NeuroScope: Reverse Engineering Deep Neural Network on Edge Devices using Dynamic Analysis

Ruoyu Wu[1], Muqi Zou[1], Arslan Khan[2], Taegyu Kim[2],
Dongyan Xu[1], Dave (Jing) Tian[1], and Antonio Bianchi[1]

[1]Purdue University, [2] The Pennsylvania State University
{wu1377, zou116, dxu, daveti, antoniob}@purdue.edu, {arslankhan, tgkim}@psu.edu

Abstract

The usage of Deep Neural Network (DNN) models in edge devices (e.g., IoT devices) has surged. In this usage scenario, the inference phase of the DNN model is executed by a dedicated, compiled piece of code (i.e., a DNN binary). From the security standpoint, the ability to reverse engineer such binaries (i.e., recovering the original, high-level representation of the implemented DNN) enables several applications, such as stealing DNN models, gray/white-box adversarial machine learning attacks and defenses, and backdoor detection. While a few recent works proposed dedicated approaches to reverse engineer DNN binaries, these approaches are fundamentally limited in the type of DNN binaries they support.

To address these limitations, in this paper, we propose NeuroScope, a novel data-driven approach based on dynamic analysis and machine learning to reverse engineer DNN binaries. This compiler-independent and code-feature-free approach enables NeuroScope to support a larger variety of DNN binaries across different DNN compilers and hardware platforms, including binaries implementing DNN models using an interpreter-based approach. We demonstrate NeuroScope's capability by using it to reverse engineer DNN binaries unsupported by previous approaches with high accuracy. Moreover, we showcase how NeuroScope can reverse engineer a proprietary DNN binary compiled with a closed-source compiler and enable gray-box adversarial machine learning attacks.

1 Introduction

In recent years, there has been a surge in the usage of Deep Neural Network (DNN) models to perform a variety of tasks, such as image recognition, natural language processing, and autonomous driving. While some of these models are deployed on high-end GPU servers (and are accessible online), others are deployed locally on edge devices, such as IoT devices and other devices [2,51] in consideration of low latency, high availability, and better privacy. An example of this scenario is a smart-camera performing on-device face recognition [8]. In this case, the inference phase of a DNN is executed by a dedicated, compiled piece of code, which we call a *DNN binary*.

From the security standpoint, the ability to reverse engineer such binaries (i.e., recovering the original, high-level representation of the implemented DNN) enables stealing DNN models, which are considered valuable intellectual properties [62]. Additionally, it also facilitates several downstream applications, including gray/white-box adversarial machine learning attacks and defenses [41,46], backdoor detection [30], or binary patching [63]. However, traditional, general-purpose decompilers [32,45] are not suitable for this task, since they can only recover a C-like representation of the code [38,64,66,72].

Recent works proposed dedicated approaches to reverse engineering DNN binaries to tackle this issue. Specifically, DND [66] uses symbolic execution to reverse engineer DNN binaries generated by dedicated DNN compilers. Additionally, BTD [38] and NNReverse [19] first use dedicated machine learning classifiers to identify the types of DNN operators implemented in the DNN binary and then recover the high-level representation of the implemented DNN using strategies specific to certain DNN compilers.

Besides compiler dependency, the existing approaches target a specific type of DNN binaries in which each DNN operator is implemented as a standalone function (i.e., DNN binaries compiled using the *Ahead of Time* approach). For this reason, they cannot handle DNN binaries that utilize the *interpreter-based* approach, where a runtime interprets a data-only, proprietary representation of a DNN. DNN binary code does not reflect the structure of implemented DNNs by design. Hence, the existing approaches, which rely on code features (either extracted by symbolic execution or machine learning techniques), cannot extract a high-level representation of the DNNs from these binaries. Reliance on specific code features also forces current approaches to retrain the machine learning models or develop additional compiler-specific features for each DNN compiler, CPU architecture, or even compiler version. Similarly, these approaches are unsuitable for reverse engineering DNN binaries leveraging hardware accelerators, such as NPUs, because the internals of these accelerators and the code they execute cannot be directly observed.

To address these limitations, we propose NeuroScope, a novel data-driven approach based on dynamic analysis to reverse engineer DNN binaries. This approach enables NeuroScope to support a variety of DNN binaries, including binaries implementing DNN models using the interpreter-based approach. Our key observation is that, although DNNs may be implemented differently by various compilers on different hardware platforms, the mathematical semantics of the DNN operators (the basic building blocks of a DNN) remain the same. Therefore, by examining the data that different operators receive and output, we can infer the semantics of operators in an SDK-independent and hardware-independent fashion.

Accordingly, NeuroScope first uses dynamic analysis to identify the DNN operators implemented in the binary, and then captures pairs of input/output data, i.e., pairs of data being processed by and returned from each DNN operator. Then, it uses dedicated machine learning models to infer the DNN semantics from input/output pairs enriched with statistical features extracted from the input/output pairs. To acquire the dataset for training such models, we propose a generic data synthesizer that synthesizes input/output pairs of each DNN operator type.

NeuroScope successfully recovers five different DNN models (LeNet-5 [69], ResNet-18 [31], MobileNet v2 [52], Char-RNN [1], and LSTM-MNIST [10]) used in three different edge devices (NXP i.MX RT1050, TI SK-TDA4VM, and NXP i.MX 8M Plus). In addition, we demonstrate how we use NeuroScope to reverse engineer a proprietary DNN binary compiled with a closed-source compiler and showcase how NeuroScope's capability can be used to perform more powerful gray-box adversarial machine learning attacks instead of black-box ones.

In summary, these are the contributions of this paper:

1. We propose NeuroScope, a compiler-independent and hardware-independent DNN binary reverse engineering framework. NeuroScope adopts a data-driven approach based on dynamic analysis to bypass limitations affecting existing DNN reverse engineering tools.

2. We propose using machine learning models to infer the semantics of DNN operators from their input and output, and we develop a generic data synthesizer that synthesizes the dataset for training such models.

3. We show how NeuroScope can successfully recover the high-level representation of three different DNN models deployed with three different SDKs and on three different hardware platforms. Additionally, we showcase how this capability can be used to reverse engineer a proprietary DNN binary. The results can be used to boost adversarial machine learning attacks by enabling gray-box attacks in place of black-box ones.

NeuroScope's source code is available via Github at https://github.com/purseclab/NeuroScope.

2 Background

2.1 Deep Neural Network

Deep Neural Networks (DNNs) represent a category of machine learning algorithms that utilize a series of cascaded DNN operators to extract and transform features. DNN operators, including convolution, pooling, and activation, serve as the fundamental building blocks of DNNs. Each DNN operator receives the output of preceding operators as its input (or the initial input to the DNN model in the absence of prior operators) and calculates its output based on its operator type and attributes. DNN operator attributes are the properties of the operator that define the operator's behavior, such as the kernel size and stride of a convolution operator, and the number of output channels of a fully connected operator. The input and output of each DNN operator are tensors, which represent multidimensional data.

DNN Architecture. DNN architecture refers to the structure of nodes that constitute a DNN, which is represented as a directed computation graph, where each node, as a DNN operator, performs a specific mathematical operation on its inputs. Specifically, the DNN architecture includes the number of operators, the type/attributes of each operator, and the topology among operators. The DNN architecture is crucial because it influences the network's ability to learn and generalize from data. Developing DNN model architectures usually demands significant human effort and extensive computing resources. Consequently, the architecture itself has evolved into intellectual property and a primary target for attackers [27, 68]. Furthermore, knowledge of a DNN's architecture can serve as a basis for subsequent attacks, including the model weights stealing [49], adversarial attacks [18], membership inference [17, 56], and data reconstruction [14].

2.2 DNN Inference on Edge Devices

To help developers deploy DNN on edge devices, vendors usually provide their specific (and usually proprietary) software development kits (SDKs), such as NXP eIQ ML software development environment [54] and Texas Instruments Edge AI Studio [12], which take a DNN model as input and generate a binary that can be executed on vendors' hardware. There are two primary approaches to generate such a binary: *ahead-of-time (AOT)* approach and *interpreter-based* approach [66].

For AOT, each DNN operator in a DNN model is compiled into a separate function, where an AOT compiler specializes the function for the specific operator attributes. For instance, two convolution operators with different operator attributes (e.g., kernel size), though sharing the same operator type, will be compiled into two different functions.

On the contrary, the interpreter-based approach generates a binary that contains a machine learning runtime library and a DNN configuration file, which is often encoded in a proprietary format by SDKs, such as the format used by NXP eIQ DeepViewRT inference [4]. To execute the DNN, the runtime first loads and parses the DNN configuration file into a computation graph, and then interprets the computation graph to perform the DNN inference. Due to the portability and flexibility, the interpreter-based approach is widely adopted, especially when deploying DNN on edge devices with hardware accelerators, such as Tensorflow Lite for Microcontrollers (TFLM) [22], Edge Impulse [3], and DeepViewRT [4], Though implemented differently by different vendors, the inference workflow is similar, as shown in Listing 1.

```
void main(){
  // Load DNN model.
  model = load_model();
  // Determine backends (e.g., CPU, accelerators).
  model->set_backends(get_backends());
  // Partition into subgraphs according to backends.
  model->partition_graph();
  // Perform inference.
  while (data = get_data()) {
    model->set_input(data);
    inference(model);
    output = model->get_output();
  }
}

void inference(model){
  // Invoke subgraph in topological order.
  for (subgraph : model->subgraphs()) {
    if (subgraph->backend() == CPU) {
      for (op : subgraph.operators()) {
        op->invoke();
      }
    } else {
      subgraph->invoke_at_accelerator();
    }
  }
}
```

Listing 1: Pseudocode of simplified DNN inference workflow.

