DeepSniffer: A DNN Model Extraction Framework Based on Learning Architectural Hints

Xing Hu[1], Ling Liang[1], Shuangchen Li[1], Lei Deng[1,2], Pengfei Zuo[1,3], Yu Ji[1,2], Xinfeng Xie[1]
Yufei Ding[1], Chang Liu[4], Timothy Sherwood[1], Yuan Xie[1]
University of California, Santa Barbara[1] Tsinghua University[2]
Huazhong University of Science and Technology[3] Citadel Securities[4]
{xinghu,lingliang,shuangchenli,leideng,xinfeng,yuanxie}@ucsb.edu,pfzuo@hust.edu.cn
jiy15@mails.tsinghua.edu.cn,{yufeiding,sherwood}@cs.ucsb.edu,liuchang2005acm@gmail.com

Abstract

As deep neural networks (DNNs) continue their reach into a wide range of application domains, the neural network architecture of DNN models becomes an increasingly sensitive subject, due to either intellectual property protection or risks of adversarial attacks. Previous studies explore to leverage architecture-level events disposed in hardware platforms to extract the model architecture information. They pose the following limitations: requiring a priori knowledge of victim models, lacking in robustness and generality, or obtaining incomplete information of the victim model architecture.

Our paper proposes DeepSniffer, a learning-based model extraction framework to obtain the complete model architecture information without any prior knowledge of the victim model. It is robust to architectural and system noises introduced by the complex memory hierarchy and diverse runtime system optimizations. The basic idea of DeepSniffer is to learn the relation between extracted architectural hints (e.g., volumes of memory reads/writes obtained by side-channel or bus snooping attacks) and model internal architectures. Taking GPU platforms as a showcase, DeepSniffer conducts model extraction by learning both the architecture-level execution features of kernels and the inter-layer temporal association information introduced by the common practice of DNN design. We demonstrate that DeepSniffer works experimentally in the context of an off-the-shelf Nvidia GPU platform running a variety of DNN models. The extracted models are directly helpful to the attempting of crafting adversarial inputs. Our experimental results show that DeepSniffer achieves a high accuracy of model extraction and thus improves the adversarial attack success rate from 14.6%~25.5% (without network architecture knowledge) to 75.9% (with extracted network architecture). The DeepSniffer project has been released in Github[1].

• **Computer systems organization → Architectures**; • **Computing methodologies → Machine learning**; • **Security and privacy → Domain-specific security and privacy architectures**.

Keywords domain-specific architecture; deep learning security; machine learning

ACM Reference Format:
Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, Yuan Xie. 2020. DeepSniffer: A DNN Model Extraction Framework Based on Learning Architectural Hints. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3373376.3378460

1 Introduction

Machine learning approaches, especially deep neural networks (DNNs), are transforming a wide range of application domains, such as computer vision [24, 45], speech recognition [61], and language processing [10, 48, 56]. Computer vision, for example, has seen commercial adoption of DNNs with impacts across the automotive industry, business service, consumer market, agriculture, government sector, and so forth [41]. Such maturing DNN technologies start to power existing industries.

DNN model characteristics, especially, model architectures (e.g., number of layers, layer connection topology, layer types, and the layer dimension sizes) are critical information for deep learning applications. By extracting such information, attackers can not only counterfeit the intellectual property of the DNN design, but also conduct more efficient adversarial attacks towards the DNN system [28, 44]. Previous studies have confirmed that the details of the model architecture information ultimately affect the success rate of

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378460

[1]https://github.com/xinghu7788/DeepSniffer

adversarial attacks that induce DNNs to misclassify a well-recognized output. The vulnerability to adversarial examples becomes one of the major risks for applying DNNs in safety-critical scenarios. Therefore, model extraction attacks, which reveal the internal model characteristics information, become an important attack model in DNN systems [44, 55, 57].

Previous algorithm-level studies mainly conduct model extraction through detecting the decision boundary of the victim black-box DNN models [44]. However, such approaches demand significant computational resources and huge time overhead: given the pre-knowledge of the total number of layers and their type information, it still takes 40 GPU-days to search a 7-layer network architecture with a simple chained topology [44]. Even worse, this approach cannot accommodate state-of-the-art DNNs with complex topology, e.g., DenseNet [20] and ResNet [17], due to the enlarged search space of possible network architectures.

Due to the limitation of the algorithm-level model extraction, some studies begin to explore architecture-level events to extract model-related information. These architecture-level events disposed in hardware platforms during the execution of DNN models are referred to as the *architectural hints* in the rest of this paper. For instance, Insecure Render [32] leverages a regression model to learn the number of input neurons of the Rodinia backpropagation algorithm with performance counter data on GPU. Cache Telepathy [62] focuses on a specific implementation of DNN on CPU, i.e., GEMM, and builds an analytical model to estimate the DNN layer dimension with the number of GEMM calls and their arguments. ReverseCNN [18] targets DNN hardware accelerators and calculates the possible dimension sizes with the assumption that full feature map and weight data trace are visible across the memory bus.

Although previous studies provide a great leap as the initial attempts to extract DNN model information with architectural hints, they pose the following limitations: 1) **Requiring a priori model knowledge**: They often work with a priori knowledge of the victim models, such as layer type or DNN architecture [32]. 2) **Lack of generality and robustness**: They rely on detailed (ad hoc) characterization and *analytical modeling* of dimension sizes and then infer the layer type or layer topology based on predicted dimension sizes. They may work under some specific implementations, but not robust and generally applicable to common scenarios with diverse runtime system optimization and architecture-level noises [18, 62]. 3) **Incomplete model extraction:** Their extraction methods obtain incomplete information about the DNN model architectures (i.e., either dimension sizes or neuron number). Thus, in terms of effectiveness evaluation, there is no direct evidence to show the relation between extracted information and end-to-end attack effectiveness.

In this work, we propose *DeepSniffer*, a framework to obtain the **complete** model architecture with **no priori knowledge** of the victim model and it is **robust** to system-level and architecture-level noises. The complete model architecture extraction includes the following steps: *run-time layer sequence identification*, *layer topology reconstruction*, and *dimension size estimation*. Among these steps, the *run-time layer sequence identification* is the most fundamental one and is missing in previous work [18, 32, 62], since they either take the layer type or neural network architecture as known information or impractically assume that the single-layer architecture hints can be easily distinguished. We map the *run-time layer sequence identification* to a sequence-to-sequence prediction problem and address it using learning-based approaches. One of the most important differences that DeepSniffer distinguishes from previous work is that it decouples layer sequence prediction from dimension size prediction, thus being more generally applicable and robust to noises.

We further propose and experimentally demonstrate end-to-end attacks in the context of an off-the-shelf Nvidia GPU platform with full system stacks, which urges the demand to design secure architecture and system to ensure the DNN security. In summary, we make the following contributions:

- We observe that complex system stack, run-time dynamics, and optimized memory hierarchy systems introduce both system-level and architecture-level noises in architectural hints. Previous studies are not feasible enough to handle such issues.
- We map the fundamental step of model extraction, i.e, run-time layer sequence identification, to a sequence-to-sequence prediction problem and adopt learning-based approaches to conduct accurate and robust run-time layer sequence prediction.
- We showcase the effectiveness of DeepSniffer to conduct model extraction with two sets of architectural hints under two attack scenarios. We experimentally demonstrate our methodologies on an off-the-shelf GPU platform. With the easy-to-get off-chip bus communication information, the extracted network architectures exhibit very small differences from those of the victim DNN models.
- We conduct an end-to-end attack to show that the extracted neural network architectures boost adversarial attack effectiveness, improving the attack success rate from 14.6%∼25.5% to 75.9% compared to cases without neural network architecture knowledge. DeepSniffer project has been released in the Github.

2 Background and Challenges

In this section, we introduce the background of DNN model characteristics and existing model extraction techniques at architectural perspective.

2.1 Model Characteristics

Model extraction attacks aim to explore the model characteristics of DNNs for establishing a near-equivalent DNN model [55], which is the initial step for further attacks. The model characteristics that an adversary may extract include the following: (1) ***network architecture*** consists of layer depth and types, connection topologies between layers, and layer dimensions (including the number of channels, feature map size, weight kernel size, stride, and padding in each layer). (2) ***parameters*** include the weights, biases, and Batch Normalization (BN) parameters. They are updated during the stochastic gradient descent (SGD) in the training process. (3) ***hyper-parameters*** refer to the configurations during training, including the learning rate, regularization factors, and momentum coefficients, etc.

