Extracting DNN Architectures via Runtime Profiling on Mobile GPUs

Anonymous Submission

Abstract. 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 paper explores the feasibility of model extraction attacks on mobile devices using 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 paper presents a novel approach to identifying the original DNN architecture by utilizing duration and count binning of the profile as a side-channel, even in the presence of optimization-induced noise and obfuscation in the optimization process. To the best of our knowledge, this is the first-ever solution to extract architectures from optimized DNN models deployed on mobile GPUs, even with obfuscation in the optimization process.

Keywords: deep neural network · model extraction · side-channel

1 Introduction

Deep Neural Networks (DNNs), have achieved remarkable performance improvements in classification and regression problems over the last decade [KSH12, SVL14, VSP+17]. As a consequence, artificial intelligence (AI) providers are deploying DNNs in various domains ranging from autonomous vehicles [JGBG20] health monitoring [ZYC+16]. Critically, the emerging business models to monetize DNNs, including licensing, pay-per-use, or pay-per-install, view DNNs as intellectual property (IP). Therefore, a compromise to the DNN IP is also a compromise to the provider's business model.

Previous work has demonstrated the feasibility of a model extraction attack which enables an adversary to steal a DNN [TZJ+16, XAQ21, LZX+22]. 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, the creation of a DNN incurs cost at each stage of the machine learning pipeline, including data collection [IEE22] and DNN design [CS23], training, and maintenance [SHG+15], making the IP a valuable asset. The training cost alone may eclipse \$1m [SPS20]. This investment is lost if an adversary steals the DNN. Second, an adversary may calculate the gradient of a stolen DNN, enabling them to complete further adversarial attacks with much more potency [PMG+17]. They could craft maliciously perturbed inputs which fool the DNN, giving the adversary control of the model's output, or could reconstruct training data [AM18, AMKS21]. Either of these attacks may compromise the system in which the DNN operates.

Model extraction attacks are usually split into two steps. The first step, architecture extraction [HLL+20, MXL+22, WZZ+20], 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 [YFT20, HDK+18, LS20, JMKM20], GPU [PHW+22, NNQA18, LWC+19], and DNN accelerators [HZS18], power consumption [JMKM20, XCC+20], electromagnetic emanations [YMY+20, MXL+22, BBJP19, CW21], PCIe bus snooping [ZCZL21, HLL+20], and application profiles [ZCZL21, JMKM20, PHW+22, NNQA18, WZZ+20, HLL+20].

In this work, we consider architecture extraction attacks applied to mobile devices. There is a considerable lack of security measures implemented in deployed AI applications on these devices with 41% of mobile applications implementing no protections for their models [SSLM21]. 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, Arm Mali, and Apple Neural Engine [ITK+19]. A large-scale study by Sun et. al [SSLM21] 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) [CMJ+18] 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.

Overcoming compiler-based optimization in a model extraction attack is a challenge. As discussed in previous work [HLL+20], the TVM compiler stack may be employed as a defense strategy against architecture extraction. Operator-level fusion results in significant changes to the duration of each operator's computation. This makes it challenging for adversaries to accurately predict the computation performed by each operator. Graph-level fusion circumvents architecture extraction approaches which predict one layer at a time because each operator may implement multiple layers. We take the challenge one step further by considering obfuscation in the optimization process when the fused operators have arbitrary names. Changed operator names occur in existing work by Huang et. al [HTC+23] which focuses on privacy of data inputs to a DNN.

In our study, we simulate the optimization problem with the Apache TVM framework. We demonstrate an effective architecture extraction attack on a deployed DNN model on the Arm Mali G52 GPU using a TVM runtime profile as a side-channel that yields a sequence of layer durations. Despite the challenge of optimization-induced noise and obfuscation in the optimization process, we show that we can accurately predict the DNN architecture with a feature engineering technique that bins the layer durations. This approach allows us to precisely identify the original DNN architecture. Our work differs from that of Wu et. al [WKT+22], Zhang et. al [ZWW], and Liu et. al [LYW+22], who wrote decompilers for compiled TVM binaries, because we consider obfuscation in our threat model, a different hardware backend, and a different side-channel. Additionally, our attack will work even if the AI provider obfuscates the file format of the TVM binary as proposed by Chen et. al [CYM+21] since we only use the runtime profile, while the decompiler technique will not work.