The workflow can be summarized as follows: ❶ The runtime first loads the DNN model. ❷ The runtime then determines which hardware backends are used for DNN inference, such as neural processing unit (NPU) and digital signal processor (DSP). Hardware accelerators usually only support a subset of DNN operators which are computationally intensive, such as convolution and matrix multiplication [35]. For other operators, the runtime needs to fall back to the CPU backend for operator execution [9], which is one of the reasons why the CPU backend code is usually included in the binary even when hardware accelerators are available. ❸ The runtime then partitions the computation graph into multiple subgraphs, each of which contains only the operators that are supported by the same backend. ❹ With the subgraphs partitioned, the runtime feeds the input data to the DNN model and iterates over the subgraphs in the topological order. For subgraphs executed on the hardware accelerators, the runtime transfers them to the hardware accelerator, and they are executed on the hardware accelerator as a whole. In other words, a subgraph executed on the hardware accelerator is a black box where we can only observe its input and output, instead of the intermediate states of the operators in the subgraph. On the other hand, for subgraphs executed on the CPU, each operator in the subgraph is invoked sequentially, leaving the intermediate states of the operators in the subgraph observable.

Note that, for DNN binaries generated with the interpreter-based approach, each *type* of the DNN operator has a generic implementation. For instance, if a DNN contains multiple `Convolution` operators, the same `Convolution` operator implementation will be invoked multiple times, with different inputs, operator attributes, and parameters.

3 Motivation

Deploying DNNs on edge devices is becoming prevalent to enable intelligence features, such as the usage of face recognition in smart cameras [8]. Since the firmware on edge devices are, in general, accessible to the end users, the firmware implementing DNN models are vulnerable to reverse engineering attacks [19, 38, 66, 70], which recover the DNN model from the binaries with static or dynamic binary analysis.

A few recent works have explored this topic. Specifically, DnD [66] assumes certain code patterns (e.g., nested loop structures) and data structure layouts exhibited by its target DNN compilers (i.e., glow [43] and TVM [20]), and develops pattern matching heuristics that are specifically designed for the target DNN compilers to recover the semantics. BTD [38] and NNReverse [19] operate in three steps. First, they recover the DNN operator types using binary function similarity, i.e., they train NLP models on the assembly code generated by a DNN compiler. Then, they use the trained models to classify the operator type from the assembly code. Finally, they recover the operator attributes and parameters with compiler-specific heuristics. We summarize the previous works in Table 1 and compare them against NEUROSCOPE.

Table 1: NEUROSCOPE and representative works for DNN binary reverse engineering.

Works	Features	Compiler Scheme	Targeted Hardware
DnD [66]	Binary Code	AOT	CPU
BTD [38]	Binary Code	AOT	x86 CPU
NNReverse [19]	Binary Code	AOT	CPU
NEUROSCOPE	I/O Behavior	Interpreter/AOT	CPU with accelerator

Features. The existing approaches use binary code as features to recover the DNN model. Specifically, DnD and BTD rely on compiler-specific code patterns and data structure layouts to develop compiler-specific heuristics to recover the DNN model. BTD and NNReverse train an NLP model on the binary code generated by the target DNN compiler to classify the operator types within the binary generated by the same DNN compiler. BTD even requires training dedicated models for different versions and different optimization levels of the same DNN compiler [38]. As mentioned in Section 2.2, since DNNs on edge devices are deployed with a diverse set of vendor-specific, proprietary, and frequently updated DNN compilers, it is challenging to develop and maintain the heuristics/models for each DNN compiler configuration.

Compiler Scheme. Existing works focus on DNN binaries generated by AOT-based DNN compilers. Those DNN binaries are standalone binaries and can only run fixed DNNs. However, many DNN binaries on edge devices are generated by interpreter-based DNN compilers [3, 4, 29], which encodes a DNN into a configuration file in a proprietary format and loads this DNN from the file only during inference. Reverse engineering from DNNs generated by interpreter-based DNN compilers is challenging for static-analysis-based approaches, such as DnD and NNReverse. Specifically, to execute a DNN, an interpreter reads and parses hyperparameters from its configuration file, constructs a computation graph on the fly, and iterates through the graph to invoke the corresponding operator functions. Unfortunately, the dynamic nature and multiple levels of abstractions of DNN binaries compiled with the interpreter-based approach make static reverse engineering difficult. For instance, to determine operator types, static-analysis-based approaches must identify semantics and dependencies between fields in a configuration file and their corresponding operator function invocations. This process requires precise binary-level interprocedural analysis, which is not supported by previous static-analysis-based approaches [66]. Note that BTD can handle DNN binaries produced only by AOT-based DNN compilers, although it employs dynamic binary analysis built upon Intel Pin.

Targeted Hardware. Existing works only support DNN binaries that are executed on a CPU. For example, BTD only supports the binaries on an x86 CPU, while many DNNs on edge devices run on hardware accelerators whenever possible.

The goal of this work is to develop a framework that addresses the aforementioned limitations and complements the capabilities of the aforementioned state-of-the-art works. In the following paragraphs, we describe the challenges in fulfilling the aforementioned goals and provide an overview of how NEUROSCOPE addresses them.

Challenge 1: Supporting diverse and proprietary DNN SDKs and hardware platforms. DNNs are compiled and deployed into different hardware platforms, with different vendor-specific SDKs. To a certain extent, the existing works "overfit" target compilers, and it is challenging to have a generic and easily extensible solution that recovers the DNN semantics by analyzing the binary code due to variations of DNN compiler semantics.

Solution 1: Our key observation is that, although DNNs may be implemented differently by various compilers on different hardware, the mathematical semantics of DNN operators, the basic building blocks of a DNN, remain identical. More importantly, the mathematical semantics can be inferred by examining their input-output behaviors, which are usually SDK- and hardware-independent compared with examining their binary code. For instance, given the same input, the output of a `Convolution` operator should exhibit similar patterns across different DNN SDKs and hardware platforms regardless of how they are implemented. To capture the input-output behaviors, we develop a dynamic analysis framework to automatically locate the functions that implement each DNN operator, identify the memory buffers that hold input/output tensors, and record each input/output pairs as a pair of one-dimensional arrays (Section 6.1).

Challenge 2: Inferring semantics from input/output pairs with unknown tensor shape, operator attributes, and parameters. As mentioned earlier, we propose to infer the semantics of DNN operators by examining their input-output behaviors. However, inferring the semantics from input/output pairs requires reasoning about the numeric relationship between two sequences. Furthermore, we simply cannot enumerate all the possible DNN operators on the input tensor and select the one that matches the output tensor because of the unknown shape of the input/output tensors we acquire from memory and the unknown operator attributes and parameters. For instance, an array we acquire from memory containing the values $[1,2,3,4,5,6,7,8]$ may represent a tensor of shape $(2,2,2)$ (i.e., $[[[1,2],[3,4]],[[5,6],[7,8]]]$), or a tensor of shape $(4,2)$ (i.e., $[[1,2],[3,4],[5,6],[7,8]]$). Additionally, even if we know the shape of the input/output tensors, the operator attributes and parameters are still unknown and encompass a wide range of possibilities, making it infeasible to enumerate all the possibilities.

Solution 2: We propose using machine learning to infer the DNN semantics from input/output pairs. Specifically, we utilize the Seq2Seq model [58] to encode the input/output pairs into a vector and enrich the encoded vector with statistical features we extract from the input/output pairs (e.g., lengths and averages) to recover the operator type and attributes (Section 5.2).

To acquire the dataset for training such a model, we propose a data synthesizer that synthesizes input/output pairs of each supported DNN operator type (Section 5.1). To ensure the synthesized input/output pairs are valid, the data synthesizer considers the constraints of each operator type (e.g., for `Convolution` operator, the kernel size should be smaller than the input width/height).

Challenge 3: Part of the DNN may be executed using hardware accelerators. As described in Section 2.2, the runtime on the edge device partitions the DNN computation graph into multiple subgraphs, each of which usually contains multiple operators that are supported by the same backend. Some subgraphs of the DNN may be executed using hardware accelerators available on the edge device. The internal function of these accelerators is a black box, and we can only observe the input and output of the whole subgraph, instead of the intermediate states of each operator, preventing us from inferring the semantics of each individual DNN operator executed by a hardware accelerator.

Solution 3: For flexibility and portability considerations, current DNN inference runtimes usually include the CPU backend code for all the operators in the binary even when hardware accelerators are available [9, 22]. In this way, when hardware accelerators are not available, the runtime can robustly fall back to the CPU backend for subgraph execution, where we can observe the input/output of each operator in the subgraph. Leveraging this observation, we use various approaches to prevent the runtime from using the hardware accelerators, forcing it to fall back to the CPU backend for subgraph execution, and then use the aforementioned techniques to infer the semantics of each operator in the subgraph (Section 6.1.1). Note that this approach relies on a DNN runtime's capability to switch its execution from a hardware accelerator to a CPU (i.e., CPU fallback) because we cannot directly reverse-engineer a DNN running on a hardware accelerator. We discuss the limitations of this approach and potential solutions in Section 10.

4 NEUROSCOPE Overview

NEUROSCOPE supports recovering DNN architecture (i.e., operator types, attributes, and topology) from DNN binaries and outputs the recovered architecture in ONNX format [25].

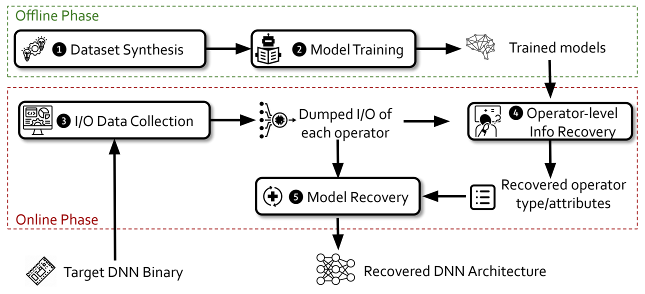

Figure 1: An overview of NEUROSCOPE pipeline.

Figure 1 shows an overview of NEUROSCOPE, which consists of two phases: an offline phase (Section 5) and an online phase (Section 6).

