March 2024

Extracting DNN Architectures Via Runtime Profiling On Mobile GPUs

Dong Hyub Kim
University of Massachusetts Amherst

EXTRACTING DNN ARCHITECTURES VIA RUNTIME PROFILING ON MOBILE GPUS

A Thesis Presented

by

DONG HYUB KIM

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

February 2024

Electrical and Computer Engineering

EXTRACTING DNN ARCHITECTURES VIA RUNTIME PROFILING ON MOBILE GPUS

A Thesis Presented

by

DONG HYUB KIM

Approved as to style and content by:

Sandip Kundu, Chair

Daniel Holcomb, Member

Russell Tessier, Member

Christopher V. Hollot, Department Chair
Electrical and Computer Engineering

ABSTRACT

EXTRACTING DNN ARCHITECTURES VIA RUNTIME PROFILING ON MOBILE GPUS

FEBRUARY 2024

DONG HYUB KIM

B.S., INHA UNIVERSITY

M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Sandip Kundu

Due to significant investment, research, and development efforts over the past decade, deep neural networks (DNNs) have achieved notable advancements in classification and regression domains. As a result, DNNs are considered valuable intellectual property for artificial intelligence providers. Prior work has demonstrated highly effective model extraction attacks which steal a DNN, dismantling the provider's business model and paving the way for unethical or malicious activities, such as misuse of personal data, safety risks in critical systems, or spreading misinformation. This thesis explores the feasibility of model extraction attacks on mobile devices using aggregated runtime profiles as a side-channel to leak DNN architecture. Since mobile devices are resource constrained, DNN deployments require optimization efforts to reduce latency. The main hurdle in extracting DNN architectures in this scenario is that optimization techniques, such as operator-level and graph-level fusion, can obfuscate the association between runtime profile operators and their corresponding

DNN layers, posing challenges for adversaries to accurately predict the computation performed. The thesis presents a novel approach for identifying the original architecture of a Deep Neural Network (DNN) based on analyzing its GPU call profile as a side-channel. Even when the optimization process has obscured layer information and introduced noise, the proposed approach can effectively determine the original structure. Additionally, we propose extraction of hyperparameters layer-by-layer from sub-layer patterns. No existing solution has extracted architectures from optimized DNN models deployed on mobile GPUs, especially in the presence of obfuscation or optimization. This research is the first to do so.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Deep Neural Networks (DNNs), have achieved remarkable performance improvements in classification and regression problems over the last decade [22, 36, 38]. This success has led artificial intelligence (AI) providers to integrate DNNs into diverse domains, such as autonomous vehicles [20] and health monitoring [49]. Importantly, the evolving business strategies for capitalizing on DNNs, such as licensing, pay-per-use, or pay-per-install models, perceive DNNs as valuable intellectual property (IP). Consequently, any compromise to the integrity of DNN IP also poses a threat to the provider's underlying business model.

Previous work has demonstrated the feasibility of a model extraction attack which enables an adversary to steal a DNN [25, 37, 43]. The consequences of this attack are devastating as the adversary will be able to release the model to the public, sell it as their own, create a competing product, or bypass paywalls. Besides breaking the provider's business model, model extraction attacks present several threats. First, DNN models are costly to develop. Each phase in the machine learning pipeline, spanning from gathering data [17] to designing DNN models [8], training, and maintaining them [33], incurs substantial expenses, underscoring the valuable nature of the model's intellectual property (IP). The training cost alone may eclipse $1m [34]. This investment is lost if an adversary steals the DNN. Second, an adversary may easily calculate the gradient of a stolen DNN, enabling them to mount further adversarial attacks with much more potency [30]. They could craft maliciously perturbed inputs which fool the DNN, giving the adversary control of the model's output, or

could reconstruct training data [1, 2]. Such attacks compromise the system in which the DNN operates.

Model extraction attacks are usually split into two steps. The first step, architecture extraction [14, 27, 39], steals the architecture of a DNN, which are the mathematical operations that define the model's computation. These operations are semantically split into DNN layers based on the type of operation such that there is no data dependency within a layer. The second step steals the parameters of the DNN which are found through training. Architecture extraction attacks typically use side-channel data and have been demonstrated in many different threat scenarios. Effective side-channels include memory and cache accesses to the CPU [13, 21, 24, 44], GPU [23, 29, 31], and DNN accelerators [15], power consumption [21, 42], electromagnetic emanations [3, 9, 27, 45], PCIe bus snooping [14, 50], and application profiles [14, 21, 29, 31, 39, 50].

In this work, we consider architecture extraction attacks applied to mobile devices. AI providers may run DNNs on mobile devices to preserve data privacy (the data never leaves the device) to limit server load (edge devices bear the computational load rather than the cloud) and to reduce latency. Because mobile devices are hardware constrained and running a DNN is computationally expensive, many hardware providers include special purpose hardware accelerators to speed-up machine learning workloads, such as the Qualcomm Snapdragon Neural Processing Engine (SNPE), Arm Mali, and Apple Neural Engine [18]. A large-scale study by Sun et. al [35] found that 54% of mobile applications include GPU support.

In addition to hardware support, AI providers reduce the DNN computational load through software optimization. Deep learning compilers like Apache TVM (Tensor Virtual Machine) [6] reduce computational load through operator-level and graph-level optimization techniques. Operator-level fusion involves applying optimizations such as vectorization, loop tiling, and reordering within each layer of the model, while

graph-level fusion combines layers to reduce the number of DRAM accesses, thereby achieving a latency benefit. For example, when there is a sequence of convolution, batch normalization, and ReLU layers, there are nine DRAM accesses required for parameters, input and output tensors. However, fusing these three layers into one layer reduces the DRAM accesses to three.

Compiler-based optimizations pose a significant hurdle in model extraction attacks, as they obscure model architectures. The TVM compiler stack itself may be employed as a defense strategy against architecture extraction k [14]. The main challenges are:

- *Operator-level fusion*: Operator-level fusion significantly alters the duration of individual operator computations, making it challenging for attackers to accurately predict their functions.

- *Graph-level fusion*: By merging multiple layers into single operators, graph-level fusion renders traditional layer-by-layer extraction methods ineffective.

- *Obfuscation*: Model architecture extraction confronts additional challenge when fused operators have arbitrary, uninformative names, a technique previously employed to protect data privacy in DNNs [16].

In our study, we employ the Apache TVM framework for optimization targeting Arm Mali G52 GPU platform. Our goal is to perform faithful model architecture extraction despite optimization. Using the TVM runtime profile as a side-channel, we extract a sequence of layer durations. Despite the inherent challenges posed by optimization-induced noise and obfuscation during the optimization process, our study reveals that employing a feature engineering technique that involves binning the layer durations allows for accurate prediction of the DNN architecture. This innovative approach enables precise identification of the original DNN architecture.

Our research distinguishes itself from the work of Wu et. al [41], Zhang et. al [48], and Liu et. al [26], who used decompilers for compiled TVM binaries that are specific to a hardware backend. Their method relies on binary analysis while our method is based on runtime profile. Binary analysis does not not deal well with obfustaction. In fact, obfuscation was proposed as a countermeasure against attack [7]. In contrast, our work is not tied to a specific hardware backend, file format of the TVM binary as proposed by Chen et. al [7], or any keywords, since we rely solely on runtime profile. Our contributions include:

1. DNN model architecture extraction from runtime profiles even when the model is optimized in a way that fuses multiple DNN layers.

2. We present a novel binning approach for profile analysis that can discern runtime features, even in the absence of feature information due to obfuscation.

3. We demonstrate that our approach achieves 100% accuracy on the Arm Mali G52 GPU with OpenCL when all models from the model zoo use fixed optimization level.

4. We further demonstrate that our method maintains >97.2% accuracy for all model zoo models, even when they are subjected to diverse optimization levels.

5. To the best of our knowledge, this is the first-ever solution to extract architectures from optimized DNN models on mobile GPUs even with obfuscation in the optimization process.

6. Lastly, we also present a layer-by-layer reconstruction solution for arbitrary models that also achieves high architecture extraction accuracy.

CHAPTER 2

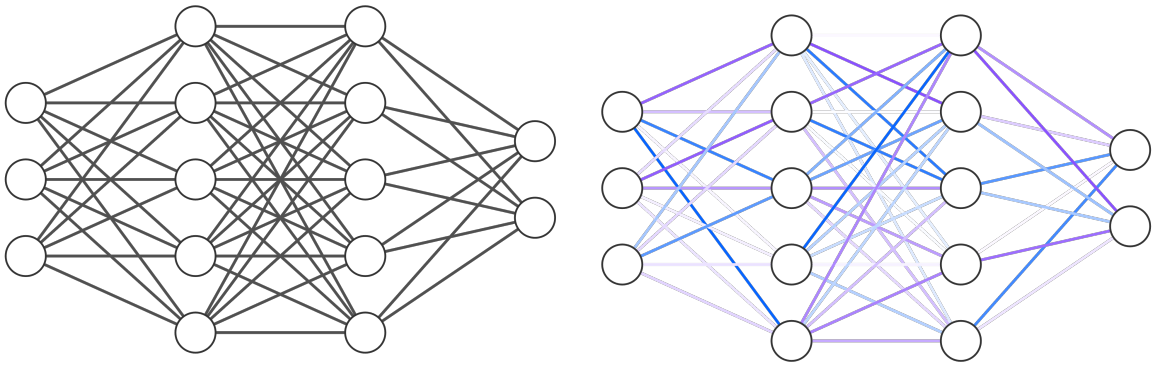
BACKGROUND AND RELATED WORK

2.1 Deep Neural Networks

The focus of this study is on feed-forward neural networks for image classification tasks. Fundamentally, these models learn a function $\mathcal{X} \to \mathcal{Y}$, where the network input $\mathcal{X} \subseteq \mathbb{R}^d$ encodes an image and the network output $\mathcal{Y} \subseteq \mathbb{R}^k$ is a categorical distribution among k classes. For some input $x \in \mathcal{X}$ and corresponding output $y \in \mathcal{Y}$, argmax(y) represents the classifier's predicted class for the input x.

The difference between a DNN architecture and a DNN model is illustrated in Figure 2.1. A DNN architecture f is the computational graph specifying equations for computing the output of the network for some input. One component of the architecture is the sequence of DNN layers, such as fully-connected, convolution, max-pooling, or batch normalization layers. Each of these layers includes hyperparameters such as padding, stride length, and number of nodes. The other architectural component is the connections between the layers, or how the data flows between layers in the computational graph. In feed-forward networks, a node will link to some subset of nodes in later layers of the network.

The DNN layer sequence, layer hyperparameters, and layer connections are all design choices when creating a DNN architecture, so the space of possible architectures is extremely large. Finding a high-performing architecture within this space is an active research problem [8]. When designing an AI application, providers are incentivized to use existing state-of-the-art architectures rather than develop their own to avoid the cost of this search [4].



DNN Architecture

Parameterized DNN Architecture

Figure 2.1: A DNN architecture is an unparameterized computation graph specifying how to compute an output of the network given some input. The parameters (weight and bias) of a DNN are found through a training process, resulting in a specific parameterization of an architecture, or a model. Distinct edge colors in a Parameterized DNN Architecture figure signify variations in parameter values being assigned.

As opposed to a DNN architecture, a DNN model f^θ is an architecture f parameterized by θ, a set of parameters obtained from training the model, including biases, weights, and batch normalization parameters. Typically, the parameters θ are found according to a training process

$$\text{argmin}_\theta \sum_{x_i \in \mathcal{D}} L(y_i, f^\theta(x_i)) \tag{2.1}$$

which is the sum of a loss function L between the ground truth value y and the model's prediction $f^\theta(x)$ over a labeled dataset $\mathcal{D}$. The training hyperparameters, such as batch size, learning rate, and number of epochs, are not represented in the model. Once the training process is complete, the parameters θ are fixed. Passing an input to the model and computing the output when the parameters are fixed is called inference.