1.1 Contributions

- We present a method to extract DNN architecture from runtime profiles even when the model is optimized in a way that fuses multiple DNN layers.
- We present a novel binning approach for profile analysis that can discern runtime features, even in the absence of feature information due to obfuscation.
- We demonstrate our approach achieves 100% accuracy on the Arm Mali G52 GPU with OpenCL.
- We further demonstrate that we can maintain >97.2% accuracy across all model optimization levels.
- 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.

1.2 Paper Organization

The rest of this paper is organized as follows. In Section 2, we discuss the notation we follow, the components of a model extraction attack, and profiling the TVM runtime. We introduce our binning attack and threat models in Section 3. In Section 4, we detail the DNN architectures, software stack, and hardware considered in our experiments. We present experimental results of architecture extraction across various optimization levels and threat scenarios in Section 5. We suggest mitigations against this attack in Section 6. Finally, in Section 7, we conclude the paper.

2 Preliminaries

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 $y \in \mathcal{Y}$, $\text{argmax}(y)$ represents the classifier's predicted class for the input x.

The difference between a DNN architecture and a DNN model is illustrated in Figure 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

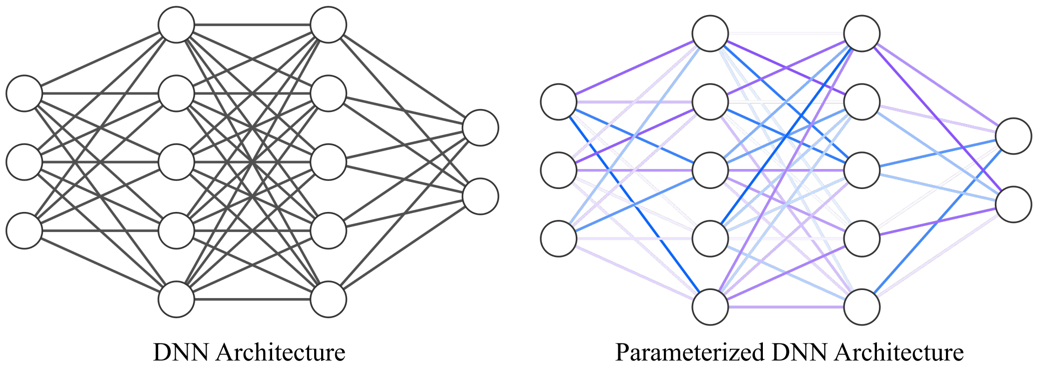


Figure 1: A DNN architecture is an unparameterized computation graph specifying how to compute an output of the network given some input. The parameters of a DNN are found through a training process, resulting in a specific parameterization of an architecture, or a model.

extremely large. Finding a high-performing architecture within this space is an active research problem [CS23]. 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 [CJM20].

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

$$\text{argmin}_\theta \sum_{x_i \in \mathcal{D}} L(y_i, f^\theta(x_i)) \tag{1}$$

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 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 [CAD+21, AM18, AMKS21]. 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 [GCY+21, YDZ+22, YYZ+20, JCB+20, CJM20, dSBB+21], 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

$$\mathcal{A}(\mathcal{H}(f^\theta)) = \mathbb{E}[f|\mathcal{H}(f^\theta)] = \hat{f} \tag{2}$$

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

$$\frac{1}{N}\sum_{i=1}^{N} \mathbb{1}(f_i = \hat{f}_i) \tag{3}$$

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:

$$\text{argmin}_{\hat{\theta}} \sum_{x_i \in \mathcal{D}} L(f^\theta(x_i), \hat{f}^{\hat{\theta}}(x_i)) \tag{4}$$

which is similar to the DNN training process described in Equation 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 [MFLG19, YDZ+22].

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 [MXL+22].

2.4 TVM Runtime Profiles

2.4.1 Apache TVM

Apache TVM [CMJ+18] 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

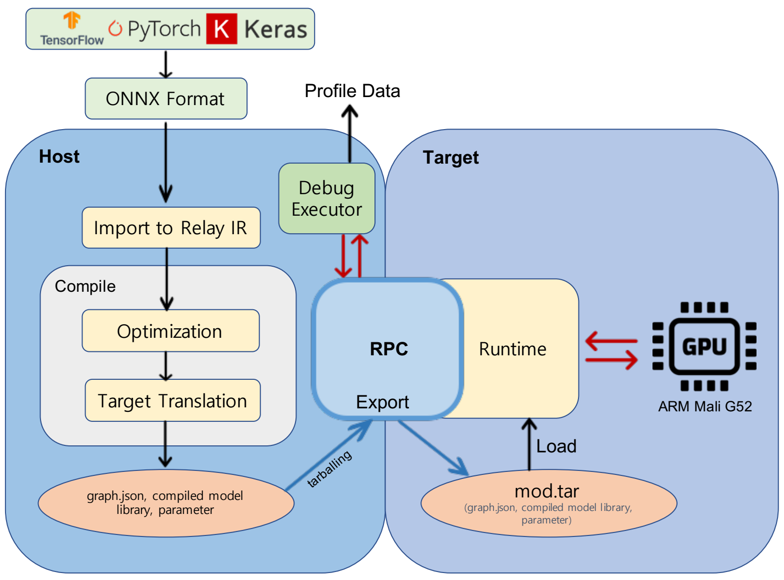


Figure 2: TVM Profiling system diagram.

