine and EMC controller is available. For this paper, we do not use the utility with `sudo` access. This makes our attack model stronger, as an attacker does not require privileged access to monitor the system traces. Without `sudo` and running tegrastats as a background process, the system traces collection phase of the attack is stealthy and passive. Section 4.3 explores available statistics from tegrastats without `sudo` access.

4. Model Fingerprinting Attack

4.1. Threat Model

Our DNN architecture fingerprinting method is motivated by application/website fingerprinting [30], [41] approaches. The core idea is that the attacker can build a classifier from global-statistic traces collected using the same target device that hosts the victim DL applications. We consider a *closed-world* scenario, where the attacker is aware of network architectures that can be running on the victim device. These include popular and state-of-the-art deep learning network architectures that are frequently used for computer vision tasks. This enables us to create a labeled dataset used to train the classifier via supervised learning.

Attackers Goal. The attacker's primary goal is to learn about the victim DL application; in particular, the DNN architecture. The attacker aims to do this *remotely* and *passively*, by only collecting aggregate system-level traces from the victim device. With knowledge about which common model architectures the victim DL application uses, the attacker can explore a variety of downstream adversarial attacks.

Attackers Knowledge. Our proposed attack makes the following three basic assumptions. (1) To infer the victim DNN architecture, the attacker must know the type of victim edge device(s). The attacker can purchase the same device, which we refer to as the attacker's device. The attacker's device runs DNN models and collects system traces aiming to mimic the victim device. (2) Only one DNN runs at a given time on the victim device, without any other background processes. Hence, the global statistics reflect the running targeted application. This is a reasonable assumption given that the target edge device is resource-constrained and typically deployed for specific application. (3) The attacker also assumes the victim DNN belongs to one of the known families of network architectures (in particular, VGG, ResNet, DenseNet, SqueezeNet,

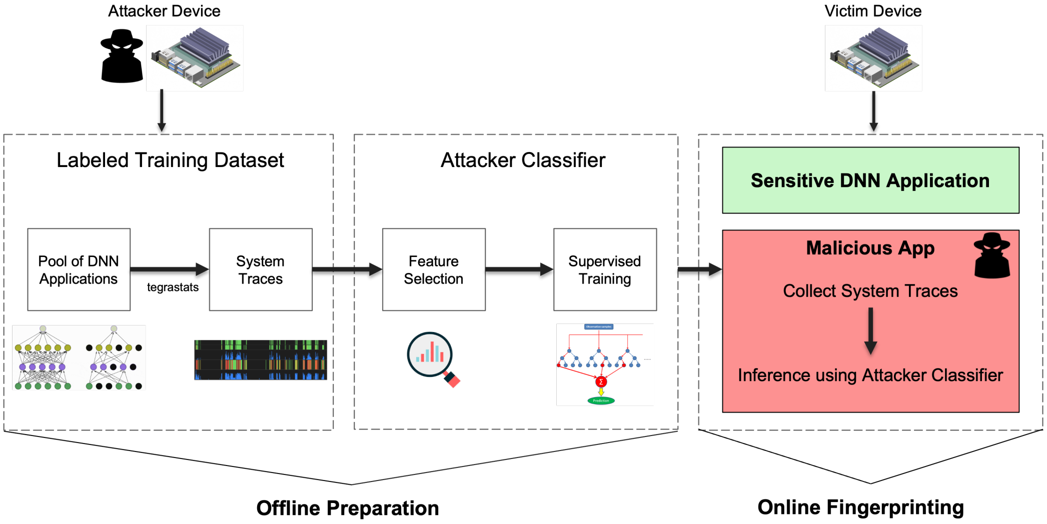


Figure 1: Attack pipeline for the model architecture fingerprinting.

AlexNet, MobileNet, and Inception). However, any prior knowledge of weights or parameters is not required. Off-the-shelf model architectures are convenient for users who want an existing pipeline and implementation to perform their DL/ML task. Existing models are commonly used after they are either pre-trained on known datasets and classes like ImageNet or retrained on custom datasets. This closed-world approach gives the attacker a pool of applications with the different DNN architectures to train and test an ML-based attacker classifier. The training and testing phases for this classifier are performed on the attacker device. Then the classifier is used to fingerprint the model architecture running on the victim's device.

Attackers Capability. The attacker uses a non-privileged performance counter utility tool, tegrastats, and gains access to the victim system either through remote log-in or an installed malicious application[5]. In both approaches, the attacker does not need to gain any physical access to the device. The attacker collects the system traces as a background process to avoid detection. Background processes are run independently of the user without interaction, making the trace-collection phase *stealthy*. Thus, the side-channel information collection is passive, not requiring the use of cache-based methods like Prime+Probe or Flush+Reload. It is also important to note that various utilities disable per-process monitoring from unauthorized users [61], hiding more fine-grained side-channel leakages. This is why most prior work utilized remote cache-based attacks or required physical probings to gain the finer-grained memory statistics. However, using global statistics allows us to overcome this barrier. With knowledge of the victim's model architecture, the attacker can then launch improved adversarial attacks. Architecture-specific targeted adversarial examples can then be generated to attack the victim DL application.

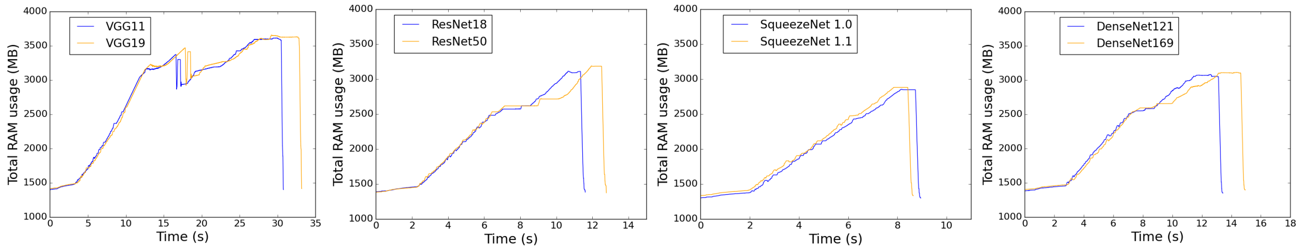
Indeed, our demonstrated attacker can identify (fingerprint) the model architecture family with minimal assumptions compared to the plethora of state-of-the-art side-channel leakage-based attacks discussed in Section 2. For instance, many model architecture extraction attacks—which reveal fine-grained knowledge—utilize micro-architectural side-channel information that requires the attacker to run very sophisticated processes (e.g., Flush+Reload, Prime+Probe, etc., to flush cache). In contrast, our attack utilizes a simple utility function provided at the user level to collect global statistics, making the attack more passive, (i.e., not invasive,) coarse-grained, and stealthy.

4.2. Attack Overview

Our work leverages system-level information to fingerprint popular model architectures deployed on CPU-GPU edge devices. Figure 1 outlines the model fingerprinting attack. An attacker can download common ML model architectures available through Pytorch, TensorFlow, ONNX, etc. (see Table 1) and develop typical inference pipelines with the pool of models on its device(s). Specifically, We consider a typical DNN model inference pipeline for image classification that consists of three necessary stages: (1) loading the data/image to perform DL task on, (2) loading the model and weights, and (3) forward-pass data through a model to get predicted label (running inference task). Stages (1) & (2) can be performed in either order, but (3) requires both the prior stages completed. We assume that a victim DL inference application also exhibits these pipeline stages. As the programs are run on resource-constrained edge devices, we do not expect any more stages associated with training and re-training.

4.2.1. Offline Preparation. Now that the attacker has all models and inference pipelines ready, they can run single-batched inference on randomized input images and collect their system statistics. Our model fingerprinting attack relies on the attacker's ability to create a supervised machine learning classifier. To be able to build and train a classifier, the attacker needs to generate a training/testing dataset from traces collected from its own device(s). The dataset consists of system traces (i.e., memory, CPU, GPU) collected when running the DNN inference application on the device. The attacker simulates the inference pipeline using the common DNN architectures. The inputs can be any images fed into the DL system as we are not interested in any particular type of inputs or outputs but only the memory and processor usage of the specific model architecture. The monitoring tool runs

[5] This is a common method for application/website fingerprinting.



(a) VGG Family (b) ResNet Family (c) SqueezeNet Family (d) DenseNet Family

Figure 2: Examples of total RAM usage for different DNN model families collected via tegrastats on Jetson Nano. Memory usage patterns are similar within the same family and distinctive between separate families. To demonstrate the different shapes of the memory traces, we do not scale the time-axis. The remaining families are in Figure 8.

as a background process, collecting (labeled) traces while programs are running. This enables the attacker to collect a labeled dataset on the specific device. The next step is for the attacker to leverage its generated dataset to develop a supervised ML model for time-series classification to categorize the different DNN models.

4.2.2. Online Fingerprinting. After the classifier is built, the attacker can launch online attack by collecting system traces on the victim device and feeds them into its trained classifier to infer the victim network architecture. The attacker may infect an existing application or create a malicious application to download on the targeted device. The purpose of this application would be to (1) run the system monitoring tool (tegrastats in our scenario for targeted Jetson devices) and (2) process and feed those time series traces to the trained classifier for inference. Once installed on the victim device, the malicious app would not need any privileged access. It can run the monitoring tool in the background as a stealthy process to avoid easy detection. We assume only one DL application is running on the device at a time and the associated DNN model is from one of the popular model architecture families. Hence, any changes in system statistics from the idle state can be attributed to the running application and fed into the classifier for inference. The result from the attacker's classifier identifies the running model on the victim device, the knowledge with which the attacker can then exploit to perform transferable adversarial attacks.

4.3. Feature Description

tegrastats reports collected statistics as time-series data. Hence, we have the aggregated system traces for the entire duration of the target DL application. The utility reports various statistics, and the three we will focus on are memory, CPU, and GPU usage. Other reported statistics include usages for audio/video engines, EMC controller, and thermals (temperatures for CPU & GPU). The EMC controller, audio, video engine statistics are not applicable to our scenario. Notice that the *thermal* information is unreliable. Victim applications can be running back-to-back, causing the temperatures to rise. It takes some time for the temperatures to fall back to the level of an idle state. If the temperatures are still high from the previous run, those readings will contaminate the concurrent run. Hence, we only track reliable parameters like GPU, CPU loads, and memory usages, but did not utilize temperature as a feature.