During the offline phase, given a set of input/output pairs, NEUROSCOPE trains a classification neural network, M_T, to recover the operator type, and a set of dedicated regression neural networks, M_{Att}, to recover the operator attributes. To this aim, as the first step, NEUROSCOPE synthesizes a training dataset consisting of numerous input/output pairs of each supported DNN operator (Step ❶in Section 5.1). Then, NEUROSCOPE trains M_T and M_{Att} on the synthesized training dataset. Later on, these trained models can, given input/output pairs of an operator, recover its type and attributes, respectively (Step ❷in Section 5.2).

In the online phase, NEUROSCOPE automatically reverse-engineers the DNN architecture from the target DNN binary using the collected input/output data and the trained neural networks. Specifically, NEUROSCOPE first locates where the mathematical operations for different DNN operators happen in the binary and the input/output buffers for each operator. Then, it dumps the data in the input/output buffers (Step ❸in Section 6.1). Later, NEUROSCOPE recovers the operator-level information (i.e., operator type and attributes) of each DNN operator from the dumped input/output data using the M_T and M_{Att} (Step ❹in Section 6.2) trained during the offline phases. Finally, NEUROSCOPE recovers the DNN topology i.e., the interconnections of the different DNN operators, by identifying data dependencies between their input/output buffers. As a last step, by combining the topology information with the already recovered operator-level information, NEUROSCOPE recovers the complete DNN architecture (Step ❺in Section 6.3).

Note that the offline phase is conducted only once to train M_T and M_{Att}, and the online phase is conducted for each run of the DNN binaries (i.e., once per input).

5 NEUROSCOPE Offline Phase

In this section, we describe the offline phase consisting of two steps: (1) dataset synthesis (Section 5.1) and (2) model training (Section 5.2). The offline phase is conducted only once, and the trained neural networks will be used in the online phase to recover the DNN architecture from different DNN binaries on different edge devices.

5.1 Dataset Synthesis

As the first step, NEUROSCOPE synthesizes a dataset consisting of numerous input/output pairs of each supported DNN operator to train M_T and M_{Att}. We chose to synthesize a dataset instead of collecting from existing DNN binaries because we aim at training the generic M_T and M_{Att} to support binaries compiled by different and proprietary DNN SDKs.

To synthesize the dataset, NEUROSCOPE repeatedly chooses a DNN operator type, generates random input tensor shape, operator attributes, and operator parameters, and it computes the corresponding output tensor. Since there are inherent constraints between the input tensor shape and operator attributes, in order to ensure the correctness of the synthesized dataset, NEUROSCOPE considers these constraints when generating the input tensor shape and operator attributes. Taking the convolution operator (`Conv`) as an example, its output shape size can be calculated as $Out = (In - K + 2*P + 1)/S$, where *Out*, *In*, *K*, *P* and *S* denote the output size, input size, kernel size, padding size, and striding size, respectively. Furthermore, *Out* and *In* must be positive integers, while *K*, *P*, *S* must be non-negative integers. Besides the constraints between input tensor shape and operator attributes, operator attributes also have their own valid ranges [27]. For instance, the kernel size (K) of a convolution operator is usually in the range $[1,7]$, while the padding size (P) is usually in the range $[0,(K-1)/2]$ [27]. Table 3 summarizes these operator attributes and their valid ranges.

Given the above constraints, NEUROSCOPE randomly generates the input tensor shapes, operator attributes, and operator parameters satisfying the constraints, and computes the corresponding output tensors. Finally, NEUROSCOPE stores the input/output tensor pairs along with their operator types, attributes, and parameters in the dataset. Note that if a DNN operator has two inputs, we represent them as a single tensor by concatenating them.

5.2 Model Training

In this section, we first introduce the challenges and design choices in modeling the problem of recovering DNN operators as a learning task (Section 5.2.1), and then we introduce the architectures of M_T and M_{Att} that we use to recover the operator type and attributes (Section 5.2.2).

5.2.1 Challenges and Design Choices

We formulate the problem of recovering DNN operators as follows: given an input/output tensor pair $(\vec{I},\vec{O})$, where $\vec{I}$ and $\vec{O}$ are the input and output tensors of a DNN operator, the goal is to recover the operator type and its attributes. We enumerate the challenges in modeling the problem as a learning task as follows.

The first challenge is the inherent complexity of a multi-class classification scenario. If we model recovering the operator type and its attributes using a single classification model, the number of output classes will be large, making the classification hard to train. For instance, `Conv` operator has 5 attributes (i.e., number of input channels, number of kernels, kernel size, padding size, striding size, number of kernels), each of which can take many choices. Assuming that each attribute can have ten integer values, consequently, this leads to 10^5 attribute combinations, solely for a `Conv` operator.

To tackle this challenge, we use a "divide-and-conquer" approach, decomposing the task into predicting the operator type and predicting each attribute separately. Specifically, given an input/output tensor pair of an operator, NEUROSCOPE first recovers its operator type with a classification model (M_T), and then, based on the operator type recovered, recovers each of its attributes using dedicated regression models (M_{Att}). Note that each attribute (e.g., kernel size and padding size) is recovered by a dedicated regression neural network model.

The second challenge is the difficulty of modeling the relationship between input and output tensors of an operator. Unlike conventional sequence classification modeling, where the model receives a single sequence as input, in this scenario, rather than just reasoning one single sequence, the model needs to capture the numerical relationship between two sequences (i.e., input and output tensors of an operator). Some existing work [23,47] applies Sequence to Sequence LSTM neural network (Seq2Seq) [58] with attention mechanism [39] to capture the dependency between two sequences. However, we find that the attention mechanism, such as a transformer architecture, is not suitable for our problem, since the sequences in our scenario are sometimes too long (e.g., the input sequence of a convolution operator can contain $64*224*224 = 3211264$ elements), which makes the attention mechanism computationally expensive. However, without the attention mechanism (as shown in the ablation study in Section 8.1.2), the Seq2Seq alone fails to capture the dependency between two sequences, possibly due to the lack of the ability to capture the long-range dependency between two sequences, leading to low accuracy of the recovered operator type and attributes.

To tackle this challenge, we propose to enhance the Seq2Seq neural network with statistical features extracted from input/output sequences (as shown in Table 2) to accurately capture the characteristics within and between the input and output sequences. For example, we count the number of zeros in the output tensor to distinguish `Conv` and `Conv+Relu` operators. This is because the `Relu` operator, fused in the `Conv+Relu` operator, transforms all negative elements to zeros, resulting in more zeros in the output tensor than the `Conv` operator alone. These many zeros serve as a strong indicator to distinguish `Conv` and `Conv+Relu` operators. We will elaborate on how the extracted statistical features are used in combination with the Seq2Seq neural network in Section 5.2.2.

Table 2: Extracted statistical features. $\vec{I}$, $\vec{O}$, $len(X)$, and $range(X)$ denote the input sequence, output sequences, length of a sequence X, and range of a sequence X, respectively.

Feature Type	Feature
Sequence-level features	Number of zero in $\vec{I}$, $\vec{O}$
	Mean of $\vec{I}$, $\vec{O}$
	Minimum of $\vec{I}$, $\vec{O}$
	Maximum of $\vec{I}$, $\vec{O}$
Inter-sequence features	$len(\vec{I})/len(\vec{O})$
	$range(\vec{I})/range(\vec{O})$

5.2.2 Architectures of M_T and M_{Att}

Figure 2 shows the architectures of M_T and M_{Att}. For both M_T and M_{Att}, the input sequence (*Input*) of an operator is fed into the first LSTM layer (I), and its final hidden state is used as the initial hidden state of the second LSTM layer (O) that processes the output sequence ($Output$) of the same operator. Both the input and output sequences are processed at the tensor element level, where each element is a floating point. Then, the final output state of the second LSTM layer (O), along with each of the extracted statistical features in Section 5.2.1 (F), is used as an input of the two consecutive Fully Connected (FC) layers. For M_T, the last FC layer outputs a vector of the shape $(1, C)$, where C is the number of output classes, and a Softmax layer is applied to convert the output of the last FC layer to a probability distribution over the output classes. On the other hand, for M_{Att}, the last FC layer outputs a floating-point value, which is the predicted value of the attribute.

The design of M_T and M_{Att} is inspired by neural program synthesis techniques [23, 47]. Those techniques use the Seq2Seq neural network to capture dependencies between input and output sequences. However, as shown in the experiments in Section 8.1.2, the Seq2Seq neural network alone is not sufficient to capture dependencies between input and output sequences. To enhance the Seq2Seq neural network, we fuse the extracted statistical features and Seq2Seq neural network with two consecutive FC layers. This combination empirically shows a higher accuracy than that of the Seq2Seq neural network alone. Note that we did not find it beneficial to use an additional FC layer to fuse the statistical features and the output of the Seq2Seq neural network.

We describe the hyperparameters of M_T and M_{Att} and how they are fine-tuned in Section 7.

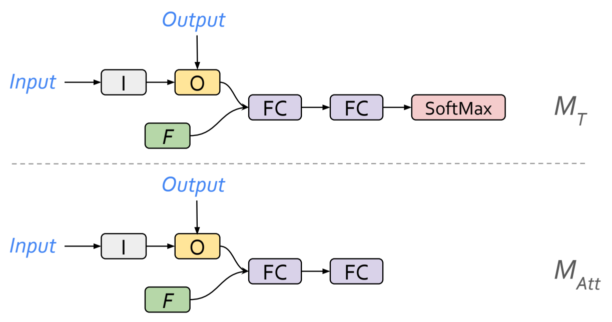

Figure 2: The top and bottom figures show the architectures of M_T and M_{Att}, respectively. They share a similar architecture except that M_T has a Softmax layer at the end.