shown by Figure 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 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 [HTC+23]. In Figure 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.

3 Attack Methodology

3.1 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 [SSLM21] and may either enable the TVM profiler through a remote procedure call or SSH session. In either case the RPC 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 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.

3.2 Attack Framework

Figure 3 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 [JMKM20, PHW+22, WAK23, XCC+20, YMY+20], 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

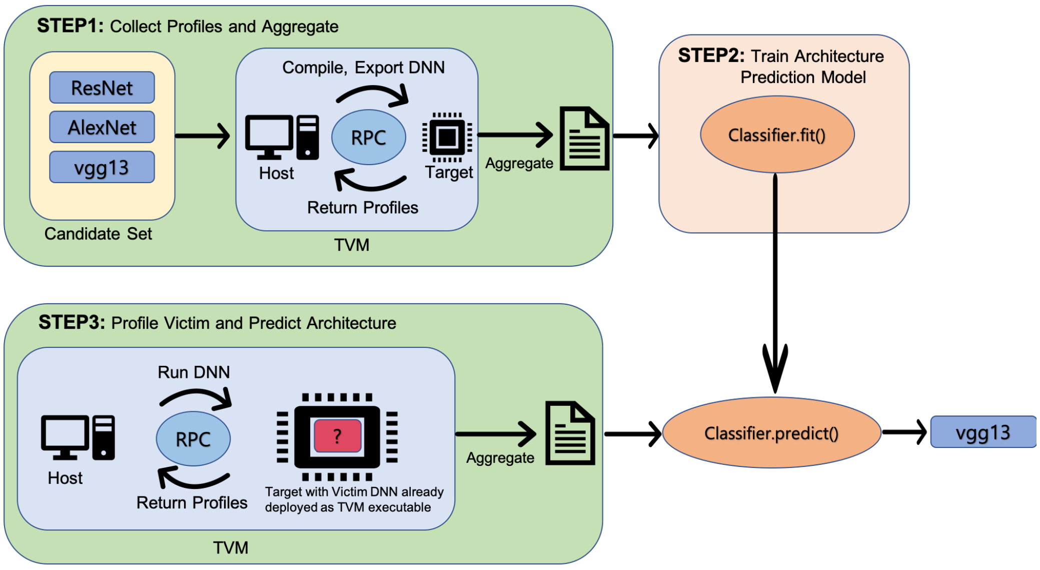

STEP1: Collect Profiles and Aggregate
ResNet
AlexNet
vgg13
Candidate Set
Compile, Export DNN
RPC
Host
Target
Return Profiles
TVM
Aggregate
STEP2: Train Architecture Prediction Model
Classifier.fit()
STEP3: Profile Victim and Predict Architecture
Run DNN
RPC
Host
Return Profiles
?
Target with Victim DNN already deployed as TVM executable
TVM
Aggregate
Classifier.predict()
vgg13

Figure 3: Step-by-step illustration of proposed architecture extraction attack.

(Step 3). Having this, the adversary would then run a parameter extraction attack to complete the DNN model theft.

3.2.1 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 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 [HLL+20]. However, when choosing our approach we had to take into account the effect of TVM optimizations which may fuse DNN layers. A convolution 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 [WAK23]. 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.

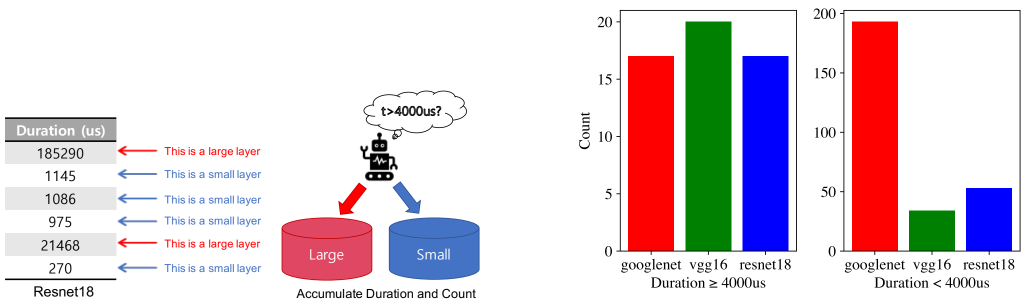


Figure 4: Sortation of layer duration from a ResNet18 model into 2 bins (left), Binned count distribution for different models (right).