Memory. tegrastats delineates two main segments of the memory statistics: physical memory (RAM) statistics and virtual memory (SWAP) statistics. For the former case, the utility reports the total RAM in use (in MB) out of the total available RAM, which is device dependant. This is the predominant source of data for analyzing memory usage. Alongside the RAM usage, information about the Largest Free Block (lfb) is also reported. This statistic is not as useful for our purpose as it is mainly static and does not change considerably during the application run or provide helpful insight for classification. The next statistics reported is the virtual memory usage (SWAP) in use (in MB) out of the total available SWAP. However, the SWAP memory is only used when all available RAM is used and additional memory is required. Thus, the SWAP is not helpful for devices with adequate RAM for the DL applications (like Jetson Nano 4GB). Finally, the amount of SWAP cached (in MB) is also reported.

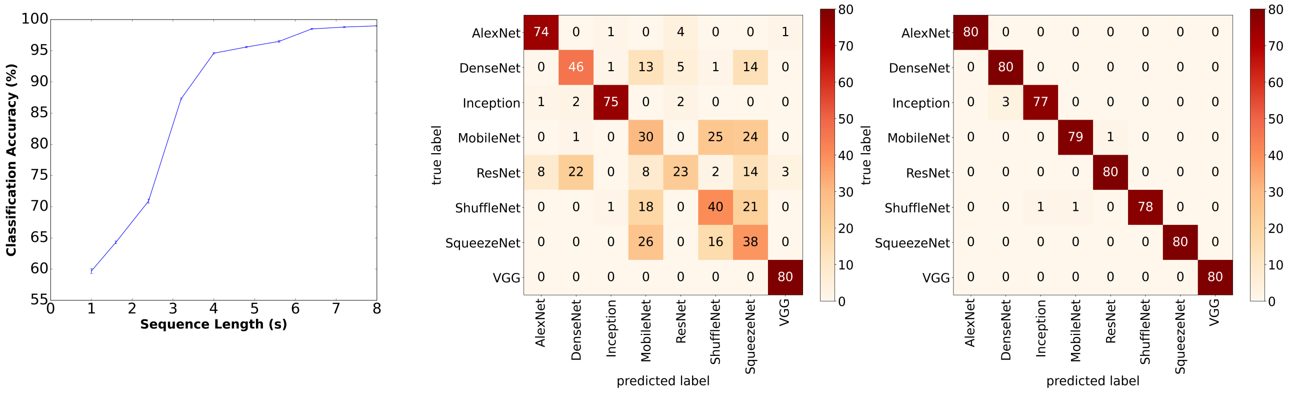
CPU. tegrastats reports the CPU load and frequency for all available CPUs. These are rough estimations that are acquired from time spent on the system idle process as reported by the Linux kernel (in `/proc/stat`). Thus, for every CPU core, the load (in percentage) and its associated frequency are recorded.

GPU. The GPU is a specialized processor designed initially to accelerate graphics rendering. However, the modern trend is to use General-Purpose GPUs (GPGPUs) for non-specialized tasks outside of graphics. The GPU is optimized for throughput and has an extraordinary computation capability. For example, deep learning involves a massive amount of computation that can be parallelized, and therefore, using GPGPUs for DL has become a norm. tegrastats reports the percentage of the GPU engine (GR3D) being used relative to the current running frequency.

4.4. Feature Analysis

Prior works [9], [14], [15], [26], [27] have utilized more fine-grained memory or per-process side-channel information to enable systematic and precise/exact model extraction. We aim only to use global aggregate statistics, which are more course-grained. Our goal is to classify victim model architectures into known architecture families.

4.4.1. Memory. Model extraction/reverse-engineering research identifies memory-access patterns as a valuable source for model extraction [9], [13], [14]. Furthermore, memory-access patterns have been used to reconstruct network architectures, so memory usage proved to have hints about network architecture. DeepSniffer observed that the runtimes for layers within a network vary across different



(a) Varying sequence lengths

(b) Sequence length of 1.5s

(c) Sequence length of 8s

Figure 3: 3a shows classification accuracies for different chosen sequence lengths. 3b and 3c exhibit confusion matrices for the smallest (1.5s) and the largest (8s) sequence lengths, respectively.

models and layers of a DNN have static execution order related to their computational graph. This implies that regardless of execution times, the memory pattern observed corresponds to the computational graph of a DNN [9]. Furthermore, one of the key reasons to inspect RAM usage is because the Jetson Devices are shared memory systems, which means the memory between CPU and GPU is shared. For DL applications, this is important as typically, the inputs, models, model weights all get loaded from the CPU onto the GPU for inference. Usually, GPUs have their memory for faster access. As the CPU and GPU share the main memory, the main memory also reflects the GPU memory utilization. Hence, RAM usage is an important feature for fingerprinting model architectures.

4.4.2. GPU. There have also been model extraction techniques through GPU side-channels, looking into context-switching and CUDA spy applications. We aim to explore CPU-GPU edge devices. These devices are built to deploy AI on edge. The GPU is a core aspect of DL tasks and the dominant platform to train and run DNNs. DNNs require extensive computation, and this is where GPUs excel, accelerating inference on the GPU. Different architectures have different numbers and types of layers, which require different computations. Hence, GPU usage is an obvious candidate to fingerprint DNN applications.

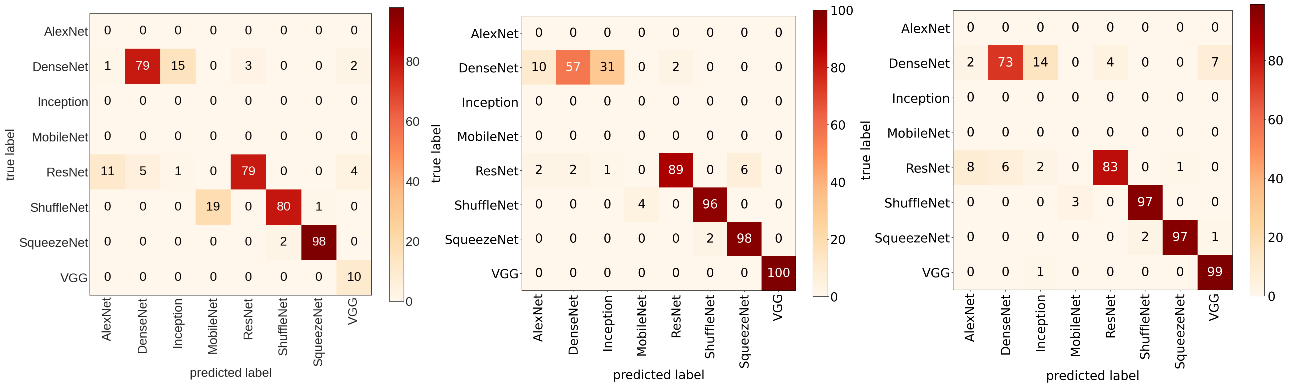
4.4.3. CPU. Many modern DL technologies rely on CPUs and GPUs working together. While CPU statistics have not explicitly been used for model reverse engineering/extraction, CPU load has been shown to fingerprint applications [30]. The CPU frequencies themselves are not a viable differentiator between different DNN models. More so, the device can be used on performance mode, which means the clocks are locked to their maximum. This makes frequencies redundant as they are always at a static maximum. Unlike RAM and GPU usages, we have multiple CPUs and hence data for each CPU load. A process is scheduled onto an available core, and the operating system (OS) determines the scheduling policy. This indicates that the victim application process can be running on any available CPU core. Therefore, we decide to use all available CPU loads as features.

4.5. Classification

Our task is to use the system-level information as features to classify a running DNN into a known architecture family. As previously seen, the features are all collected throughout a running application as time-series data using (non-`sudo`) utility tool `tegrastats`. Hence, our classification problem becomes, with each feature is a time-series itself, a multivariate classification problem.

We combat this by utilizing the *sktime* [62] library, which is a new open-source, scikit-learn compatible, Python library for machine learning with time series. *sktime* provides a variety of state-of-the-art classifiers. We leverage the interval-based Time-Series Forest Classifier [63] (based on Random Forest Classifier). For each of our time-series features, `sktime` builds a classifier and ensembles them. The classifier has two requirements. The first is that the time series must be equally spaced (in terms of observed time). This case is satisfied as our sampling rate is constant (see Section 5.1.3). The second requirement is that all the time series data have to be of the same sequence length. However, as the DL applications runs in real-time, the execution time varies depending on various factors, including the hardware/software environments, and in our case, also the choice of DNN model. We take the middle X seconds of the data, where X is determined experimentally. We define the 'middle' by computing the middle value of a given trace (in time axis), and taking X/2 seconds of data from each side of that middle point. We test our classifier with a range of values for X and pick the sequence length where the accuracy is highest and stable (see Section 5.2). Furthermore, for each trace, we normalize the values using scikit learn's `MinMaxScaler()` to the range [0-1]. For memory, normalization is important. Inherently, there is a certain amount of RAM in use (by the OS itself). This may differ from case to case, and without normalization, it can skew our results and cause dependencies on device-specific values.

The attacker will build its classifier based on the same device as the victim, but it still needs to be generalized to be able to adhere to slight variations (like the initial offset caused by RAM in use). Normalized values allow us to classify a DNN model based on the memory-usage pattern or "shape" rather than raw values. The CPU and GPU usages are already in the range of [0%-100%]. They are still normalized to [0-1] for convenience. We utilize the default parameters provided by `sktime` for the classifier, for some vital parameters these are: `n_estimators` = 200 (number of trees in the forest),



(a) All Features

(b) RAM only

(c) RAM and CPU

Figure 4: Confusion matrices for the transferability experimentations on Jetson Nano—classification conducted on Test set 2—using all features (4a), most prominent feature—RAM (4b), and RAM+CPU (4c).

criterion set as `entropy` (the function to measure quality of a split), and `max_depth` = `None` (the maximum depth of the tree).