6 Online Phase

The online phase is conducted on the target DNN binary on the edge device to recover the DNN architecture, which consists of the following steps (as shown in Figure 1): ❸ I/O data collection (Section 6.1), ❹ Operator-level Info Recovery (Section 6.2), and ❺ Model Recovery (Section 6.3).

6.1 I/O Data Collection

In this section, we describe how NeuroScope collects the input/output data of the executed DNN operators, which will be used to recover the operator-level information and the DNN model in the following steps. The I/O data collection consists of the following two steps: (1) *Forcing the DNN runtime to CPU mode* and (2) *Locating DNN operators and I/O buffers*. In the first step, as discussed in Section 3, in order to have observability of the DNN runtime execution at an operator level, NeuroScope forcibly executes the DNN runtime on a CPU by hiding the hardware accelerator from the machine learning runtime (Section 6.1.1). Then, NeuroScope locates the implementations of all the used DNN operators and their corresponding buffers that hold the input and output data (Section 6.1.2).

6.1.1 Forcing the DNN Runtime to CPU Mode

To ensure the observability of the DNN execution at an operator level, NeuroScope forcibly executes the DNN runtime on a CPU. Embedded machine learning runtimes with hardware acceleration support typically use Memory-Mapped I/O (MMIO) to interface with the hardware accelerators, where an accelerator's I/O memory is directly mapped to the normal address space, and a proprietary accelerator library is provided as a shared library to manipulate the memory space. To hide the hardware accelerator from the machine learning runtime, we list the following techniques we used on the edge devices we analyzed:

- *Library Hiding*: We can remove the shared libraries that are used to communicate with the hardware accelerator.
- *Configuration Tampering*: Some machine learning runtimes, such as TIDL [13], use configuration files to specify the hardware accelerator to be used. We can modify such configuration files to disable the usage of the hardware accelerator.

Due to the robustness of the machine learning runtime, we empirically found that the DNN binaries with hardware acceleration support can still run in the CPU mode normally after applying the above techniques. As shown in Section 8, all the SDKs used for evaluation supported running in the CPU mode. Note that our CPU mode enforcement technique may be applicable only to certain edge devices. We discuss the limitations of this CPU fallback approach and potential solutions in Section 10.

6.1.2 Locating DNN Operators and I/O Buffers

In this section, we describe how NeuroScope realizes the operator-level I/O observability by locating the DNN operator implementations and their corresponding buffers that hold the I/O data.

Locating DNN Operator Implementations. As discussed in Section 2.2, unlike the AOT approach that previous works [66] focus on, the DNN binaries compiled with the interpreter-based approach has generic implementations for each type of DNN operator. To locate the DNN operator implementations, we develop dedicated static and dynamic binary analyses. Specifically, NEUROSCOPE first conducts static analysis to locate DNN operator candidate functions. NEUROSCOPE considers functions as DNN operator implementation candidates if they have nested loops whose depth is more than 2 *and* contain vectorized instructions (e.g., ARM NEON instructions) since DNN operators are usually implemented with vectorized instructions with nested loops [66]. This heuristic is based on the observation that DNN operators, such as convolutions and matrix multiplications, inherently involve repetitive computations over multi-dimensional data structures. These operators are typically implemented using multiple nested loops with vectorized instructions.

Next, NEUROSCOPE runs a target DNN binary and sets breakpoints at entry addresses of a subset of candidate functions. During execution, NEUROSCOPE monitors its execution to determine whether those candidate functions are executed. NEUROSCOPE can detect a candidate function execution because an occurrence of a set breakpoint hit indicates the execution of that candidate function. The reason NEUROSCOPE only focuses on a subset of candidate functions is that some edge devices have limitations on the number of breakpoints that can be set at the same time. Since we only focus, at each iteration, on a subset, we need to repeat the process iteratively until NEUROSCOPE covers all candidate functions. All the candidate functions that are executed are considered DNN operator implementations.

Locating I/O Buffers. Once the DNN operator implementations invoked by the target DNN binary are found, NEUROSCOPE locates their I/O data buffers and identifies tensor element data types of data within those buffers. To this end, we leverage the observation that the I/O buffers are frequently accessed by the operator implementations in the nested loops.

Specifically, for each DNN operator implementation, NEUROSCOPE first conducts static binary analysis to identify the entry point of the nested loops and all the memory access instructions (e.g., `ldr` and `str`) within those loop bodies. Then, NEUROSCOPE sets breakpoints at these memory access instructions, resumes execution from the entry point of the nested loops, and it collects the accessed memory addresses until a target loop exits.

To locate the output buffer, NEUROSCOPE uses DBSCAN [24] to cluster the collected memory addresses updated by DNN operators and then identifies the cluster with the largest number of memory addresses as the output buffer. The rationale behind this approach is that the majority of memory addresses written by DNN operators are concentrated within the output buffer, which is typically a contiguous memory region. Similarly, for the input buffers, NEUROSCOPE uses DBSCAN to cluster the collected memory addresses read by DNN operators and then identifies the cluster with the largest number of memory addresses as the input buffer. The difference in locating the input and output buffers is that, for input buffers, there may be two (e.g., `Add` operators have two inputs). If two clusters have an equal number of memory addresses, NEUROSCOPE treats them as separate input buffers. Otherwise, the cluster with the largest number of memory addresses is considered the sole input buffer.

The reason NEUROSCOPE uses DBSCAN to cluster the memory addresses is that the memory addresses of the input/output buffers are usually contiguous, and DBSCAN is suitable for clustering contiguous data points. NEUROSCOPE repeats the above process for each invocation of the DNN operator implementation, since, for some runtime, the I/O buffers are dynamically allocated, and the location of the I/O buffers can be different across different invocations.

Additionally, NEUROSCOPE identifies the tensor element data type (e.g., 32-bit floating point or 8-bit integer) of I/O buffers by analyzing the memory access instructions. For instance, if the memory access instruction is `str` that writes 32-bit data from a floating-point register to memory, NEUROSCOPE considers the atomic data type of the buffer as a 32-bit floating point.

Finally, NEUROSCOPE dumps the data within the identified input/output buffers to be used by *Operator-level Info Recovery* (Section 6.2) and *Model Recovery* (Section 6.3).

6.2 Operator-level Info Recovery

NEUROSCOPE uses the M_T and M_{Att} trained in the offline phase to recover the operator type and attributes of every DNN operator from their corresponding input/output buffer dumps. Specifically, NEUROSCOPE feeds N different inputs to the target DNN binary, and it collects the input/output buffer dumps of each operator, as described in Section 6.1.2. That is, for each operator in the target DNN binary, NEUROSCOPE collects N different input/output buffer dumps. Then, for each operator, NEUROSCOPE feeds these N different input/output buffer dumps separately into the M_T and decides the operator type of each operator based on the majority vote of the predicted operator types. If multiple operator types are predicted with the same highest vote, NEUROSCOPE randomly selects one of them as the recovered operator type. After the operator type is determined, NEUROSCOPE infers the associated attributes of the operator type using the M_{Att}, with the same N input/output buffer dumps and the majority vote strategy as used in the operator type recovery. N is set to 10 in our evaluation, as it empirically achieves good performance.

Table 3 shows the attributes that NEUROSCOPE recovers. As shown in the previous work [27, 38, 66], an operator's attributes can be deduced deterministically from other attributes of the same operator, or can be deduced from the attributes of the predecessor operator. For instance, the number of input channels (C_{in}) of a convolution operator (`Conv`) is equal to the number of kernels (C_{out}) of the predecessor operator. Therefore, NEUROSCOPE only needs to recover some attributes of each operator type, and the other attributes can be inferred from the recovered attributes. We mark the attributes that NEUROSCOPE needs to recover with ✓ in Table 3. To recover these marked attributes, NEUROSCOPE feeds the dumped I/O data of each operator invocation into its associated M_{Att} and uses the rounded predicted value as the recovered attribute.

Table 3: A list of attributes of operators. `Conv` and `FC` denote the convolution and fully-connected operators. The `Pooling` operator includes both `MaxPool` and `AvgPool` since their attributes are identical. `RNN` and `LSTM` denote the recurrent neural network (RNN) operator and long short-term memory (LSTM) operator, respectively. ✓ in the column **P** denotes if the attribute is predicted by a dedicated M_{Att}, the attributes without the ✓ are inferred from the other correlated attributes. The assumed ranges are aligned with the recent works [27,40]

Operator	Attribute	Description	P
Conv	C_{in}	Number of input channels. Value is decided by the input or C_{out}	
	C_{out}	Number of kernels. Its value is 2^n where n is in range [0, 8]	✓
	K	Kernel size. Value is in range [1, 7].	✓
	S	Stride size. Value is in {1, 2}.	✓
	P	Padding size.	✓
FC	F_{in}	Input size. Value is decided by F_{out} or F_{out}	
	F_{out}	Output size.	✓
Pooling	K	Kernel size. Value is in range [1, 7].	✓
	P	Padding size. Value is in {0, 1}.	✓
	S	Stride size. Its value is in {1, 2}	✓
RNN	R_{hidden}	Hidden state size.	✓
LSTM	R_{hidden}	Hidden state size.	✓

6.3 Model Recovery

As the final step, NEUROSCOPE recovers the DNN topology and combines it with the recovered operator-level information to recover the complete DNN model.

Recovering DNN Topology. NEUROSCOPE recovers the DNN topology by examining the data dependencies between the identified I/O buffers of every operator invocation and their invocation order. Specifically, NEUROSCOPE keeps track of addresses of I/O buffers of each operator invocation and connects two operator invocations if the successor invocation's inputs match the predecessor invocation's outputs. Through this process, NEUROSCOPE identifies all the connections within the DNN topology and eventually recovers the DNN topology.

Recovering DNN Model. NEUROSCOPE simply combines the recovered DNN topology with the recovered operator-level information, and outputs a high-level representation of the recovered DNN architecture in the ONNX format [25].

7 Implementation

We implemented NEUROSCOPE in around 3,200 lines of Python code.