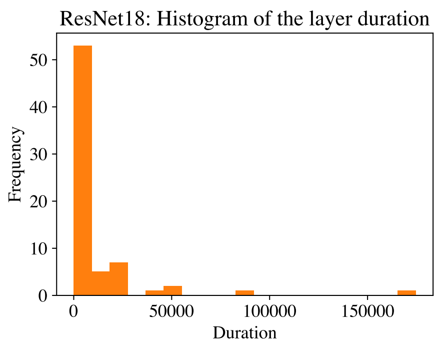


Figure 5: Distribution of ResNet18 layer durations at TVM optimization level 0. The x axis is in microseconds.

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) [GWBV02], which has been used in prior work to shrink the width of side-channel data [PHW+22, WAK23]. This also reduces the complexity of training the architecture prediction model.

Scenario 2: TVM is an IR-based optimizing enabled framework. Therefore there could be always the case that the AI provider tunes their own layer and names them 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 4. 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 $[4000, \infty)$. Therefore a TVM operator with *operator_duration* = 5000us 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 4 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 5.

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. The adversary will validate that the model generalizes well with high architecture prediction accuracy before moving on to the online phase.

3.2.2 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. 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.

4 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+).

In our study, we utilized the Open Neural Network Exchange (ONNX) format to prepare 34 deep learning image classification models converted from the PyTorch model zoo [PyT21], which served as our candidate set. We show these in Table 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 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

Table 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 [PyT21] except for the ResNext architectures due to lack of support for group convolution TVM operations on the Arm Mali G52 GPU.

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	

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 5. 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.

5 Results

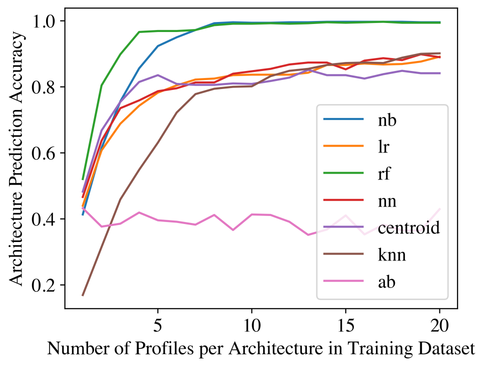
In this section, we test our proposed architecture extraction attack using Scenario 1 and Scenario 2. The organization of this section is as follows. First, we validate that the attack achieves sufficient accuracy in the easiest case holding the TVM optimization level constant, and we investigate the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

5.1 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 6 we show that the architecture extraction attack can achieve



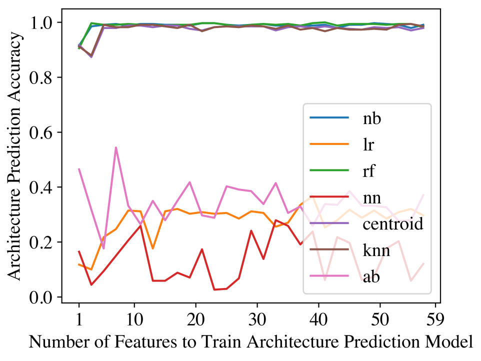


Figure 6: Architecture prediction accuracy by number of training profiles per architecture (left) and number of features (selected by RFE using a Random Forest model) used to train the architecture prediction model (right), for Scenario 1 considering only TVM optimization level 0.

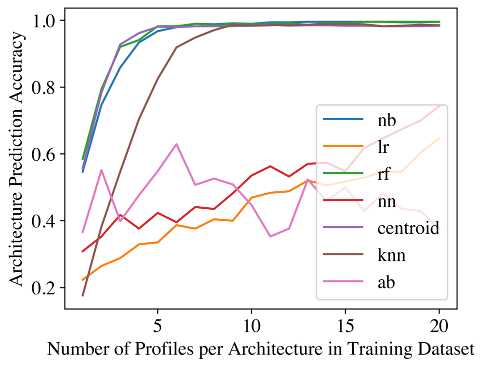


Figure 7: Architecture prediction accuracy by number of training profiles per architecture for Scenario 2 considering only TVM optimization level 0.