A DNN model is inherently valuable due to the significant costs incurred at each stage of the machine learning pipeline required to create it. The collection of a dataset D and the training process described in Equation 2.1 is expensive. If the

DNN architecture is novel and achieves state-of-the-art performance, this further increases the value of the model.

2.1.1 White-Box vs. Black-Box Access

When an adversary has white-box access to a DNN model f^θ, they possess complete knowledge of both the model's architecture f and its parameters θ. Conversely, an adversary with black-box or query access to a model f^θ has no knowledge of the model's architecture or parameters, and can only provide an input $x \in \mathcal{X}$ and observe the model's output $f^\theta(x)$. From an adversarial standpoint, white-box access is far more advantageous than black-box access since the adversary can compute the model's gradient, which significantly enhances the effectiveness and reduces the cost of adversarial attacks. In this work we assume a black-box model to extract the model architecture. A surrogate model may be trained with the extracted architecture to create a *gray-box* model that is functionally a close approximation of the victim model.

2.2 Adversarial Machine Learning Attacks

Model evasion and model inversion are two adversarial machine learning attacks that exploit vulnerabilities inherent to DNN algorithms [1,2,5]. Model evasion attacks cause a DNN to produce wildly incorrect outputs using only a slight perturbation on an input. For image classification domains, the unperturbed image and the perturbed image look identical to a human, but the DNN misclassifies the perturbed image. A model evasion attack could cause a self-driving car to classify a stop sign as a speed-limit 60MPH sign or cause a facial recognition system to admit an adversary through a security checkpoint. Model inversion attacks allow an adversary to recover data used to train a DNN, which causes privacy concerns when the training data is sensitive. Critically, the efficacy of both of these attacks escalates if the adversary has white-box

access to the victim DNN and can calculate the gradient of the network. Such attacks may be accelerated significantly using the gray-box model described above.

2.3 Model Extraction Attack

A model extraction attack steals a DNN model f^θ. Before running the attack, the adversary has black-box access to the victim model, and after running the attack, the adversary will have a gray-box surrogate model that closely approximates the victim model. To run this attack, the adversary first steals the architecture of the victim model (architecture extraction) and then the adversary steals the parameters (parameter extraction). Since the architecture and parameters together completely define a DNN model, running these two steps is equivalent to stealing a DNN.

More specifically, after running architecture extraction, the adversary has a prediction $\hat{f}$ of the victim architecture f and uses that prediction to instantiate a surrogate model $\hat{f}^{\hat{\theta}}$ whose role is to mimic the victim model f^θ. The best case result for the adversary is when $f = \hat{f}$, the surrogate and victim have the same architecture, and $\theta = \hat{\theta}$, the surrogate and victim have the same parameters.

This work focuses on architecture extraction only. There are many existing parameter extraction methods [4, 10, 11, 19, 46, 47], most of which require the adversary to complete architecture extraction prior to running the algorithm.

2.3.1 Architecture Extraction

The goal of architecture extraction is to generate a prediction $\hat{f}$ for the architecture of the victim model f^θ, which has architecture f. This is usually done with a side-channel attack, where the adversary collects side-channel information during DNN inference, and then trains a model $\mathcal{A}$ to map from the side-channel information to a predicted DNN architecture. This architecture prediction model $\mathcal{A}$ takes as input side-channel information collected about the victim model $\mathcal{H}(f^\theta)$ and outputs a guess

of the DNN architecture that produced that side-channel information. Formally, this is implemented as

$$\mathcal{A}(\mathcal{H}(f^\theta)) = \mathbb{E}[f|\mathcal{H}(f^\theta)] = \hat{f} \tag{2.2}$$

The architecture prediction accuracy of $\mathcal{A}$ is calculated as the percent of correct predictions over a set of N DNN architectures

$$\frac{1}{N}\sum_{i=1}^{N} \mathbb{1}(f_i = \hat{f}_i) \tag{2.3}$$

where $\mathbb{1}$ is the indicator function evaluating to 1 if its argument is true and 0 otherwise.

The architecture prediction model $\mathcal{A}$ is typically a supervised learning model, and thus must be trained offline on a dataset of labeled side-channel examples. The amount of data required depends on the stochasticity in the side-channel and the compatibility between the side-channel data format, the architecture prediction model $\mathcal{A}$, and the training algorithm. The adversary is required to collect this dataset and therefore prefers architecture extraction methods which minimize the amount of data required and the complexity of training $\mathcal{A}$.

2.3.2 Parameter Extraction

The goal of parameter extraction is to steal the functionality of the victim model f^θ. After completing architecture extraction, the adversary has a prediction $\hat{f}$ for the victim model architecture f. The adversary creates a surrogate model $\hat{f}^{\hat{\theta}}$ and attempts to set the surrogate model parameters $\hat{\theta}$ to mimic the victim model parameters θ. This is done through an adaptation of knowledge distillation, in which a "teacher" DNN (the victim model) labels data on which the "student" DNN (the surrogate model) is trained:

$$\text{argmin}_{\hat{\theta}} \sum_{x_i \in \mathcal{D}} L(f^\theta(x_i), \hat{f}^{\hat{\theta}}(x_i)) \tag{2.4}$$

which is similar to the DNN training process described in Equation 2.1 except that the optimization parameters are $\hat{\theta}$ and the ground-truth label is replaced by the victim model's output $f^{\theta}(x_i)$. Evaluation of parameter extraction methods may either rate the surrogate model's performance on the problem domain or compare the similarity of the victim and surrogate model's outputs on the same input.

2.3.3 Adversarial Motivation for Model Extraction

An adversary's motivation for model extraction may be theft or reconnaissance. A theft-motivated adversary may damage the owner of the DNN IP by creating a competing service, selling the IP as their own, bypassing paywalls, or releasing the IP to the public. In this case, performing architecture extraction is not strictly necessary as the adversary could instantiate a surrogate model with sufficient capacity (i.e. neuron and layer count) for the problem domain and run parameter extraction directly. However the results of parameter extraction are improved if architecture extraction is completed first [28, 47].

A reconnaissance-motivated adversary seeks to mount further attacks like model evasion or model inversion. Since the adversary has white-box access to the extracted surrogate model, they may calculate the gradient of the surrogate model to perform these attacks with higher potency. Performing architecture extraction is necessary to achieve higher misclassification for model evasion and better approximation of the training data distribution for model inversion [27].

2.4 TVM Runtime Profiles

2.4.1 Apache TVM

Apache TVM [6] is an open source machine learning compiler framework for CPUs, GPUs, and machine learning accelerators. It enables DNN workload optimization across a variety of hardware backends.

2.4.2 TVM Compiler

The TVM compiler transforms a DNN implementation in a high level machine learning library into a deployable module optimized for a specific set of hardware operators. As shown by Figure 2.2, first, TVM converts the high level DNN computation graph into a framework-agnostic Intermediate Representation (IR) such as Relay IR and Tensor IR. Relay IR represents end to end model graph and Tensor IR represents low level operator implementation. Then the compiler applies different IR-specific enhancements including graph-level and operator-level optimization. The optimization level is a user-defined parameter which can take the values 0 (no optimization), 1, 2, or 3 (maximum optimization). Finally, through a code generation tool, TVM generates a hardware-specific model executable.

The compiler output executable includes 3 components: 1) the execution graph represented in json format, 2) the TVM operator library that comprises of compiled functions (i.e. OpenCL kernel script) specifically optimized for the target hardware, and 3) the parameter blobs of the model.

After the 3 artifacts are generated, model is converted into a `.tar` file format. The `.tar` file contains 2 object files containing script of those 3 artifacts. This `.tar` file is then ready to be exported to target storage and loaded to the runtime environment that is waiting inside the target device. When the `.tar` file is loaded on the runtime, components of the model are linked together and finally ready to be executed by the device.

To execute the DNN on the hardware, TVM runtime loads the `.tar` file and unzips it to obtain the `graph.json`, compiled operator library, and parameters. Then it constructs a graph from the `graph.json` file. The graph holds information on the name of the operators and the shape of the input and output data at each layer. The implementation of the operators comes from the compiled operator library. When inference is invoked, runtime module reads graph and calls the operators implemented

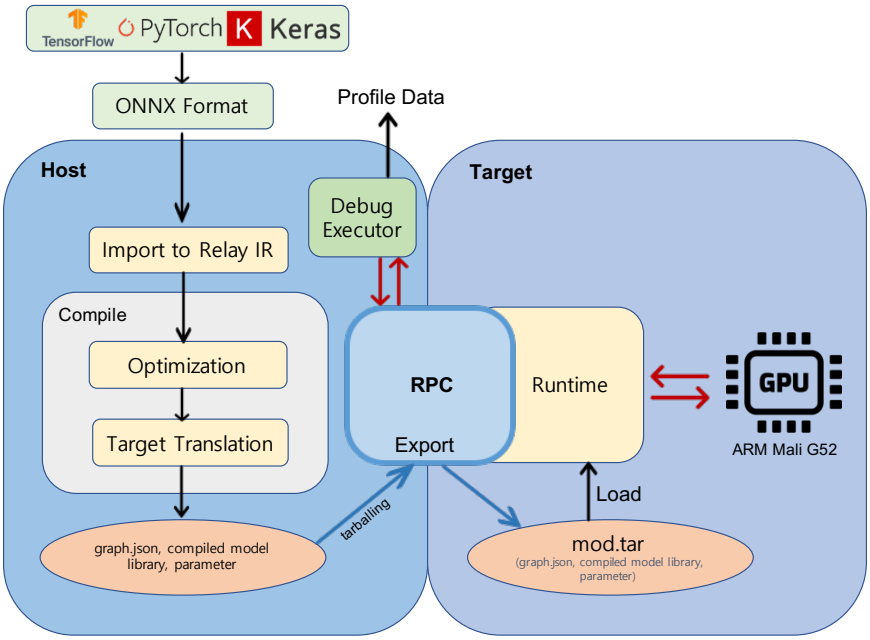


Figure 2.2: TVM Profiling system diagram.

in the compiled operator library sequentially, thereby executing by making calls to the underlying hardware API.

2.4.3 TVM Debug Executor (Profiler)

The TVM Debug Executor (referred to henceforth as the profiler) is a TVM object that can invoke the TVM runtime to run and profile the execution of deep learning models on target hardware platform. The intended purpose of the profiler is to identify and diagnose performance issues like bottlenecks, errors, and other problems that may occur during the execution of a model on specific hardware backend. The profiler yields a log containing the names of the operations being executed and their duration. The names are available due to a pairing of the profiler with the TVM runtime module allowing the profiler to see the operator names in the `graph.json` file. These operator names are automatically generated by the TVM compiler, but the operator names are arbitrary and could be changed for operator obfuscation [16]. In Figure 2.2, the red arrows indicate the commands invoked by profiler, which may

function like a remote controller that has a performance profiling ability while running DNN inference.

There are multiple options for profiling the TVM runtime without physical access to a device. A TVM remote procedure call (RPC) object enables runtime profiling across a network connection without credentials if the RPC server is running on the target device. Additionally, profiling can be performed via an SSH session or local terminal.

CHAPTER 3

ATTACK METHODOLOGY

In this chapter, we consider the attack scenario where the candidate DNN set are known. In Chapter 5, we present attack methodology where the candidate DNN architecture set is not known *a priori*.

3.1 Attack Methodology

3.1.1 Threat Model

Adversary's Objective: We consider DNNs compiled with the TVM framework at various optimization levels and deployed onto a mobile device. The adversary's objective is to extract the architecture of a victim DNN which has already been deployed from the AI provider to the mobile device.

Adversary's Capabilities: The adversary knows a candidate set of DNN architectures from which the victim DNN model is likely drawn. Also, the adversary can profile the TVM runtime during DNN inference. In order to do this, the adversary can find the path to the `mod.tar` file within an application using filename analysis like keyword matching [35] and may either enable the TVM profiler through a remote procedure call or SSH session. In either case the RPC (Remote Procedure Call) server process or Linux user must be able to access the `mod.tar` file to perform inference.

