# LibSteal: Model Extraction Attack towards Deep Learning Compilers by Reversing DNN Binary Library

Jinquan Zhang[1], Pei Wang[2] and Dinghao Wu[1]
[1]*College of Information Science and Technology, The Pennsylvania State University, University Park, USA*
[2]*Individual Researcher, USA*
*jxz372, dwu@psu.edu, uraj@apache.org*

Keywords: Model Extraction Attack; Neural Network Architecture; Deep Learning Compiler; Reverse Engineering;

Abstract: The need for Deep Learning (DL) based services has rapidly increased in the past years. As part of the trend, the privatization of Deep Neural Network (DNN) models has become increasingly popular. The authors give customers or service providers direct access to their created models and let them deploy models on devices or infrastructure out of the control of the authors. Meanwhile, the emergence of DL Compilers makes it possible to compile a DNN model into a lightweight binary for faster inference, which is attractive to many stakeholders. However, distilling the essence of a model into a binary that is free to be examined by untrusted parties creates a chance to leak essential information. With only DNN binary library, it is possible to extract neural network architecture using reverse engineering. In this paper, we present *LibSteal*. This framework can leak DNN architecture information by reversing the binary library generated from the DL Compiler, which is similar to or even equivalent to the original. The evaluation shows that *LibSteal* can efficiently steal the architecture information of victim DNN models. After training the extracted models with the same hyper-parameter, we can achieve accuracy comparable to that of the original models.

## 1 INTRODUCTION

Machine learning models, especially deep neural networks (DNNs), have been widely deployed to tackle challenging problems in computer vision (He et al., 2016; Krizhevsky et al., 2012; Simonyan and Zisserman, 2014), speech recognition (Graves et al., 2013; Hinton et al., 2012), natural language processing (Collobert and Weston, 2008), and autonomous driving (Kato et al., 2015). Compared to other machine learning technologies, the outstanding performance of DNNs on recognition and prediction tasks (LeCun et al., 2015; Schmidhuber, 2015) has seen its commercial adoption with impacts across the field. It also increases the demand for Deep Learning (DL) based services and the need to deploy deep learning model on edge devices like mobile phones (Wu et al., 2019; Liang et al., 2018). For example, to help users who are blind or have low vision, some DNN models need to be deployed on the phone so that users can use them to identify nearby objects more conveniently. Also, giant AI providers provide the so-called service privatization to sell their high-quality DNN models to other companies and organizations with a license fee.

However, over the past decades, the explosion of both DNN frameworks (Chollet et al., 2015; Abadi et al., 2016; Markham and Jia, 2017; Chen et al., 2015; Paszke et al., 2017) and hardware backend (e.g., CPUs, GPUs, and FPGAs) increase the difficulty of deploying DL based services to target platforms. Such deployment requires significant manual effort due to the vast gap between DNN frameworks and the hardware backend. The deep learning compiler kills two birds with one stone and draws the attention of many stakeholders. Several DL compilers have been proposed by both industrial and academic actors recently, such as TVM (Chen et al., 2018), Tensor Comprehension (Vasilache et al., 2018), Glow (Rotem et al., 2018), nGraph (Cyphers et al., 2018), and XLA (Leary and Wang, 2017). The DL compilers take the models from different DL frameworks as input and compile them into a lightweight binary with faster inference efficiency for the target hardware platform. However, the DNN programs generated from DL compilers make it possible for an attacker to leak the internal work of DNN models by reversing or decompiling the stand-alone programs (Liu et al., 2022; Wu et al., 2022; Chen et al., 2022). Unlike the existing work targeting the whole DNN executables, in this paper, we narrow down the threat model where we only have access to the DNN binary library. We propose a framework named LibSteal to leak the network architecture information of the target DNN model by using only DNN binary library.