100% accuracy while reducing the computational complexity of the attack. The left subfigure demonstrates 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 shortens the length 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 Gaussian Naive Bayes and Random Forest architecture prediction models perform best.

Scenario 2: We run a similar experiment for Scenario 2. Since the dataset is already narrow with 4 features, do not consider shrinking the width of the data. Figure 7 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 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.

Table 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%

5.2 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 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 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%

5.3 Train all Optimization Levels, Test on Each Optimization 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 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.

5.4 Further Optimizing the Attack

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 8, 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 3 with 8 profiles per architecture per optimization level.

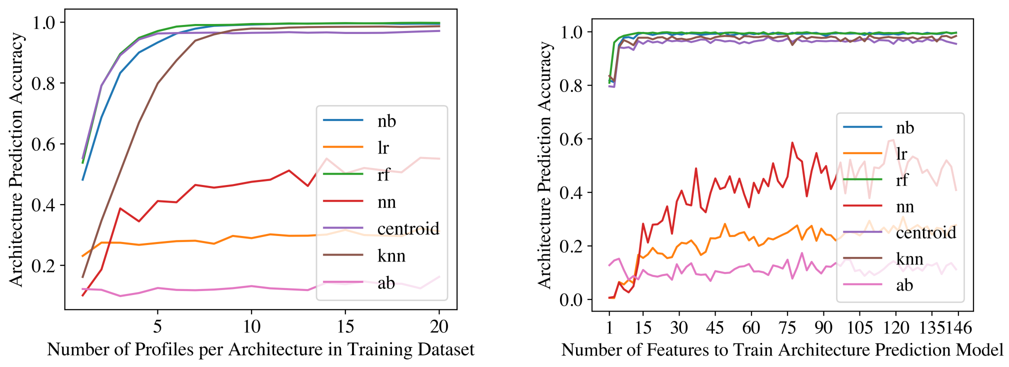


Figure 8: Architecture prediction accuracy by number of training profiles per architecture per optimization level (left) and number of features (selected by RFE using a Random Forest model) used to train the architecture prediction model (right), for Scenario 1, training and testing on all optimization levels.

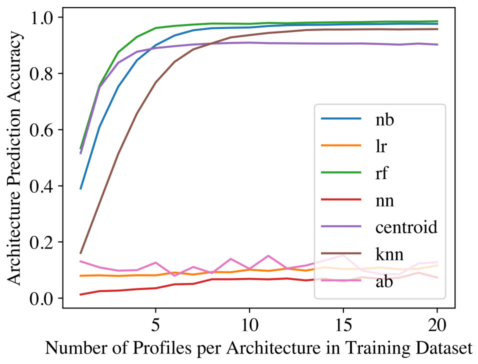


Figure 9: Architecture prediction accuracy by number of training profiles per architecture per optimization level for Scenario 2, training and testing on all optimization levels.

This means that the data collection requires only $8 \cdot 34 \cdot 4 = 1088$ 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 7, these results show that the same reduction in attack complexity may be achieved despite considering all optimization levels instead of one.

6 Defense Strategies

The accuracy of the binning attack across optimization levels demonstrates that as long as the adversary is aware of the optimization stack, they may add all possibilities to their training data. This cat-and-mouse game can continue with the provider adding more optimization transformations and the adversary collecting data with those transformations. One option to end the cat-and-mouse game would be for the AI provider to implement and apply proprietary optimization techniques that the adversary cannot replicate. However, requires too much overhead for most providers, is a challenging problem, and may not achieve the same computational load reduction as open-source techniques. Another mitigation would be constant-time obfuscation by the addition of dummy computations. The dummy computations would add noise to the profile data such that the side-channel data is nearly constant regardless of the DNN architecture. This could be implemented across the entire candidate set or subsets of the candidate set. Unfortunately, constant-time obfuscation comes with a significant performance overhead which opposes the TVM optimization objective in the first place.

Below are other defense strategies specific to the threat models considered in this work.

6.1 Scenario 1: Elevated Privileges

Defending against scenario 1 is more difficult to defend due to a stronger adversary; they may profile OpenCL at the application level and observe both the name and duration of each kernel operation. A comprehensive mitigation for similar vulnerabilities demonstrated with the CUDA GPU API is to require elevated permission to use the profiling tools [NNQA18]. Unfortunately, adversaries with administrator privileges will maintain access to the OpenCL profile side-channel, and to defend against them, preventing installation or use of profiling tools is the best option.