Dataset Synthesis and Models. The dataset synthesis and neural network models are implemented in Python using PyTorch. Regarding the dataset synthesis, NEUROSCOPE currently supports 12 DNN operators, including `Add`, `AvgPool`, `Concat`, `Conv`, `Conv+Relu`, `Fully Connected`, `LSTM`, `MaxPool`, `Relu`, `RNN`, `Softmax`, and `Transpose`. As shown in the recent DNN reverse-engineering works [27,68], these operators are the most common operators composing widely deployed DNN models, such as ResNet [31] and MobileNet [33]. We discuss NEUROSCOPE's limitation regarding unsupported operators/architectures (e.g., `Transformer Block` operator in the transformer architecture [61]) and the NEUROSCOPE extension plan as future work in Section 10.

Regarding the hyperparameters of M_T and M_{Att}, we set the hidden size of an LSTM layer to 16, and the output size of the first FC layers to 32. To determine the optimal hyperparameters, we conducted a grid search over a range of sizes: 8, 16, 32, and 64. Each configuration for those two sizes was used to train the models on the training synthesized dataset and evaluated on the validation synthesized dataset. As a result, we selected the configuration with 16 for the hidden size of an LSTM layer and 32 for the first FC layers because it shows the best performance on the validation dataset. We use cross entropy loss and mean squared error loss as the loss functions to train M_T and M_{Att}, respectively, and Adam optimizer to train those models with a learning rate of 0.001. We did not tune the learning rate, as we found that the default learning rate worked well for our models.

Static Analysis. We implement our static analysis to identify the nested loop structures and some specific types of instruction (e.g., vectorized instruction and memory access instructions) using angr [57]. For the bare-metal edge devices, we perform static analysis on the firmware binary, and for the edge devices with an operating system, we perform static analysis on the shared library that contains the machine learning runtime.

Dynamic Analysis. We utilize `ptrace` for Linux-based edge devices to set breakpoints and collect memory access information. For bare-metal edge devices, we utilize SEGGER J-Link [6], a hardware debugger, to enable using GDB. We discuss more details about how we can enable our dynamic analysis in other scenarios in Section 10.

According to the static analysis results, the dynamic analysis sets breakpoints and collects necessary information when a breakpoint is triggered. During the collection of memory access information, the analysis process can be slowed down by the presence of complicated nested loops. To mitigate this issue, we introduce a timeout for the analysis of each operator implementation. If the analysis of an operator implementation exceeds the timeout, we finish the analysis of the current operator implementation. If a timeout occurs, the boundary of the I/O buffer is determined by comparing the memory content before and after the invocation of the operator implementation. We empirically set the timeout as ten minutes, which is sufficient in our experiments.

8 Evaluation

In this section, we evaluate the effectiveness of NEUROSCOPE. First, we describe the synthesized dataset we use to train our neural network models, and we measure the models' performance on the synthesized dataset (Section 8.1). Then, we evaluate the full pipeline of NEUROSCOPE on two real-world edge devices with different vendor-specific SDKs and show the effectiveness of NEUROSCOPE in recovering DNN architectures from the edge devices (Section 9.1). Lastly, we present a case study to demonstrate how to use NEUROSCOPE to reverse engineer a proprietary DNN model shipped with the NXP i.MX 8M Plus board, a System-on-Chip (SoC) designed for edge AI applications, and then launch an adversarial attack by leveraging the recovered DNN architecture (Section 9.2).

8.1 Synthesized Dataset and Evaluation

Here, we first describe the synthesized dataset we use to train our neural network models and how we train M_T and M_{Att} in Section 8.1.1. Then, we evaluate the performance of our trained models on the synthesized dataset in Section 8.1.2.

8.1.1 Synthesized Dataset and Training

We synthesize 100,000 input/output pairs for each operator we support. In total, the synthesized dataset contains 1.2 million pairs of input/output tensors. We split the synthesized dataset into training, validation, and testing datasets with a ratio of 8:1:1. Models were trained on the training dataset and validated on the validation dataset. Models were trained for 30 epochs with batch size 128, and the model with the best overall validation accuracy was selected as the final model. Training for each model took approximately 8 hours, using a single NVIDIA A100-80GB GPU.

8.1.2 Evaluation on Testing Synthesized Dataset

In this section, we evaluate the performance of our trained models on the testing dataset.

9 Confusion matrix of the M_T on the testing synthesized dataset.

Operator type recovery. We show the accuracy of M_T on recovering the operator type on the testing synthesized dataset, and, as an ablation study, the accuracy of M_T without using

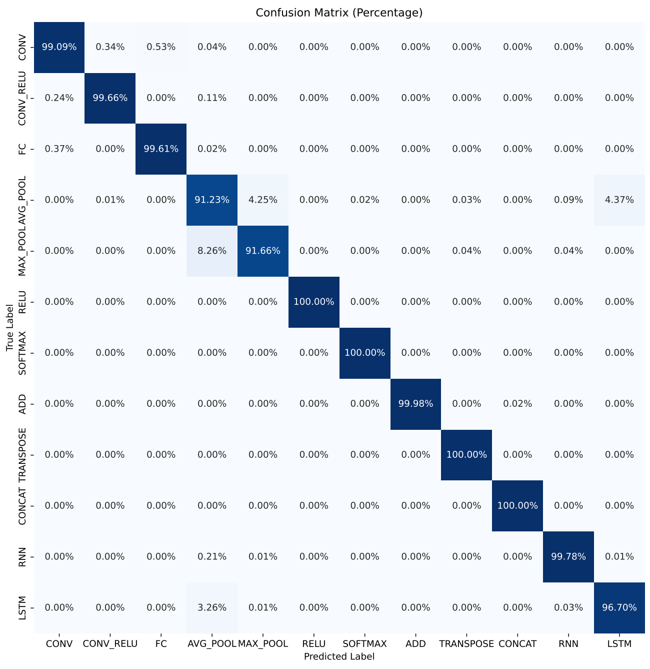

Figure 3: The confusion matrix of the M_T on the testing synthesized dataset.

the statistical features (i.e., only with the Seq2Seq model), in Table 4. As shown in the table, M_T achieves high accuracy in recovering the operator type, demonstrating the effectiveness of M_T. Furthermore, M_T achieves significantly higher accuracy using the statistical features compared to not using them. This difference is more distinct when recovering computationally complex operator types, such as `Conv`, `Conv+Relu`, and `FC` (e.g., 99.09% vs. 82.61% for `Conv`). This indicates the effectiveness of incorporating statistical features in recovering the operator type.

Although the accuracy for `AveragePool` and `MaxPool` (around 91%) is slightly lower than other operator types, our investigation of the confusion matrix (shown in Figure 3) shows these two pooling operators are misclassified (i.e., `AveragePool` is misclassified as `MaxPool` or vice versa) because their functions in DNNs are similar. Additionally, the number of pooling operators in DNNs is usually much smaller than that of other operators, such as `Conv` and `FC`, which makes their misclassification less impactful.

Operator attribute recovery. Since the neural networks that recover the operator attributes are dedicated to each operator type, we first split the testing dataset into subsets based on the operator type and then evaluated the accuracy of each dedicated neural network on the corresponding subset. We consider the attribute recovery accurate if the predicted value after rounding is equal to the ground truth value. As shown in Table 5, the dedicated neural networks achieve high accuracy in recovering the operator attributes.

Table 4: Accuracy of operator type recovery on the testing synthesized dataset. We consider the recovery accurate if the predicted operator type is equal to the ground truth. The **Accuracy** column shows the accuracy of M_T, and the **Accuracy w/o Features** column shows the accuracy of the M_{Att} without using the statistical features.

Operator	Accuracy	Accuracy w/o Features
`Conv`	99.09%	82.61%
`Conv+Relu`	99.66%	86.05%
`FC`	99.61%	88.19%
`AvgPool`	91.23%	76.59%
`Maxpool`	91.58%	71.92%
`Relu`	100.00%	96.22%
`Softmax`	100.00%	99.95%
`Add`	100.00%	99.78%
`Transpose`	100.00%	96.26%
`Concat`	100.00%	98.68%
`RNN`	99.78%	84.96%
`LSTM`	96.70%	83.36%
Overall	98.13%	88.71%

Table 5: Accuracy of operator attributes recovery on the testing synthesized dataset. We consider the recovery results to be accurate if the predicted value is equal to the ground truth. `Conv`, `FC`, `MaxPool`, `AvgPool`, `RNN`, and `LSTM` respectively denote the convolution, fully-connected, max pooling, average pooling, RNN, and LSTM operators.

Operator	Attribute	Recovery Accuracy
`Conv`	Kernel size	99.75%
	Number of kernels	99.97%
	Stride size	99.10%
	Padding size	99.68%
`FC`	Output size	100.00%
`MaxPool`	Kernel size	94.62%
	Padding size	94.34%
	Stride size	97.26%
`AvgPool`	Kernel size	93.51%
	Pad size	94.72%
	Stride size	97.21%
`RNN`	Hidden size	92.56%
`LSTM`	Output size	93.14%

9.1 Evaluation on Edge Devices

In this section, we show our evaluation of NEUROSCOPE on two real-world edge devices and show the effectiveness of NEUROSCOPE in recovering DNN architectures from the edge devices. We describe the target edge devices and DNN models used for our evaluation in Section 9.1.1 and then explain the evaluation setup in Section 9.1.2. We demonstrate the evaluation results in Section 9.1.3.

9.1.1 Evaluation Targets

We choose the two edge devices shown in Table 6 for evaluating NEUROSCOPE.

Table 6: The edge device used for evaluation, their corresponding SDKs for deploying DNNs, their operating system support and accelerator support.