The architecture information includes the layer types, attributes, dimensions, and connectivity of the layers. In order to get this information, we designed our framework into three parts: binary analyzer, layer identification, and search engine. The binary analyzer slices the program into layer functions and extracts their I/O dimensions. Also, we find the nested loops of each control flow graph (CFG) at this step to identify the most significant features of each layer because the computation of the DNN layers mostly depends on the matrix. According to our observation, each layer has its unique computation pattern and these computation patterns remain the same across DNN models. Therefore, at the layer identification part, we first iterate possible layer attributes to generate layer functions with the same I/O dimensions and use these functions to make up the layer repo. Then we compare the similarity between the layer functions of the victim model and the generated layer functions in the layer repo to obtain the layer types and attributes. As for the search engine, we first build a directed graph based on I/O dimensions. After that, we search for the possible connection between the layers heuristically.

We implement the prototype of LibSteal based on Uroboros (Wang et al., 2015) and adopt the idea proposed by Asm2Vec (Ding et al., 2019) to accomplish the similarity comparison between layer functions. To demonstrate the practicality and the effectiveness of our attack framework, we evaluate it against binaries compiled from four widely-used DNN models, MNIST (LeCun, 1998), VGG (Simonyan and Zisserman, 2014), ResNet (He et al., 2016), and MobileNet (Howard et al., 2017) using TVM (Chen et al., 2018). The experimental result shows that our framework can effectively extract the neural network architecture information. The reconstructed models have similar or even equivalent network architecture to the original. We then re-trained the extracted models and they all achieved accuracy comparable to that of the original models.

In summary, we make the following contributions:

- We narrow down the threat model from the DNN executable to the DNN binary library. With limited input, we are able to leak essential information about the DNN model architecture.
- We design and implement the framework LibSteal to achieve our goal, which consists of three parts and combines various techniques to deliver a decent pipeline.

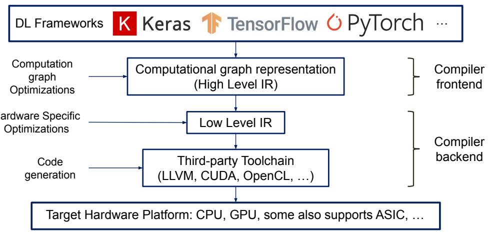

DL Frameworks
Keras
TensorFlow
PyTorch
...
Computation graph Optimizations
Computational graph representation (High Level IR)
Compiler frontend
Hardware Specific Optimizations
Low Level IR
Compiler backend
Code generation
Third-party Toolchain (LLVM, CUDA, OpenCL, ...)
Target Hardware Platform: CPU, GPU, some also supports ASIC, ...

Figure 1: The overview of deep learning compilers' commonly adopted design architecture.

- We have evaluated our framework on four widely-used model binaries using the TVM as the DL compiler. The results indicate that our framework can handle MNIST, VGG, ResNet, and MobileNet DNN models. With the stolen information, the reconstructed models have similar or even equivalent network architecture to the original and can achieve inference accuracy comparable to that of the original.

# 2 BACKGROUND

## 2.1 Deep Learning Compiler

The main purpose of deep learning compiler (Leary and Wang, 2017; Chen et al., 2018; Cyphers et al., 2018; Vasilache et al., 2018) is to reduce the manual effort required when deploying DL models to various hardware backends. As shown in Figure 1, the DL compiler takes different types of DNN frameworks, like TensorFlow (Abadi et al., 2016), PyTorch (Paszke et al., 2017), MXNet (Chen et al., 2015), Caffe2 (Markham and Jia, 2017), and Keras (Ketkar, 2017), as input and generates the optimized code for the different target hardware platforms. Most DL Compilers support CPU and GPU, and some also support Application-specific integrated circuits (ASICs) like TPU (Jouppi et al., 2017).