Model extraction is the initial step for further adversarial attacks. With the extracted model characteristics, the adversary is able to build the substitute models for adversarial examples generation and then use these examples to attack the victim black-box model [2, 14, 38, 50]. Among all of the model characteristics, the network architecture is the most fundamental one for DNN security. Previous studies demonstrate that with the knowledge of the network architecture, the adversary is able to explore the extraction of model parameters, hyper-parameters, and even training data [55, 57]. In addition, previous work [28, 44] also observe that the network architecture similarity between the substitute and victim models plays a very important role for the success of adversarial attacks. Hence, this work mainly focuses on the neural network architecture extraction.

2.2 Model Extraction Techniques

Due to the importance of the neural network architecture, some initial studies are proposed to extract model architecture from an architectural perspective [18, 32, 62]. Insecure Render [32] infers the neuron number with GPU performance counter information for the specific algorithm (Rodinia Backpropagation) with the knowledge of the victim model. ReverseCNN [18] and CacheTelepathy [62] analytically compute the potential dimension spaces based on architectural hints in DNN accelerator and CPU cache. Then, they infer the layer type and topology based on the predicted dimension sizes.

These studies pose the following limitations: 1) Require knowledge or information of the victim model [32], which raises the difficulty for common use. 2) Rely on accurate dimension estimation for layer type and topology prediction, which is neither robust nor general applicable. ReverseCNN [18] assumes that all the feature maps and weight data are visible in the memory bus, which is not true in CPU/GPU platforms with complex system stack. Cache Telepathy [62] considers the scenario that all the Conv and FC layers are implemented with basic GEMMs (general matrix multiply). In GPU platforms, such an assumption is not practical considering there are many diverse implementations for Conv layers, such as Winograd or FFT-based approaches. In addition, accurate dimension estimation is extremely challenging in general purpose platforms with the existence of both system-level and architecture-level noises (more details in Section 4). 3) Extract imprecise or incomplete model architecture and it lacks evidence to show the effectiveness of such extracted information. Insecure Render [32] merely infers the neuron number. ReverseCNN and CacheTelepathy rely on the dimension sizes to predict the architecture and only obtain the potential network architecture candidate space. With the increasing complexity and depth of victim models, such candidate space may get too large for effective attacks.

To this end, we propose DeepSniffer, a learning-based framework to obtain the model architecture with no priori knowledge of the victim model. It decouples the layer sequence identification and topology reconstruction from dimension size prediction, thus being more robust to both system-level and architecture-level noises and applicable to more common cases. We also experimentally conduct an end-to-end attack to show the model extraction effectiveness.

3 Attack Model and Arch-Hints

The methodology of DeepSniffer can leverage available architectural hints to conduct model extraction. In this work, we showcase the adoption of DeepSniffer in GPU platforms. The threat model of this showcase is as shown in Figure 1, which mainly focuses on edge security where the adversary is able to physically access the victim platform. Specifically, the attacker can physically access one GPU platform encapsulating a victim DNN model for model extraction. Such a physical access based attack is practical and harmful, because the adversary is able to attack all the other devices sharing the same DNN model with the extracted model information from one device. Note that, we consider a threat model in which the adversary does not have any knowledge about the victim models including what family the DNN models belong to, what software code those models are implemented with, or any other information about the operation of the device under attack that is not directly exposed through externally accessible connections. The extraction attack is fully passive and only has the ability to observe architectural side-channel information over time.

To understand what architectural hints can be obtained in the hardware platforms, we first illustrate the overview of commonly-used GPU platforms in Figure 1b [42]. The CPU and GPU are connected by the PCIe bus, and the host and device memories are attached to the CPU and GPU through DDR and GDDR memory buses, respectively. This architecture is widely used in many real industrial products, including most of the existing L3 autopilot systems [53, 58].

We consider the following two attack scenarios according to the obtained architectural hints. Table 1 summarizes the available architectural hints and the extracted DNN model types under these two scenarios:

1) Scenario-1 (Side-channel attack): Previous studies show electromagnetic (EM) emanations can easily obtain the off-chip events [6] and even perform memory profiling [13]. Therefore, we can obtain the read and write memory access volume (R_v, W_v) by EM side-channel attacks. The kernel execution time (Exe_{Lat}) can be obtained either by EM side-channel attacks on interconnections between host and GPU or co-locating CUDA spy [32]. Under such a attack scenario, DeepSniffer is able to reconstruct DNN models with simple chained topology.

2) Scenario-2 (Bus snooping attack): The adversary passively monitors the memory bus and PCIe events. By observing the memory access trace through the GDDR memory bus, the adversary obtains the kernel read/write access volume (R_v/W_v) and memory address traces. Since there are control messages passing through the PCIe bus when a kernel is launched and completed, the adversary can determine the kernel execution latency (Exe_{Lat}) by monitoring the time between kernel launching and completing. Under this attack scenario, DeepSniffer is able to reconstruct DNN models with complex topology, as presented in Section 5.2.

Bus snooping is a well understood, practical, and low-cost attack that has been widely demonstrated [1, 5, 19, 21]; some mature prototypes and toolkits have already been developed, such as HMTT-v4 [1]. We assume that the adversary cannot access the data passing through buses, can only access the addresses, and thus the adversary described above can work even when data is encrypted. The address snooping is also much easier than the data snooping because of its lower frequency in GDDR5.

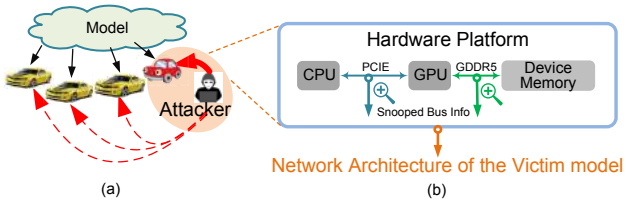


Figure 1. Illustration of the attack model. (a) Hack-one, attack-all-others with the extracted model. (b) GPU platform overview.

Table 1. Available architectural hints.

	Available Architectural Hints	Victim Model
Scenario-1	Exe_{Lat}, R_v, W_v	Chained DNN
Scenario-2	Exe_{Lat}, R_v, W_v, mem address trace	Complex DNN

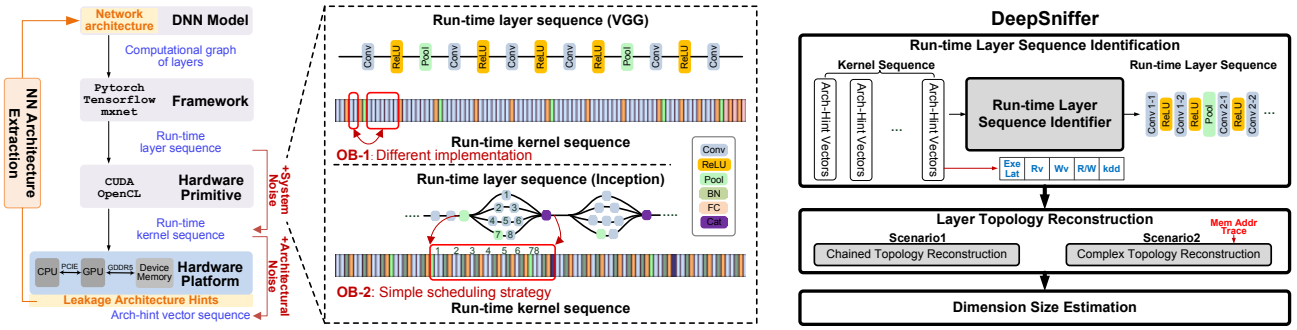
4 Observation and Design Overview

Understanding the transformation from computational graphs of DNN models to architectural hints is the initial step to learn how DeepSniffer works. In the following, we first present the workflow of DNN system stack dealing with a DNN model inference, which transforms the DNN computational graphs to architectural hints. We then present our observations about the available architectural hints when executing this flow. Motivated by these observations, we finally present the design overview of DeepSniffer.

DNN System Stack Noises: The detailed workflow is shown in Figure 2a. The computational graph of a DNN model is processed by the deep learning framework, hardware primitive libraries, and hardware platform. First, the deep learning framework optimizes the network architecture of the DNN model to form a framework-level computational graph of layers that is a representation of a composite function as a graph of connected layer operations. The framework then transforms this high-level computational graph abstraction to hardware primitives of run-time layer execution sequence. Then, the run-time hardware primitive libraries, such as cuDNN library [36], launch the well-optimized kernel sequence according to the layer type. Finally, such kernel sequences are executed on the hardware platform, which exhibit architectural hints, including the memory access pattern and the kernel execution latency.

It is challenging to recover the model architecture based on kernel sequences of architectural hints, because of the existence of architectural and system noises. **Architectural noise:** The comprehensive memory system optimization, such as the shuffling address mapping and complex memory hierarchy, raises the difficulty to obtain and identify the complete memory traces for accurate dimension size estimation. **System noise:** System run-time dynamics introduces the noises to architectural hint sequence in the further step. DNN layers are transformed into GPU-kernels dynamically during run-time, with various implementations (e.g. Winograd and FFT). The dynamic, not one-to-one correspondence mapping between layers and kernels makes it difficult to even figure out the number of layers and layer boundary in a kernel sequence, not to mention the corresponding layer dimension size.