Hardware	**NXP i.MX RT1050**	**TI SK-TDA4VM**
DNN SDK	NXP eIQ TFLM [54]	TI EdgeAI TIDL [13]
Operating System	Bare-metal	Linux-based
Accelerator	No	NPU

Regarding the models used in our evaluation, aligned with prior model extraction attacks [66, 68], we use the LeNet-5 [69], ResNet-18 [31], Char-RNN [1], and LSTM-MNIST [10] models as the target DNN models for our evaluation. We acquire the LeNet-5 and ResNet-18 models from ONNX Model Zoo [26] and trained the Char-RNN and LSTM-MNIST models by following their training instructions [1,10].

To deploy the target models on the target edge devices, for each target model and each target edge device, we write an application that iteratively reads inputs from a file and performs the target DNN model inference by invoking the interfaces provided by the corresponding SDKs, and compile the application into a binary executable. Note that we deploy the Char-RNN and LSTM-MNIST models only on the TI SK-TDA4VM board because the DNN SDK [54] of the NXP i.MX RT1050 board does not support RNN and LSTM operators.

9.1.2 Evaluation Setup

We run the target DNN binary with an input and collect the input/output buffer data of the invoked DNN operator. We repeat this process 1,000 times for each target DNN binary to collect 1,000 input/output buffer dumps for each DNN operator. For the LeNet-5 and the LSTM-MNIST models, the inputs are randomly selected from the MNIST [69] and QMNIST [67] datasets. For the target DNN model of ResNet-18, the inputs are randomly selected from the ImageNet dataset [5] and the Cifar-100 dataset [36]. Regarding the target DNN model of Char-RNN, the inputs are randomly selected from the text data provided by PyTorch [1].

To enable dynamic analysis, for the NXP i.MX RT1050 board, we use a hardware debugger (i.e., SEGGER J-Link PRO debug probe) to enable GDB debugging. On the Linux-based Texas Instruments SK-TDA4VM board, instead, we rely on `ptrace` offered by Linux. Since the dynamic analysis (i.e., identifying the operators and their corresponding input/output buffer) can be prolonged, to facilitate the evaluation, we only perform the dynamic analysis for the first inference of each target DNN binary and use the analysis results for the remaining inferences.

9.1.3 Recovery Accuracy

To evaluate the accuracy of NEUROSCOPE, we evaluate (1) the correctness of the recovered DNN topology, (2) the accuracy of M_T and M_{Att}, and (3) the accuracy of NEUROSCOPE to recover DNN architecture. Note that the accuracy of NEUROSCOPE is different from the accuracy of M_T and M_{Att} since NEUROSCOPE aggregates multiple recovered operator types/attributes with the majority voting to decide the final operator type/attributes as mentioned in Section 6.2.

DNN Topology Recovery. We manually compare the recovered DNN topology (i.e., how different operators are connected with each other) with the ground truth DNN topology of the target DNN models, and we report that, for each target DNN binary, the recovered DNN topology is identical to the ground truth DNN topology.

Accuracy of M_T and M_{Att}. As mentioned in Section 9.1.2, for each target DNN binary, we collect 1,000 input/output buffer dumps for each DNN operator. To evaluate the accuracy of M_T, we feed these 1,000 input/output buffer dumps to the trained M_T, compare the predicted operator types with the ground truth operator types, and we show the accuracy of each operator type recovery and the overall accuracy in Table 7. The overall accuracy is calculated as the number of operators whose types are correctly recovered divided by the total number of operators in the ground truth DNN architecture. Table 7 shows the accuracy of M_T. NEUROSCOPE achieves high accuracy in recovering the operator types for the target DNN binaries on the edge devices. We observe that the accuracy distribution is similar to the synthesized dataset, where the accuracy of `AveragePool` and `MaxPool` are slightly lower than other operator types, indicating that the synthesized dataset is representative of the input/output tensor pair distribution in real-world DNN binaries.

Similarly, we evaluate the accuracy of the operator's attributes recovery, and we show the accuracy of each operator's attributes in Table 8. As shown in the table, NEUROSCOPE also achieves high accuracy in recovering the operator attributes.

Accuracy of NEUROSCOPE. We randomly divide the collected 1,000 input/output buffer dumps into 100 groups each of which contains 10 input/output buffer dumps. For each group, we separately feed the input/output buffer dumps to the trained M_T and M_{Att}, use the majority voting to decide the final operator type/attributes, and recover the DNN architecture. We evaluate the correctness of NEUROSCOPE by comparing these 100 recovered DNN architecture with the ground truth DNN architecture of the target DNN models and show the percentage of the recovered DNN architectures that are identical to their respective ground truth DNN architecture in Table 9. As shown in Table 9, NEUROSCOPE shows high accuracy in recovering the DNN architectures from the target DNN binaries. We investigate the recovered DNN architectures that are not identical to the respective ground truth DNN architectures and find that all the errors are caused by the misclassification between `AvgPool` and `MaxPool` operators.

Table 7: Accuracy of operator type recovery for the target DNN binaries. The denotations of the hardware/model combinations are as follows: **NXP/L** (NXP i.MX RT1050/LeNet-5), **NXP/R** (NXP i.MX RT1050/ResNet-18), **TI/L** (TI TDA4VM/LeNet-5), **TI/R** (TI TDA4VM/ResNet-18), **TI/C** (TI TDA4VM/Char-RNN), and **TI/LS** (TI TDA4VM/LSTM-MNIST). The overall accuracy is calculated as the weighted average of the accuracy of each operator type, based on the number of occurrences of each operator type in the target DNN binaries. N/A denotes that the operator type does not exist in the target DNN model.

Operator	NXP/L	NXP/R	TI/L	TI/R	TI/C	TI/LS
`Conv`	N/A	97.6%	N/A	96.8%	N/A	N/A
`Conv+Relu`	99.9%	100.0%	100.0%	100.0%	N/A	N/A
`FC`	99.6%	99.6%	96.4%	100.0%	N/A	N/A
`AvgPool`	N/A	85.9%	N/A	91.4%	N/A	N/A
`MaxPool`	94.0%	N/A	94.3%	N/A	N/A	N/A
`Relu`	N/A	100.0%	N/A	100.0%	N/A	N/A
`Softmax`	N/A	100.0%	N/A	100.0%	100.0%	100.0%
`Add`	N/A	100.0%	N/A	100.0%	N/A	N/A
`RNN`	N/A	N/A	N/A	N/A	98.84%	N/A
`LSTM`	N/A	N/A	N/A	N/A	N/A	96.26%
Overall	98.6%	98.6%	97.4%	99.0%	98.85%	96.26%

Table 9: Percentage of the recovered DNN architecture that is identical to the ground truth DNN architecture. The denotations of the hardware/model combinations are as follows: **NXP/L** (NXP i.MX RT1050/LeNet-5), **NXP/R** (NXP i.MX RT1050/ResNet-18), **TI/L** (TI TDA4VM/LeNet-5), **TI/R** (TI TDA4VM/ResNet-18), **TI/C** (TI TDA4VM/Char-RNN), and **TI/LS** (TI TDA4VM/LSTM-MNIST).

	NXP/L	NXP/R	TI/L	TI/R	TI/C	TI/LS
Identical percentage	100%	92%	100%	99%	100%	100%

9.2 Case Study

In this section, we demonstrate how we use NEUROSCOPE to recover DNN architectures from a real-world proprietary DNN binary shipped with the NXP i.MX 8M Plus board. Since we do not have access to the ground truth DNN architecture, we cannot directly verify the correctness of the recovered DNN architecture. Nevertheless, we show the effectiveness of NEUROSCOPE in boosting adversarial attacks by leveraging the recovered DNN architecture. Specifically, we first launch adversarial attacks on the target DNN binary, both with and without knowledge of the recovered DNN architecture, using a gray-box adversarial attack [41] and a black-box adversarial attack [46], respectively. Then, we compare the target DNN binary's inference accuracy on the generated

Table 8: Accuracy of operator attributes recovery for the target DNN binaries. The denotations of the hardware/model combinations are as follows: **NXP/L** (NXP i.MX RT1050/LeNet-5), **NXP/R** (NXP i.MX RT1050/ResNet-18), **TI/L** (TI TDA4VM/LeNet-5), **TI/R** (TI TDA4VM/ResNet-18), **TI/C** (TI TDA4VM/Char-RNN), and **TI/LS** (TI TDA4VM/LSTM-MNIST). N/A denotes that the operator type does not exist in the target DNN binaries.

Operator	**Attribute**	**NXP/L**	**NXP/R**	**TI/L**	**TI/R**	**TI/C**	**TI/LS**
`Conv`	Kernel size	99.3%	99.2%	98.8%	98.4%	N/A	N/A
	# of kernels	98.2%	97.4%	99.7%	98.3%	N/A	N/A
	Stride size	97.4%	98.8%	98.2%	99.0%	N/A	N/A
	Padding size	98.8%	98.5%	99.1%	98.4%	N/A	N/A
`FC`	Output size	99.3%	98.0%	100.0%	99.7%	N/A	N/A
`MaxPool`	Kernel size	93.1%	N/A	93.4%	N/A	N/A	N/A
	Padding size	93.5%	N/A	94.7%	N/A	N/A	N/A
	Stride size	97.3%	N/A	97.5%	N/A	N/A	N/A
`AvgPool`	Kernel size	N/A	95.9%	N/A	96.8%	N/A	N/A
	Pad size	N/A	94.3%	N/A	96.6%	N/A	N/A
	Stride size	N/A	96.8%	N/A	97.1%	N/A	N/A
`RNN`	Hidden size	N/A	N/A	N/A	N/A	91.6%	N/A
`LSTM`	Output size	99.3%	98.0%	100.0%	99.7%	N/A	93.9%

adversarial example (i.e., the lower the accuracy, the higher the attack success rate). Our evaluation demonstrates that the adversarial attack success rate is boosted by leveraging the recovered DNN architecture, proving the effectiveness of NEUROSCOPE.