The key component of the DL Compiler is the multi-level intermediate representation (IR) (Li et al., 2020), which is an abstraction of the program and is mainly used to apply the optimization at different levels. The compilation process can be divided into the frontend and backend. At the compiler frontend, the DL models will be transferred to the high-level IR, also known as computational graph representation. The computational graph is independent of the target hardware platform. At this level, several different-grained optimizations will be applied to the computation graph. At the compiler backend, the high-level IR will further be represented as low-level IR for hardware-specific optimizations and code generation on the target hardware platform. The DL com-

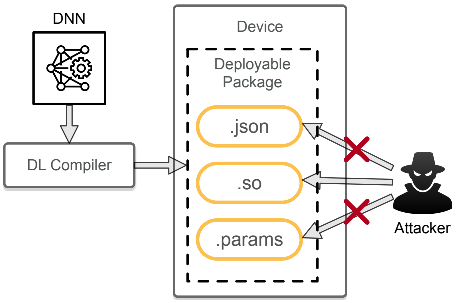


Figure 2: Threat Model. The DL compiler compiles the DNN model and deploys it on the edge device. The attacker can only access the .so part (i.e., shared library).

piler also supports the existing infrastructures such as LLVM (Lattner and Adve, 2004) to utilize the third-party toolchain for further code generation and optimizations. The generated package consists of three parts: a shared library file with all the layer functions, a JSON file with the connectivity of the network architecture, and a data file with the parameter weights. Our target in this paper is the shared library file.

## 2.2 Deep Neural Networks

Deep Neural Network (DNN) is a sub-area of Deep Learning in Artificial Neural networks (ANNs) with multiple layers between the input and output. A DNN can be represented as a function $y = f(x)$ where the input $x \in \mathbb{R}^n$ and the output $y \in \mathbb{R}^m$. In order to get the function, massive computation and a large amount of data are required for the training. A DNN model has the following essential characteristics: (1) **Neural network architecture** consists of layer types, layer dimensions, connection topology between layers, and layer attributes. (2) **Hyper-parameters** are the configuration data for the training process. They control the efficiency of the training process and affect the final performance of the model. (3) **Parameters** are updated during the training process. The performance of the model significantly relies on them. In our paper, we aim to steal the network architecture information of the DNN model through our framework.

## 2.3 Binary Reverse Engineering

Binary reverse engineering is the process of analyzing a binary program to identify the program's components and functionalities. It is imperative to understand the inner logic of the program for further operations, especially when the source code is unavailable. There are plenty of mature reverse engineering frameworks from both academia (Brumley et al., 2011; Wang and Shoshitaishvili, 2017; Wang et al., 2015; Song et al., 2008) and industry (Eagle, 2011). These frameworks are designed to recover as much information as possible on the target program for different purposes, which can either be maintaining the legacy code, understanding the behavior of the malware, or exploring the vulnerabilities in the program. In this paper, we implement our framework based on the Uroboros (Wang et al., 2015) to analyze the DNN binaries.

# 3 THREAT MODEL

This paper uses TVM (Chen et al., 2018) as the target DL Compiler. Having a concrete target is mostly to ease the technical discussions. The essence of our work applies to most major DL compilers on the market. As shown in Figure 2, the deployable package generated by TVM consists of three parts: a JSON-formatted specification file, a shared library (.so), and parameter weights (.params). Among them, both the JSON file and the shared library are important for inferring information about the neural network architecture. The JSON file contains the connection topology between the layers of the DNN models, and the shared library includes all the unique layer functions. In this paper, we assume that the attacker is motivated to leak the DNN model architecture information for malicious usage and can only access the shared library. This scenario is reasonable and has a practical impact because, as far as we know, the JSON file is a specific implementation of the TVM, and other DL compilers do not have this design. Even the TVM community plans to remove this text file in the future. We also assume that the victim DNN binary libraries are not obfuscated because, to the best of our knowledge, DL compilers themselves do not apply any software defensive mechanism to the generated binary.