6.2 Scenario 2: Encrypt Models in Deployment

In scenario 2, the adversary is able to write their own program that executes a TVM computational graph present on the device in debug mode. This may be mitigated if the `model.tar` file is encrypted by the AI provider prior to deployment and decrypted at runtime. Then the adversary's code cannot execute the computational graph without the encryption key. A comprehensive study by Sun et. al found that only 59% of mobile apps are encrypted [SSLM21].

7 Conclusion

In this paper, we presented a method to extract a DNN architecture from TVM runtime profiles, despite various levels of DNN optimization that fuses TVM operations. We also consider a more challenging scenario, when the TVM operator names are 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. To the best of our knowledge, this is the first work to extract DNN architectures from optimized DNN models on mobile GPUs despite obfuscation in the optimization process.

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.

Our findings highlight the importance of carefully selecting a profiling technique and preprocessing steps that can effectively capture the model architecture while being robust to optimization-based deep learning compilers. This attack enhances an adversary's ability to steal a DNN in a model extraction attack, which causes security, privacy, and financial concerns to the AI industry. This work highlights the importance of research in developing resilient techniques to defend against such attacks.

8 Acknowledgement

Redacted.

9 Availability

Code is available at https://github.com/{redacted}.

References

[AM18] Naveed Akhtar and Ajmal S. Mian. Threat of adversarial attacks on deep learning in computer vision: A survey. *IEEE Access*, 6:14410–14430, 2018. https://doi.org/10.1109/ACCESS.2018.2807385.

[AMKS21] Naveed Akhtar, Ajmal Mian, Navid Kardan, and Mubarak Shah. Threat of adversarial attacks on deep learning in computer vision: Survey II. *CoRR*, abs/2108.00401, 2021. https://arxiv.org/abs/2108.00401.

[BBJP19] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. CSI NN: reverse engineering of neural network architectures through electromagnetic side channel. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX*, pages 515–532. USENIX Association, 2019. https://www.usenix.org/conference/usenixsecurity19/presentation/batina.

[CAD$^+$21] Anirban Chakraborty, Manaar Alam, Vishal Dey, Anupam Chattopadhyay, and Debdeep Mukhopadhyay. A survey on adversarial attacks and defences. *CAAI Trans. Intell. Technol.*, 6(1):25–45, 2021. https://doi.org/10.1049/cit2.12028.

[CJM20] Nicholas Carlini, Matthew Jagielski, and Ilya Mironov. Cryptanalytic extraction of neural network models. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO*, volume 12172 of *Lecture Notes in Computer Science*, pages 189–218. Springer, 2020. https://doi.org/10.1007/978-3-030-56877-1_7.

[CMJ$^+$18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 578–594. USENIX Association, 2018. https://www.usenix.org/conference/osdi18/presentation/chen.

[CS23] Krishna Teja Chitty-Venkata and Arun K. Somani. Neural architecture search survey: A hardware perspective. *ACM Comput. Surv.*, 55(4):78:1–78:36, 2023. https://doi.org/10.1145/3524500.

[CW21] Lukasz Chmielewski and Leo Weissbart. On reverse engineering neural network implementation on GPU. In *Applied Cryptography and Network Security Workshops - ACNS*, volume 12809, pages 96–113. Springer, 2021. https://doi.org/10.1007/978-3-030-81645-2_7.

[CYM+21] Zhi Chen, Cody Hao Yu, Trevor Morris, Jorn Tuyls, Yi-Hsiang Lai, Jared Roesch, Elliott Delaye, Vin Sharma, and Yida Wang. Bring your own codegen to deep learning compiler. *CoRR*, abs/2105.03215, 2021. https://arxiv.org/abs/2105.03215.

[dSBB+21] Jacson Rodrigues Correia da Silva, Rodrigo Ferreira Berriel, Claudine Badue, Alberto F. De Souza, and Thiago Oliveira-Santos. 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.

[GCY+21] Xueluan Gong, Yanjiao Chen, Wenbin Yang, Guanghao Mei, and Qian Wang. Inversenet: Augmenting model extraction attacks with training data inversion. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI*, pages 2439–2447. ijcai.org, 2021. https://doi.org/10.24963/ijcai.2021/336.

[GWBV02] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. Gene selection for cancer classification using support vector machines. *Mach. Learn.*, 46(1-3):389–422, 2002. https://doi.org/10.1023/A:1012487302797.

[HDK+18] Sanghyun Hong, Michael Davinroy, Yigitcan Kaya, Stuart Nevans Locke, Ian Rackow, Kevin Kulda, Dana Dachman-Soled, and Tudor Dumitras. 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.