9.2.1 Target Details

The NXP i.MX 8M Plus board is a Linux-based SoC designed for edge AI applications and equipped with a neural processing unit (NPU) that accelerates DNN inference. Au-Zone DeepViewRT [4], a proprietary and closed-source SDK, is provided for developing DNN applications on this board and to leverage the NPU for DNN inference. In this case study, we use a proprietary DNN binary shipped with DeepViewRT SDK as the target DNN binary, which is a binary executable that takes an image as input and outputs the classification result of the image. From the labels of the classification result, we observe that the DNN model is an image classification model with the same labels as the ImageNet dataset [5].

9.2.2 DNN Architecture Extraction Attack

Since the NXP i.MX 8M Plus board is a Linux-based edge device, we use `ptrace` to perform the dynamic analysis of the target DNN binary. To force the target DNN binary to run fully on a CPU, we modify the DeepViewRT runtime's configuration file to remove the use of NPU acceleration. We use NEUROSCOPE to identify the DNN operator functions that are invoked by the target DNN binary, dump the input/output buffers of each DNN operator function invocation, and recover the DNN architecture. The recovered DNN architecture contains 58 operators, with consecutive pairs of `Conv` and `Relu` operators, followed by `AveragePool` and `Softmax` operators at the end. We investigate the recovered DNN architecture and find that it is identical to MobileNet v2 [52], a popular DNN architecture for image classification on mobile and embedded devices. However, since we do not have the source code/model of the target DNN binary, we cannot directly and completely verify the correctness of the recovered DNN architecture.

9.2.3 Boosting Adversarial Attack

In the context of DNN adversarial attacks, an adversary aims to manipulate the predicted labels of a DNN model by introducing minimal, often imperceptible, disturbances to the input images that cause the DNN model to erroneously predict an incorrect label [46]. Previous works show that adversarial attacks can be boosted if the adversary has knowledge of the DNN architecture [41] (i.e., the gray-box attack). Specifically, with the knowledge of the DNN architecture, gray-box attacks train a surrogate model with the same architecture as the target DNN model, conduct white-box attacks on the surrogate model to generate adversarial examples, and then use the adversarial examples to attack the target DNN model.

We demonstrate that the DNN architecture recovered by NEUROSCOPE can be used to conduct gray-box adversarial attacks. Specifically, we first randomly select 1,000 images from the ImageNet dataset [5] as the input images and then record the corresponding output logits. We also evaluate the inference accuracy of the target DNN binary on these images by comparing the predicted label with the ground truth label. The inference accuracy of the target DNN binary on the 1,000 images is 74.4%.

To launch the gray-box attack, given the observation that the recovered DNN architecture is identical to the MobileNet v2 model, we use the pre-trained MobileNet v2 model on TorchVision [44] as the starting point and fine-tune the pre-trained model. Specifically, as the fine-tuning procedure, we train the pre-trained model with the 1,000 images and their corresponding output logits as the training dataset, and we train the model for 50 epochs. We use the fine-tuned model as the surrogate model and use foolbox [50] to generate adversarial examples with the surrogate model.

For the black-box attack, we start with the ResNet-50 pre-trained model on TorchVision, fine-tune it with the same procedure and training hyperparameters as the gray-box attack, and use the fine-tuned model to generate adversarial examples with the same 1,000 images.

Table 10 shows the target DNN binary's accuracy on the adversarial examples generated by the black-box attack and the gray-box attack enabled by NEUROSCOPE. The `epsilon` denotes the perturbation magnitude, which represents the maximum allowed perturbation to the original image. Larger `epsilon` results in more noticeable perturbations. As shown in Table 10, the inference accuracy under the gray-box attack is significantly lower than the original inference accuracy (i.e., 74.4%). More importantly, given the same `epsilon`, the inference accuracy under the gray-box attack enabled by NEUROSCOPE is lower than the inference accuracy under the black-box attack, indicating that the adversarial attack is boosted by leveraging DNN architecture recovered by NEUROSCOPE.

Table 10: Accuracy of the victim DNN binary on the generated adversarial examples with and without the knowledge of the recovered DNN architecture. The lower the accuracy, the more successful the adversarial attack is.

epsilon	Black-box attack	Gray-box attack
0.01	63.8%	61.2%
0.1	49.9%	37.6%
0.3	35.6%	23.0%

10 Discussion

Handling Hardware Accelerators. NEUROSCOPE currently supports edge devices equipped with hardware accelerators by leveraging their CPU fallback capability and analyzing the I/O behaviors of the code running on their CPUs. This design is based on the observation that the CPU fallback is a common design choice for DNN runtimes on edge devices with hardware accelerators, and it allows DNN runtimes to execute DNN code on a CPU when a hardware accelerator is unavailable. As mentioned in Section 6.1.1, different mechanisms can be used to trick the runtime into using the CPU backend. For instance, if the runtime looks for shared libraries [11] or a device file to use an accelerator, we can simulate an environment without such an accelerator by pre-loading modified versions of the used shared libraries [48]. Unfortunately, some DNN runtimes (e.g., TensorRT on Nvidia Jetson) do not offer this CPU fallback feature, limiting the applicability of NEUROSCOPE.

To overcome this limitation, as future work, we plan to support this type of DNN runtime by developing the capability of monitoring the interactions between a CPU and a hardware accelerator (e.g., by monitoring memory-mapped I/O (MMIO)). This capability will enable capturing the I/O behaviors of DNN code running on a hardware accelerator. Then, we can use the captured I/O data to recover the semantics of DNN operators with trained neural networks as NEUROSCOPE does. Alternatively, for accelerators using an open ISA and providing debugging capability, we could adapt our dynamic analysis approach to analyze code running on those accelerators directly.

Supporting DNN Parameter Recovery. Some DNN operators (e.g., `Convolution` operators) carry parameters (i.e., weights). During inference, similar to the input data, the parameters are loaded from a buffer and used to perform the computation. While the parameter buffers are usually smaller than the input/output data buffers, they can be identified using the same dynamic analysis approach we use to identify the input/output data buffers (i.e., clustering the memory access information). To precisely recover the parameters from the parameter buffers, we need information about how the parameters are stored in the buffer (e.g., the layout of the parameters in the buffer). As future work, we can leverage the operator type and attributes recovered by NEUROSCOPE to guide the recovery of the parameters. Specifically, given the recovered operator type and attributes, we could enumerate the possible parameter layouts. Then, for each possible parameter layout, we could extract the parameters from the parameter buffer according to the layout and verify by executing the operator with the recovered parameters and input data and comparing the output with the output we acquire from the output buffer.

Supporting Additional DNN Operators. As shown in the previous research [27, 38], the DNN operators supported by NEUROSCOPE are sufficient to cover many popular DNN families. One limitation is that NEUROSCOPE does not currently support some DNN operators, such as the Transformer Block in the transformer architecture [61] or the TopK operator. We believe NEUROSCOPE can be extended to support these operators by synthesizing a dataset for them and retraining the classification and regression neural network models. Based on our experience, we estimate that adding support for a new operator requires 1 to 2 person-days of work, including 1 day for retraining for the machine-learning models, i.e., M_T and M_{Att}.

Enabling Debugging Features on Edge Devices. In scenarios where edge devices operate under an OS and DNN binaries are executed as applications, software debuggers are typically available. For bare-metal edge devices, to enable debugging features, we can connect hardware debugging tools, such as SEGGER J-Link [6], to a debug port or directly soldered to debug pins. Note that we can identify debug pins by checking a target board's specification or employing hardware reverse engineering tools, such as JTAGulator [7]. If they are not applicable, we can utilize alternative methods, such as firmware rehosting [21], and binary firmware patching [65], to enable debugging access.

Handling DNN Binaries with Heavy Optimizations and Obfuscations. We assume that our target DNN binaries are compiled with DNN compilers, using their default compilation options, that are shipped with the DNN SDK toolchains. Because their compilation process usually does not employ heavy optimizations and obfuscations, we do not consider the impacts of heavy optimization or obfuscation in our experiments. In principle, optimization and obfuscation should not fundamentally affect the effectiveness of our approach because they do not alter the I/O behaviors of DNN code. However, in practice, they may make it difficult for NeuroScope to accurately locate the DNN operators and I/O buffers within DNN binaries. In future work, we plan to investigate the impact of heavy optimization and obfuscation on the effectiveness of our approach and develop techniques to mitigate the impacts of these challenges.

Possible Defenses Against NeuroScope. Users can employ specific techniques to hinder NeuroScope's analysis. For instance, a user familiar with NeuroScope may try to disrupt its dependencies to thwart NeuroScope-based reverse engineering. For instance, users can prevent NeuroScope's dynamic analysis by disabling the debugging features (e.g., removing debug ports and disabling `ptrace`). Unfortunately, in this case, NeuroScope cannot reverse engineer the DNN binary. Furthermore, it is possible to obfuscate the DNN binaries [53], making the identification of DNN operators and I/O buffers more challenging.

11 Related Work

DNN Extraction Attacks. DNN Model extraction attacks reveal a DNN's model hyperparameters and/or parameters. Three categories of DNN extraction attacks have been proposed. The first category of extraction attack, including this work, leverages static or dynamic binary analysis to recover DNN models from the compiled DNN binaries [19, 38, 66, 70]. The second category of extraction attack queries a black-box DNN model and then trains a substitute model to approximate input-output behaviors from the victim DNNs [60, 62]. These attacks usually cannot recover DNN architecture/parameters and require significant computational resources, which is a key limitation when applied to edge devices. The third category of attack exploits hardware or side channels, such as PCIe traffic [34, 71], cache-based side-channel [68] and electromagnetic [15, 27, 28, 40], to launch attack. With the DNN model extracted, the adversary can launch other attacks, such as white-box adversarial attacks [18, 59], which assume an attacker has prior knowledge of a victim DNN model, such as model hyperparameters and parameters.