# 4 ATTACK DESIGN

## 4.1 Overview

As shown in Figure 3, our attack framework consists of three parts, the binary analyzer, the layer identification module, and the search engine. First, we feed the DNN library to the binary analyzer. The analyzer disassembles the binary and slices it into different layer functions. We leverage the information from the data section to extract the layer dimensions for each layer function. Moreover, we apply nested loop analysis to identify each layer function's existing nested loop. The details will be discussed in Section 4.2.

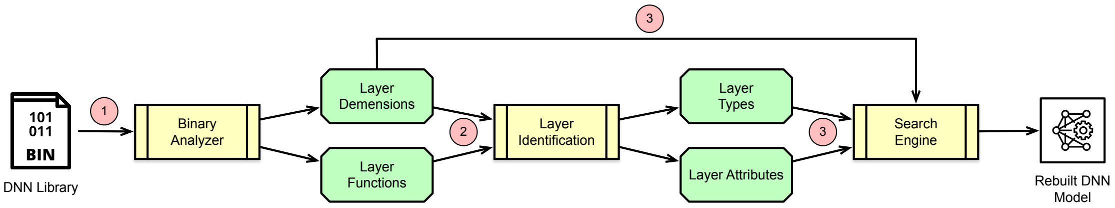


Figure 3: LibSteal Workflow Overview. ① The binary analyzer takes DNN binary library as input and disassembles it to get the sliced layer function. With further analysis, we also extract the layer dimensions and find the nested loop at this step; ② At the layer identification step, we leverage the immutable computation pattern to compare the similarity of unknown victim layer functions with customized candidate layer functions and identify their layer types and attributes. ③ Based on layer types, attributes, and dimensions, the search engine finds the connectivity between layers and rebuilds the network architecture of the model.

The next step is to identify the layer type and attributes of the layer functions. We use the layer dimensions to generate all possible layer functions with the same I/O dimension and store them in the candidate layer repository. Since the computation pattern of each kind of layer remains the same across different models, the layers with the same layer dimensions, type, and attributes will lead to very similar layer functions. Therefore, in order to obtain layer types and attributes, we use collected layer candidates to train the representative learning model so that we can check the similarity between victim layer functions and candidate layers functions. We present the detail in Section 4.3.

In the last step, we use the search engine to recover the model architecture topology connection using the information collected in the above two steps. We first build a directed graph based on the layer dimensions. Then in order to make the search process more efficient, we make some heuristic pruning based on the layer types. Along with the built graph, we explore a result containing all layers and get the connection topology between layers.

## 4.2 Binary Analysis

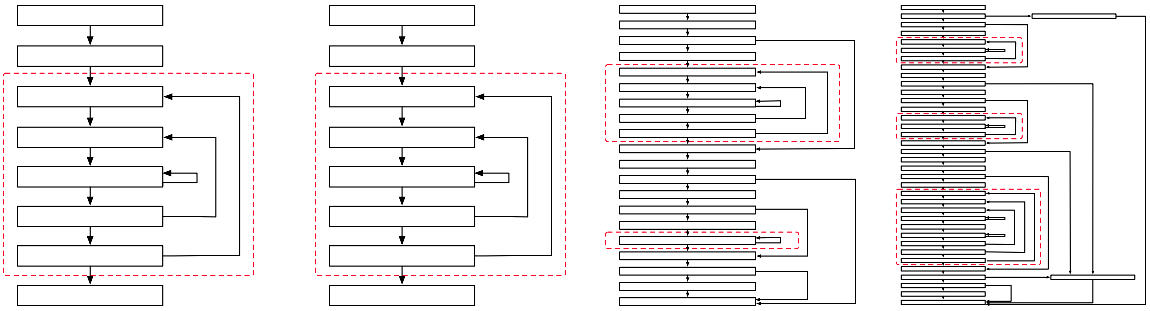
The Binary Analyzer uses Uroboros (Wang et al., 2015) as the binary reverse engineering framework to disassemble and analyze the victim DNN library, which help us to slice the program into separated layer functions and recover the precise control flow graph of each function.