Observations: To analyze the influence of such dynamics, we perform experiments on an off-the-shelf GPU platform with PyTorch [40] and cuDNN [36]. Figure 2b shows the transformations from the layer sequence of DNN models to the run-time kernel sequence, taking the VGG and Inception as illustrative examples. We have the following two observations. *OB-1*): Run-time kernel implementations vary across different models and even across time for the same model. For example, in the run-time kernel sequence of Figure 2b, the blue bars represent Conv kernels. The boxed two sets of Conv kernels in the VGG kernel sequence are different from each other with different implementions. *OB-2*): The kernel sequences of different layers have a static execution order related to the original computational graph of a DNN model. Such a characteristic exists in different DNN models with either a chained topology (e.g., VGG) or a complex



(a) (b) (c)

Figure 2. (a) Computational graph transformations through DNN system stack. (b) System noise during run-time layer sequence to kernel sequence transformations. (c) DeepSniffer overview.

topology (e.g., Inception). For instance, the highlighted Inception block in Figure 2b has 8 layer operations with branch topology. It is observed that the kernel sequence of every branch is executed in order when running on the GPU platform. Hence, such simple scheduling method of run-time layer sequence provide the opportunity to extract the victim model architecture.

Design Overview: The design overview of DeepSniffer is shown in Figure 2c. DeepSniffer proposes a run-time layer sequence identification methodology which learns the single kernel feature of architectural hints during kernel execution and inter-kernel/layer context probability for higher prediction accuracy. With the predicted layer sequence, DeepSniff then conducts the layer topology reconstruction and dimension size estimation to get the complete DNN architecture.

As the most fundamental step, run-time layer sequence identification translates the kernel-grained architectural hint sequence back to run-time layer sequence. Based on the observation that it can be mapped to a typical sequence-to-sequence translation problem, we propose a run-time layer sequence identification methodology based on deep learning techniques which learns both the single kernel feature and inter-kernel/layer context association for high prediction accuracy. Unlike previous work [18, 62], the run-time layer sequence prediction does not rely on the exact calculation of the dimension size parameters and is hence more robust and generally applicable.

Layer topology reconstruction is conducted based on monitoring the read-after-write (RAW) memory access pattern under the bus snooping attack scenario. To note, such methodology only needs partial memory traces for layer dependency analysis instead of the complete memory trace, which is much more practical than ReverseCNN [18].

After the first two steps, the skeleton of the DNN architecture is obtained. In observing that the ReLU kernels usually exhibit very large read cache misses, we estimate the dimension sizes based on ReLU read volume. The dimension size prediction is not precise in our work, but we take the dimension size prediction as a less important step than the other two and showcase a highly effective end-to-end adversarial attack with imprecise dimension sizes.

5 DeepSniffer Design

This section introduces the three steps of model extraction in DeepSniffer: run-time layer sequence identification, layer topology reconstruction, and dimension size estimation, as summarized in Figure 2c.

5.1 Run-time Layer Sequence Identification

After comprehensively investigating modern DNN models, we consider the following layers in this work: Conv (convolution), FC (fully-connected), BN (batch normalization), ReLU (rectified linear unit), Pool (pooling), Add, and Concat (concatenation), because most of the state-of-art neural network architectures can be represented by these basic layers [17, 45, 49, 52, 63, 64]. Note that it is easy to integrate other layers into DeepSniffer if necessary.

5.1.1 Problem Formalization

Formally, the run-time layer sequence identification problem can be described as follows: We obtain the architectural hint vectors of kernel sequence X with temporal length of T as an input. At each time step t, kernel feature X_t $(0 \leq t < T)$ can be described as a multiple-dimension tuple of architectural hints. Note that this tuple can be extended if the attack scenarios expose more architectural hints. The label space is a set of layer sequences comprised of all typical layers. The goal is to train a layer sequence identifier h to predict the run-time layer sequence (L) having the minimal distance to the ground-truth layer sequence (L^*).

The run-time layer sequence identification involves two internal correlation models: *kernel model* and *layer-sequence model*. The *kernel model* correlates the relationship between the architectural hints and the kernel type. The *layer sequence model* correlates the probabilistic distribution between the layers. We observe that the process of predicting the run-time layer sequence is similar to that of the speech recognition, as shown in Figure 3, which also involves two parts: an acoustic model converting acoustic signals to phonemes and

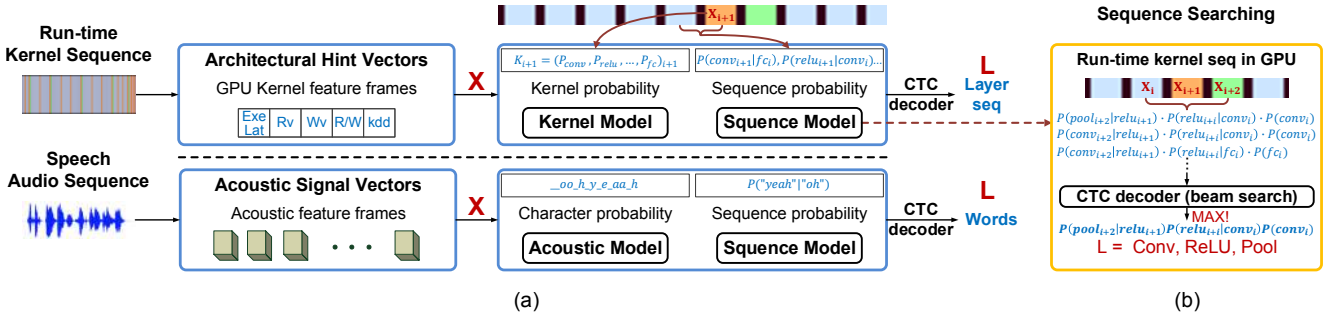


Figure 3. Context-aware layer sequence identification. (a) Identification flow (map the layer sequence identification to a speech recognition problem); (b) CTC decoder searches the sequence with highest probability.

a language model computing sequence probabilistic distribution on the words or sentences. Therefore this problem can be mapped to a speech recognition problem due to the high similarity of these two problems. Based on this insight, DeepSniffer leverages the auto speech recognition (ASR) techniques [15, 16] as a tool for run-time layer sequence identification. In the following subsection, we first show the intrinsic features of these two models.

5.1.2 Kernel and Layer Features

Architectural Hints of A Single Kernel. During DNN model execution, every layer conducts a series of kernel operation(s) for the input data and delivers output results to the next kernel(s), thus dataflow volume through kernels and the computation complexity constitutes the major parts of kernel features. As introduced in Section 3, we can determine the following architecture hints in Attack Scenario-1: 1) Kernel execution time ($\mathbf{Exe_{Lat}}$); 2) The kernel read volume ($\mathbf{R_v}$) and write volume ($\mathbf{W_v}$) through the memory bus; We can also calculate: 3) Input/output data volume ratio ($\mathbf{I_v/O_v}$) of each kernel, where the output volume (O_v) is equal to the write volume of this kernel and input volume (I_v) is equal to the write volume of the previous executed kernel. For the bus snooping attack scenario, we additionally use 4) kernel dependency distance (**kdd**) to indicate the topology influence. *kdd* is defined as the maximum distance between this kernel and the previous dependent kernels during the kernel sequence execution, which is a metric to encode the layer topology information in the kernel features. We regard this tuple (Exe_{Lat}, R_v, W_v, I_v/O_v, kdd) as one frame of kernel features.

We observe that although the kernels of different layers have their own features according to their functionality, it is still challenging to predict which layer a kernel belongs to, based on *kernel model* only. Our experiment results show that, on average, 30% of kernels are identified incorrectly with the executed features only and this error rate increases drastically with deeper network architectures (above 50%). The details of the experimental results are shown in Section 6.2.5. In summary, it is challenging to accurately predict layer sequence by considering single kernel architectural features only.

Inter-Layer Sequence Context. We observe that the temporal association of the layer sequence offers the opportunity for the better model extraction. Specifically, given the previous layer, there is a non-uniform likelihood for the following layer type, which is referred to as the *inter-layer context*. Such temporal association information between layers (aka. layer context) is inherently brought by the DNN model design philosophy. For example, there is a small likelihood that an FC layer follows a Conv layer in DNN models, because it does not make sense to have two consecutive linear transformation layers. Recalling the design philosophy of some typical NN models, e.g., VGG [45], ResNet [17], GoogleNet [49], and DenseNet [20], there are some common empirical evidences in building the network architecture: 1) the architecture consists of several basic blocks iteratively connected, 2) the basic blocks usually include linear operation first (Conv, FC), possibly following normalization to improve the convergence (BN), then non-linear transformation (ReLU), then possible down-sampling of the feature map (Pool), and possible tensor reduction or merge (Add, Concat).