5. Experimental Evaluation

5.1. Experiment Setup

5.1.1. Device Specifications. From the Jetson family of devices, we selected the Jetson Nano for our original experimentation. The Jetson Nano has a Quad-core ARM A57 (@ 1.43 GHz) CPU, 128-core Maxwell GPU, and 4 GB (64-bit LPDDR4 25.6 GB/s) Memory. The device utilizes the Linux4Tegra operating system based on Ubuntu 18.04, and we use the JetPack SDK version 4.5. There are two power modes — 10W (default) and 5W for less energy use. We collect our training data with the applications running on the 10W mode. Furthermore, we use the Jetson Nano on performance mode, which disables the DVFS governor and locks the clocks to their maximums. This is the ideal mode to run DL applications utilizing the GPU. For AI performance, the Jetson Nano can deliver 472 GFLOPs.

5.1.2. Inference Pipeline Implementation. Using PyTorch, we develop realistic DL programs consisting of the basic inference pipeline: loading data, loading model (weights), and performing inference (see Section 4.2.1 for details). Since we use the PyTorch framework, pretrained image classification models are provided by the torchvision models subpackage [46]. All pretrained models except InceptionV3 expect the input images to be 224x224 and normalized in the same way. InceptionV3 expects input images to be 229x229. We run inference on the ImageNet ILSVRC Test set images. Next, we use the PyTorch Dataloaders to randomly shuffle and sample the input images. Then, the images are normalized and loaded onto the GPU for single-batch inference. Hence, the data loading stage for all applications is identical with image input from the ImageNet Test set. The models are chosen from the attackers' model pool, and the model and (pretrained) weights are loaded onto the GPU. Finally, the single-batched inference is performed to yield a predicted class. All applications follow this structure, with the difference being in which model is used.

5.1.3. Data Collection. Before any data is collected, we set the Jetson Nano to performance mode. We start tegrastats as a background process. Next, we run each DL application one at a time. After every application finishes execution, the resultant trace is collected and stored with the label of the corresponding model run. Application/website fingerprinting research has also shown that higher sampling rates correlate to an increase in classification accuracy [30] for fingerprinting. Therefore, we maximize the sampling rate to distinguish between model architectures so the system-level traces can give a more accurate distinction. For our experiments, we set our sampling rate to every 1ms (where the default is 1000ms for tegrastats). It is important to note that while we select the sampling interval to be 1ms, in reality the utility samples at about 1.6ms. (As seen experimentally, not mentioned by NVIDIA!)

5.1.4. Dataset. Table 1 shows all the model families and the variants we consider for our experiments. There are two datasets we use. The first is Train/Test 1, for which we consider all the model variants in both the training and testing set. The entire dataset includes 3200 samples, with 400 samples from each with model families seen in Table 1. If a model family has more than one variant, there are equal sub-samples of each variant, adding up to 400 samples total for the family. To generate the Train set 1 and Test set 1, we use an 80:20 train-test split ratio.

Test set 1 provides the classification results for predicting model families from variants that are present in the training dataset. Test set 2 is created with model variants that do not exist in the training set. We created this dataset to analyze the transferability (see Definition 5.1) of our proposed attack. For model families where no other model variants exist or are not readily available, we did not consider them in the Test 2 set (N/A on Table 1). Hence, we have 5 model families in Test set 2. We include 100 samples for each family. Again there are equal sub-samples if there are variants within a family.

5.2. Training with Complete Feature Set

First, we attempt to leverage all features: CPU, GPU, and memory (RAM). The execution times for each application vary, and the classifier requires that the time series be of the same sequence length. Hence, we experimentally test sequence lengths ranging from 1.5 second to 8 seconds (on Test 1). Figure 3a demonstrates that the classification accuracy peaks and becomes stable at 99%

for a sequence length of 7-8 seconds. However, there is a significant decrease in accuracy when using a sequence length under 3 seconds. This shows that under 3 seconds, data-trace is not adequate or distinguishable amongst other families for the classifier.

We demonstrate the confusion matrices for the lowest (Figure 3b) and highest (Figure 3c) chosen sequence lengths on Test set 1. We notice there is high accuracy in identifying Alexnet, Inception, and VGG families. These network architectures are larger. A majority of ResNet samples are misclassified as DenseNet, both being residual networks. There is also noticeable misclassification amongst the smaller networks: MobileNet, SqueezeNet, and ShuffleNet. Furthermore, ResNet is also amongst these misclassifications, possibly due to ResNet18 belonging in the set, which is also a small network. Using a sequence length of 5s considerably improves the classification. Based on our results, a sequence length of 8s is used for the remaining experiments, which gives a classification accuracy of 99% on Test set 1.

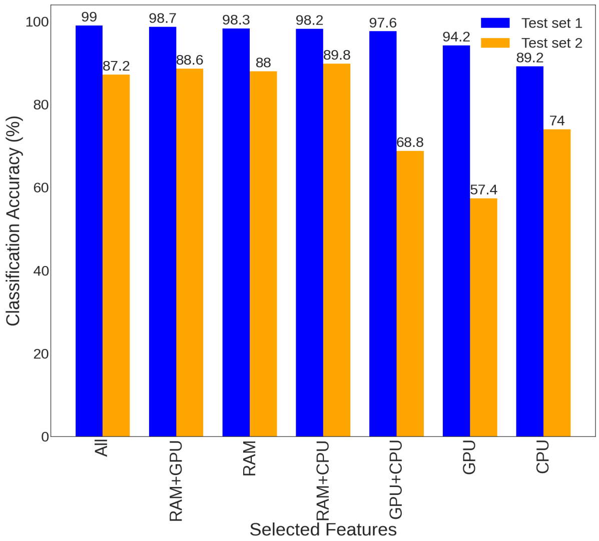


Figure 5: Classification Accuracies for different combination of features on Test set 1 and Test set 2 for the Jetson Nano device.

5.3. Feature Ablation Study

We perform feature ablation experiments to identify the importance and impact of our features—RAM, GPU, and CPU. We train classifiers with all possible combinations of these features. Figure 5 shows our findings for feature ablation performed on Test set 1 and 2. This section will discuss the findings from Test set 1. Next, we will discuss the Test set 2 findings in Section 5.4. We notice that classification accuracy with only GPU usage is 94.2% and only CPU usage is 89.2%, resulting in an accuracy drop of 4.8% and 9.8%, respectively, when compared to previous experiment that utilize complete feature set. On the other hand, using only RAM as a feature suffers an accuracy drop of less than 1%. The classification accuracy remains within 98%-99% for all combinations with RAM included as a feature. However, the classification accuracy decreases for feature combinations without RAM. This implies that RAM usage is an essential feature. We observe that the memory footprints of the applications remain broadly consistent. The CPU and GPU usages are comparatively less deterministic. Numerous system software and hardware factors affect the CPU, GPU control flow, and execution of instructions. Some of these are explored in Section 6.5.

5.4. Transferability

Definition 5.1 (Transferability)**.** A machine learning-based attack is *transferable* if it can classify instances that are not members of the attacker's training dataset with a reasonable accuracy.

Our attack looks at state-of-the-art models. However, a victim can be running a modified architecture while keeping the original backbone (e.g., changing the layers). These constitute unseen new variants. To understand the impact of new variants that do not exist in the training dataset, i.e., *transferability* of the proposed attack (per Definition 5.1), we evaluate the classifier's performance and feature selection on Test set 2 of Table 1. Test set 2 contains variants of models that do not exist in the training set. When using all features to perform classification, the accuracy on Test set 2 drops to 87.2%, which is an 11.2% decrease compared to accuracy on Test set 1. For all RAM-included combinations, the accuracy drop is around 8%-11% compared to on Test set 1. The best performing feature combination is RAM and CPU. Similar to the previous feature ablation study, the feature combinations without RAM suffer a significant accuracy drop. Notably, when using only GPU usage as an exclusive feature to build the classifier yields, we achieve the worst performance on Test set 2. From this experiment, we identify RAM usage patterns as being essential for the transferability of our model fingerprinting attack.

We demonstrate the confusion matrices for our best performing combination (RAM and CPU) and the most vital single feature (RAM) in Figure 4c. When only using RAM as input feature, we notice that DenseNet samples are misclassified predominantly as AlexNet or Inception. In Figure 2, we observe visually that the pattern or "shape" of DenseNet (Figure 11d) memory usage resembles that of Inception (Figure 8d). However, when both RAM and CPU usages are used to train the classifier, DenseNet can be more accurately predicted and are mostly misclassified as Inception. In both scenarios, DenseNet stands out from the remaining classes, having high precision but low recall. This signifies that while the memory usage is a strong feature, it helps to have CPU/GPU features to distinguish the DNN families that are closely related.

5.5. Platform Portability

Definition 5.2 (Portability)**.** An attack is *portable* to a new platform if the original attack pipeline on the new platform is comparatively as successful as on the original platform.

In this section, we evaluate the capacity of our fingerprinting strategy through the lens of *platform portability* according to the above definition. We reproduce all experimentations—performed on Jetson Nano—on two new platforms, i.e., the Jetson Xavier NX [59] and the Jetson TX2 [58].

5.5.1. Device Specifications. The *Jetson Xavier NX* has a 6-core NVIDIA Carmel ARM CPU, 384-core NVIDIA

Features Dataset	All	RAM	GPU	CPU	RAM+GPU	RAM+CPU	GPU+CPU
NX Test set 1	98.5%	98.6%	94.5%	83.2%	**98.9%**	98.2%	94.7%
NX Test set 2	79.8%	76.8%	79%	65.2%	**86.6%**	77.4%	77.2%
TX2 Test set 1	98.9%	**99.7%**	97.5%	90.8%	99.5%	97.9%	97.1%
TX2 Test set 2	**95.6%**	88.8%	60.6%	89.6%	93.4%	94.0%	82.0%

TABLE 2: *Platform portability* classification accuracy results for Jetson Xavier NX and Jetson TX2 for all features.