### 4.2.1 Layer Dimensions

This section shows how we infer the layer dimensions. Figure 4 shows a part of a layer function compiled from VGG16, which is abstracted from the actual result, where we only replace the recovered symbol with a more readable one. We find that every layer function will check the constraint of the data dimensions before the computation so that the memory will not mess up during the runtime. Therefore, we locate the Basic Block invoking the error report function. In line 9, the code loads the address of __TVMAPISetLastError, which sets the last error message before return, to the register rax. The function is indirectly called in line 11 with the error message set to STR in line 10. According to the message carried by STR, this is the exception caused by the mismatch of the input and actual data dimensions. When we trace back to line 2 in Figure 4, we find the comparison between a memory-loaded number and a constant number in line 1 and figure out that one of the data dimension numbers is equal to 1. Following the same routine, we get a set of numbers. The numbers can be separated into groups based on their memory ad-

```
    section .text
     ...
    LAYER_FUNC:
     ...
 1   cmp dword [rax], 0x1
 2   jne LABEL
 3   cmp dword [rax + 0x8], 0x40
 4   jne (label)
 5   cmp dword [rax + 0x10], 0x20
 6   jne (label)
 7   cmp dword [rax + 0x18], 0x20
 8   jne (label)
     ...
    LABEL:
 9   mov rax, qword [LABEL_GOT]
10   lea rdi, [STR]
11   call qword [rax]
12   pop rcx
13   ret
     ...
    section .rodata
14   STR: "Assert fail" ...
     ...
    section .got
15   LABEL_GOT: qword __TVMAPISetLastError
```

Figure 4: Example of a layer function from VGG16 DNN model.



(a) Victim layer (b) MaxPooling2D layer (c) AveragePooling2D layer (d) Conv2D layer

Figure 5: The comparison between layer functions CFG with the same I/O dimensions. All four layer functions have the same input dimension (1,64,32,32) and the same output dimension (1,64,16,16).

dress. For example, four numbers in Figure 4 are in one group. According to our observation, each group represents one data dimension of the layer function. Therefore, the data dimension extracted from Figure 4 is $1,64,32,32$. Typically, one layer function only has one input dimension and one output dimension. However, some layers require multiple inputs, like `Add` layers whose input can be a list of tensors with the same shape. Fortunately, all constraints of dimension data are checked in order.

### 4.2.2 Nested Loop Analysis

The main goal of nested loop analysis is to find each layer function's most significant computation features. After the DNN model is compiled into the binary, the layer functions contain computation code and trivial instructions like push/pop and data load/store, which will influence the similarity check in the next step. Moreover, as shown in Figure 5, different types of layer functions have different numbers of nested loops, which is reasonable because the computation of pooling layers is relatively more straightforward than the Conv2D layer. This feature can be used to validate the result of the layer identification process to increase accuracy.

In order to find the nested loop from each layer function, we use Algorithm 1 to apply the analysis. In this algorithm, we demonstrate CFG as $G = (V,E)$ where $V$ is the set of basic blocks, and $E$ is the set of directed flow between basic blocks. In the first step (Line 1-13), we traverse the whole CFG beginning from the entry basic blocks (Line 2-3) and timestamp each basic block when they are popped out from the stack (Line 10-11). In the second step (Line 14-23), we first create a reversed CFG, $G' = (V',E')$ from the original CFG, $G$ (Line 14-16). The direction of flows of $G'$ are opposite from $G$. After that, we search $V'$ to find the basic block with the latest timestamp (Line 18-20). And starting from this basic block, we try to traverse the reversed CFG, $G'$ (Line 21). All the basic blocks reached by this point are no doubt members of a nested loop. We then eliminate these basic blocks from $V'$ (Line 22) and continue the job until all basic blocks are revisited. For the record, we also consider the self-loop as the nested loop.