We consider this threat model in two scenarios. In Scenario 1, the TVM-generated operator names are unchanged, so this scenario is easier to attack. In Scenario 2, the AI provider may obfuscate the operator names by choosing arbitrary names. In both

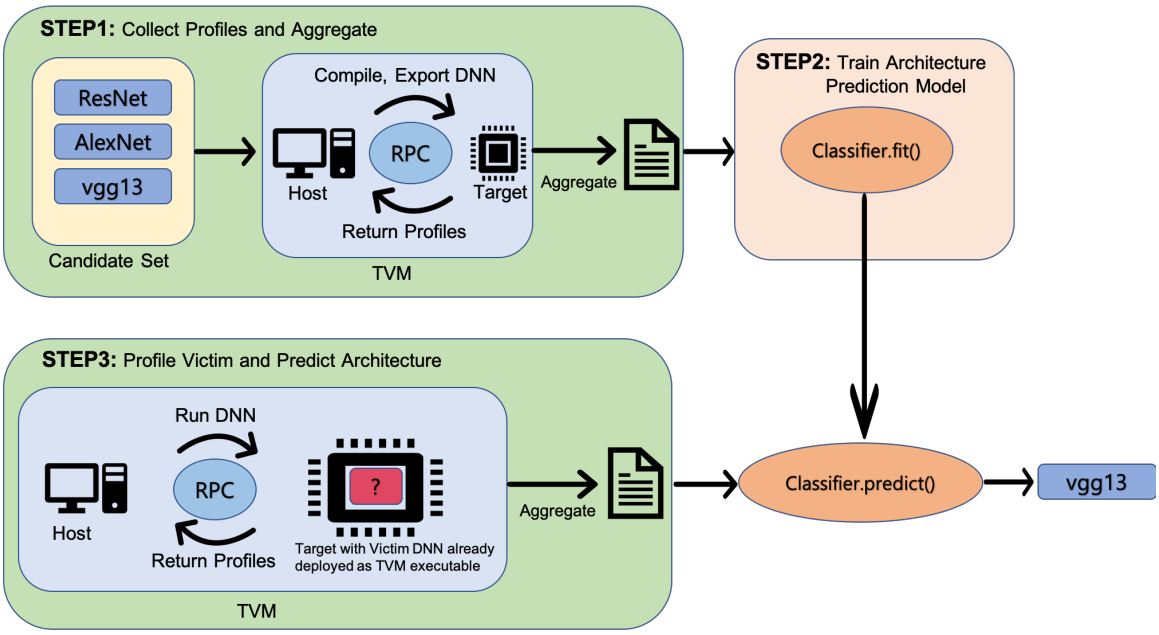


Figure 3.1: Step-by-step illustration of proposed architecture extraction attack.

scenarios we consider DNNs optimized to all TVM optimization levels sweeping from no optimization to maximum optimization.

This threat model would be applicable in a situation where a malicious smartphone user downloads an AI application containing a TVM-compiled model. Then the user may find the `mod.tar` file in the application and profile it locally or through any aforementioned method.

We note that the TVM compiler runs with a target hardware backend in mind, and different backends will run the same operations with different speeds, so the TVM runtime profile is hardware dependent. The adversary will run the online and offline steps of the attack on the same device for both Scenario 1 and Scenario 2.

3.1.2 Attack Framework

Figure 3.1 shows the proposed architecture extraction attack framework. The attack operates in a similar fashion to a generic machine learning classification workflow. In an offline step, the adversary collects a dataset of side-channel information

labeled by the DNN architecture (Step 1). Then the adversary trains a supervised learning model on this dataset to predict DNN architecture given some side-channel information (Step 2). As in several prior works [21, 31, 40, 42, 45], we formulate the prediction problem as a multiclass classification task over a candidate set of architectures. In the online attack, the adversary collects side-channel information on the victim DNN and uses the trained architecture prediction model to predict the victim DNN architecture (Step 3). Having this, the adversary would then run a parameter extraction attack to complete the DNN model theft.

3.1.2.1 Offline Preprocessing

Data Collection: Step 1 of the attack, the adversary must collect side-channel data $\mathcal{H}(f^\theta)$ on a candidate set of K DNN architectures $\{f_1, f_2, ..., f_K\}$. The side-channel data $\mathcal{H}(f^\theta)$ is a TVM profile, and the label for the data is the architecture f. The adversary collects N profiles per architecture to generate a dataset of size $K \cdot N$. As long as the architecture prediction accuracy can be maintained, the adversary prefers smaller values of N to reduce the effort of data collection. For each architecture in the candidate set, the adversary will deploy a DNN onto the target device according to the blue arrows in Figure 2.2 and profile the DNN inference according to the red arrows. After collecting this dataset, the adversary may perform preprocessing to format the data so that the architecture prediction model may achieve high prediction accuracy. We outline these methods by the threat scenario below.

Scenario 1: In Scenario 1, the names of TVM operators are not obfuscated. The TVM profile and side-channel data $\mathcal{H}(f^\theta)$ consists of a sequence of tuples (*operator_name, operator_duration*). The data may be formatted and preprocessed in many ways conducive to training an architecture prediction model, including leaving the data unprocessed [14]. However, when choosing our approach we had to take into account the effect of TVM optimizations which may fuse DNN layers. A con-

volution layer, for example, may be fused with a batch normalization layer and an activation layer. So the duration of a TVM operator may include the latency of a forward pass on multiple DNN layers. We choose to adapt an aggregation feature engineering step which has been demonstrated to be successful in similar attacks on CUDA profiles [40]. The aggregation groups all of the *operator_duration* data by the *operator_name* which enables 3 features per *operator_name*:

- The total duration spent on a TVM operator over the whole DNN inference.
- The percentage of the total DNN inference time spent on a TVM operator.
- The number of times a TVM operator was invoked over the whole DNN inference.

We also add two more features: 1) the total number of TVM operator invocations, and 2) the total duration of all invocations. By preprocessing the data in this way, the architecture prediction model may associate a set of aggregated TVM operator features with a specific architecture for each TVM optimization level.

Due to differences in layer compositions among the models, some operators are present in one model but not in another. For example, a transpose operator is present in ShuffleNet but not in VGG13. This effect is also seen by varying the TVM optimiztion level. We fill the missing values with zero so that the data is complete.

Aggregating TVM operator features in this manner creates a wide dataset from which a few features could be sufficient to perform architecture extraction. To narrow the dataset, the adversary may rank the features using Recursive Feature Elimination (RFE) [12], which has been used in prior work to shrink the width of side-channel data [31, 40]. This also reduces the complexity of training the architecture prediction model.

Scenario 2: TVM is an intermediate representation (IR)-based optimization enabled framework. Therefore there could always be the case that the AI provider

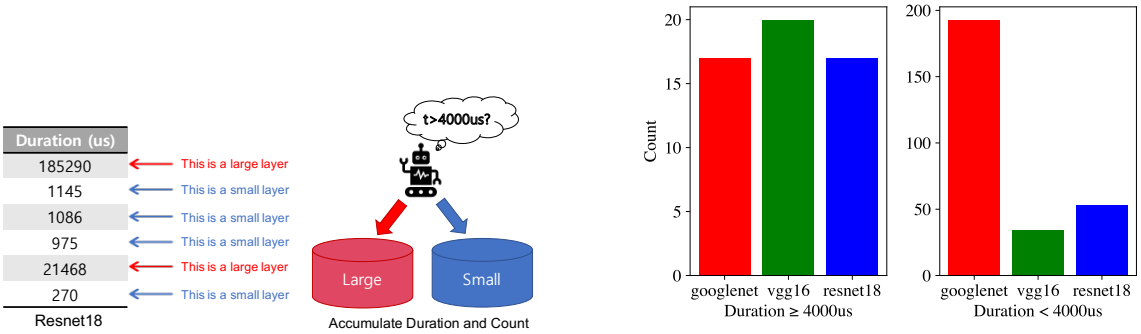


Figure 3.2: Sortation of layer duration from a ResNet18 model into 2 bins (left), Binned count distribution for different models (right).

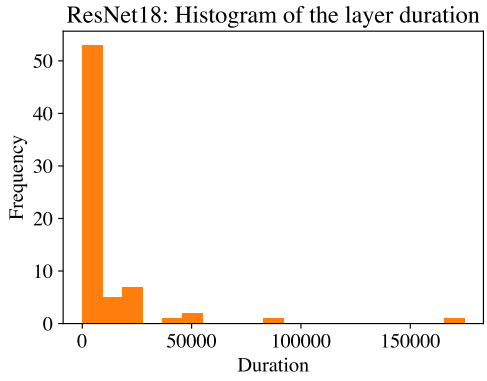


Figure 3.3: Distribution of ResNet18 layer durations at TVM optimization level 0. The x axis is in microseconds.

tunes their own layer names differently. Since Scenario 1 makes a prediction based on a layer's name, the previous approach will not work because when the adversary profiles the victim DNN in Step 3, the operator names will not match any of the operator names from the dataset collected in Step 1.

In this scenario, the names of the TVM operators are obfuscated. Like in Scenario 1, the TVM profile and side-channel data $\mathcal{H}(f^\theta)$ is a sequence of tuples (*operator_ name, operator_ duration*), except the challenge is that the operator names are arbitrary.

In response to this challenge, we adopt a binning feature engineering technique on only the *operator_ duration features*, shown in Figure 3.2. The binning technique partitions the nonnegative real numbers $\mathbb{R}$ into M bins and sorts each *operator_ duration* feature into the bin that includes its duration in microseconds. For example, we use $M = 2$ and bins representing the intervals $[0, 4000)$ and $[4000, \infty)$, where the number 4000 was determined experimentally using a search process. Therefore a TVM operator with *operator_ duration* $= 5000\mu s$ would be sorted into the second bin. From each bin, we extract 2 features:

- The sum of the operator durations in that bin.
- The number of the operators durations which were assigned to that bin.

We call these two features *duration* and *count* respectively. Figure 3.2 shows a visualization of these features for the large bin for GoogleNet, VGG16, and ResNet18.

The motivation for the binning approach is that the distribution of layer durations in inference is sensitive to the DNN architecture. For example, a convolution layer with many large filters, padding, and a low stride length will have a much longer duration than one with a few small filters, no padding, and a large stride length, which will in turn have a longer duration than most non-convolutional layers. The sequence of layers is specific to each architecture and generates a long-tail distribution

of layer durations which acts as a fingerprint to identify the architecture. We show an example of this distribution in Figure 3.3.

Training an Architecture Prediction Model: In Step 2 of the attack, the adversary has completed data collection and preprocessing. They will train an architecture prediction model to map from the side-channel data to a DNN architecture from the candidate set according to Equation 2.2. The adversary will validate that the model generalizes well with high architecture prediction accuracy before moving on to the online phase.

3.1.2.2 Online Attack

Step 3 of the attack follows a similar procedure to the training process, with the difference being that the victim model is now profiled and treated as a black box. The adversary collects a profile of the deployed victim DNN following the red arrows in Figure 2.2. Then the adversary will preprocess the data using aggregation or binning depending on the threat scenario. Finally, the adversary feeds the profile to the architecture prediction model that was trained in Step 2, resulting in the prediction of the victim model architecture.

CHAPTER 4

EXPERIMENTAL SETUP

In this Chapter, we present our experimental findings from the attack procedure described in Chapter 3.

4.1 Experimental Setup

This section presents an in-depth view of the internal setup of the TVM system on both the host machine and the target device to facilitate the experiment. All experiments were completed using the Hard Kernel Odroid N2+ Single Board Computer with the ARM Mali G-52 GPU, Quad-Core ARM Cortex A73 CPU, and Dual-Core Arm Cortex A53 CPU. The versions of relevant software packages includes the following: Python v3.7.3, TVM v0.11.dev0, ONNX v1.21.0, PyTorch v1.12.1, and OpenCL v2.0 (supported by the Odroid N2+).