Volta GPU (with 48 Tensor Cores), and 8GB (128-bit LPDDR4x, 59.7GB/s) Memory. Furthermore, the device has two NVDLA Engines, which are the NVIDIA DL accelerators. There are three power modes: 10W, 15W, and 20W, which differ in the number of active CPUs. For instance, at default 15W, only two of six CPUs are active; the rest are turned off. To avoid testing the various power modes, we collect data from the Jetson NX using the default 15W (with two active CPUs) mode. Next, the *Jetson TX2* has a Dual-Core NVIDIA Denver 2 and a Quad-Core ARM Cortex-A57 MPCore CPU (hence, total six cores), a 256-core NVIDIA Pascal GPU, and 8GB (128-bit LPDDR4, 59.7GB/s) Memory. Similar to the NX, on the default mode the TX2 only has four active CPUs. There are two power modes: 7.5W and 15W. For AI performance, the Jetson NX can deliver 21 TOPS and the Jetson TX2 can deliver 1.33 TFLOPS.

5.5.2. Experimentations. The experiment setup is identical to the Nano settings in Section 5.1. The same DL applications are run as we collect the NX and TX2 system traces (still using tegrastats) to construct Train, Test set 1, and Test set 2 of NX and TX2. These sets entail the same models and variants as the test sets used for the Nano experiments, only that the system traces are collected on the NX and TX2 platforms instead. It is important to note that both the NX and TX2 have a significant performance boost over the Nano, having more powerful resources (Section 5.5.1). This means that an attacker has to re-evaluate suitable sequence lengths to develop its classifier, as the execution times vary between platforms. Furthermore, one other difference with the NX lies in the number of CPUs available. In the Nano experiments, we leverage the traces from four CPUs while we have only two in this device as we chose the default power mode. This is why the attacker needs to know the target device Section 4.1 and build a new classifier targeting that device.

5.5.3. Results. We conduct the same experiment as in Section 5.2, e.g., training the classifier with all RAM, CPU, and GPU features with a range of sequence lengths from 1s to 5s (smaller range due to the improved execution time for DNN applications). Again, we notice a similar trend as seen in Figure 3a, where the accuracies stabilize at about 98.5% for sequence lengths of 3.5s-5s. We chose a sequence length of 4.5s and reproduced classification accuracy results on Test set 1 and Test set 2 bot both TX2 and NX.

We observe some akin trends between the fingerprinting results achieved on the Nano, NX, and TX2. We find that the model fingerprinting method still achieves high classification accuracy on a different platform. RAM is again a vital feature, with which accuracy on known model variants (Test set 1) remains high at around 98% for both platforms. The combination of RAM and GPU statistics produces the (marginally) highest accuracy (98.9%) on NX Test set 1. For TX2 Test 1, the higher accuracies are achieved using only RAM (99.7%), and the combination of RAM and GPU (99.5%) as features.

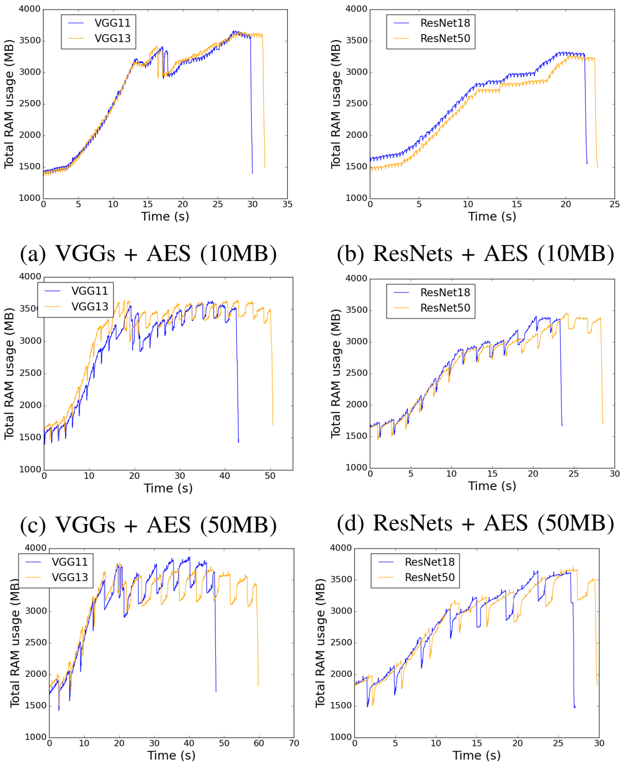
Transferability. We also execute the transferability experiment (i.e., fingerprinting unseen variants of models) on the NX and TX2. Here, we observe some differences. Using the Nano as our platform, we obtained an average accuracy of 88.4% with RAM-based features combinations Figure 5. However, we observe a considerable decrease in classification accuracy for our transferability experiment on the NX test set 2 (See Table 2). Only the combination of RAM and GPU features enables attackers classifier to obtain performance close to that noticed on the Nano with an accuracy of 86.6%. The remaining feature combinations yield an accuracy of about 77%-79%, with the notable exception of CPU as a standalone feature (lowest-performing with 65.2%). If we focus on RAM as a single feature, we notice that the primary performance degrade stems from the misclassification of DenseNet as ResNet (see Figure 7a). Observing the memory-usage patterns or “shape” for both ResNet and DenseNet families on the NX (Figure 9), we discern that they are visually similar. We encounter the evaluation in Section 5.4, where when the memory pattern usages for families are very similar, other system information aids the classification. Likewise, in our NX experiment, using RAM aided with GPU statistics as features enables us to improve detecting DenseNets (Figure 7c).

On the other hand, the TX2 test set 2 classification performance, i.e., transferability, is excellent (Table 2) and better than that of Nano and NX. Using GPU alone has the lowest performance here (60%), but the remaining feature combinations perform considerably well. In particular, using all features and combination of RAM and CPU achieve an accuracy of 95.6% and 94.0%, respectively, which are close to the performance on Test set 1. From the confusion matrix for Test set 2 (trained on all features) we notice that ResNets are the only model that have poor performance (Figure 7n).

5.6. Robustness to a Background Noise or Application

Definition 5.3 (Robustness)**.** A machine learning-based attack is *robust* to modifications to the original pipeline/assumptions if the attack — the original classifier — is comparatively as effective as after the modifications.

We explore the impact of a background application running simultaneously with our targeted (victim) application. While it is realistic to assume the DL inference application is the only running program on a resource-constrained edge device such as the Jetson Nano, other secondary or housekeeping applications can be running in the background. These applications could involve image processing, video encoding/decoding, cryptography-based



(a) VGGs + AES (10MB) (b) ResNets + AES (10MB)

(c) VGGs + AES (50MB) (d) ResNets + AES (50MB)

(e) VGGs + AES (100MB) (f) ResNets + AES (100MB)

Figure 6: *Background noise:* Memory usage on Jetson Nano for the original pipeline (classifier) with a background application running — AES algorithm on a plaintext of fixed sizes, e.g., 10MB, 50MB, and 100MB. The remaining figures are in Figure 13.

applications for secure communication, etc. Hence, we choose to explore the impact of such a secondary application running in the background, competing with the system resources needed for our attack. We decided to explore the AES algorithm (encryption and decryption) on randomized plaintext of different sizes for the background application. To simulate this scenario, we run our DL inference applications (on the Nano) parallel with an AES application that runs encryption and decryption repeatedly on plaintext of three distinct fixed sizes, i.e., 10MB, 50MB, and 100MB. We chose these sizes to see how the system traces are affected by different strain magnitudes on the main memory (the crucial feature in our model architecture fingerprinting).

To test the robustness of our attack to the background noise, we use the trained classifier from Section 5.1 (trained on pretrained models with no noise with all features) and test on data with our induced background noise. For these test sets, we collect traces from the same application pool and test sets (Test set 1 and 2) as shown in Table 1 under the same experimental setup as Section 5.1 while running the AES application repeatedly in parallel.

Table 3 summarizes the classification accuracy on these two test sets (Test set 1 & 2) for three scenarios: 10MB (AES BG 10MB), 50MB (AES BG 50MB), and 100MB (AES BG 100MB) plaintexts. As expected, introducing background noise affects our classifier's performance. For AES encryption/decryption running on 10MB plaintext, the disturbance/noise in memory fluctuates by 10MB (as seen in the memory traces in Figure 6). Since this is not considerable memory stress, we notice from the confusion matrices (Figure 7e, Figure 7f) that certain models are harder to distinguish/classify. In particular, ResNets and DenseNets suffer the most. Since the ResNet family consists of varied model sizes (ResNet18, ResNet34, ResNet50, ResNet101, ResNet152), it is likely that the bigger models (ResNet101, ResNet152) are misclassified as DenseNets or Inception and the smaller models (ResNet18) as ShuffleNet. The same can be said about DenseNets. We also notice that with smaller (10MB) noise, the classifier performance is better (86.4%) for model variants that exist in the training set than variations that the classifier has not seen (69.6%).

Dataset / Background app	Test set 1	Test set 2
AES BG 10MB	86.4%	69.6%
AES BG 50MB	42.6%	38.6%
AES BG 100MB	16.9%	21.4%

TABLE 3: *Robustness* classification accuracy results for traces with a background noise. Model was trained on original pretrained models pipeline on Jetson Nano (no noise) and Test set accuraries show classification performance on Test sets with background noise of varying memory-stress (10MB, 50MB, 100MB).

The extreme case, AES BG 100MB, results demonstrate too much noise/disturbance in the data, which thwarts the classification (to the 17%-21% range for both test sets). Figure 6 visually shows that we lose the distinguishability in our memory consumption patterns with such a larger noise. In practice, running a memory-hungry application like AES BG 100MB would also affect the victim model's performance and inference time significantly. Hence, such a background process is unlikely to co-exist. We include this test case here for completion. In addition, AES 50MB results show that it is between 10MB and 100MB in terms of classification and application runtime performance as expected. However, from the confusion matrices (Figure 7g, Figure 7h) we see that AlexNet and VGGs in particular are still classifiable. This demonstrates the general trend that our classification performance decreases with increasingly memory-hungry secondary applications, but the primary running victim application suffers from decreased runtime performance.