**Algorithm 1** Nested Loop Analysis Algorithm

```
    Input: $G = (V,E)$ {Control flow graph}
 1: $stack \leftarrow \emptyset$
 2: $stack.push(G.entryBlock)$
 3: $visitedBlocks \leftarrow \{G.entryBlocks\}$
 4: while not $stack.empty()$ do
 5:     $u \leftarrow stack.top()$
 6:     if $v \in u.successors$ and $v \notin visitedBlocks$ then
 7:         $stack.push(v)$
 8:         $visitedBlocks.add(v)$
 9:     else
10:         $stack.pop()$
11:         $timestamp(u)$
12:     end if
13: end while
14: $V' \leftarrow V$
15: $E' \leftarrow reversed(E)$ {the directions of all edges are
    opposite}
16: $G' \leftarrow (V',E')$
17: $nestedLoops \leftarrow \emptyset$
18: while not $V'.empty()$ do
19:     for $u \in V'$ do
20:         $v \leftarrow u : v \ ? \ u.timestamp > v.timestamp$
21:     end for
22:     $blockSet \leftarrow traverse(G',v)$
23:     $V' \leftarrow V' - blocksSet$
24:     if $blockSet.size() > 1$ or $v.isSelfLoop()$ then
25:         $nestedLoops.add(blockSet)$
26:     end if
27: end while
```

## 4.3 Layer Identification

### 4.3.1 Layer Generator

The fundamental of layer identification is that for two layers, even if they are from different models, as long as they share the same layer type, I/O dimensions, and layer attributes, they are compiled into very similar layer functions. For many layer types, the relationship between the output and input dimensions is based on their attributes. We take the Conv2D layer as an example. Assume the input dimension is $(N,C,H,W)$, where $N$ is the sample number, $C$ is the channel number, and $H,W$ is the resolution of the data. Then we have

$$H_{out} = \lfloor \frac{H + 2\cdot P - D\cdot(K-1) - 1}{S} + 1 \rfloor$$
$$W_{out} = \lfloor \frac{W + 2\cdot P - D\cdot(K-1) - 1}{S} + 1 \rfloor$$

where $P$ is the value of padding, D is the value of dilation, $K$ is the value of kernel size, and $S$ is the value of stride. Therefore, for each pair of I/O layer dimensions, we generate possible candidate layer functions to form a layer repo. As shown in Figure 5, (a) is the computation part of the layer function we discuss in Figure 4. After applying the method we present in Section 4.2, we get its I/O dimensions as $(1,64,32,32)$ and $(1,64,16,16)$. There are several possible layers with this kind of I/O dimensions. We choose three representative layers to illustrate our approach: MaxPooling2D with `pool_size=2` (b), AveragePooling2D with `pool_size=2` (c), and Conv2D with `filters=64`, `kernel_size=2`, `strides=2`, `padding=same` (d). As we can see, the CFG of the victim layer function shown in (a) is the same as the layer function of MaxPooling2D with `pool_size=2` (b).

### 4.3.2 Layer Function Representation Learning

Our representation learning model is built based on the idea proposed by Asm2Vec (Ding et al., 2019). After finishing the training of the representation learning model, we first use the nested loop analysis approach in Section 4.2.2 to lift the computation pattern of candidate layer functions. We then use the trained model to produce the vector of each candidate layer function. The vector of the victim function is also generated from the nested loop of the functions. Finally, we calculate the similarity score and make the inference decision for the victim function.

As for the example shown in Figure 5, the similarity between the victim layer and MaxPooling2D is 0.9356, while its similarity with AveragePooling2D is 0.522 and with Conv2D is 0.272. Therefore, we determine that the layer function from VGG16 is MaxPooling2D with `pool_size=2`. This result is reasonable because the computation patterns MaxPooling2D and AveragePooling2D are similar. They follow the same routine to slide through the data. However, the key operations are different.

## 4.4 Model Reconstruction

After gathering valuable information, including layer dimensions, types, and attributes, we can reconstruct the model architecture.