AlexNet	VGG11_bn	DenseNet121	MnasNet0_75
ResNet18	VGG13	DenseNet169	MnasNet1_0
ResNet34	VGG13_bn	DenseNet201	MnasNet1_3
ResNet50	VGG16	DenseNet161	ShuffleNet_v2_x0_5
ResNet101	VGG16_bn	GoogleNet	ShuffleNet_v2_x1_0
ResNet152	VGG19	MobileNet_v2	ShuffleNet_v2_x1_5
Wide_ResNet50	VGG19_bn	MobileNet_v3_large	ShuffleNet_v2_x2_0
Wide_ResNet101	SqueezeNet1_0	MobileNet_v3_small	
VGG11	SqueezeNet1_1	MnasNet0_5	

Table 4.1: The 34 PyTorch vision architectures forming the candidate set in the proposed architecture extraction attack. These are all of the architectures present in Torchvision v0.10.0 [32] except for the ResNext architectures due to lack of support for group convolution TVM operations on the Arm Mali G52 GPU.

In our study, we used the Open Neural Network Exchange (ONNX) format to prepare 34 deep learning image classification models converted from the PyTorch model zoo [32], which served as our candidate set shown in Table 4.1. For each TVM optimization level (0, 1, 2, and 3) we profile each model from the candidate set 20 times, resulting in a dataset of 680 profiles per optimization level.

To preprocess the data, in Scenario 1, we use recursive feature elimination (RFE) to rank the aggregated TVM operator features to narrow the dataset and make the learning task easier. In Scenario 2, we set $M = 2$ bins with the intervals $[0, 4000)$ and $[4000, \infty)$. We observe that the choice of the binning threshold is not critical to achieve high architecture prediction accuracy due to low intra-class variance and high inter-class distance in the distribution of layer durations from models in our candidate set. The choice of the binning threshold becomes more important when distinguishing between DNN architectures with highly correlated layer duration distributions such as the one shown for ResNet18 in Figure 3.3. Therefore the intervals were sufficient for experimentation on this candidate set.

To train the architecture prediction model in Step 2 of our attack, we test seven different supervised learning classifiers from Scikit-learn to predict the victim architecture. These classifiers include Gaussian Naive Bayes (nb), Random Forest (rf), Logistic Regression (lr), Multi-layer Perceptron (nn), Nearest Centroid (nc), K Nearest Neighbors (knn), and AdaBoost (ab). We train these classifiers using the aggregated and profiled data obtained from Step 1 of the attack, allowing them to learn patterns and make predictions on the victim architecture.

4.2 Results

In this section, we test our proposed architecture extraction attack using Scenario 1 and Scenario 2. The organization of this section is as follows. First, we validate that the attack achieves sufficient accuracy in the easiest case holding the TVM

optimization level constant, and we investigate the minimum effort required by the adversary in data collection and architecture prediction model training to realize the best performance. Second, we test the generalization of models trained on one or two optimization levels and tested on all optimization levels separately. Third, we train architecture prediction models on all optimization levels and assess the accuracy on each level. Finally, as in the first section, we find the minimum attack requirements for the adversary when considering all optimization levels.

4.2.1 Train with Optimization Level 0, Test on Optimization Level 0

We first show that the proposed architecture extraction attack is feasible in the easiest case where only TVM optimization level 0 (no optimization) is applied. In subsequent sections, we consider all optimization levels.

The dataset of profiles at optimization level 0 contains 19 unique TVM operators across the 34 architectures in the candidate set. The data preprocessing for Scenario 1 generates 59 features from these operators, and 4 features for Scenario 2.

Scenario 1: In Figure 4.1 we show that the architecture extraction attack can achieve 100% accuracy while reducing the computational complexity of the attack. The left subfigure shows that N, the number of profiles collected per architecture in Step 1 of the attack, may be reduced to 8 while maintaining this accuracy. This reduces the size of the dataset and reduces the adversary's data collection effort. The adversary could collect $8 \cdot 34 = 272$ profiles instead of 680 with no accuracy loss.

The right subfigure ranks the 59 features by RFE using a random forest model, and selects the top B features to train the architecture prediction model. We find that 4 features are sufficient to determine the DNN architecture. We observe that the total duration feature, which was ranked first by RFE, enables over 90% architecture prediction accuracy alone. This result narrows the dataset reduce the complexity of training the architecture prediction model. We find that in both experiments, the

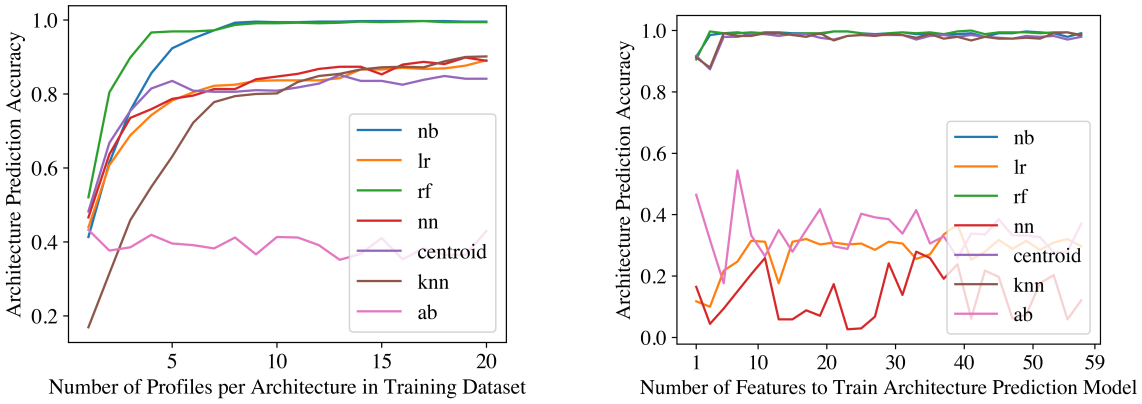


Figure 4.1: Architecture prediction accuracy by number of training profiles per architecture (left) and number of features (selected by RFE using a Random Forest model) used to train the architecture prediction model (right), for Scenario 1 considering only TVM optimization level 0.

Gaussian Naive Bayes and Random Forest architecture prediction models perform best.

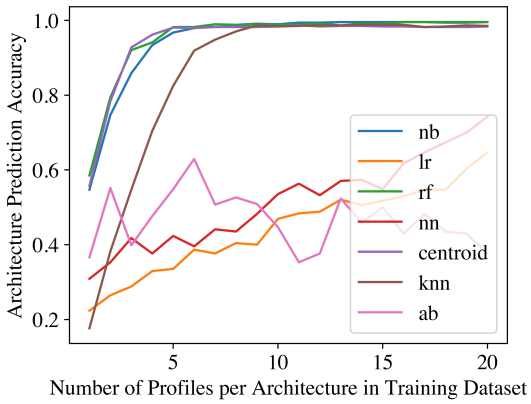


Figure 4.2: Architecture prediction accuracy by number of training profiles per architecture for Scenario 2 considering only TVM optimization level 0.

Scenario 2: We run a similar experiment for Scenario 2. Considering the dataset is already constrained with only 4 features, it is decided not to observe the relationship between feature size and the accuracy in this particular scenario. Figure 4.2 shows that only 5 profiles per architecture are necessary to achieve 100% architecture prediction accuracy for the Gaussian Naive Bayes, Random Forest, and Nearest Centroid

Table 4.2: Architecture prediction accuracy of a Random Forest classifier trained on one or two optimization levels and tested on all optimization levels using the binning technique in Scenario 2. Reported accuracy is on the Victim Set.

Train Set	**Victim Set**			
	opt_level0	opt_level1	opt_level2	opt_level3
opt_level0	99.4%	24.1%	23.5%	20.5%
opt_level1	30.4%	98.9%	79.4%	71.6%
opt_level2	26.4%	71.6%	99.2%	74.7%
opt_level3	26.5%	67.8%	81.6%	97.3%
opt_level0 & opt_level1	99.4%	99.8%	79.5%	77.0%
opt_level0 & opt_level2	99.4%	80.0%	99.8%	77.9%
opt_level0 & opt_level3	99.4%	83.8%	85.4%	97.6%
opt_level1 & opt_level2	32.3%	99.5%	99.8%	76.4%
opt_level1 & opt_level3	32.6%	99.7%	83.6%	97.8%
opt_level2 & opt_level3	26.4%	83.8%	99.7%	99.7%

models, and 8 for the K Nearest Neighbors model. This means the adversary would only need to collect $5 \cdot 34 = 170$ profiles. Interestingly, as compared to training with 59 features in Scenario 1, the 4 features in Scenario 2 admit a smaller dataset to achieve the same accuracy, meaning that the binning features are particularly representative of the DNN architecture.

4.2.2 Generalization of Optimization Levels

Next, we test if an architecture prediction model trained on one or two TVM optimization levels can generalize to other optimization levels. In Scenario 1, the total number of TVM operator grows from 19 to 48 when adding optimization levels 1-3. The TVM operator names at optimization levels 1-3 have almost no overlap with the names at optimization level 0 due to the transformed feature names resulting from the fusion or change of layers. For example, when moving from optimization level 0 to 1, ResNet18's convolution operator, with name `convolution`, becomes fused

with relu to become `convolution_relu`. Therefore it is not feasible to conduct this experiment using Scenario 1 and we perform this experiment using only Scenario 2.

We present these results in Table 4.2 where we consider a Random Forest architecture prediction model, the best performing model in the previous experiments. The first four rows of the table test the accuracy of the classifier trained with one optimization level and tested on each level separately. Consistent with the results from the previous section, we find that the accuracy is high when the training and testing optimization levels are the same. Notably, the prediction accuracy significantly decreased when the classifier was trained with no optimization (level 0) and tested on any level of optimization (level 1, 2, or 3) or when it was trained with any level of optimization and tested on profiles with no optimization. This poor accuracy is attributed to the application of graph fusion optimization, which occurs when the optimization is raised from 0 to 1. This process involves fusing a larger layer with smaller layers, thereby reducing the number of smaller layers and causing classifier to predict inaccurately.

This result prompted us to conduct an additional experiment such as training the classifier with 2 different optimization levels. The last six rows of the table show that this approach may yield much better prediction accuracy across all optimization levels, especially when the training set included optimization level 0 and one optimized level. We observe poor accuracy when level 0 was not included in the training set and the classifier was tested on level 0.

Our results indicate a high correlation between optimization level and prediction accuracy, and we suggest that adversaries include a wide range of optimized version of models in their training set to achieve accurate predictions when trying to attack optimized model.

Table 4.3: Architecture prediction accuracy of architecture prediction models trained on all optimization levels and tested on each optimization level. There were 146 features for Scenario 1 and 4 for Scenario 2.

Classifier & Scenario		**Victim Set**			
		opt_level0	opt_level1	opt_level2	opt_level3
Naive Bayes	Scenario 1	99.5%	98.9%	99.7%	100%
	Scenario 2	98.5%	97.9%	98.5%	95.1%
Random Forest	Scenario 1	99.7%	99.8%	100%	100%
	Scenario 2	99.7%	99.4%	98.6%	97.2%
K Nearest Neighbors	Scenario 1	98.3%	98.3%	97.9%	97.5%
	Scenario 2	97.6%	94.5%	95.0%	92.6%
Nearest Centroid	Scenario 1	98.2%	96.0%	97.0%	97.5%
	Scenario 2	95.1%	88.9%	90.0%	89.5%
MLP Classifier	Scenario 1	50.5%	50.1%	34.1%	84.4%
	Scenario 2	5.8%	10.7%	8.9%	5.8%
Logistic Regression	Scenario 1	31.1%	27.6%	20.3%	42.3%
	Scenario 2	5.8%	12.7%	10.5%	8.8%
Adaboost	Scenario 1	20.5%	8.8%	2.9%	20.5%
	Scenario 2	17.6%	5.9%	5.8%	5.8%

4.2.3 Train with all Optimization Levels, Test on Each Level

The cross-optimization generalization results from the previous section indicate that architecture prediction is highest on all optimization levels when multiple levels are included in the training data. In this section, we consider architecture prediction models trained on a dataset of 20 profiles per architecture per optimization level, resulting in a dataset of size $20 \cdot 34 \cdot 4 = 2720$ profiles. We train architecture prediction models on this dataset using data preprocessing from Scenario 1 and Scenario 2. We then test the accuracy of these models on all optimization levels.