Program Induction and Synthesis by Examples. Program induction and synthesis automatically generate programs that satisfy the given requirements or specifications. The approach that is most relevant to our work is neural program synthesis by examples, which utilizes neural networks to synthesize programs from input/output examples [16, 23, 47]. There are also some works that leverage program synthesis to deobfuscate obfuscated programs [37, 42]. TF-Coder [55] is a recent work that uses neural networks to synthesize TensorFlow programs from natural language descriptions of the program and input/output examples. However, it requires the user to provide the natural language description of the program, which is not available in our setting. Additionally, it requires the knowledge of the exact information of the input/output examples, for example, the shape/dimension of the input/output tensors, which are hard to infer given the flat memory region our dynamic analysis identifies.

12 Conclusions

In this paper, we present NeuroScope, a novel, data-driven approach for reverse-engineering DNN binaries on edge devices through a combination of dynamic analysis and machine learning. Our approach does not rely on specific code features of the analyzed DNN binary, enabling NeuroScope to support more DNN binaries, specifically those implementing DNN models using an interpreter-based approach, than the existing approaches. Our evaluation shows that NeuroScope can accurately recover three different DNN models compiled by three different SDKs on three different edge devices. As a case study, we demonstrate that NeuroScope can be used to reverse-engineer a proprietary DNN binary which is compiled by a closed-source SDK, and the reverse-engineered results can be used to enable gray-box adversarial attacks.

Acknowledgments

This project was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contracts number N6600120C4031, the National Science Foundation (NSF) under contract number CNS2145744, the Office of Naval Research (ONR) under contract number N000142312157, and Lockheed Martin. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of United States Government, any agency, or corporation thereof.

References

[1] Classifying names with a character-level rnn. https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html.

[2] Deep learning meets the internet of things. https://shorturl.at/bILT5.

[3] Edge impulse. https://edgeimpulse.com/.

[4] eiq® inference with deepviewrt™. https://www.nxp.com/design/design-center/software/eiq-ml-development-environment/eiq-inference-with-deepviewrt:EIQ-INFERENCE-DEEPVIEWRT.

[5] Imagenet. https://image-net.org/index.php.

[6] J-link debug probes by segger – the embedded experts.

[7] Jtagulator. https://grandideastudio.com/portfolio/security/jtagulator.

[8] Nxp edgeready mcu-based solution for secure face recognition | nxp semiconductors.

[9] onnxruntime architecture. https://onnxruntime.ai/docs/reference/high-level-design.html.

[10] Sequence classification with lstm on mnist. https://notebook.community/santipuch590/deeplearning-tf/dl_tf_BDU/3.RNN/ML0120EN-3.1-Review-LSTM-MNIST-Database.

[11] Tensorflow lite delegates. https://www.tensorflow.org/lite/performance/delegates. (Accessed on 08/14/2024).

[12] Texas instruments edge ai studio. https://dev.ti.com/edgeaistudio/.

[13] Texasinstruments/edgeai-tidl-tools: Edgeai tidl tools and examples. https://github.com/TexasInstruments/edgeai-tidl-tools/tree/master.

[14] Borja Balle, Giovanni Cherubin, and Jamie Hayes. Reconstructing training data with informed adversaries. In *2022 IEEE Symposium on Security and Privacy*, pages 1138–1156. IEEE, 2022.

[15] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel. In *Proceedings of the USENIX Security Symposium*, 2019.

[16] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. Autopandas: neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

[17] Nicholas Carlini, Steve Chien, Milad Nasr, Shuang Song, Andreas Terzis, and Florian Tramer. Membership inference attacks from first principles. In *2022 IEEE Symposium on Security and Privacy*, pages 1897–1914. IEEE, 2022.

[18] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2017.

[19] Simin Chen, Hamed Khanpour, Cong Liu, and Wei Yang. Learning to reverse dnns from ai programs automatically. *arXiv preprint arXiv:2205.10364*, 2022.

[20] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie 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 *USENIX Symposium on Operating Systems Design and Implementation*, 2018.

[21] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. Halucinator: Firmware re-hosting through abstraction layer emulation. In *Proceedings of 29th USENIX Security Symposium*, 2020.

[22] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. Tensorflow lite micro: Embedded machine learning for tinyml systems. *Proceedings of Machine Learning and Systems*, 3:800–811, 2021.

[23] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR, 2017.

[24] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231, 1996.

[25] Linux Foundation. ONNX. https://onnx.ai/.

[26] Linux Foundation. ONNX model zoo. https://github.com/onnx/models.

[27] Yansong Gao, Huming Qiu, Zhi Zhang, Binghui Wang, Hua Ma, Alsharif Abuadbba, Minhui Xue, Anmin Fu, and Surya Nepal. Deeptheft: Stealing dnn model architectures through power side channel. In *2024 IEEE Symposium on Security and Privacy*. IEEE, 2024.

[28] Cheng Gongye, Yukui Luo, Xiaolin Xu, and Yunsi Fei. Side-channel-assisted reverse-engineering of encrypted dnn hardware accelerator ip and attack surface exploration. In *2024 IEEE Symposium on Security and Privacy*, pages 1–1. IEEE Computer Society, 2023.

[29] Google. Tensorflow lite for microcontrollers. https://www.tensorflow.org/lite/microcontrollers.

[30] Wenbo Guo, Lun Wang, Xinyu Xing, Min Du, and Dawn Song. Tabor: A highly accurate approach to inspecting and restoring trojan backdoors in AI systems. *arXiv preprint arXiv:1908.01763*, 2019.

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

[32] Hex-Rays. https://hex-rays.com/.

[33] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[34] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, et al. Deepsniffer: A dnn model extraction framework based on learning architectural hints. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[35] Bo-Yuan Huang, Steven Lyubomirsky, Yi Li, Mike He, Thierry Tambe, Gus Henry Smith, Akash Gaonkar, Vishal Canumalla, Gu-Yeon Wei, Aarti Gupta, Zachary Tatlock, and Sharad Malik. Specialized accelerators and compiler flows: Replacing accelerator apis with a formal software/hardware interface. 2022.

[36] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[37] Jaehyung Lee and Woosuk Lee. Simplifying mixed boolean-arithmetic obfuscation by program synthesis and term rewriting. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2351–2365, 2023.

[38] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, Xiaofei Xie, and Lei Ma. Decompiling x86 deep neural network executables. In *32nd USENIX Security Symposium*, pages 7357–7374, 2023.

[39] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[40] 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 *Proceedings of the USENIX Security Symposium*, 2022.

[41] Dongyu Meng and Hao Chen. Magnet: a two-pronged defense against adversarial examples. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 135–147, 2017.

[42] Grégoire Menguy, Sébastien Bardin, Richard Bonichon, and Cauim de Souza Lima. Search-based local black-box deobfuscation: understand, improve and mitigate. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2513–2525, 2021.

[43] Meta. glow. https://github.com/pytorch/glow.

[44] Meta. Torchvision datasets. http://pytorch.org/vision/main/datasets.html.

[45] NSA. Ghidra. https://ghidra-sre.org.

[46] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security*, 2017.

[47] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

[48] Kevin Pulo. Fun with ld_preload. In *linux. conf. au*, volume 153, page 103, 2009.

[49] Adnan Siraj Rakin, Md Hafizul Islam Chowdhuryy, Fan Yao, and Deliang Fan. Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories. In *2022 IEEE Symposium on Security and Privacy*, pages 1157–1174. IEEE, 2022.

[50] Jonas Rauber, Wieland Brendel, and Matthias Bethge. Foolbox: A python toolbox to benchmark the robustness of machine learning models. In *Reliable Machine Learning in the Wild Workshop, International Conference on Machine Learning*, 2017.

[51] Tirias Research. Smart inference devices. https://www.tiriasresearch.com/wp-content/uploads/2020/04/TIRIAS_Research-Smart_Inference_Devices.pdf.

[52] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018.

[53] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *Acm computing surveys*, 49(1):1–37, 2016.

[54] NXP Semiconductors. eiq® ml software development environment | nxp semiconductors. https://www.nxp.com/design/software/development-software/eiq-ml-development-environment:EIQ.

[55] Kensen Shi, David Bieber, and Rishabh Singh. Tf-coder: Program synthesis for tensor manipulations. *ACM Transactions on Programming Languages and Systems*, 44(2):1–36, 2022.

[56] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2017.

[57] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2016.

[58] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

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

[60] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *Proceedings of the USENIX Security Symposium*, 2016.

[61] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[62] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2018.

[63] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[64] Hongwei Wu, Jianliang Wu, Ruoyu Wu, Ayushi Sharma, Aravind Machiry, and Antonio Bianchi. Veribin: Adaptive verification of patches at the binary level.

[65] Jianliang Wu, Ruoyu Wu, Daniele Antonioli, Mathias Payer, Nils Ole Tippenhauer, Dongyan Xu, Dave Jing Tian, and Antonio Bianchi. Lightblue: Automatic profile-aware debloating of bluetooth stacks. In *30th USENIX Security Symposium*, pages 339–356, 2021.

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

[67] Chhavi Yadav and Léon Bottou. Cold case: The lost mnist digits. *Advances in Neural Information Processing Systems*, 32, 2019.

[68] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *Proceedings of the USENIX Security Symposium*, 2020.

[69] Corinna Cortes Yann LeCun and Chris Burges. Mnist handwritten digit database. http://yann.lecun.com/exdb/mnist/.

[70] Jinquan Zhang, Pei Wang, and Dinghao Wu. Libsteal: Model extraction attack towards deep learning compilers by reversing dnn binary library. In *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering*, 2023.

[71] Yuankun Zhu, Yueqiang Cheng, Husheng Zhou, and Yantao Lu. Hermes attack: Steal DNN models with lossless inference accuracy. In *Proceedings of the USENIX Security Symposium*, 2021.

[72] Muqi Zou, Arslan Khan, Ruoyu Wu, Han Gao, Antonio Bianchi, and Dave Jing Tian. D-helix: A generic decompiler testing framework using symbolic differentiation. In *33rd USENIX Security Symposium*, pages 397–414, 2024.