At first, we build a directed graph, $G_{nn} = (V_{nn}, E_{nn})$, where each layer represents a vertex $v$, and an edge $e$, pointing from $v_1$ to $v_2$ means the output dimension of $v_1$ matches the input dimension of $v_2$. Also, we divide the layer into supportive layers (e.g., ReLu, BN) and functional layers (e.g., Conv2D, Pooling2D). As we mentioned before, the shared library only contains unique functions. The victim model will obviously use the most supportive layers multiple times. Therefore, to reduce the search space, we assume that each functional layer can only be used once and should all be used in the final reconstructed model, and as for the supportive layers, we do not limit their usage. Moreover, once a functional layer finds out that it can link multiple supportive layers, we directly add them to the path. For example, if a Conv2D layer finds itself connected to a ReLu layer and a BN layer, we will combine them as Conv2D → BN → ReLu structure. Additionally, when we meet the layer with multiple inputs, we trace back the data flow graph to find another input and link it back to the former path. Finally, along the directed graph $G_{nn}$, we search from input layer to the output layer and reconstruct the DNN model.

# 5 EVALUATION

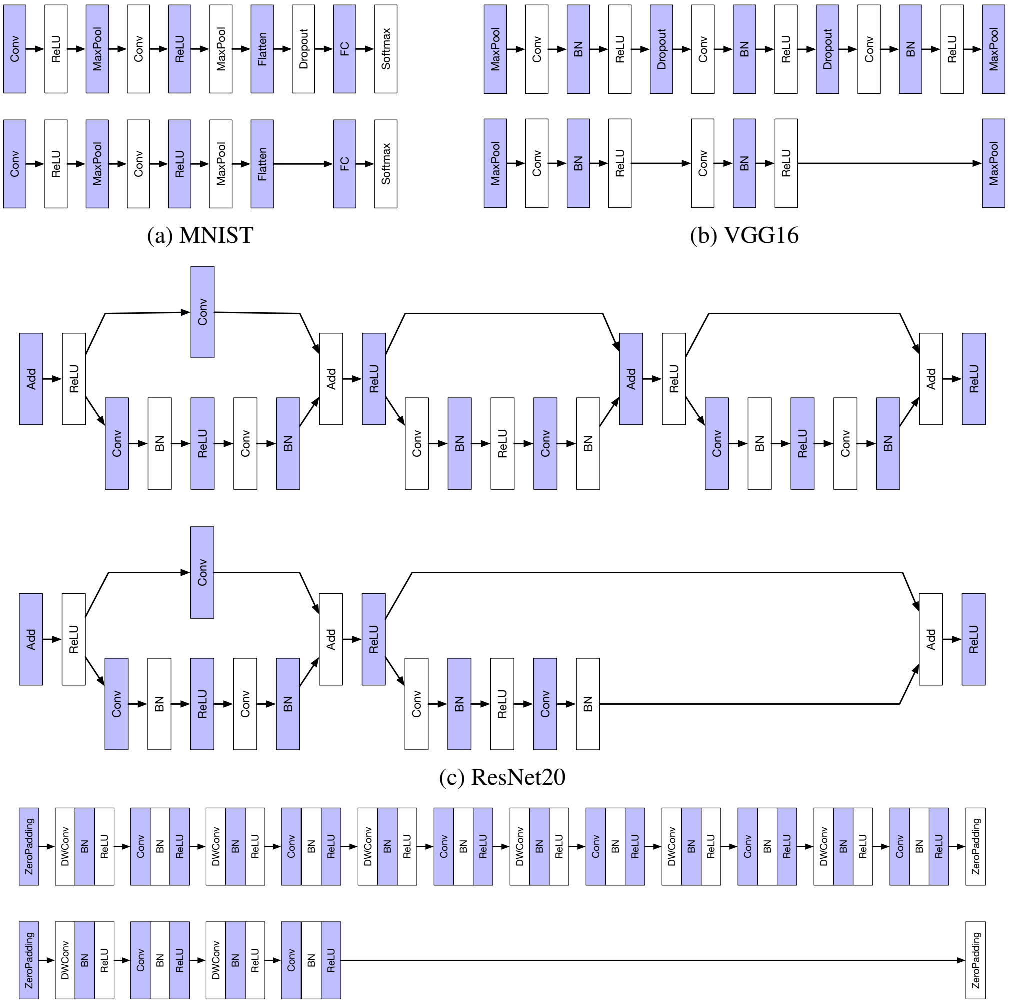
## 5.1 Experiment Setup

**Environment:** All the experiments are run on the Ubuntu 18.04 LTS server with `NVIDIA TITAN XP` GPU and dual-core 2.20 GHz `Intel(R) Xeno(R) Silver 4114` CPU. We pick CUDA 10.3 as the GPU programming interface. For the training of the representative learning model, we choose embedding dimension $d = 200$, 25 negative samples, 3 random walks, and a learning rate of 0.025.

**Victim Model:** We evaluate our attack on four widely-used DNN models: MNIST, VGG16,

Table 1: Victim Models Information.

| | Datasets | Input Shape | # of Parameters | # of layers | # of layer types |
|---|---|---|---|---|---|
| MNIST | MNIST | (28,28,1) | 34,826 | 11 | 7 |
| VGG16 | CIFAR-10 | (32,32,3) | 150,001,418 | 60 | 7 |
| ResNet20 | CIFAR-10 | (32,32,3) | 19 274,442 | 72 | 8 |
| MobileNet | CIFAR-10 | (32,32,3) | 3,239,114 | 91 | 9 |



(a) MNIST

(b) VGG16

(c) ResNet20

(d) MobileNet

Figure 6: Architecture Comparison. This figure partially shows the difference of the network architecture between original models and extracted models. MNIST