These results are presented in Table 4.3. Our architecture extraction attack achieves at least 99.7% accuracy for Scenario 1 and at least 97.2% accuracy for Scenario 2 using a Random Forest model. As in previous experiments, the Gaussian Naive Bayes, K Nearest Neighbors, and Nearest Centroid models had similarly high performance. We show that, despite the challenge of obfuscated TVM operator names in Scenario 2, the binning feature engineering achieves nearly the same accuracy as Scenario 1 when the TVM operator names are visible, illustrating the strength of this approach. There is only a slight ($<$5.6%) decrease in accuracy as the optimization level increases in Scenario 2 for the highest performing models.

4.2.4 Further Attack Optimization

The previous results show that an adversary will have highest success including all TVM optimization levels in their training data. In this case, we may test the extent to which the training dataset length and width may be decreased, as we showed before when considering only one optimization level.

Scenario 1: In Figure 4.3, we show that the Gaussian Naive Bayes, Random Forest, Nearest Centroid, and K Nearest Neighbors models all converge to their architecture prediction accuracy from Table 4.3 with 8 profiles per architecture per optimization level. This means that the data collection requires only $8 \cdot 34 \cdot 4 = 1088$

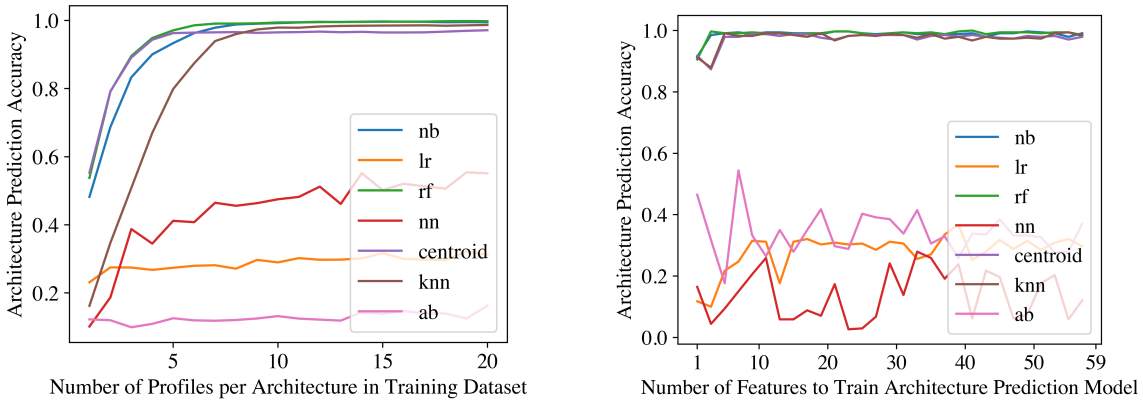


Figure 4.3: Architecture prediction accuracy by number of training profiles per architecture per optimization level (left) and number of features (selected by RFE using a Random Forest model) used to train the architecture prediction model (right), for Scenario 1, training and testing on all optimization levels.

profiles instead of the 2720 collected previously, a 60% decrease. Also, we show that 4 out of the 146 features are sufficient to maintain the architecture prediction accuracy, a 97.2% decrease in the dataset width. These most important features were the total duration of the DNN inference, the percentage of the inference time spent on the TVM operator `nnmaxpool2d`, the percentage of the inference time spent on the TVM operator `nndenseadd`, and the total duration of all invocations to the `nnconv2dnnbiasaddnnrelu` operator. Significantly, only a subset of these features are present in profiles at different optimization levels. For example, optimization level 0 does not include the `nndenseadd` or `nnconv2dnnbiasaddnnrelu` operators.

Scenario 2: Similarly, we show that in Scenario 2, the architecture prediction accuracy converges between 5 and 13 profiles per architecture per optimization level. An adversary would select the highest performing model, Random Forest, converging with 6 profiles and a dataset size of 816. Compared to Figure 4.2, these results show that the same reduction in attack complexity may be achieved despite considering all optimization levels instead of one.

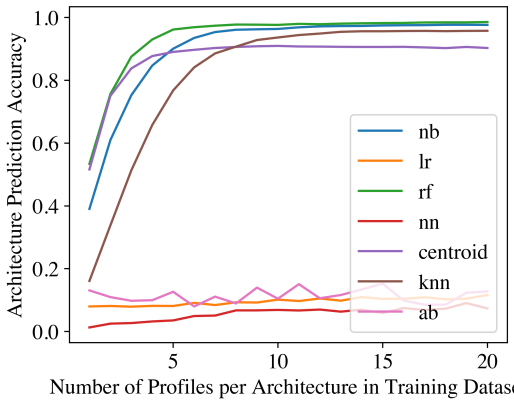


Figure 4.4: Architecture prediction accuracy by number of training profiles per architecture per optimization level for Scenario 2, training and testing on all optimization levels.

CHAPTER 5

LAYER-BY-LAYER RECONSTRUCTION OF MODEL ARCHITECTURE

In this Chapter we consider attack scenario where the candidate DNN set is not known *a priori*. Hence, the hyperparameters must be extracted layer-by-layer.

5.1 Motivation

In the earlier chapter, we successfully demonstrated the efficacy of predicting DNN architecture from candidate set. However, the challenge of extracting the hyperparameters of a victim DNN with permuted configuration or an unknown DNN model remains. In this section, We discuss DNN model extraction attack focusing on hyperparameters extraction. Hyperparameters of a DNN are parameters configured by AI developers. When constructing DNN models, developers specify layer parameters such as number of channels, kernel size, padding, stride, and more, aiming to achieve various objectives such as classification accuracy, performance enhancement or efficient computing. If we have information about the hyperparameters, we can estimate the duration, representing the amount of computation (such as FLOPs, MAC operations) required. The key question of this section is to know if it is possible to determine this relationship in reverse. In other words, We want to know if we can extract hyperparameters if we possess sequential kernel data obtained from GPU profiles, which incorporate duration information in it.

When employing traditional methods, defining this relationship is challenging, given that a singular duration is influenced by diverse set of hyperparameters, including the number of channels, feature map size, and kernel size. This challenge

prompts us to adopt machine learning methods to overcome the limitations of traditional approaches. Utilizing the associations between duration and hyperparameters in known models, we train our machine learning model that predicts hyperparameters. Our research is focused on predicting the hyperparameters of the convolution layer in CNN models, which involves input image's height, input image's width, the number of the input channel, output image's height, output image's width, the number of the output channels and the kernel size. The convolution layer stands out as the most challenging, given that it undergoes primary shape changes and is affected by the largest number of factors influencing its duration. Successfully determining the hyperparameters of the convolution layer essentially acts as a key to completing the information for the remaining layers. This is due to the fact that the remaining layers are mostly influenced by the dimension of the image, a factor that can be automatically determined once the convolution layers are accurately compromised.

5.2 Methodology

5.2.1 Pre-processing

In this hypothetical scenario, the adversary possesses sequential profile data obtained from running a known DNN model on the target mobile GPU as a train dataset. The train dataset comprises details about layer types, duration of the layer, and known hyperparameters such as input shape, output shape, and kernel size.

Initially, the adversary identifies a convolution sub-layer pattern within the sequential data that appears to be linked to the duration of the targeted convolution layer. Subsequently, the adversary extracts durations and hyperparameters from the layers in designated pattern, creating a single row where these durations function as features and hyperparameters function as labels. To train the feature augmentation sub-model, as detailed in the modeling method section, we categorize hyperparameters based on their values and add them as an additional column, represented as

bin numbers. This algorithm iterates through all 34 DNN models from PyTorch zoo, making a single file that contains 1171 rows. In our research, we define a layer pattern as `convolution`, `bias_add`, `relu`. In this pattern, the input and output shapes of `bias_add` and `relu`, as well as the output shape of `convolution`, are identical. It is noteworthy that predicting hyperparameters associated with `relu` and `bias_add` from their durations is easier, as it only involves considerations of the number of channels and the image size. This design allows the durations of `bias_add` and `relu` to influence the output shape of the convolution layer, simplifying the prediction task by focusing relatively solely on predicting kernel size and input shape. In contrast, if the model were exclusively trained with convolution layers, it would need to consider input shape, kernel size, and output size simultaneously, making the prediction process more challenging.

By establishing the pattern first, the prediction process becomes more accurate. However, for the adversary to successfully extract hyperparameters, they must consider the pattern's impact on duration and selectively exclude profile layer type that are unlikely to be relevant to its hyperparameters during the prediction process.

5.2.2 Training Regression Model and Feature Augmenting Classification Sub-models

The adversary's regression model aims to predict the hyperparameters of the victim DNN model solely based on the duration information from the profile data. However, it is observed that training the regression model using only the durations of `convolution`, `relu`, and `bias_add` leads to poor prediction performance. To enhance the model's capability, additional features are introduced by employing sub-models that serves more hints to regression model by classifying hyperparameter categories, previously categorized as bins in the preprocessing phase. The first sub-model is designed to take only duration information as input, producing `output_w_class` and

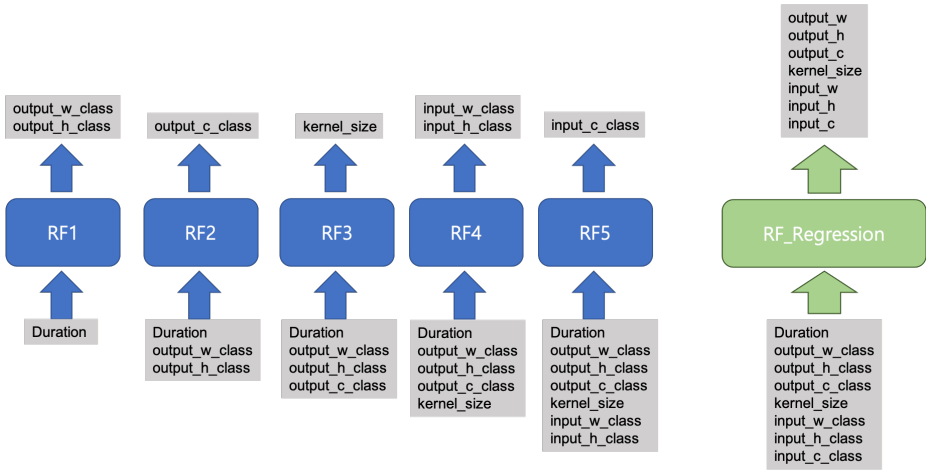


Figure 5.1: Training phase, each feature augmentation sub-models and the regression model are trained with labels

`output_h_class` as outputs. The subsequent sub-model, taking both the duration and the output of the first sub-model as input, generates `output_c_class` as its output. This sequential propagation continues through five sub-models until predictions for all the desired result categories are achieved. In the training phase, all these sub-models are trained with labels without propagation. The main regression model is intended to take all the predictions generated by the sub-models, along with the durations, as inputs to produce the final regression result. Similar to the training approach for sub-models, the regression model is trained using labels instead of the predictions obtained from the propagated predictions of the sub-models. The test accuracy of sub-models can be influenced by adjusting the number of hyperparameter bins. As the number of hyperparameters increases, accuracy tends to decrease, while precision increases. Conversely, reducing the number of hyperparameters results in increased accuracy but lower precision. The choice of the number of bins for categorizing hyperparameters is carefully set to maintain accuracy above 90%, while also maximizing precision. In our experiment, Random Forest Models are used both in classification and the regression task with number of estimators set to 100 for optimal performance.