5.7. Robustness to Modified DNN Models

Our attack identifies popular DNN model families on their pre-trained ImageNet weights. It is realistic for applications to use these state-of-the-art popular models on various applications. A user can directly run these pretrained models in many cases, especially since they are trained on ImageNet with 1000 classes. However, it may also be beneficial for these models to be adapted to other tasks or different classification requirements in certain conditions. Hence, transfer learning [64] is a popular methodology that aims to boost the performance of classifiers on new fields by transferring the obtained learning knowledge of previous but different and related-source fields. Generally, model weights are kept as original, and the last few layers are retrained for the new target field.

The studied state-of-the-art models in this work are also commonly used for transfer learning tasks. Hence, we modify our pipeline to adjust for CIFAR10 [65] dataset in this section. CIFAR10 only has ten output classes, and the images are smaller in size (32×32). Hence, the last layer of the models was modified to output ten labels, and CIFAR test set data was shuffled and passed into the model for inference. The rest of the experiment setup remains the same as Section 5.1. Only InceptionV3 is missing from this dataset since the network cannot take in the smaller input size.

Similar to the background application/noise study in Section 5.6, we investigate the robustness of our attack to this variation on our original classifier trained on original ImageNet pretrained data by testing it on the Test set 1 % 2 from this new CIFAR10 pipeline traces. Our classifier achieves *71.7%* and *82.4%* accuracy on Test set 1 & 2 respectively. Again, we turn to the confusion matrices to observe specifically which model families are being classified well and which are not. We find that AlexNet performs poorly in this experiment, which none classified correctly in Test set 1 (see Figure 7k). This is the main contributor to drive the accuracy down. The other models follow similar trends to those already seen in prior sections, where DenseNets are often misclassified as Inception and MobileNets as ShuffleNets. The cause of this misclassification is most likely the similar model sizes. ShuffleNets, SqueezeNets, and VGGs are predicted with high accuracy. We notice similar trends in Test set 2 as seen from Figure 7l.

5.8. Attacking a Different DL Framework: TensorFlow

Other than PyTorch, TensorFlow [66] is the alternative and popular open-source framework for deep learning. Both frameworks are optimized for NVIDIA GPUs and can utilize cuDNN and CUDA kernels for improved performance. One of the key differences between the two frameworks is code execution. PyTorch natively implements dynamic computational graphs, while TensorFlow implements static computational graphs.

We replicate our experimentation done on PyTorch in Section 5.1 but using TensorFlow and Keras pretrained models on the Jetson Nano. Keras is a library that runs on top of TensorFlow, and the Keras Applications API provides us with popular ImageNet pre-trained models [67], similar to `torchvision` for PyTorch. Not all models available on `torchvision` are found on Keras, hence we take the subset of model families and variants that are available, and divide them into a similar fashion seen in Table 1: Train/Test set 1 including DenseNets (121, 201), MobileNets (MobileNetV2), ResNets (ResNet50), and Inception (InceptionV3) and test set 2 including DenseNet201 and MobileNet. Form our original model families chosen in PyTorch; only a few are available on TensorFlow. We noticed that while VGG16,19 and ResNet101,152 were available on Keras, we could not collect traces from these models since they were too large. The Jetson Nano faces an out-of-Memory (OOM) error while running these models — most likely due to allocation of static graphs requiring more memory, and the Nano is a memory-constraint device.

Using our RF classifier trained on Train/Test set 1 on all features, we achieved *99.1%* and *94.0%* accuracy when testing this trained model on Test set 1 and 2, respectively. One possible reason for having a comparatively higher accuracy on Test set 2 is fewer classes and models in our TensorFlow dataset. We observe from these results that TensorFlow's system utilization for DL inference can also exhibit differences amongst different model families. Thus, we still can identify that the traces are classifiable with this preliminary study on TensorFlow. We also present the confusion matrix for both Test set 1 (Figure 7o) and Test set 2 (Figure 7p) experimentation.

5.9. Enhancing Adversarial Attacks

Adversarial attacks involve adversaries manipulating an input by adding imperceptible perturbations to deceive a machine learning system (e.g., DNNs) [68], [69]. For example, non-targeted attacks [12] modify inputs such that the victim model produces an arbitrary incorrect output, while targeted attacks [12] lead the models to produce a specific (incorrect) label. In a black-box setting, the adversary generates the adversarial examples without knowing the victim model. In some instances, the adversary may have limited knowledge, such as the model's architecture, referred to as a semi-black-box attack [68]. Therefore, in the black-box (and semi-black-box) setting, the adversary builds a substitute (or an ensemble of) model(s) to generate the adversarial examples. In our case, the attacker does not know what model (architecture) is running on the device but knows that it is a popular state-of-the-art readily available model (i.e., limited knowledge). In our case, the attacker generates adversarial examples from an ensemble of models. Here, we illustrate that the proposed model family fingerprinting attack boosts the adversarial attack performance with the extracted knowledge of the model family in the semi-black-box setting.

5.9.1. Experimental Setup. DeepSniffer [9] conducted experiments to show that the knowledge of model architecture improves the success rate of (targetted) adversarial attacks [9]. We take inspiration from their work to develop our testing pipeline. For experimentation, we develop our pipelines using TensorFlow [66] for the pretrained models and the Adversarial Robustness Toolbox (ART) library [70] for the adversarial attack. All models are trained using transfer learning on CIFAR10 dataset. In our experiments, we use DenseNet121 as our victim model. We use the DeepFool [18] attack from the ART library to attack the model by generating adversarial examples from an ensemble of different models and scenarios — which we refer to as a set. In the first scenario sets, the ensemble examples were generated using the models belonging to the same family: 1) ResNet Family (ResNet50, ResNet101, ResNet152), 2) MobileNet Family (MobileNet, MobileNetV2), and 3) VGG Family (VGG16, VGG19). The second scenario sets include ensemble examples generated from a random mix of models: 4) Mix 1 (MobileNet, ResNet50, EfficientNet), 5) Mix 2 (MobileNet, VGG16, EfficientNet), 6) Mix 3 (MobileNet, DenseNet121, EfficientNet), and 7) Mix 4 (ResNet152, MobileNetV2, DenseNet201).

Lastly, we generate adversarial examples from an ensemble of the victim model's architecture family, 8) DenseNet Family (DenseNet121, DenseNet169, DenseNet201). For each ensemble, we randomly select 6000 images from the CIFAR10 test set, with an even distribution amongst models inside an ensemble (e.g., for

Adversarial examples generated by DeepFool on the ensemble of	Classification accuracy of victim model DenseNet121		Accuracy drop (%)
	W/o adv. perturbation (%)	W/ adv. perturbation (%)	
ResNets (ResNet50, 101, 152)	84.14	61.02	23.12
MobileNets (MobileNet, V2)	83.43	59.46	23.97
VGGs (VGG16, 19)	82.73	51.07	31.66
Mix 1 (MobileNet, ResNet50, EfficientNet)	83.85	63.9	19.95
Mix 2 (MobileNet, VGG16, EfficientNet)	83.69	58.08	25.61
Mix 3 (MobileNet, DenseNet121, EfficientNet)	83.72	53.55	30.17
Mix 4 (ResNet152, MobileNetV2, DenseNet201)	83.07	52.02	31.05
DenseNets (DenseNet121, 169, 201)	83.13	28.23	54.9

TABLE 4: The knowledge of the victim model family, DenseNet, strengthens the adversarial attack, DeepFool [18].

ResNet Family, 2000 images each from all three chosen ResNet models). From these selected images, we generate adversarial examples using the DeepFool attack. This process is repeated for all sets. Finally, we feed the sets with and without the adversarial attack to the victim model, DenseNet121, to analyze the classification performance.

5.9.2. Results. Table 4 summarizes our findings. The baseline classification accuracy for benign examples, i.e., without adversarial perturbation, is around 83% for each set. We notice that adversarial examples generated from the wrong model family sets are less efficient. Using the ResNet family or MobileNet family to generate the adversarial examples drops the victim model accuracy by about 23%-24%. Adversarial examples from the VGG family perform slightly better, managing to drop the accuracy by 31.66%. We also demonstrate the effect of using a random mix of models to generate adversarial examples in Mix sets 1-4. We notice that mix sets without having a variant of DenseNet (Mix 1 & 2) reduce the accuracy by 19.95% and 25%, respectively. On the other hand, if a variant of the victim model exists in the set, the examples generated are more effective but still only able to decrease the victim model accuracy by 30%-31%. Finally, we notice that with the adversarial examples generated from the DenseNet family set (from which the victim model belongs), the victim model accuracy dropped by a substantial 54.9%, i.e., almost double than all other cases.

To conclude, with the knowledge of the victim model family, an adversary can generate much more effective adversarial examples from an ensemble of that family. We demonstrate that this knowledge of model architecture family in our semi-black-box scenario has a greater success at degrading victim model classification performance than an attacker using either a random mixture of models or guessing the wrong model family.

6. Discussion

6.1. DNNs and Memory Usage Distinguishability

We observe that memory usage is a vital feature for distinguishing between DNN families explored in our paper. This is intuitive and corroborates the recent findings that DNNs are significantly different in terms of their memory utilizations [71], [72]. Biano et al. [71] presents a thorough analysis of state-of-the-art DNNs for image classification tasks in terms of computational cost and accuracy. Specifically, they explore accuracy rate, model complexity, memory usage, computational complexity, and inference time. The most exciting finding for our work complements their observation of a linear relationship between model complexity (i.e., the initial static allocation of the model parameters) and the total memory utilization. Especially, Biano et al. claim that *"model complexity can be used to estimate the total memory utilization reliably."* Therefore, the core observation of our introduced attack demonstrates the opposite direction of the statement is also true, such as we estimate the model complexity employing the total memory utilization. Furthermore, we notice that the variants of a family of DNN models are grouped from this work.

6.2. Disturbances in Memory Utilization

We assume that only one application would be running at a time on the edge device. We consider this realistic, as many edge devices are dedicated to a specific DNN task. However, there may be cases where a device can be running other background applications or even multiple DNN tasks. The OS periodically performs context switching (to achieve concurrency), typically ranging from 10-50 milliseconds. The initial task is halted (and its progress is saved), and the OS switches to a new task for execution. The initial task is resumed from where it was halted. This would cause disturbances or interruptions in memory utilization. Furthermore, modern CPUs have multiple cores, which can also exploit parallelism for processes and tasks. For tasks running in parallel, it introduces more disturbances in memory utilization.