Although DNN architectures evolve rapidly, the basic design philosophy remains the same. Even for the state-of-the-art autoML technical direction of Neural Architecture Search (NAS), which uses the reinforcement learning search method to optimize the DNN architecture, it also follows the similar empirical experience [64]. Therefore, such layer context generally exists in the network architecture design, which can be leveraged for layer identification.

5.1.3 Context-aware Layer Sequence Identification

Based on the analysis of kernel and inter-layer features, we adopt the Long Short-Term Memory (LSTM) model with a Connectionist Temporal Classification (CTC) decoder to build the context-aware layer sequence identifier h. The combination of LSTM model and CTC decoder is commonly used in the automatic speech recognition [15, 16]. As shown in Figure 3, given the input sequence $X = (X_1, .., X_T)$, the object function of training layer sequence identifier h is to minimize

the CTC cost for a given target layer sequence L^*. The CTC cost is calculated as follows:

$$cost\,(X) = -logP\,(L^*|h\,(X)) \quad (1)$$

where $P\,(L^*|h\,(X))$ denotes the total probability of an emission result L^* in the presence of input X.

An Example for Layer Sequence Prediction. The layer sequence prediction workflow is simplified as shown in Figure 3a. For the (i)th frame of the kernel sequence, its kernel architectural hint vectors are X_i. The layer sequence identifier first conducts the kernel classification based on X_i and obtains its probability distribution K_i of being Conv, ReLU, BN, Pool, Concat, Add, and FC.

$$K_i = \{P_{conv}, P_{relu}, P_{bn}, P_{pool}, P_{concat}, P_{add}, P_{fc}\}_i \quad (2)$$

The layer sequence identifier then uses a sequence model to estimate the conditional probability with the probability distribution of prior kernels: (K_0, K_1, ..., K_i). With the whole kernel feature sequence, the CTC decoder uses the beam search to find out the layer sequence with the largest conditional possibility as output (L). The layer sequence predictor has better prediction accuracy when there is less difference between the predicted layer sequence (L) and the ground-truth layer sequence (L*). The experimental details of the model training, validation, and testing are introduced in Section 6.1.

In the further step, we illustrate the detailed working mechanisms of a simplified CTC decoder in Figure 3b. In the monitor window (X_i, X_{i+1}, X_{i+2}), the CTC decoder searches throughout the searching space containing all of the potential layer sequences, such as (Conv, ReLU, Pool), (FC, ReLU, Conv), (Conv, Conv, Conv), etc. Then it outputs the layer sequence with the largest probability as output, which is (Conv, ReLU, Pool) in this case. In real cases, the CTC decoder is more complex and it considers the reduplication removing and adopts advanced searching algorithms [15, 16].

5.2 Layer Topology Reconstruction

DeepSniffer reconstructs the layer topology by monitoring the memory access pattern of the layers. Under the Attack Scenario-1, i.e. side-channel attack, the adversary obtains memory access events without detailed memory traces. DeepSniffer can reconstruct DNN models with chained topology. For chained DNN models without shortcut and concat interconnections, the neural network layer topology can be constructed naturally by connecting the layers in the predicted layer sequence. Under the Attack Scenario-2 (i.e., bus snooping attack), by obtaining the memory address trace in addition to kernel execution latency and read/write volumes, DeepSniffer can reconstruct DNN model architecture with much more complex topology. In this subsection, we show how the memory traffic reveals the interconnections between layers.

In the computational graph of a neural network architecture, if the feature map data of layer a is fed as the input of layer b, there should be a directed topology connection from a to b. Since this work focuses on the inference stage, there is only forward propagation across the whole network architecture. We first analyze the cache behaviors of feature map data and report the following observations:

Observation-1: *Only feature map data (activation data) can introduce read-after-write (RAW) memory access pattern in the memory bus.* There are several types of memory traffic data during the DNN inference: input images, weight parameters, and feature map data. Only feature map data is updated during inference. Feature map data is written to the memory hierarchy and read as the input data of the following layer(s). The input image and parameter data are not updated during the entire inference procedure. Therefore, the RAW memory access pattern is introduced only by the feature map data.

Observation-2: *Feature map data has a very high possibility to introduce RAW access pattern, especially for the convergent and divergent layers.* We examine the read cache misses of the feature map data in kernels of convergent and divergent layers at branches. The convergent layer receives feature map data from layers in different branches. For example, Add and Concat are the main convergent layers in neural network models. The read cache-miss rate of an Add layer is more than 98% and that of a Concat layer is more than 50%, as shown in Figure 4. The divergent layer outputs feature map data to several successor layers on different branches. We observe that GPU kernels execute the layers through one branch by one branch manner. Moreover, the memory traffic volume in the convergent layer and the successors of divergent layers have much higher memory traffic volume than the ground-truth weight data size. Since the CUDA library implements extreme data reuse optimizations that prioritize the weight tensor, the feature map needs to be flushed into memory and then read again due to a long reuse distance [42].

These two observations indicate that the RAW access pattern can be used to determine the interconnections among different layers. We propose a layer topology reconstruction scheme as follows: DeepSniffer scans all the layers in the run-time layer sequence. For layer i, all the addresses of its read requests constitute $ReadSet_i$ and that of write requests constitute $WriteSet_i$. DeepSniffer searches all its antecedent layers $layer_j \in (layer_0, layer_1, ..., layer_{i-1})$ in the sequence and checks whether $ReadSet_i \cap WriteSet_j = \emptyset$. If it is not $\emptyset$, DeepSniffer adds the connection between layer i and layer j. At the end, DeepSniffer checks whether there is any layer that doesn't have any successors in the topology, and eliminates the orphan layers by adding the connection to their following layer in the run-time layer sequence.

Note that, we do not require the complete memory address trace of all the feature map data, but only partial segments in order to identify the connections between different layers, which is robust to the memory traffic filtering.

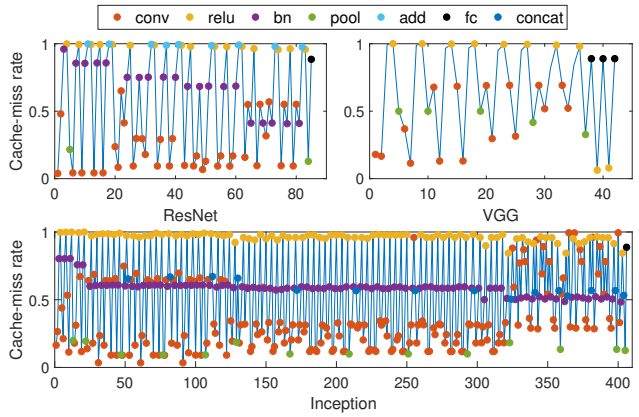


Figure 4. Read cache-miss rate of kernels in VGG11, ResNet18, and Inception.

5.3 Dimension Size Estimation

After completing the first two steps, we obtain the skeleton of the neural network architecture, based on which the dimension size estimation is conducted. Dimension size estimation includes the following two steps: 1) Layer feature map size prediction; 2) Dimension space calculation. In this section, we explain how to estimate the dimension size parameters according to the memory read and write volume.

We first characterize the cache miss rate in the GPU platform. For most DNN models, ReLU kernels have a stable high cache miss rate, surpassing 98%, as shown in Figure 4. Hence, the read volume through the bus R_v is almost the same as the input feature map size of the DNN model. Then the write volume W_v can be estimated which is equal to R_v. Based on this observation, we can obtain the input and output sizes of ReLU layers. Dimension parameters of DNN models are estimated based on the sizes of ReLU layers.

Step-1: Layer feature map size prediction. In neural networks, the previous layer's feature map output acts as the feature map input of the current layer, and thus the feature map output size (the feature map height/width and channel number for Conv or neuron number for FC) of the previous layer is equal to the feature map input size of the current layer. Hence, given the input size of a ReLU layer, the output size of the previous BN/Add/Conv/FC layer and the input size of the next Conv/FC layer can be estimated. Since the ReLU layer is almost a standard layer in every basic block, the feature map sizes of the layers in the victim model can be estimated by broadcasting the ReLU size to their adjacent layers. Add and Concat, which are the convergent layers and only exist in DNN models with complex interconnections, conduct element-wise add and concatenate operations for input feature map from different branches. After reconstructing the layer topology, output size of an Add operation is calculated as the input feature map size in each branch and that of a Concat operation is the sum of input feature map sizes in branches.

Step-2: Dimension Space Calculation. With the constructed layer topology and input/output size of every layer, we calculate the following dimension space: the input (output) channel size IC_i (OC_i), the input (output) height IH_i (OH_i), the input (output) width IW_i (OW_i), the weight size ($K \times K$), and the convolution padding P and stride S.