5.2.3 Inference Phase with Result Propagation in the Connected Model

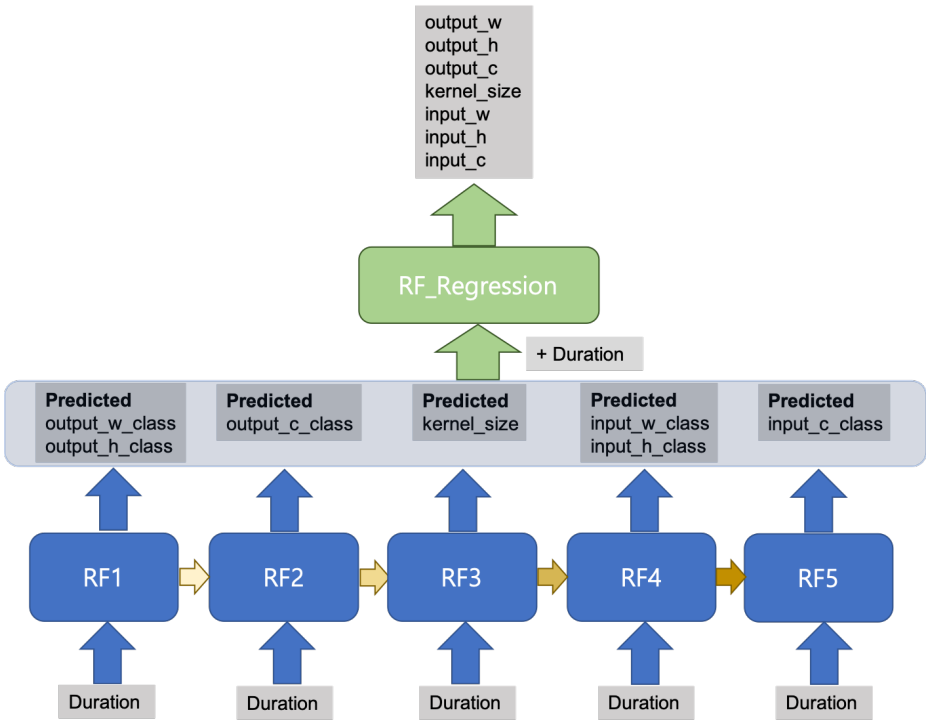


Figure 5.2: Complete connected model in inference phase with prediction result propagation.

Following the training of the feature-augmenting sub-models and the regression model with their respective labels, these models are interconnected. The complete connected model collectively takes only duration information as input, propagate predictions, and generate the final regression result. The order of prediction in feature augmenting sub-models is strategically designed to predict `relu` and `bias_add` side hyperparameters first. As detailed in the pre-processing section, the input and output shapes of `relu` and `bias_add`, as well as the output shape of `convolution`, are identical. Predicting hyperparameters for `relu` and `bias_add` is relatively straightforward, given that the size of the feature map and the number of channels are the key factors determining these hyperparameters. Once the output side of the convolution layer is predicted, it is sequentially propagated to predict the kernel size and the input side of the convolution layer. When the prediction of sub-models is finished, the regression

model takes both the duration and the classification predictions generated by the sub-models as input, producing the final regression output.

5.2.4 Post-processing

With the generated regression output, post-processing is necessary for three reasons described next. The first reason is to minimize regression errors, thereby improving the overall precision of the prediction. The second reason is to ensure a consistent layer shape flow. Failure to achieve this consistency could lead to issues when compiling the reconstructed DNN. The third reason is to fill in shape hyperparameters for the remaining layers, mostly `max_pooling`. In this context, We have made 4 post-processing method for the prediction result.

1. We convert the predicted hyperparameter regression output to the closest label from the known DNN train dataset.

2. If the convolution pattern is consecutive, we convert the input shape of the subsequent layer pattern to the output shape of the current layer pattern (e.g., `current_conv` - `current_bias_add` - `current_relu` - `subsequent_conv` - `subsequent_bias_add` - `subsequent_relu`).

3. If the convolution pattern is not consecutive and there is another type of layer in between, we fill the shape hyperparameters of the target layer with predictions from the neighboring layer patterns (e.g., `prev_conv` - `prev_bias_add` - `prev_relu` - `max_pooling` - `next_conv` - `next_bias_add` - `next_relu`).

4. Taking into account that the image size (excluding channel) follow a decreasing function in whole layers, we iterate through each predicted layer. At each step, we refer to the image sizes of the next five layers from the current layer and adjust the current layer's image size to the maximum value among these five values. During the iteration, a slightly different approach is taken for considering

the input and output image sizes. the input image size is determined by referring to the input image sizes of the next five layers. Similarly, the output image size is determined by referring to the output image sizes of the next five layers. Additionally for the output image size, we ensure that the output image size does not exceed the input image size in the same layer.

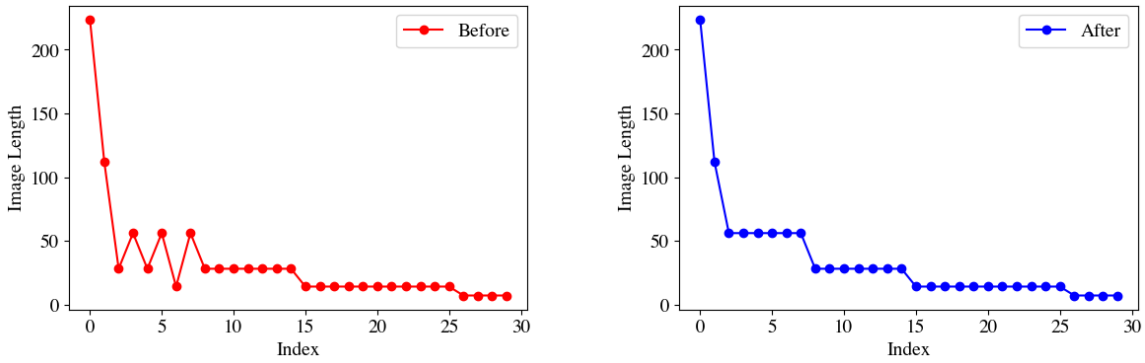


Figure 5.3: Image Length of the prediction of ResNet34 before (Left) and after (Right) 4th method applied.

The first method aims to align with common DNN design conventions, thereby reducing prediction errors. We've observed that deep learning hyperparameters often fall within a limited value pool favored by DNN developers. Additionally, this rule has the effect of rounding the predicted hyperparameter. The second method is implemented to address the second reason. Ensuring shape consistency in consecutive pattern layers is crucial to prevent errors during DNN compilation. This method also contributes to reducing prediction errors. The third method is designed to fill in missing hyperparameters for layers not present in the pattern, completing the overall reconstruction process. The final method prevents the feature map size fluctuation by smoothing out the predicted image size to the maximum one. It is important to note that these methods should be applied carefully and with flexibility while examining the result data, as changing the value of predictions may potentially causing correct predictions to become incorrect.

5.3 Result

We evaluate the efficacy of hyperparameter extraction through various dimensions. Initially, we examine the test accuracy of the models we trained, including each sub-model and the complete connected model. Next, using the complete connected model, we attempt to extract hyperparameters from representative models in the zoo and assess the regression error. Lastly, we select VGG19 and compare the performance of the reconstructed model with the original one.

5.3.1 Model Performance

Given that our extraction model comprises numerous small machine learning models, we evaluate the performance of each sub-model using diverse metrics. In the context of classification for feature augmenting sub-models, accuracy serves as the metric of choice. For the regression model, responsible for producing the final regression result, we employ normalized average error to describe the errors associated with each output on varying scales fixed to 100. The normalized average error is described as follows.

$$NormalizedError_i = \frac{|\hat{y}_i - y_i|}{\hat{y}_{\max} - \hat{y}_{\min}} \times 100 \tag{5.1}$$

$$NormalizedAverageError = \frac{\sum_{i=1}^{n} NormalizedError_i}{\mathrm{n}} \tag{5.2}$$

After evaluating each sub-model and the regression model, we assess the performance of the complete connected model, which involves the result propagation process. In the preprocessing phase, a total of 1171 pattern rows were extracted. Each pattern row consists of durations for three layers (`convolution`, `bias_add`, `relu`), resulting in the extraction of 3513 layers from 34 models in the zoo. In each distinct sub-model, the test size is adjusted empirically to achieve the highest accuracy. For sub-models responsible for feature augmentation, all test accuracies are observed to be above 90%. In the regression model, utilizing a total of 117 samples derived from a test size of 10%, the normalized average error for all predicted hyperparameters is

under 3 with values normalized between the range of 0 and 100. During testing with the complete connected model involving result propagation, a slight increase in error is observed compared to testing only the regression model. This increase in error is anticipated to stem from misclassifications during the propagation process, given that the accuracy of the sub-models was not 100%. It is observed that employing feature augmentation with prediction result propagation demonstrates a $\frac{1}{20}$ reduction in Mean Squared Error (MSE) for the regression model compared to not using these techniques.

Table 5.1: The accuracy of classification sub-models for the purpose of feature augmentation, along with the testing configurations applied to these sub-models.

	RF1	RF2	RF3	RF4	RF5
Input	Duration	Duration Output_w_class Output_h_class	Duration Output_w_class Output_h_class Output_c_class	Duration Output_w_class Output_h_class Output_c_class Kernel_Size	Duration Output_w_class Output_h_class Output_c_class Kernel_size Input_w_class Input_h_class
Output	Output_w_class Output_h_class	Output_c_class	Kernel_Size	Input_w_class Input_h_class	Input_c_class
Test Acc	Width 92.68% Height 91.61%	92.54%	91.49%	Width 99.15% Height 99.15%	90.09%
Test Size	0.4	0.4	0.2	0.2	0.2
Number of Bins	8	4	(5)	8	4

5.3.2 Predicting Representative DNN Victim Models

We proceed to evaluate how well our model performs when actively attacking victim DNN. The previous results focused on the model's performance, while this evaluation incorporates the effectiveness of post-processing. To conduct this assessment, we select 11 representative victim DNN models from the PyTorch zoo. Next, we extract the `convolution`, `bias_add`, `relu` pattern from the victim model and predict with our trained complete connected model. Following the prediction phase, we carry out post-processing to finalize the extraction process. The metric for evaluating the results is the normalized average error, which is consistent with the metric used

Table 5.2: The normalized average error calculated for each specific output of the regression model, as well as for the complete connected inference model which involves the prediction result propagation

		RF_Regressor (test size = 0.1)	Inference
Input		Duration Output_w_class Output_h_class Output_c_class Kernel_size Input_w_class Input_h_class Input_c_class	Duration Propagation of the prediction
Output		Output_w Output_h Output_c Kernel_size Input_w Input_h Input_c	Output_w Output_h Output_c Kernel_size Input_w Input_h Input_c
Normalized Average Feature Error (scale = 100)	Output_w	0.71	1.54
	Output_h	0.71	1.54
	Output_c	2.30	2.80
	Kernel_Size	1.28	1.14
	Input_w	1.53	1.38
	Input_h	1.53	1.38
	Input_c	1.39	2.59

to evaluate the regression model. When only considering sequential models, The observation shows that three DNNs from the VGG family are perfectly extracted, while VGG19 shows the most inaccurate predictions (3.23), as indicated by the average error across multiple outputs. The most inaccurate victim including complex chained model (lower part of the table partitioned by line) was ShuffleNet v2 (3.82).