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

[HTC+23] Po-Hsuan Huang, Chia-Heng Tu, Shen-Ming Chung, Pei-Yuan Wu, Tung-Lin Tsai, Yi-An Lin, Chun-Yi Dai, and Tzu-Yi Liao. 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.

[HZS18] Weizhe Hua, Zhiru Zhang, and G. Edward Suh. Reverse engineering convolutional neural networks through side-channel information leaks. In *Proceedings of the 55th Annual Design Automation Conference, DAC*, pages 4:1–4:6. ACM, 2018. https://doi.org/10.1145/3195970.3196105.

[IEE22] IEEE. The radical scope of tesla's data hoard. https://spectrum.ieee.org/tesla-autopilot-data-scope, Aug 2022.

[ITK+19] Andrey Ignatov, Radu Timofte, Andrei Kulik, Seungsoo Yang, Ke Wang, Felix Baum, Max Wu, Lirong Xu, and Luc Van Gool. 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*, pages 3617–3635. IEEE, 2019. https://doi.org/10.1109/ICCVW.2019.00447.

[JCB+20] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. High accuracy and high fidelity extraction of neural networks. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX*, pages 1345–1362. USENIX Association, 2020. https://www.usenix.org/conference/usenixsecurity20/presentation/jagielski.

[JGBG20] Joel Janai, Fatma Güney, Aseem Behl, and Andreas Geiger. Computer vision for autonomous vehicles: Problems, datasets and state of the art. *Found. Trends Comput. Graph. Vis.*, 12(1-3):1–308, 2020. https://doi.org/10.1561/0600000079.

[JMKM20] Nandan Kumar Jha, Sparsh Mittal, Binod Kumar, and Govardhan Mattela. Deeppeep: Exploiting design ramifications to decipher the architecture of compact dnns. *ACM*, 17(1):5:1–5:25, 2020. https://doi.org/10.1145/3414552.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems*, pages 1106–1114, 2012. https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[LS20] Yuntao Liu and Ankur Srivastava. GANRED: gan-based reverse engineering of dnns via cache side-channel. In Yinqian Zhang and Radu Sion, editors, *CCSW'20, Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 41–52. ACM, 2020. https://doi.org/10.1145/3411495.3421356.

[LWC+19] Sihang Liu, Yizhou Wei, Jianfeng Chi, Faysal Hossain Shezan, and Yuan Tian. Side channel attacks in computation offloading systems with GPU virtualization. In *2019 IEEE Security and Privacy Workshops, SP*, pages 156–161. IEEE, 2019. https://doi.org/10.1109/SPW.2019.00037.

[LYW+22] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, Xiaofei Xie, and Lei Ma. Decompiling x86 deep neural network executables. *CoRR*, abs/2210.01075, 2022. https://doi.org/10.48550/arXiv.2210.01075.

[LZX+22] Yuntao Liu, Michael Zuzak, Daniel Xing, Isaac McDaniel, Priya Mittu, Olsan Ozbay, Abir Akib, and Ankur Srivastava. 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*, pages 312–315. IEEE, 2022. https://doi.org/10.1109/AICAS54282.2022.9869995.

[MFLG19] Seyed-Iman Mirzadeh, Mehrdad Farajtabar, Ang Li, and Hassan Ghasemzadeh. Improved knowledge distillation via teacher assistant: Bridging the gap between student and teacher. *CoRR*, abs/1902.03393, 2019. http://arxiv.org/abs/1902.03393.

[MXL+22] Henrique Teles Maia, Chang Xiao, Dingzeyu Li, Eitan Grinspun, and Changxi Zheng. Can one hear the shape of a neural network?: Snooping the GPU via magnetic side channel. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX*, pages 4383–4400. USENIX Association, 2022. https://www.usenix.org/conference/usenixsecurity22/presentation/maia.

[NNQA18] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael B. Abu-Ghazaleh. Rendered insecure: GPU side channel attacks are practical. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS*, pages 2139–2153. ACM, 2018. https://doi.org/10.1145/3243734.3243831.

[PHW$^+$22] Kartik Patwari, Syed Mahbub Hafiz, Han Wang, Houman Homayoun, Zubair Shafiq, and Chen-Nee Chuah. DNN model architecture fingerprinting attack on CPU-GPU edge devices. In *7th IEEE European Symposium on Security and Privacy, EuroS&P*, pages 337–355. IEEE, 2022. https://doi.org/10.1109/EuroSP53844.2022.00029.