Based on the fact that the input size of each layer is the same as the output size of the previous layer, and there are some tensor constraints during computation as shown in Table 2, we are able to search the possible solution for every layer. Since we target the computer vision applications, the $IC_0 = 3$. We assume the feature map height and weight are the same and stride=1 (which are the common configuration in lots of DNN models). By iterating over possible kernel sizes (1, 3, 5 ..), we can estimate the other configuration parameters with the constraints in Table 2.

Notice that, we neither assure nor aim to obtain the precise dimension size parameters. Instead, we randomly select the possible sets of dimension parameters which satisfy the constraints in Table 2 as the configuration of the extracted DNN architecture. We conduct empirical experiments showing that with the neural network sequence and topology, we can achieve good attack effectiveness even though the dimension parameters are different from the victim model (More analysis in Section 6.4.2).

Table 2. Dimension space calculation.

Layer OP	Constraints & Estimation
Conv	$OH_i = \lfloor (IH_i + 2P - K)/S \rfloor + 1$ $OW_i = \lfloor (IW_i + 2P - K)/S \rfloor + 1$ $OH_i \times OW_i \times OC_i = O_i/N$
Pool	$OH_i = \lfloor (IH_i + 2P - K)/S \rfloor + 1$ $OW_i = \lfloor (IW_i + 2P - K)/S \rfloor + 1$ $OC_i = IC_i, OH_i \times OW_i \times OC_i = O_i/N$
FC	$OC_i = Oi/N$
BN	$OH_I = IH_i, OW_I = IW_i, OC_i = IC_i$
ReLU	$OH_I = IH_i, OW_I = IW_i, OC_i = IC_i$
Add	$OH_i = IH_{i_j}, OW_i = IW_{i_j}, OC_i = IC_{i_j}$
Concat	$OH_i = IH_{i_j}, OW_i = IW_{i_j}, OC_i = \sum_j IC_{i_j}$

6 Experimental Results

In this section, we evaluate the accuracy and robustness of the proposed network architecture extraction under side-channel attack and bus snooping attack scenarios.

6.1 Evaluation Methodology

To validate the feasibility of stealing the memory information during inference execution, we conduct the experiments on the hardware platform equipped with Nvidia K40 GPU [35]. The DNN models are implemented based on PyTorch framework [40], with CUDA8.0 [60] and cuDNN optimization library [36]. We use the GPU performance counter [34] to emulate bus snooping for kernel execution latency, kernel write, and read access volume information collection.

As an initial step for network architecture extraction, we first train the layer sequence identifier based on an LSTM-CTC model for layer sequence identification. The detailed training procedure is as follows.

Training: In order to prepare the training data, we first generate 8500 random computational graphs of DNN models and obtain the kernel architectural features. Two kinds of randomness are considered during random graph generation: topological randomness and dimensional randomness. At every step, the generator randomly selects one type of block from sequential, Add, and Concat blocks. The sequential block candidates include (Conv, ReLU), (FC, ReLU), and (Conv, ReLU, Pool) with or without BN. The FC layer only occurs when the feature map size is smaller than a threshold. The Add block is randomly built based on the sequential blocks with shortcut connection. The Concat block is built with randomly generated subtrack number, possibly within Add blocks and sequential blocks. The dimension size parameters – such as the channel, stride, padding, and weight size of Conv and neuron size of FC layer – are randomly generated to improve the diversity of the random graphs. The input size of the first layer and the output size of the last layer are fixed during random graph generation, considering that they are usually fixed in one specific target platform. We randomly select 80% of the random graphs as the ***training set*** and other 20% as the ***validation set*** to validate whether the training is overfitting or not.

Testing: To verify the effectiveness and generalization of our layer sequence identifier framework, we examine various commonly-used DNN models as the ***test set***, including VGG [45], ResNet [17], and Nasnet [64] to cover the representative state-of-the-art DNN models.

6.2 Layer Sequence Identification Accuracy

In this section, we first evaluate the layer sequence identification accuracy. Then we analyze the importance of the layer context information and the influence of noises in architectural hints.

6.2.1 Evaluation Metric

We quantify the prediction accuracy with the layer prediction error rate (LER), similar to those being used in speech recognition problems. It is the mean normalized edit distance between the predicted sequence and label sequence which quantifies the prediction accuracy [15, 16]. The detailed prediction calculation is as follows [15].

$$LER = \frac{ED(L, L^*)}{|L^*|} \tag{3}$$

where $ED(L, L^*)$ is the edit distance between the predicted layer sequences L and the ground-truth layer sequence L^*, i.e. the minimum number of insertions, substitutions, and deletions required to change L into L^*. $|L^*|$ is the length of ground-truth layer sequence.

6.2.2 Side-Channel Attack Scenario

We first evaluate the accuracy on the randomly generated DNN models, as the blue bars shown in Figure 5a. For DNN models with chained topology only, the average prediction error rate of layer sequence identification is about 0.06. For neural networks with shortcut and concat topology, the average LER of layer sequence identification is about 0.07 and 0.12. We then evaluate the accuracy of the typical sequential DNN models. The LER of AlexNet and VGG19 are 0.02 and 0.017 respectively, as shown in Table 3. Such results indicate that, under the side-channel attack, the proposed methodology can accurately identify the layer sequence. The LER is a little higher for DNN models with more complex topology.

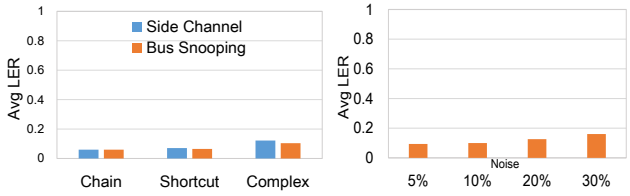


Figure 5. (a) Average prediction error rate of layer sequence identification under side-channel and bus snooping attacks. (b) Robustness to architectural hint noise under bus snooping attack scenario.

Table 3. Prediction error rate on typical networks.

Side-Channel		Bus Snooping			
AlexNet	VGG19	ResNet34	ResNet101	ResNet152	Nasnet_large
0.020	0.017	0.040	0.067	0.068	0.144

6.2.3 Bus Snooping Attack Scenario

Under the bus snooping attack scenario, the adversary has additional kernel dependency distance statistics since they can obtain memory address traces. Therefore, the adversary can achieve more accurate identification than that under side-channel attack scenario. As the red bars shown in Figure 5a, the average LER of layer sequence identification for random generated sequential models with chained layer topology, shortcut models with Add operations, and complex DNN models with Add and Concat is 0.06, 0.06 and 0.1 respectively.

Furthermore, we evaluate the accuracy in identifying several state-of-the-art DNN models, as shown in Table 3. For ResNet families, the prediction LER is lower than 0.07. For NasNet, the LER increases slightly due to the much deeper and complex connections. We take ResNet34 as an example to present the detailed results in Table 4. In summary, our proposed method is generally effective in correctly identifying the layer sequence. There may exist small deviation between the predicted sequence and ground-truth sequence. Thus we conduct end-to-end experiments in the Section 6.4, which shows that the extracted neural network architecture, although having a little deviation from the victim architecture, can still boost the attacking effectiveness.

6.2.4 Robustness to Hint Noises

We conduct experiments to analyze the accuracy sensitivity of the identifier taking in the kernel features with noises. Taking the bus snooping attack as an example, When kernel

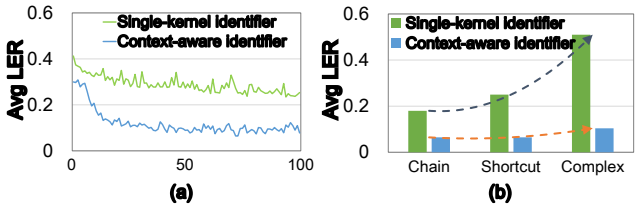


Figure 6. (a) Average prediction error rate comparison between single-kernel identifier and context-aware identifier during training process. (b) Average prediction rate comparison with different victim DNNs

Table 4. Identification results

DNN Model	Ground-truth Sequence	Predicted Sequence
ResNet18 (ErrorRate 0.032)	Conv BN ReLU MaxPool Conv BN ReLU Conv BN ADD ReLU Conv BN ReLU Conv BN ADD ReLU Conv BN Conv BN Add ReLU Conv BN ReLU Conv BN ADD ReLU Conv BN ReLU Conv BN Conv BN Add ReLU Conv BN ReLU Conv BN ADD ReLU Conv BN Conv BN Add ReLU Conv BN ReLU Conv BN FC	Conv ~~BN~~ ReLU MaxPool Conv BN ReLU Conv BN ADD ReLU Conv BN ReLU Conv BN ADD ReLU Conv BN Conv BN Add ReLU Conv BN ReLU Conv BN ADD ReLU Conv BN ReLU Conv BN Conv BN Add ReLU Conv BN ReLU Conv BN ADD ReLU Conv BN Conv BN Add ReLU Conv BN ReLU Conv BN ReLU FC

execution feature statistics are affected by random noises within 5%, 10%, 20%, or 30% of amplitude, the average error rate of the layer prediction increases from 0.08 to 0.16, as shown in Figure 5b. The results indicate that the layer sequence identifier is not sensitive to architectural hint noises.