5.3.3 Comparing DNN Performance of Reconstructed vs Original

We explore how close the extracted victim model are to the original model in terms of performance. We exclusively consider sequential models, as it is simpler to

Table 5.3: The normalized average error evaluated under attack conditions by the complete connected inference model when applied to representative victim DNNs selected from PyTorch zoo

Model	Normalized Average Error (Scale = 100)						
	input c	input h	input w	kernel size	output c	output h	output w
VGG11	0.00	0.00	0.00	0.00	0.00	0.00	0.00
VGG13	0.00	0.00	0.00	0.00	0.00	0.00	0.00
VGG16	0.00	0.00	0.00	0.00	0.00	0.00	0.00
VGG19	2.34	4.29	4.29	0.0	2.38	4.68	4.68
ResNet18	0.00	1.78	1.78	2.04	0.00	0.00	0.00
ResNet34	5.00	0.83	0.83	2.85	5.00	0.00	0.00
Densenet121	6.15	0.83	0.83	0.48	1.74	1.66	1.66
Densenet169	2.07	0.03	0.03	0.00	0.30	0.07	0.07
ShuffleNet v2	2.18	5.20	5.20	3.70	0.80	4.86	4.86
SqueezeNet	6.12	1.52	1.52	2.19	6.06	2.15	2.15
GoogleNet	7.32	0.38	0.38	2.00	4.53	0.76	0.76

probe impact of the hyperparameters difference, while complex chained models may introduce performance noise due to their architectural characteristic. We make the assumption that a perfectly predicted model, such as VGG11, behaves exactly the same as the original model, while a poorly predicted model, such as VGG19, exhibits the most significant performance differences. In this context, we choose VGG19 for comparison in this section, anticipating that other models with better predictions will show smaller performance differences. We observed that majority of differences occurs from front part of the architecture as in table 5.4. For the layers where correct predictions were made, the remaining layers were snipped. We train and test with CIFAR10 dataset with batch size = 64 and run for 20 epochs. Even with a different image size from ImageNet (224x224), it is still possible to use CIFAR10 (32x32) as a dataset. This is feasible because the model itself manages the change in image size, rather than configuring the image size layer by layer. We configure Stochastic Gradient Descent (SGD) as an optimizer with a learning rate of 0.1 and momentum of 0.9. To ensure robustness of the result, we conduct the test 10 times and calculate the average test accuracy and test loss for both the original and reconstructed DNN. The

average test accuracy shows 79.15% in original VGG19 and 76.8% in reconstructed VGG19. Based on this testing, we conclude that the most poorly reconstructed model exhibits less than a 3% performance difference, and we can anticipate obtaining closer performance traces from other better-predicted models.

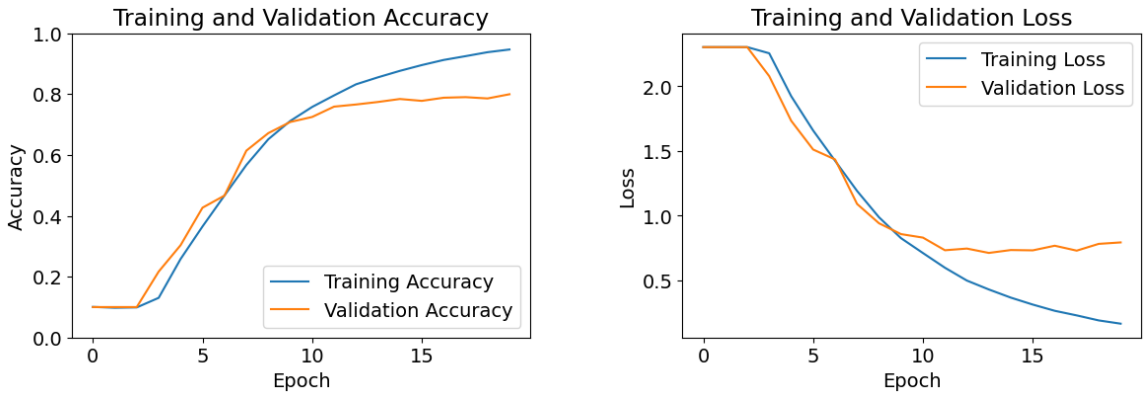


Figure 5.4: Loss(left), Accuracy(right) Trace of the Original DNN.

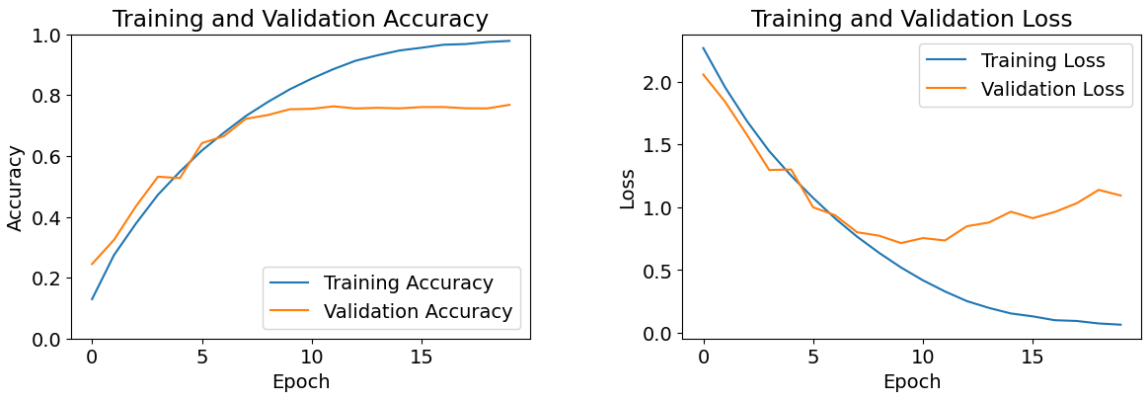


Figure 5.5: Loss(left), Accuracy(right) Trace of the Reconstructed Victim DNN.

Table 5.4: The major difference between the original and reconstructed DNN is observed in the front part of VGG19. For the layers where correct predictions were made, the remaining layers were truncated.

Layer Pattern		Input_c	Input_h	Input_w	Kernel_Size	Output_c	Output_h	Output_w
Conv Bias ReLU	Original	3	224	224	3	64	224	224
	Reconstructed	3	224	224	3	64	224	224
Conv Bias ReLU	Original	64	224	224	3	64	224	224
	Reconstructed	64	224	224	3	256	112	112
Max Pooling	Filled with Preceding, Following Hyperparameters							
Conv Bias ReLU	Original	64	112	112	3	128	112	112
	Reconstructed	256	112	112	3	128	112	112
Conv Bias ReLU	Original	128	112	112	3	128	112	112
	Reconstructed	128	112	112	3	128	112	112
Max Pooling	Filled with Preceding, Following Hyperparameters							
Conv Bias ReLU	Original	128	56	56	3	256	56	56
	Reconstructed	128	112	112	3	128	112	112
Conv Bias ReLU	Original	256	56	56	3	256	56	56
	Reconstructed	128	112	112	3	256	56	56
Below Layers Identical (truncated)								

CHAPTER 6

CONCLUSION

We presented a method to extract a DNN architecture from TVM runtime profiles. We considered two cases. In the first case, the target victim model belongs to to a known architecture family, *i. e.* candidate set is known. This allows candidates to be profiled and matched. We then consider the second case, where the victim architecture is unseen before attack.

For the first case, we considered a number of scenarios ranging from the simplest case of no optimization to various levels of DNN optimization that fuses TVM operations. We also consider a more challenging scenario, where the TVM operator names are deliberately obfuscated. In response to this challenge, we propose a binning technique based on duration partitioning that is robust to changes in DNN computation graphs caused by the TVM optimization transformation.

For the second case, where the the victim architecture has not been seen, we propose a hyperparameter extraction process to overcome the limitation of the candidate matching.

Our experiments demonstrate that the adversary can predict the victim DNN architecture from a candidate set of architectures with 100% accuracy if the optimization level is known, at least 99.8% accuracy if the optimization level is unknown and the TVM operators are visible, and at least 97.2% accuracy if the optimization level is unknown and the TVM operators are obfuscated.

For unseen victim models, the hyperparameter extraction process achieves a normalized average error below 2 (on a scale of 100) for predicted hyperparameters, and

the reconstructed model exhibits a maximum classification accuracy difference of 3% compared to the original DNN.

Our novel step-by-step attack approach overcomes all existing model extraction attack defenses, demonstrating a critical vulnerability in current AI security. The success of our attack raises concerns about security, privacy, and financial risks for the entire AI industry. This work highlights the importance of research in developing resilient techniques to defend against such attacks.

BIBLIOGRAPHY

[1] Akhtar, Naveed, Mian, Ajmal, Kardan, Navid, and Shah, Mubarak. Threat of adversarial attacks on deep learning in computer vision: Survey II. *CoRR abs/2108.00401* (2021). https://arxiv.org/abs/2108.00401.

[2] Akhtar, Naveed, and Mian, Ajmal S. Threat of adversarial attacks on deep learning in computer vision: A survey. *IEEE Access 6* (2018), 14410–14430. https://doi.org/10.1109/ACCESS.2018.2807385.

[3] Batina, Lejla, Bhasin, Shivam, Jap, Dirmanto, and Picek, Stjepan. CSI NN: reverse engineering of neural network architectures through electromagnetic side channel. In *28th USENIX Security Symposium, USENIX* (2019), Nadia Heninger and Patrick Traynor, Eds., USENIX Association, pp. 515–532. https://www.usenix.org/conference/usenixsecurity19/presentation/batina.

[4] Carlini, Nicholas, Jagielski, Matthew, and Mironov, Ilya. Cryptanalytic extraction of neural network models. In *Advances in Cryptology - CRYPTO* (2020), Daniele Micciancio and Thomas Ristenpart, Eds., vol. 12172 of *Lecture Notes in Computer Science*, Springer, pp. 189–218. https://doi.org/10.1007/978-3-030-56877-1_7.

[5] Chakraborty, Anirban, Alam, Manaar, Dey, Vishal, Chattopadhyay, Anupam, and Mukhopadhyay, Debdeep. A survey on adversarial attacks and defences. *CAAI Trans. Intell. Technol. 6*, 1 (2021), 25–45. https://doi.org/10.1049/cit2.12028.

[6] Chen, Tianqi, Moreau, Thierry, Jiang, Ziheng, Zheng, Lianmin, Yan, Eddie Q., Shen, Haichen, Cowan, Meghan, Wang, Leyuan, Hu, Yuwei, Ceze, Luis, Guestrin, Carlos, and Krishnamurthy, Arvind. TVM: an automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018* (2018), Andrea C. Arpaci-Dusseau and Geoff Voelker, Eds., USENIX Association, pp. 578–594. https://www.usenix.org/conference/osdi18/presentation/chen.

[7] Chen, Zhi, Yu, Cody Hao, Morris, Trevor, Tuyls, Jorn, Lai, Yi-Hsiang, Roesch, Jared, Delaye, Elliott, Sharma, Vin, and Wang, Yida. Bring your own codegen to deep learning compiler. *CoRR abs/2105.03215* (2021). https://arxiv.org/abs/2105.03215.

[8] Chitty-Venkata, Krishna Teja, and Somani, Arun K. Neural architecture search survey: A hardware perspective. *ACM Comput. Surv. 55*, 4 (2023), 78:1–78:36. https://doi.org/10.1145/3524500.

[9] Chmielewski, Lukasz, and Weissbart, Leo. On reverse engineering neural network implementation on GPU. In *Applied Cryptography and Network Security Workshops - ACNS* (2021), vol. 12809, Springer, pp. 96–113. https://doi.org/10.1007/978-3-030-81645-2_7.

[10] da Silva, Jacson Rodrigues Correia, Berriel, Rodrigo Ferreira, Badue, Claudine, Souza, Alberto F. De, and Oliveira-Santos, Thiago. Copycat CNN: are random non-labeled data enough to steal knowledge from black-box models? *CoRR abs/2101.08717* (2021). https://arxiv.org/abs/2101.08717.

[11] Gong, Xueluan, Chen, Yanjiao, Yang, Wenbin, Mei, Guanghao, and Wang, Qian. Inversenet: Augmenting model extraction attacks with training data inversion. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI* (2021), Zhi-Hua Zhou, Ed., ijcai.org, pp. 2439–2447. https://doi.org/10.24963/ijcai.2021/336.

[12] Guyon, Isabelle, Weston, Jason, Barnhill, Stephen, and Vapnik, Vladimir. Gene selection for cancer classification using support vector machines. *Mach. Learn. 46*, 1-3 (2002), 389–422. https://doi.org/10.1023/A:1012487302797.

[13] Hong, Sanghyun, Davinroy, Michael, Kaya, Yigitcan, Locke, Stuart Nevans, Rackow, Ian, Kulda, Kevin, Dachman-Soled, Dana, and Dumitras, Tudor. Security analysis of deep neural networks operating in the presence of cache side-channel attacks. *CoRR abs/1810.03487* (2018). http://arxiv.org/abs/1810.03487.