We explored such a case in Section 5.6, where we test our trained model on traces collected with a parallel secondary background application with varying memory-stress. We notice that while with significant disturbances in memory usage, the classification performance degrades, we also see that the application performance is negatively impacted. Especially for compute-constraint edge devices such as the Jetson Nano, running multiple memory-hungry applications is not realistic, especially when runtime performance is an essential factor. Furthermore, since the attacker can run the DNN models without noise on a targeted device, they may be able to identify if the victim traces collected are with or without noise (statistically) and discard those traces since the predicted model family may not be as accurate. However, with minor amplitude noise, the classifier is still able to get about 86% accuracy (see Section 5.6).

6.3. Stronger Attacker against the Odds

In Section 5.6 and Section 5.7, we explore variations and noise and their impact on our classifier's performance. Since our attacker builds their own labeled dataset, they can build a stronger classifier to account for some variations, especially if the attacker gains some more prior knowledge. For instance, if the attacker knows that the victim model inputs an image of a specific size or knows the output labels (which is possible if the model is available to the public to run on or the attacker can query it). Any additional knowledge may facilitate the attacker to encompass it into creating a labeled dataset closer to the targeted victim and build a stronger classifier. We also already discussed how an attacker may even be able to identify disturbances and noise in the observed traces from the victim since they know how the ideal/clean traces would look. A stronger attacker may be able to incorporate any additional knowledge available to their advantage when fingerprinting the model. This is an interesting future work to explore.

6.4. Generalization to Other DL Applications

We explore the idea of fingerprinting model architecture families through global system statistics. In this paper, we target image classification tasks using different DNN model architectures. Our proposed attack pipeline can be generalized to other learning tasks (e.g., object or behavior detection) as long as the attackers have access to commonly used DNN models and inference pipelines deployed on CPU-GPU edge devices (refer to the threat model in Section 4.1). This is a reasonable assumption given that DL application developers tend to reuse well-known DNN models with publicly available code from public repositories (e.g., GitHub) or online development forums (e.g., Jetson Community Projects page [73]).

6.5. Differences in Application Runtimes and Performance

Even though we define and use a typical model inference pipeline (Section 4.2.1), the run time performance of the applications, such as execution time, can be unpredictable since various factors may influence it.

From high-level programs, compilers generate instructions at runtime. As a result, compilers transform the programs in a complex and optimized manner, making it challenging to identify deterministic execution times of high-level statements. Furthermore, it is challenging to determine the execution time of instructions due to the variations in CPU performance like concurrency, data hazards, resource availability, etc. Memory systems are another source of variance– cache hits and misses can hugely impact programs' performance and execution time.

In our case, we explore the (global) memory usage in terms of the total RAM in use by the system. The pattern for the total memory usage may not be drastically invariant, with programs needing to use fixed amounts of memory during execution. However, the time taken for memory usage can differ on various accords. For instance, we utilize our device on performance mode to collect traces. This mode provides a performance boost (improving execution time) for the Jetson devices to perform computationally expensive tasks such as DNN inference. In addition, it acts to make instruction execution quicker with a higher clock rate. Our observation is that with and without performance mode, the applications exhibit similar memory usage patterns (as seen in Figure 2), with the primary difference being the execution time.

6.6. Potential Countermeasures

Our proposed method relies on profiling system statistics to fingerprint model architectures. The victim DNN application being the sole running process on the victim system allows us to exploit the system-level traces. Therefore, we can introduce noise to the system CPU, GPU, and memory profiles to disrupt the attack as a countermeasure. As discussed earlier in this paper, this may introduce a security-performance trade-off. Furthermore, while running tegrastats is stealthy and passive from the victim's perspective, it is still a running process that the OS can identify. Our proposed fingerprinting attack runs the utility tool in the background to avoid detection. A potential defense against this is to introduce an application/software that constantly checks all running processes and terminates suspicious use of the utility tool. Another solution is to disable tegrastats or make the utility accessible only with `sudo` access. This is similar to disabling hardware performance counters (HPCs) in android devices to prevent side-channel leakage-based attacks [61].

7. Conclusion and Future Work

The proliferation of GPU-enabled embedded systems such as NVIDIA Jetson devices has made DL applications—such as image recognition, object detection, and so on—deployable on edge. Nevertheless the popularity it gains, there remains potential tension for users to protect their sensitive data and for developers to preserve the intellectual property of the DNN model, including its architecture. We introduce a novel supervised machine learning-based attack to identify the running DNN model architecture family on CPU-GPU shared memory edge devices. We exploit a stealthy collection of system-level side information, including memory, CPU, and GPU usage, accessible from the user-space level to detect the victim DNN model architecture without physical or privileged access. Our rigorous feature analysis shows that sole RAM utilization can detect a model architecture with high classification accuracy. Furthermore, for DNN applications with similar memory usage patterns, combining external features such as GPU and CPU usage can give better classification.

Our results illustrate that a Random Forest-based attackers classifier built on RAM usage time-series feature alone can recognize a DNN model architecture with a 98.3% success rate. Furthermore, our presented attack has three exciting properties. Firstly, Transferability—it can detect an unknown DNN model variant with an 89.8% success rate using RAM+CPU aggregate statistics. Secondly, Platform portability—we clone the attack successfully to a different GPU-enabled edge device and obtain similar classification accuracy in stealthy fingerprinting. Lastly, because of the robustness of the attack, we employ the original attacker classifier to detect the DNN model family type when there is another memory-hungry application running and when the last layer of the DNN model is modified to adapt transfer learning. Furthermore, we assess how the extracted knowledge of the model architecture boosts the success rate of a semi-black-box adversarial attack.

A potential avenue for future work is extending our proposed attack on an open-world scenario. It would be interesting to inspect building a more robust classifier that generalizes well for more modifications and noise.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This material is based upon work supported in part by the Robert N. Noyce Trust.

References

[1] "Jetson modules." https://developer.nvidia.com/embedded/jetson-modules. Accessed: 2021-06-21.

[2] "Jetson benchmarks." https://developer.nvidia.com/embedded/jetson-benchmarks. Accessed: 2021-06-21.

[3] R. Hadidi, J. Cao, Y. Xie, B. Asgari, T. Krishna, and H. Kim, "Characterizing the deployment of deep neural networks on commercial edge devices," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 35–48, IEEE, 2019.

[4] A. Zankl, H. Seuschek, G. Irazoqui, and B. Gulmezoglu, "side-channel attacks in the internet of things: threats and challenges," in *Research Anthology on Artificial Intelligence Applications in Security*, pp. 2058–2090, IGI Global, 2021.

[5] M. Devi and A. Majumder, "Side-channel attack in internet of things: a survey," in *Applications of Internet of Things*, pp. 213–222, Springer, 2021.

[6] D. R. Gnad, J. Krautter, and M. B. Tahoori, "Leaky noise: New side-channel attack vectors in mixed-signal iot devices," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 305–339, 2019.

[7] Y. Xiang, Z. Chen, Z. Chen, Z. Fang, H. Hao, J. Chen, Y. Liu, Z. Wu, Q. Xuan, and X. Yang, "Open DNN box by power side-channel attack," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 11, pp. 2717–2721, 2020.

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

[9] X. Hu, L. Liang, S. Li, L. Deng, P. Zuo, Y. Ji, X. Xie, Y. Ding, C. Liu, T. Sherwood, *et al.*, "Deepsniffer: A DNN model extraction framework based on learning architectural hints," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 385–399, 2020.

[10] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pp. 506–519, 2017.

[11] "Windows developer: About performance counters." https://docs.microsoft.com/en-us/windows/win32/perfctrs/about-performance-counters. Accessed: 2021-09-22.

[12] Y. Liu, X. Chen, C. Liu, and D. Song, "Delving into transferable adversarial examples and black-box attacks," *arXiv preprint arXiv:1611.02770*, 2016.

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

[14] Y. Liu and A. Srivastava, "GANRED: Gan-based reverse engineering of DNNs via cache side-channel," in *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pp. 41–52, 2020.

[15] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn DNN architectures," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 2003–2020, 2020.

[16] A. Akram, M. Mushtaq, M. K. Bhatti, V. Lapotre, and G. Gogniat, "Meet the sherlock holmes' of side channel leakage: A survey of cache sca detection techniques," *IEEE Access*, vol. 8, pp. 70836–70860, 2020.

[17] G. Sangeetha and G. Sumathi, "An optimistic technique to detect cache based side channel attacks in cloud," *Peer-to-Peer Networking and Applications*, vol. 14, no. 4, pp. 2473–2486, 2021.

[18] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: a simple and accurate method to fool deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2574–2582, 2016.

[19] "DNN Model Fingerprinting Library." https://github.com/kartikp7/DNN-Model-Fingerprinting.

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

[21] L. Batina, S. Bhasin, D. Jap, and S. Picek, "CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel," in *28th USENIX Security Symposium (USENIX Security 19)*, (Santa Clara, CA), pp. 515–532, USENIX Association, Aug. 2019.

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

[23] Ł. Chmielewski and L. Weissbart, "On reverse engineering neural network implementation on gpu," in *International Conference on Applied Cryptography and Network Security*, pp. 96–113, Springer, 2021.

[24] V. Duddu, D. Samanta, D. V. Rao, and V. E. Balas, "Stealing neural networks via timing side channels," *arXiv preprint arXiv:1812.11720*, 2018.

[25] G. Dong, P. Wang, P. Chen, R. Gu, and H. Hu, "Floating-point multiplication timing attack on deep neural network," in *2019 IEEE International Conference on Smart Internet of Things (SmartIoT)*, pp. 155–161, IEEE, 2019.

[26] J. Wei, Y. Zhang, Z. Zhou, Z. Li, and M. A. Al Faruque, "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, 2020.

[27] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Rendered insecure: Gpu side channel attacks are practical," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pp. 2139–2153, 2018.