6.2.5 Why is Inter-Layer Context Important?

To analyze the importance of inter-layer context information in this section, we compare the prediction error rate of two methods: a context-aware identifier considering layer context in our work (bus snooping scenario) and a single-kernel identifier based on multi-layered perception model. The key difference between these two identifiers is whether including the sequence model in Figure 3.

We compare both the prediction error rate along the identifier training processes from the 1st to 100th epochs (Figure 6a and prediction error rate for DNN models with different architectures (Figure 6b). We draw two conclusions from this experiment: 1) DeepSniffer can achieve much better prediction accuracy with considering the layer context information. The results show that the average LER of context-aware identifier is three times lower than the single-layer identifier (Figure 6a). 2) Layer context information is increasingly important when identifying more complex network architectures. As shown in Figure 6b, compared to the simple network architecture with only chain typologies, the more complex architectures with remote connections (e.g. Add or Concat) cause higher error rates. For the single-layer identifier, the LER dramatically increases when the network is more complex (from 0.18 to 0.5); while, for the context-aware identifier, the average LER demonstrates a non-significant increase (from 0.065 to 0.104). The experimental results indicate that the layer context with inter-layer temporal association is a very important information source, especially for the deeper and more complex neural networks.

6.3 Model Size Estimation

In this section, we show the feature map size estimation results of the input and output for every layer, which is the prerequisite for dimension space estimation. For both the side-channel and bus snooping attacks, the input and output feature map sizes of every layer in DNN model are calculated based on the ReLU memory traffic volume. Therefore, we show estimation accuracy under bus snooping scenario as an example in Figure 7, which is calculated as 1 minus the deviation between the estimated size and actual size. For Conv, BN, ReLU, Add, and Concat, the estimation accuracy can reach up to 98%. The FC presents lower accuracy since the FC layer is usually at the end of the network and the neuron number decreases. Thus, the activation data of the ReLU layer may be filtered, and it is not accurate to use ReLU read transactions to estimate the FC size. We use the read access volume to predict the input and output sizes of FC layers instead. The dimension size prediction results may be platform-dependent. However, we take the dimension size prediction as the less important step than the other two and experimentally validate the effectiveness of the extracted architectures with imprecise dimension sizes.

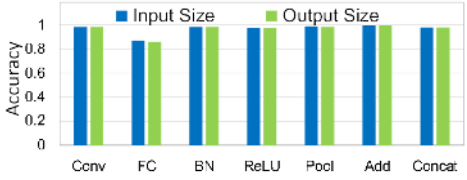


Figure 7. Layer input and output feature map size estimation (normalized to the ground-truth size).

6.4 How Effective are the Extracted Models?

The extracted network architecture can be used to conduct further-step attack. In this work, we use the adversarial attack, one of the most common attacks in the domain of neural network security, as an end-to-end attack case to show the effectiveness of the extracted network architecture.

6.4.1 Adversarial Attack with Extracted DNN Archs

In the adversarial attack, the adversary manipulates the output of the neural network model by inserting small perturbations into the input images that remain almost imperceptible to human vision [14]. The goal of adversarial attack is to search the minimum perturbation on input that can mislead the model to produce an arbitrary (*untargeted attack*)

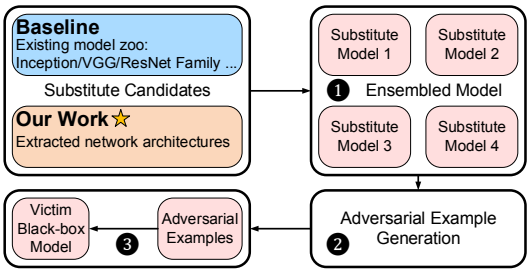


Figure 8. Adversarial attack flow.

[14] or a pre-assigned (*targeted attack*) [3, 25, 51] incorrect output. To conduct the adversarial attack against a black-box model, the adversary normally builds a substitute model first by querying the input and output of the victim model. Then the adversary generates the adversarial examples based on the white-box substitute model [37, 39, 51]. Finally, they use these adversarial examples to attack the black-box model.

In summary, the transfer-based adversarial attack flow is illustrated in Figure 8, which consists of the following steps:
1): Build substitute models. In our work, we train substitute models with the extracted network architectures, while baseline selects the typical network architectures to build the substitute model, as shown in Figure 8 .
2): Generate adversarial examples. The state-of-art solution [28] uses an ensembled method to improve the attacking success rate based on the hypothesis that if an adversarial image remains adversarial for multiple models, it is more likely to be effective against the black-box model as well. We follow the similar techniques to generate adversarial images for the ensemble of multiple models.
3): Apply the adversarial examples As the final step, the adversary attacks the black-box model using the generated adversarial examples as input data.

6.4.2 Adversarial Attack Efficiency

In this section, we show that the adversarial attack efficiency can be significantly improved with the extracted network architectures. We follow the same adversarial attack methodology in the previous work [28], which achieves better attacking success rate based on the ensemble of four substitute models. The only difference of our work is that we use the predicted network architectures to build the substitute models, as illustrated in the Figure 8.
Setup: In these experiments, we use ResNet18 [17] as the victim model for targeted attacks. Our work adopts the extracted neural network architecture to build the substitute models. For comparison, the baseline examines the substitute models established from following networks: *VGG* family [45] (`VGG11`, `VGG13`, `VGG16`, `VGG19`), *ResNet* family [17] (`ResNet34`, `ResNet50`, `ResNet101`, `ResNet152`), *DenseNet* family [20] (`DenseNet121`, `DenseNet161`, `DenseNet169`, `DenseNet201`), `SqueezeNet` [23], and `Inception` [49].
Extracted DNN Architectures: Based on the architectural hints of ResNet18, we extract DNN architectures following the three steps: run-time sequence identification, layer topology reconstruction, and dimension estimation, as shown in the Figure 9. In the run-time layer sequence identification, DeepSniffer accurately predicts the layer sequence with small errors in red color. In dimension size estimation, we randomly selects four dimension sets from the potential dimension space which satisfy the layer size and constraints in Table 2. The four dimension sets are different from the original victim ResNet18. Therefore, we validate the effectiveness of these extracted neural network models that are slightly different from the original victim model in the following.

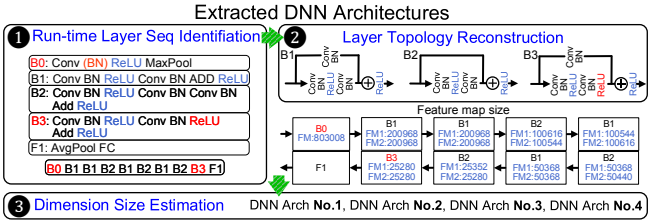


Figure 9. Extracted DNN architectures.

Adversarial Attack Effectiveness Results: First, we randomly select 10 classes, each class with 100 images from ImageNet dataset [12] as the original inputs for targeted attack tests. Then, we compare the attack effectiveness of adversarial examples generated by the following five solutions: ensembled substitute models from *VGG* family, *DenseNet* family, *Mix* architectures (squeezeNet, inception, AlexNet, DenseNet), *ResNet* family, and from extracted architectures using our proposed model extraction.

The attack success rate results are shown in Table 5. We report several observations: 1) The attack success rate is generally low for the cases without network architecture knowledge. The adversarial examples generated by substitute models with *VGG* family, *DenseNet* family, and *Mix* architectures only complete successful attacks in 14%–25.5% of the cases. 2) With some knowledge of the victim architecture, the attack success rate is significantly improved. For example, the substitute models within *ResNet* family achieve the attack success rate of 43%. 3) With our extracted network architectures – although it still has differences from the original network – the attack success rate is boosted to 75.9%. These results indicate that our model extraction significantly improves the success rate of consequent adversarial attacks.