[PMG$^+$17] Nicolas Papernot, Patrick D. McDaniel, Ian J. Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi, editors, *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 506–519. ACM, 2017. https://doi.org/10.1145/3052973.3053009.

[PyT21] PyTorch. Torchvision v0.10.0 documentation. https://pytorch.org/vision/0.10/, Jun 2021.

[SHG$^+$15] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems*, pages 2503–2511, 2015. https://proceedings.neurips.cc/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html.

[SPS20] Or Sharir, Barak Peleg, and Yoav Shoham. The cost of training NLP models: A concise overview. *CoRR*, abs/2004.08900, 2020. https://arxiv.org/abs/2004.08900.

[SSLM21] Zhichuang Sun, Ruimin Sun, Long Lu, and Alan Mislove. Mind your weight(s): A large-scale study on insufficient machine learning model protection in mobile apps. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 1955–1972. USENIX Association, August 2021.

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems*, pages 3104–3112, 2014. https://proceedings.neurips.cc/paper/2014/hash/a14ac55a4f27472c5d894ec1c3c743d2-Abstract.html.

[TZJ$^+$16] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX*, pages 601–618. USENIX Association, 2016. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/tramer.

[VSP$^+$17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems*, pages 5998–6008, 2017. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[WAK23] Jonah O'Brien Weiss, Tiago Alves, and Sandip Kundu. Ezclone: Improving dnn model extraction attack via shape distillation from gpu execution profiles, 2023. https://arxiv.org/abs/2304.03388.

[WKT$^+$22] Ruoyu Wu, Taegyu Kim, Dave (Jing) Tian, Antonio Bianchi, and Dongyan Xu. DnD: A cross-architecture deep neural network decompiler. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 2135–2152. USENIX Association, August 2022.

[WZZ$^+$20] Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. 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*, pages 125–137. IEEE, 2020. https://doi.org/10.1109/DSN48063.2020.00031.

[XAQ21] Qian Xu, Md Tanvir Arafin, and Gang Qu. Security of neural networks from hardware perspective: A survey and beyond. In *ASPDAC '21: 26th Asia and South Pacific Design Automation Conference*, pages 449–454. ACM, 2021. https://doi.org/10.1145/3394885.3431639.

[XCC$^+$20] Yun Xiang, Zhuangzhi Chen, Zuohui Chen, Zebin Fang, Haiyang Hao, Jinyin Chen, Yi Liu, Zhefu Wu, Qi Xuan, and Xiaoniu Yang. Open DNN box by power side-channel attack. *IEEE Trans. Circuits Syst.*, 67-II(11):2717–2721, 2020. https://doi.org/10.1109/TCSII.2020.2973007.

[YDZ$^+$22] Xiaoyong Yuan, Leah Ding, Lan Zhang, Xiaolin Li, and Dapeng Oliver Wu. ES attack: Model stealing against deep neural networks without data hurdles. *IEEE Trans. Emerg. Top. Comput. Intell.*, 6(5):1258–1270, 2022. https://doi.org/10.1109/TETCI.2022.3147508.

[YFT20] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX*, pages 2003–2020. USENIX Association, 2020. https://www.usenix.org/conference/usenixsecurity20/presentation/yan.

[YMY$^+$20] Honggang Yu, Haocheng Ma, Kaichen Yang, Yiqiang Zhao, and Yier Jin. Deepem: Deep neural networks model recovery through EM side-channel information leakage. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust, HOST*, pages 209–218. IEEE, 2020. https://doi.org/10.1109/HOST45689.2020.9300274.

[YYZ$^+$20] Honggang Yu, Kaichen Yang, Teng Zhang, Yun-Yun Tsai, Tsung-Yi Ho, and Yier Jin. Cloudleak: Large-scale deep learning models stealing through adversarial examples. In *27th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2020. https://www.ndss-symposium.org/ndss-paper/cloudleak-large-scale-deep-learning-models-stealing-through-adversarial-examples/.

[ZCZL21] Yuankun Zhu, Yueqiang Cheng, Husheng Zhou, and Yantao Lu. Hermes attack: Steal DNN models with lossless inference accuracy. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX*, pages 1973–1988. USENIX Association, 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/zhu.

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

[ZYC+16] Rui Zhao, Ruqiang Yan, Zhenghua Chen, Kezhi Mao, Peng Wang, and Robert X. Gao. Deep learning and its applications to machine health monitoring: A survey. *CoRR*, abs/1612.07640, 2016. http://arxiv.org/abs/1612.07640.