[28] K. Yoshida, T. Kubota, S. Okura, M. Shiozaki, and T. Fujino, "Model reverse-engineering attack using correlation power analysis against systolic array based neural network accelerator," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, IEEE, 2020.

[29] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *Communications of the ACM*, vol. 63, no. 11, pp. 139–144, 2020.

[30] N. Matyunin, Y. Wang, T. Arul, K. Kullmann, J. Szefer, and S. Katzenbeisser, "Magneticspy," *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society - WPES'19*, 2019.

[31] N. Chawla, A. Singh, M. Kar, and S. Mukhopadhyay, "Application inference using machine learning based side channel analysis," in *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, 2019.

[32] Q. Yang, P. Gasti, G. Zhou, A. Farajidavar, and K. S. Balagani, "On inferring browsing activity on smartphones via usb power analysis side-channel," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 5, pp. 1056–1066, 2016.

[33] Y. Qin and C. Yue, "Website fingerprinting by power estimation based side-channel attacks on android 7," in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pp. 1030–1039, IEEE, 2018.

[34] H. Naghibijouybari, A. Neupane, Z. Qian, and N. A. Ghazaleh, "Side channel attacks on gpus," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[35] T. van Ede, R. Bortolameotti, A. Continella, J. Ren, D. J. Dubois, M. Lindorfer, D. Choffnes, M. van Steen, and A. Peter, "Flowprint: Semi-supervised mobile-app fingerprinting on encrypted network traffic," in *Network and Distributed System Security Symposium (NDSS)*, vol. 27, 2020.

[36] N. Chawla, A. Singh, M. Kar, and S. Mukhopadhyay, "Application inference using machine learning based side channel analysis," in *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, IEEE, 2019.

[37] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *28th USENIX Security Symposium (USENIX Security 19)*, pp. 639–656, 2019.

[38] S. Jana and V. Shmatikov, "Memento: Learning secrets from process footprints," in *2012 IEEE Symposium on Security and Privacy*, pp. 143–157, 2012.

[39] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang, "Leave me alone: App-level protection against runtime information gathering on android," in *2015 IEEE Symposium on Security and Privacy*, pp. 915–930, 2015.

[40] R. Spreitzer, F. Kirchengast, D. Gruss, and S. Mangard, "Procharvester: Fully automated analysis of procfs side-channel leaks on android," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pp. 749–763, 2018.

[41] X. Zhang, X. Wang, X. Bai, Y. Zhang, and X. Wang, "OS-level side channels without procfs: Exploring cross-app information leakage on ios," in *Proceedings of the Symposium on Network and Distributed System Security*, 2018.

[42] Y. Tu, Z. Zhang, Y. Li, C. Wang, and Y. Xiao, "Research on the internet of things device recognition based on rf-fingerprinting," *IEEE Access*, vol. 7, pp. 37426–37431, 2019.

[43] K. Yang, Q. Li, and L. Sun, "Towards automatic fingerprinting of iot devices in the cyberspace," *Computer Networks*, vol. 148, pp. 318–327, 2019.

[44] H. Liu, M. Long, J. Wang, and M. Jordan, "Transferable adversarial training: A general approach to adapting deep classifiers," in *International Conference on Machine Learning*, pp. 4013–4022, PMLR, 2019.

[45] F. Tramèr, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel, "The space of transferable adversarial examples," *arXiv*, 2017.

[46] "Torchvision models." https://pytorch.org/vision/stable/models.html. Accessed: 2021-09-14.

[47] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.

[48] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[49] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[50] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1492–1500, 2017.

[51] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

[52] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "Shufflenet v2: Practical guidelines for efficient cnn architecture design," in *Proceedings of the European conference on computer vision (ECCV)*, pp. 116–131, 2018.

[53] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[54] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826, 2016.

[55] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

[56] "Nvidia edge computing - embedded systems with jetson." https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/. Accessed: 2021-09-10.

[57] "Edge Computing: Jetson Nano Developer Kit." https://developer.nvidia.com/embedded/jetson-nano-developer-kit. Accessed: 2021-09-22.

[58] "Edge Computing: Jetson TX2." https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/. Accessed: 2021-09-22.

[59] "Edge Computing: Jetson Xavier NX." https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/. Accessed: 2021-09-22.

[60] "Edge Computing: Jetson AGX Xavier." https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/. Accessed: 2021-09-22.

[61] "Android open source project: Simpleperf." https://android.googlesource.com/platform/system/extras/+/master/simpleperf/doc/README.md. Accessed: 2021-09-22.

[62] M. Löning, A. Bagnall, S. Ganesh, V. Kazakov, J. Lines, and F. J. Király, "sktime: A unified interface for machine learning with time series," *arXiv preprint arXiv:1909.07872*, 2019.

[63] "sktime: Time series forest classifier." https://bit.ly/3uUsP79. Accessed: 2022-02-14.

[64] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He, "A comprehensive survey on transfer learning," *Proceedings of the IEEE*, vol. 109, no. 1, pp. 43–76, 2020.

[65] "The CIFAR-10 dataset." https://www.cs.toronto.edu/ kriz/cifar.html. Accessed: 2022-02-16.

[66] "Tensorflow." https://www.tensorflow.org. Accessed: 2022-02-14.

[67] "Keras applications: Available models." https://keras.io/api/applications/. Accessed: 2022-02-14.

[68] N. Akhtar and A. Mian, "Threat of adversarial attacks on deep learning in computer vision: A survey," *Ieee Access*, vol. 6, pp. 14410–14430, 2018.

[69] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial machine learning at scale," *arXiv preprint arXiv:1611.01236*, 2016.

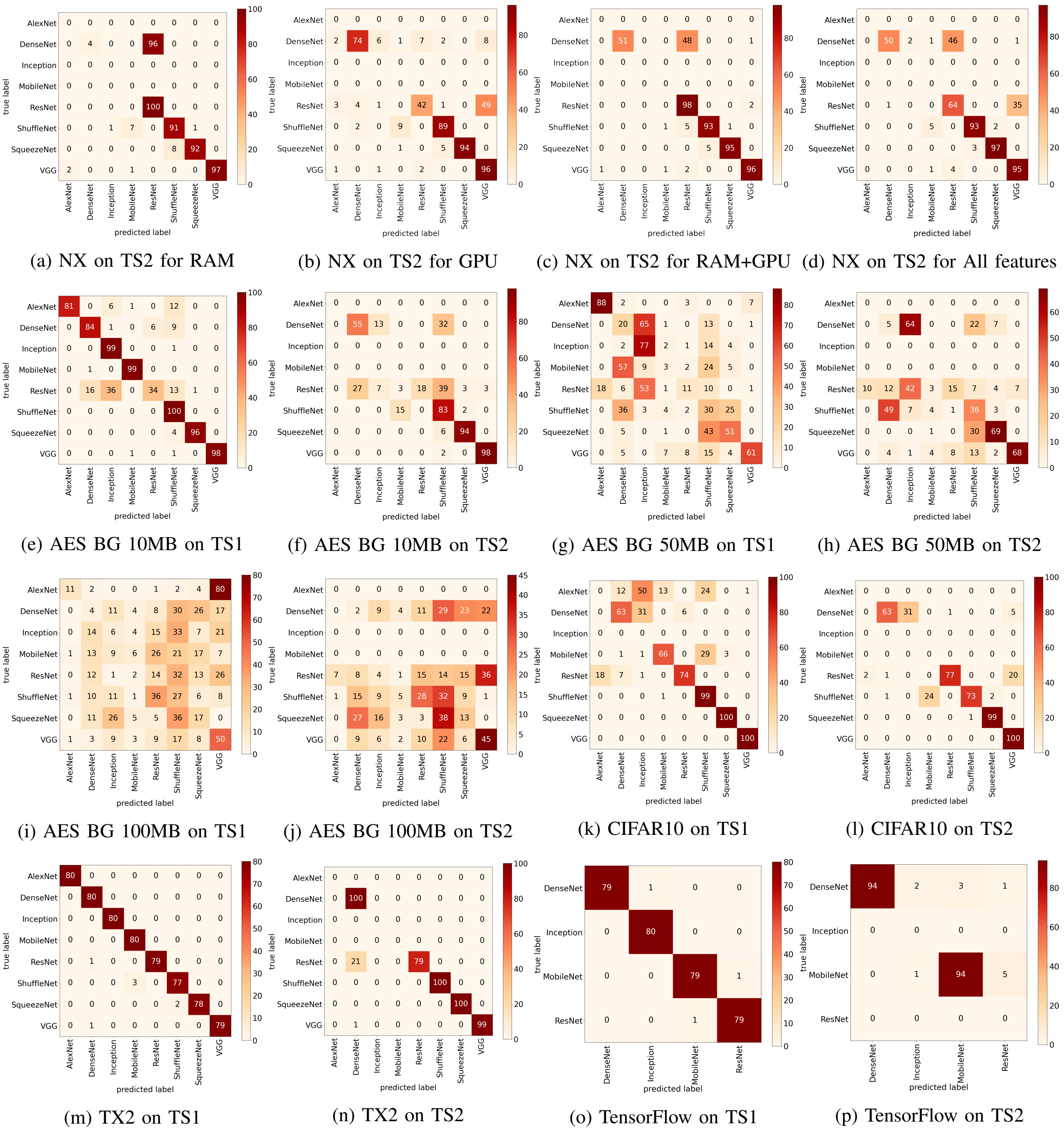
[70] M.-I. Nicolae, M. Sinn, M. N. Tran, B. Buesser, A. Rawat, M. Wistuba, V. Zantedeschi, N. Baracaldo, B. Chen, H. Ludwig, *et al.*, "Adversarial robustness toolbox v1. 0.0," *arXiv preprint arXiv:1807.01069*, 2018.

[71] S. Bianco, R. Cadene, L. Celona, and P. Napoletano, "Benchmark analysis of representative deep neural network architectures," *IEEE Access*, vol. 6, pp. 64270–64277, 2018.

[72] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Jayarajan, A. Phanishayee, B. Schroeder, and G. Pekhimenko, "Benchmarking and analyzing deep neural network training," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 88–100, IEEE, 2018.