[14] Hu, Xing, Liang, Ling, Li, Shuangchen, Deng, Lei, Zuo, Pengfei, Ji, Yu, Xie, Xinfeng, Ding, Yufei, Liu, Chang, Sherwood, Timothy, and Xie, Yuan. Deepsniffer: A DNN model extraction framework based on learning architectural hints. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne* (2020), James R. Larus, Luis Ceze, and Karin Strauss, Eds., ACM, pp. 385–399. https://doi.org/10.1145/3373376.3378460.

[15] Hua, Weizhe, Zhang, Zhiru, and Suh, G. Edward. Reverse engineering convolutional neural networks through side-channel information leaks. In *Proceedings of the 55th Annual Design Automation Conference, DAC* (2018), ACM, pp. 4:1–4:6. https://doi.org/10.1145/3195970.3196105.

[16] Huang, Po-Hsuan, Tu, Chia-Heng, Chung, Shen-Ming, Wu, Pei-Yuan, Tsai, Tung-Lin, Lin, Yi-An, Dai, Chun-Yi, and Liao, Tzu-Yi. Securetvm: A tvm-based compiler framework for selective privacy-preserving neural inference. *ACM Transactions on Design Automation of Electronic Systems* (2023). https://dl.acm.org/doi/10.1145/3579049.

[17] IEEE. The radical scope of tesla's data hoard. https://spectrum.ieee.org/tesla-autopilot-data-scope, Aug 2022.

[18] Ignatov, Andrey, Timofte, Radu, Kulik, Andrei, Yang, Seungsoo, Wang, Ke, Baum, Felix, Wu, Max, Xu, Lirong, and Gool, Luc Van. AI benchmark: All about deep learning on smartphones in 2019. In *2019 IEEE/CVF International Conference on Computer Vision Workshops, ICCV Workshops 2019, Seoul, Korea (South), October 27-28, 2019* (2019), IEEE, pp. 3617–3635. https://doi.org/10.1109/ICCVW.2019.00447.

[19] Jagielski, Matthew, Carlini, Nicholas, Berthelot, David, Kurakin, Alex, and Papernot, Nicolas. High accuracy and high fidelity extraction of neural networks. In *29th USENIX Security Symposium, USENIX* (2020), Srdjan Capkun and Franziska Roesner, Eds., USENIX Association, pp. 1345–1362. https://www.usenix.org/conference/usenixsecurity20/presentation/jagielski.

[20] Janai, Joel, Güney, Fatma, Behl, Aseem, and Geiger, Andreas. Computer vision for autonomous vehicles: Problems, datasets and state of the art. *Found. Trends Comput. Graph. Vis. 12*, 1-3 (2020), 1–308. https://doi.org/10.1561/0600000079.

[21] Jha, Nandan Kumar, Mittal, Sparsh, Kumar, Binod, and Mattela, Govardhan. Deeppeep: Exploiting design ramifications to decipher the architecture of compact dnns. *ACM 17*, 1 (2020), 5:1–5:25. https://doi.org/10.1145/3414552.

[22] Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems* (2012), Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, Eds., pp. 1106–1114. https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[23] Liu, Sihang, Wei, Yizhou, Chi, Jianfeng, Shezan, Faysal Hossain, and Tian, Yuan. Side channel attacks in computation offloading systems with GPU virtualization. In *2019 IEEE Security and Privacy Workshops, SP* (2019), IEEE, pp. 156–161. https://doi.org/10.1109/SPW.2019.00037.

[24] Liu, Yuntao, and Srivastava, Ankur. GANRED: gan-based reverse engineering of dnns via cache side-channel. In *CCSW'20, Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop* (2020), Yinqian Zhang and Radu Sion, Eds., ACM, pp. 41–52. https://doi.org/10.1145/3411495.3421356.

[25] Liu, Yuntao, Zuzak, Michael, Xing, Daniel, McDaniel, Isaac, Mittu, Priya, Ozbay, Olsan, Akib, Abir, and Srivastava, Ankur. A survey on side-channel-based reverse engineering attacks on deep neural networks. In *4th IEEE International Conference on Artificial Intelligence Circuits and Systems, AICAS* (2022), IEEE, pp. 312–315. https://doi.org/10.1109/AICAS54282.2022.9869995.

[26] Liu, Zhibo, Yuan, Yuanyuan, Wang, Shuai, Xie, Xiaofei, and Ma, Lei. Decompiling x86 deep neural network executables. *CoRR abs/2210.01075* (2022). https://doi.org/10.48550/arXiv.2210.01075.

[27] Maia, Henrique Teles, Xiao, Chang, Li, Dingzeyu, Grinspun, Eitan, and Zheng, Changxi. Can one hear the shape of a neural network?: Snooping the GPU via magnetic side channel. In *31st USENIX Security Symposium, USENIX* (2022), Kevin R. B. Butler and Kurt Thomas, Eds., USENIX Association, pp. 4383–4400. https://www.usenix.org/conference/usenixsecurity22/presentation/maia.

[28] Mirzadeh, Seyed-Iman, Farajtabar, Mehrdad, Li, Ang, and Ghasemzadeh, Hassan. Improved knowledge distillation via teacher assistant: Bridging the gap between student and teacher. *CoRR abs/1902.03393* (2019). http://arxiv.org/abs/1902.03393.

[29] Naghibijouybari, Hoda, Neupane, Ajaya, Qian, Zhiyun, and Abu-Ghazaleh, Nael B. Rendered insecure: GPU side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS* (2018), David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, Eds., ACM, pp. 2139–2153. https://doi.org/10.1145/3243734.3243831.

[30] Papernot, Nicolas, McDaniel, Patrick D., Goodfellow, Ian J., Jha, Somesh, Celik, Z. Berkay, and Swami, Ananthram. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017), Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi, Eds., ACM, pp. 506–519. https://doi.org/10.1145/3052973.3053009.

[31] Patwari, Kartik, Hafiz, Syed Mahbub, Wang, Han, Homayoun, Houman, Shafiq, Zubair, and Chuah, Chen-Nee. DNN model architecture fingerprinting attack on CPU-GPU edge devices. In *7th IEEE European Symposium on Security and Privacy, EuroS&P* (2022), IEEE, pp. 337–355. https://doi.org/10.1109/EuroSP53844.2022.00029.

[32] PyTorch. Torchvision v0.10.0 documentation. https://pytorch.org/vision/0.10/, Jun 2021.

[33] Sculley, D., Holt, Gary, Golovin, Daniel, Davydov, Eugene, Phillips, Todd, Ebner, Dietmar, Chaudhary, Vinay, Young, Michael, Crespo, Jean-François, and Dennison, Dan. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems* (2015), Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, Eds., pp. 2503–2511. https://proceedings.neurips.cc/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html.

[34] Sharir, Or, Peleg, Barak, and Shoham, Yoav. The cost of training NLP models: A concise overview. *CoRR abs/2004.08900* (2020). https://arxiv.org/abs/2004.08900.

[35] Sun, Zhichuang, Sun, Ruimin, Lu, Long, and Mislove, Alan. Mind your weight (s): A large-scale study on insufficient machine learning model protection in mobile apps. In *30th USENIX Security Symposium (USENIX Security 21)* (2021), pp. 1955–1972.

[36] Sutskever, Ilya, Vinyals, Oriol, and Le, Quoc V. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems* (2014), Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, Eds., pp. 3104–3112. https://proceedings.neurips.cc/paper/2014/hash/a14ac55a4f27472c5d894ec1c3c743d2-Abstract.html.

[37] Tramèr, Florian, Zhang, Fan, Juels, Ari, Reiter, Michael K., and Ristenpart, Thomas. Stealing machine learning models via prediction apis. In *25th USENIX Security Symposium, USENIX* (2016), Thorsten Holz and Stefan Savage, Eds., USENIX Association, pp. 601–618. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/tramer.

[38] Vaswani, Ashish, Shazeer, Noam, Parmar, Niki, Uszkoreit, Jakob, Jones, Llion, Gomez, Aidan N., Kaiser, Lukasz, and Polosukhin, Illia. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems* (2017), Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, Eds., pp. 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[39] Wei, Junyi, Zhang, Yicheng, Zhou, Zhe, Li, Zhou, and Faruque, Mohammad Abdullah Al. Leaky DNN: stealing deep-learning model secret with GPU context-switching side-channel. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN* (2020), IEEE, pp. 125–137. https://doi.org/10.1109/DSN48063.2020.00031.

[40] Weiss, Jonah O'Brien, Alves, Tiago, and Kundu, Sandip. Ezclone: Improving dnn model extraction attack via shape distillation from gpu execution profiles, 2023. https://arxiv.org/abs/2304.03388.

[41] Wu, Ruoyu, Kim, Taegyu, Tian, Dave Jing, Bianchi, Antonio, and Xu, Dongyan. {DnD}: A {Cross-Architecture} deep neural network decompiler. In *31st USENIX Security Symposium (USENIX Security 22)* (2022), pp. 2135–2152.

[42] Xiang, Yun, Chen, Zhuangzhi, Chen, Zuohui, Fang, Zebin, Hao, Haiyang, Chen, Jinyin, Liu, Yi, Wu, Zhefu, Xuan, Qi, and Yang, Xiaoniu. Open DNN box by power side-channel attack. *IEEE Trans. Circuits Syst. 67-II*, 11 (2020), 2717–2721. https://doi.org/10.1109/TCSII.2020.2973007.

[43] Xu, Qian, Arafin, Md Tanvir, and Qu, Gang. Security of neural networks from hardware perspective: A survey and beyond. In *ASPDAC '21: 26th Asia and South Pacific Design Automation Conference* (2021), ACM, pp. 449–454. https://doi.org/10.1145/3394885.3431639.

[44] Yan, Mengjia, Fletcher, Christopher W., and Torrellas, Josep. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *29th USENIX Security Symposium, USENIX* (2020), Srdjan Capkun and Franziska Roesner, Eds., USENIX Association, pp. 2003–2020. https://www.usenix.org/conference/usenixsecurity20/presentation/yan.

[45] Yu, Honggang, Ma, Haocheng, Yang, Kaichen, Zhao, Yiqiang, and Jin, Yier. Deepem: Deep neural networks model recovery through EM side-channel information leakage. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust, HOST* (2020), IEEE, pp. 209–218. https://doi.org/10.1109/HOST45689.2020.9300274.

[46] Yu, Honggang, Yang, Kaichen, Zhang, Teng, Tsai, Yun-Yun, Ho, Tsung-Yi, and Jin, Yier. Cloudleak: Large-scale deep learning models stealing through adversarial examples. In *27th Annual Network and Distributed System Security Symposium, NDSS* (2020), The Internet Society. https://www.ndss-symposium.org/ndss-paper/cloudleak-large-scale-deep-learning-models-stealing-through-adversarial-examp

[47] Yuan, Xiaoyong, Ding, Leah, Zhang, Lan, Li, Xiaolin, and Wu, Dapeng Oliver. ES attack: Model stealing against deep neural networks without data hurdles. *IEEE Trans. Emerg. Top. Comput. Intell. 6*, 5 (2022), 1258–1270. https://doi.org/10.1109/TETCI.2022.3147508.

[48] Zhang, Jinquan, Wang, Pei, and Wu, Dinghao. Libsteal: Model extraction attack towards deep learning compilers by reversing dnn binary library. https://faculty.ist.psu.edu/wu/papers/DLCompilerAttack.pdf.

[49] Zhao, Rui, Yan, Ruqiang, Chen, Zhenghua, Mao, Kezhi, Wang, Peng, and Gao, Robert X. Deep learning and its applications to machine health monitoring: A survey. *CoRR abs/1612.07640* (2016). http://arxiv.org/abs/1612.07640.

[50] Zhu, Yuankun, Cheng, Yueqiang, Zhou, Husheng, and Lu, Yantao. Hermes attack: Steal DNN models with lossless inference accuracy. In *30th USENIX Security Symposium, USENIX* (2021), Michael Bailey and Rachel Greenstadt, Eds., USENIX Association, pp. 1973–1988. `https://www.usenix.org/conference/usenixsecurity21/presentation/zhu`.