In a further step, we take a deep look at the targeted attack which leads the images in Class-755 to be misclassified as Class-255 in the ImageNet dataset. We explore the effectiveness of ensembled models with various substitute combinations, by randomly picking four substitute models from the candidate model zoo. The results are shown in the blue bars of Figure 10. We also compare the results to the cases using substitute models 1) from *VGG* family; 2) from *DenseNet* family; 3) from squeezeNet, inception, AlexNet, and DenseNet ('Mix' bar in the figure); 4) from *ResNet* family;

and 5) from extracted cognate ResNet18 model (our method) to generate the adversarial examples. As shown in Figure 10, the average success rate of random cases is only 17% and the best random-picking case just achieves the attack success rate of 34%. We observe that all good cases in random-picking (attack success rate > 20%) include substitute models from ResNet family. Our method with accurate extracted DNN models performs best attack success rate across all the cases, 40% larger than the best random-picking case and *ResNet* family cases. To summarize, with the help of the effective and accurate model extraction, the consequent adversarial attack achieves a much better attack success rate. Therefore, it is extremely important to protect the neural network architectures in the DNN system stack, which can boost the adversarial attack effectiveness.

Table 5. Success rate with different substitute models.

	VGG family	DenseNet family	Mix	ResNet family	Extracted DNN
Success rate	18.1%	25.5%	14.6%	43%	75.9%

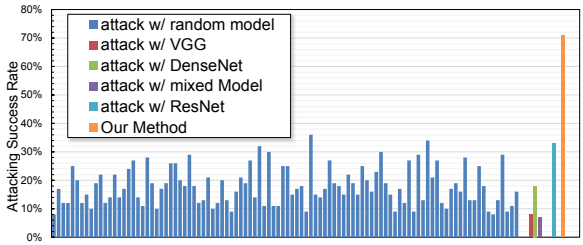


Figure 10. Explore the targeted attack success rate across different cases. Our method performs best.

7 Discussion

The standardization through the whole stack of neural network system facilitates such DNN architecture extraction. The standardized hardware platforms, drivers, libraries, and frameworks are developed to help machine learning industrialization with user-friendly interfaces. Transforming from the input neural network architecture to final hardware code depends on the compilation and scheduling strategies of DNN system stacks, which can be learned under the similar execution environment. Therefore, the adoption of these hardwares, frameworks, and libraries in the development workflow gives adversary an opportunity to investigate the execution pattern and reconstruct the network architecture based on architectural hints.

7.1 Approach Generality

The root cause of hacking the network architecture is to learn the transformations between framework-level computational graphs and kernel feature sequence. We discuss the general applicability of DeepSniffer techniques in terms of the following perspectives:

1) Different neural network architectures. Our methodology is generally applicable to various CNN models with different neural network architectures. During the training for the layer sequence predictor, we build the training set based on random graphs with basic operations provided by pytorch framework. Hence, the trained layer sequence predictor can be used to analyze any DNN models that are built based on the basic operations provide by framework (such as Conv2d, ReLU, and MaxPool2d, etc in pytorch). In addition, the predictor can be retrained with the extended training set that includes the other operations if necessary.

2) Different platforms. The overall model extraction methodology can be applied to other deep learning frameworks. As explained in Section 4, the kernel scheduling strategies of the framework give the opportunity to reverse engineer the computation graph of victim DNN models. We evaluate TensorFlow and Caffe2 (backend of the new Pytorch version), two other broadly-used frameworks, observing that they also use the similar scheduling methodology. The sequence-model-based method can be adopted in these scenarios because DeepSniffer does not rely on the exact dimension size that may differ in different frameworks, but learns the execution pattern in the target deep learning systems. Validating the proposed methods in mobile GPU platforms would be our future work.

3) Other attack models. In this work, we consider model extraction based on the minimum information that the adversary can get in the edge devices. In some other scenarios, such as machine learning cloud services [32, 62], other architectural hints may be obtained that include the API calls and GPU performance counters. DeepSniffer framework can leverage such architectural hints to explore the model extraction potentialities.

4) Algorithm optimization influence. Recently, many network quantization techniques are proposed for performance and energy optimization [22, 54]. Adopting low-precision data representations is the potential quantization method in GPU platforms. For example, users can simplify the network with int8, float16, or float32 operations provided by TensorFlow lite [54]. Our method can be applicable to such cases that don't introduce big changes the framework scheduling strategies. We still can identify the victim model architectures by learning the execution patterns in such DNN systems.

7.2 Defence Strategies

Microarchitecture Methodologies. There are a few architectural memory protection methods. *Oblivious Memory:* To reduce the information leakage on the bus, previous work proposes oblivious RAM (ORAM) [26, 27, 46], which hides the memory access pattern by encrypting the data addresses. With ORAM, attackers cannot identify two operations even when they are accessing the same physical address [46]. However, ORAM techniques incur a significant memory bandwidth overhead (up to an astonishing 10x), which is impractical for bandwidth-sensitive GPUs. *Dummy Read/Write Operations:* Another potential defence

solution is to introduce fake memory traffic to disturb the statistics of memory events. Unfortunately the noises exert only a small degradation of the layer sequence prediction accuracy, as illustrated in Section 6.2.4. As such, fake RAW operations to obfuscate the layer dependencies identification may be a more fruitful defensive technique to explore.

System Methodologies. The essence of our work is to learn the compilation and scheduling graphs of the system stack. Although the computational graphs go through multiple levels of the system stack, we demonstrate that it is still possible to recover the original computational graph based on the raw information stolen from the hardware. At the system level one could: 1) customize the overall NN system stack with TVM, which is able to implement the graph level optimization for the operations and the data layout [8]. The internal optimization possibly increases the difficulty for the attackers to learn the scheduling and compilation graph, or 2) make security-oriented scheduling between different batches during the front-end graph optimization. Although such optimizations may have negative impact on performance, they may obfuscate the adversary a view of kernel information.

8 Related Work

Exploring machine learning security issues is an important research direction with the industrialization of DNNs techniques. The related existing work mainly comes from the following two aspects.

Algorithm perspective: Adversarial attacking is one of the most important attack model which generates the adversarial examples with invisible perturbation to confuse the victim model for wrong decision. These adversarial examples can produce either the targeted [4, 7, 9, 11, 14, 33, 43, 51], or untargeted [25, 29–31, 47] output for further malicious actions. The state-of-art transfer-based adversarial attacks observe that adversarial examples transfer better if the substitute and victim model are in the same network architecture family [28, 44]. Therefore, the extracting inner network structure is important for attacking effectiveness. Consequently, model extraction work are emerged to explore the model characteristics. Previous work steal the parameter and hyperparameter of DNN models with the basic knowledge of NN architecture [55, 57]. *Seong et al.* explore the internal information of the victim model based on meta-learning [44].

Hardware perspective: Several accelerator-based attacks are proposed, either aiming to conduct model extraction [18] or input inversion [59]. However, their methodologies rely on the specific design features in hardware platforms and cannot be generally applicable to GPU platforms with full system stack. Some studies explore the information leakage in general purpose platforms. CathyTelepathy [62] explores side-channel techniques in caches to reduce the hyperparameter space of victim DNN models by inferring the configurations of GEMM operations. *Naghibijouybari et al.* show that side-channel effect in GPU platform can reveal the neuron numbers [32]. However, no direct evidence shows that how these statistics are useful to the attacking effectiveness. Targeting at the security in the edge (e.g.automotive), this work is the FIRST to propose the DNN model extraction framework and experimentally conduct an end-to-end attack on an off-the-shelf GPU platform immune to full system stack noises.

9 Conclusion

The widespread use of neural network-based applications raises stronger and stronger incentive for attackers to extract the neural network architectures of DNN models. In observing the limitations of previous work, we propose a robust learning-based methodology to extract the DNN architecture. Through the acquisition of memory access events from bus snooping, layer sequence identification by the LSTM-CTC model, layer topology connection according to the memory access pattern, and layer dimension estimation under data volume constraints, we demonstrate one can accurately recover a similar network architecture as the attack starting point. These reconstructed neural network architectures present significant increase in attack success rates, which demonstrate the importance of establishing secure DNN system stack.

Acknowledgments

We thank all anonymous reviewers for their valuable feedback. Thank Kaisheng Ma and Song Han for their comments and suggestions. This work was supported in part by NSF 1725447, 1730309, 1925717 and CRISP, one of six centers in JUMP, a SRC program sponsored by DARPA.

References

[1] 2019. HMTT: Hybrid Memory Trace Toolkit. http://asg.ict.ac.cn/hmtt/
[2] Naveed Akhtar and Ajmal Mian. 2018. Threat of Adversarial Attacks on Deep Learning in Computer Vision: A Survey. *CoRR* abs/1801.00553 (2018). arXiv:1801.00553
[3] Scott Alfeld, Xiaojin Zhu, and Paul Barford. 2016. Data Poisoning Attacks Against Autoregressive Models *(AAAI'16)*. AAAI Press, 1452–1458.
[4] Shumeet Baluja and Ian Fischer. 2017. Adversarial transformation networks: Learning to generate adversarial examples. *arXiv preprint arXiv:1703.09387* (2017).
[5] Erik-Oliver Blass and William Robertson. 2012. TRESOR-HUNT: Attacking CPU-bound Encryption *(ACSAC '12)*. ACM, New York, NY, USA, 71–78.
[6] Robert Callan, Alenka Zajić, and Milos Prvulovic. 2014. A Practical Methodology for Measuring the Side-Channel Signal Available to the Attacker for Instruction-Level Events *(MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 242–254.
[7] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 39–57.