[73] "NVIDIA Jetson Community Projects." https://developer.nvidia.com/embedded/community/jetson-projects. Accessed: 2022-02-14.

Appendix A. Additional evaluations



(a) NX on TS2 for RAM

(b) NX on TS2 for GPU

(c) NX on TS2 for RAM+GPU

(d) NX on TS2 for All features

(e) AES BG 10MB on TS1

(f) AES BG 10MB on TS2

(g) AES BG 50MB on TS1

(h) AES BG 50MB on TS2

(i) AES BG 100MB on TS1

(j) AES BG 100MB on TS2

(k) CIFAR10 on TS1

(l) CIFAR10 on TS2

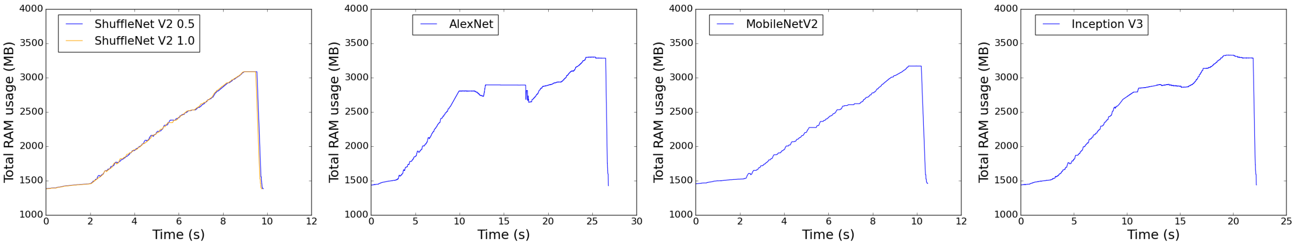
(m) TX2 on TS1

(n) TX2 on TS2

(o) TensorFlow on TS1

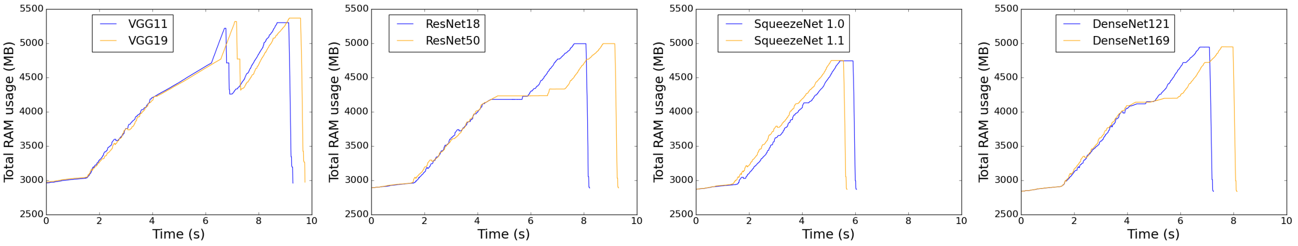
(p) TensorFlow on TS2

Figure 7: *Confusion Matrix:* Figures 7a to 7d and 7o, 7p show confusion matrices for the platform portability experimentations— 7a to 7d are for experiments conducted on NX Test set 2—RAM (7a), GPU (7b), and RAM+GPU (7c), and All features. Figures 7o and 7p are for experiments conducted on TX2 with classifier trained on all features and on Test set 1 (TS1) and Test set 2 (TS2). Figures 7e to 7j show the confusion matries for the background noise experimentations—for AES on plaintext sizes 10MB, 50MB, and 100MB on TS1 and TS2. Figures 7k and 7l show the confusion matricies for the modification to DNNs experimentation—classification on CIFAR10 pipeline on TS1 and TS2. Figures 7o and 7p shhow the confusion matricies for the different framework experimentation—trained and tested with TensorFlow data on TS1 and TS2.



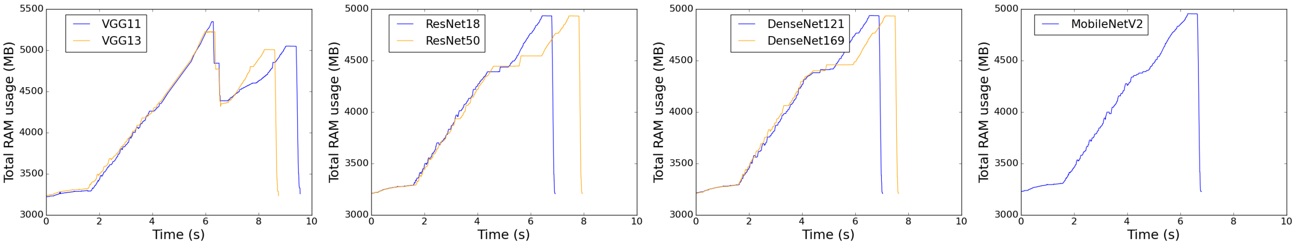
(a) ShuffleNet Family (b) AlexNet (c) MobileNet (d) Inception

Figure 8: *Jetson Nano:* (Remaining) Observed memory usage patterns on Jetson Nano



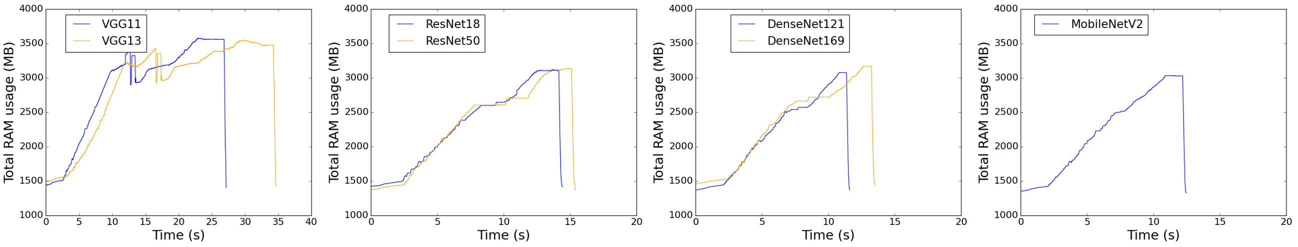
(a) VGG Family (b) ResNet Family (c) SqueezeNet Family (d) DenseNet Family

Figure 9: *Jetson Xavier NX:* Observed memory usage patterns on Jetson Xavier NX.



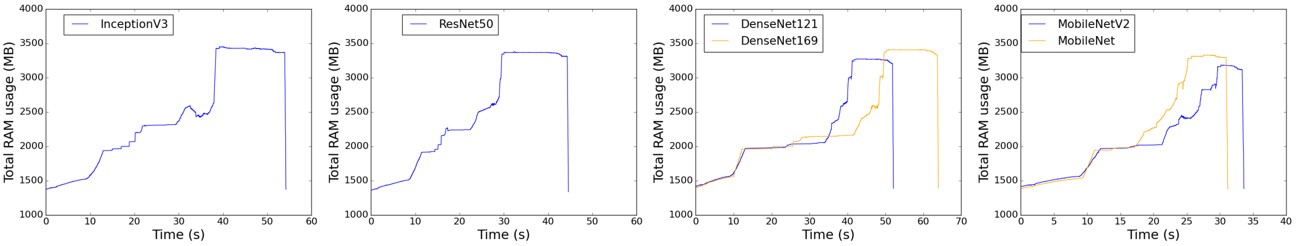
(a) VGG Family (b) ResNet Family (c) DenseNet Family (d) MobileNet

Figure 10: *Jetson TX2:* Observed memory usage patterns on the Jetson TX2. Similar to our observations with the Jetson NX.



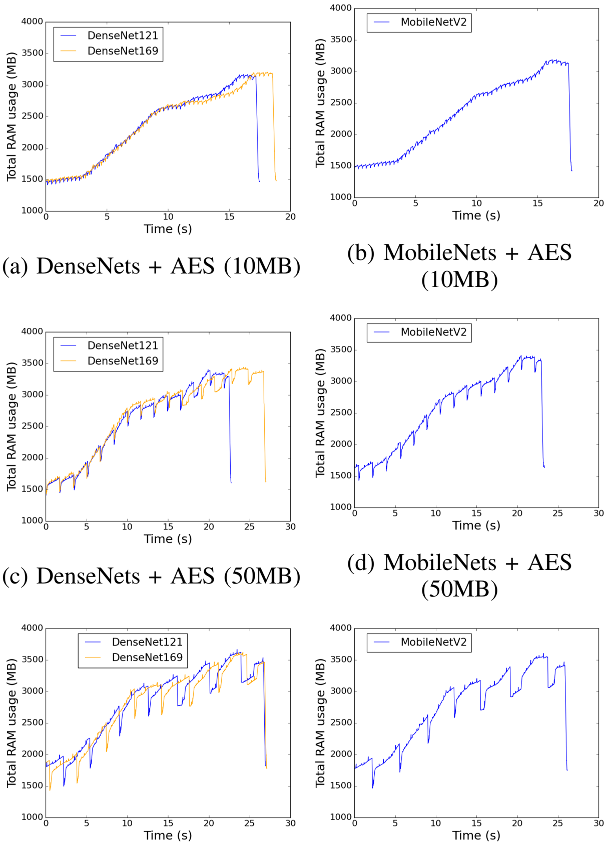
(a) VGG Family (b) ResNet Family (c) DenseNet Family (d) MobileNet

Figure 11: *CIFAR10:* Observed memory usage pattern on Jetson Nano for CIFAR10 pipeline.



(a) Inception (b) ResNet (c) DenseNet Family (d) MobileNet Family

Figure 12: *TensorFlow:* Memory consumption pattern for original ImageNet pre-trained DNN inference pipeline on the TensorFlow2.x framework.



(a) DenseNets + AES (10MB)

(b) MobileNets + AES (10MB)

(c) DenseNets + AES (50MB)

(d) MobileNets + AES (50MB)

(e) DenseNets + AES (100MB)

(f) MobileNets + AES (100MB)

Figure 13: *Background Appliication:* (Remaining) Memory usage patterns on Jetson Nano for the original pipeline (classifier) with a background application running — AES algorithm on a plaintext of fixed sizes: 10MB, 50MB, and 100MB.