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. *CoRR* abs/1802.04799 (2018). arXiv:1802.04799

[9] Moustapha Cisse, Yossi Adi, Natalia Neverova, and Joseph Keshet. 2017. Houdini: Fooling deep structured prediction models. *arXiv preprint arXiv:1707.05373* (2017).

[10] Ronan Collobert and Jason Weston. 2008. A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning *(ICML '08)*. ACM, New York, NY, USA, 160–167.

[11] Swagatam Das and Ponnuthurai Nagaratnam Suganthan. 2011. Differential evolution: a survey of the state-of-the-art. *IEEE transactions on evolutionary computation* 15, 1 (2011), 4–31.

[12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2009.* IEEE, 248–255.

[13] M. Dey, A. Nazari, A. Zajic, and M. Prvulovic. 2018. EMPROF: Memory Profiling Via EM-Emanation in IoT and Hand-Held Devices. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 881–893.

[14] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and harnessing adversarial examples. *Proceedings of the International Conference on Learning Representations* (2015).

[15] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. 2006. Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks *(ICML '06)*. ACM, New York, NY, USA, 369–376.

[16] Alex Graves and Navdeep Jaitly. 2014. Towards End-to-end Speech Recognition with Recurrent Neural Networks *(ICML'14)*. II–1764–1772.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[18] Weizhe Hua, Zhiru Zhang, and G. Edward Suh. 2018. Reverse Engineering Convolutional Neural Networks Through Side-channel Information Leaks *(DAC '18)*. ACM, New York, NY, USA, 4:1–4:6.

[19] Andrew Huang. 2003. Keeping Secrets in Hardware: The Microsoft Xbox Case Study. In *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '02)*. Springer-Verlag, London, UK, UK, 213–227.

[20] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. [n.d.]. Densely connected convolutional networks. In *CVPR 2017*.

[21] Yongbing Huang, Licheng Chen, Zehan Cui, Yuan Ruan, Yungang Bao, Mingyu Chen, and Ninghui Sun. 2014. HMTT: A Hybrid Hardware/Software Tracing System for Bridging the DRAM Access Trace's Semantic Gap. *ACM Trans. Archit. Code Optim.* 11, 1, Article 7 (Feb. 2014), 25 pages.

[22] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. In *Advances in neural information processing systems*. 4107–4115.

[23] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360* (2016).

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks *(NIPS'12)*. Curran Associates Inc., USA, 1097–1105.

[25] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. 2016. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533* (2016).

[26] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. 2015. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation *(ASPLOS '15)*. ACM, New York, NY, USA, 87–101.

[27] Chang Liu, Michael Hicks, and Elaine Shi. 2013. Memory Trace Oblivious Program Execution. In *Proceedings of the 2013 IEEE 26th Computer Security Foundations Symposium (CSF '13)*. IEEE Computer Society, Washington, DC, USA, 51–65.

[28] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. 2017. Delving into Transferable Adversarial Examples and Black-box Attacks. *ICLR* abs/1611.02770 (2017). arXiv:1611.02770

[29] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. 2017. Universal adversarial perturbations. *arXiv preprint* (2017).

[30] Seyed Mohsen Moosavi Dezfooli, Alhussein Fawzi, and Pascal Frossard. 2016. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[31] Konda Reddy Mopuri, Utsav Garg, and R Venkatesh Babu. 2017. Fast feature fool: A data independent approach to universal adversarial perturbations. *arXiv preprint arXiv:1707.05572* (2017).

[32] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. 2018. Rendered Insecure: GPU Side Channel Attacks Are Practical *(CCS '18)*. ACM, New York, NY, USA, 2139–2153.

[33] Papernot Nicolas, D. McDaniel Patrick, Jha Somesh, Fredrikson Matt, Celik Z. Berkay, and Swami Ananthram. 2015. The Limitations of Deep Learning in Adversarial Settings. *CoRR* abs/1511.07528 (2015). arXiv:1511.07528

[34] Nvidia. [n.d.]. CUDA toolkit documentation. http://docs.nvidia.com/cuda/profiler-users-guide/index.html

[35] NVIDIA. 2016. NVIDIA Tesla K40 Active GPU Accelerator. http://www.pny.com/nvidia-tesla-k40-active-gpu-accelerator.

[36] Nvidia. 2017. NVIDIA cuDNN GPU Accelerated Deep Learning. https://developer.nvidia.com/cudnn

[37] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. 2017. Practical Black-Box Attacks Against Machine Learning *(ASIA CCS '17)*. ACM, New York, NY, USA, 506–519.

[38] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael Wellman. 2016. Towards the science of security and privacy in machine learning. *arXiv preprint arXiv:1611.03814* (2016).

[39] Nicolas Papernot, Patrick D. McDaniel, and Ian J. Goodfellow. 2016. Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples. *CoRR* abs/1605.07277 (2016). arXiv:1605.07277 http://arxiv.org/abs/1605.07277

[40] PyTorch. [n.d.]. Pytorch Tutorials. http://pytorch.org/tutorials/

[41] Tractica Report. 2016. Artificial Intelligience Market Forecasts.

[42] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler. 2018. Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 78–91.

[43] Sayantan Sarkar, Ankan Bansal, Upal Mahbub, and Rama Chellappa. 2017. UPSET and ANGRI: Breaking High Performance Image Classifiers. *arXiv preprint arXiv:1707.01159* (2017).

[44] Bernt Schiele Mario Fritz Seong Joon Oh, Max Augustin. 2018. Towards Reverse-Engineering Black-Box Neural Networks. *ICLR* abs/1605.07277 (2018). https://arxiv.org/abs/1711.01768

[45] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2014). arXiv:1409.1556 http://arxiv.org/abs/1409.1556

[46] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: An Extremely Simple Oblivious RAM Protocol *(CCS '13)*. ACM, New York, NY, USA, 299–310.

[47] Jiawei Su, Danilo Vasconcellos Vargas, and Sakurai Kouichi. 2017. One pixel attack for fooling deep neural networks. *arXiv preprint arXiv:1710.08864* (2017).

[48] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 3104–3112.

[49] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. 2017. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning.. In *AAAI*. 4278–4284.

[50] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2015. Rethinking the Inception Architecture for Computer Vision. *CoRR* abs/1512.00567 (2015). arXiv:1512.00567

[51] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199* (2013).

[52] Jin-Hua Tao, Zi-Dong Du, Qi Guo, Hui-Ying Lan, Lei Zhang, Sheng-Yuan Zhou, Cong Liu, Hai-Feng Liu, Shan Tang, and Allen Rush. 2017. BENCHIP: Benchmarking Intelligence Processors. *arXiv preprint arXiv:1710.08315* (2017).

[53] TechCrunch. 2017. Nvidia is powering the world's first level 3 self-driving production car.

[54] TensorFlow. [n.d.]. Post-training quantization. https://www.tensorflow.org/lite/performance/post_training_quantization.

[55] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2016. Stealing Machine Learning Models via Prediction APIs *(SEC'16)*. USENIX Association, Berkeley, CA, USA, 601–618.

[56] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. 6000–6010.

[57] Binghui Wang and Neil Zhenqiang Gong. 2018. Stealing Hyperparameters in Machine Learning. *CoRR* abs/1802.05351 (2018). arXiv:1802.05351

[58] Waymo. 2017. Introducing Waymo's suite of custom-build, self-driving hardware. https://medium.com/waymo/introducing-waymos-suite-of-custom-built-self-driving-hardware-c47d1714563/

[59] Lingxiao Wei, Yannan Liu, Bo Luo, Yu Li, and Qiang Xu. 2018. I Know What You See: Power Side-Channel Attack on Convolutional Neural Network Accelerators. *CoRR* abs/1803.05847 (2018). arXiv:1803.05847

[60] Nicholas Wilt. 2013. *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education.

[61] Wayne Xiong, Jasha Droppo, Xuedong Huang, Frank Seide, Mike Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig. 2017. The Microsoft 2016 conversational speech recognition system. In *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*. IEEE, 5255–5259.

[62] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. 2018. Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures. *CoRR* abs/1808.04761 (2018). arXiv:1808.04761

[63] Xingcheng Zhang, Zhizhong Li, Chen Change Loy, and Dahua Lin. 2016. PolyNet: A Pursuit of Structural Diversity in Very Deep Networks. *CoRR* abs/1611.05725 (2016). arXiv:1611.05725

[64] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2017. Learning Transferable Architectures for Scalable Image Recognition. *CoRR* abs/1707.07012 (2017). arXiv:1707.07012