ResNet20, and MobileNet. The detailed information of all victim models is shown in Table 1. All the pretrained models are designed in the Keras framework (Team, 2022a; Team, 2022b) with Tensorflow as the backbone and compiled by TVM (Chen et al., 2018) to generate the binaries. We use LLVM (Lattner and Adve, 2004) as our host platform. As mentioned in Section 2, we assume that we only have access to the DNN binary library during the attack.

## 5.2 Architectural Completeness

This section compares the architectures of the original models and the extracted models. Figure 6 visualizes part of the architecture of original models and extracted models for MNIST, VGG16, ResNet20, and MobileNet. Each block in the figure represents a DNN layer. As shown in the figure, the basic architecture remains the same in extracted models. However, several layers are missing. First, the dropout layer is

Table 2: Statistics of the comparison between original model and extracted model.

| | Test accuracy | | Layer number | | | Layer type number | |
|---|---|---|---|---|---|---|---|
| | Original (%) | Extracted (%) | Original | Extracted | Percentage (%) | Original | Extracted |
| MNIST | 99.17 | 99.04 | 11 | 10 | 90.91 | 7 | 6 |
| VGG16 | 93.16 | 90.59 | 60 | 38 | 63.33 | 7 | 6 |
| ResNet20 | 91.65 | 83.92 | 72 | 49 | 68.06 | 8 | 8 |
| MobileNet | 83.16 | 75.00 | 91 | 66 | 72.53 | 9 | 8 |

used to avoid over-fitting during the training phase by setting randomly selected input units to 0, so it will be invalid during the inference.

Regarding other missing layers, as we mentioned in Section 4.3, the same type of layer with the same I/O dimensions and attributes will produce the same layer function. This rule of thumb allows us to identify the layer type and its related attributes but also leads to the situation that the layer function can be reused in the DNN executable. As shown in Figure 6, the recovery of MobileNet is least satisfying. The reason for missing a larger chunk in this model is because that the original DNN model is basically the composition of the same pattern repeated for four times. Since our attack targets the DNN library, it is difficult to infer the number of repetitions solely from the code, as the same code can be executed for arbitrary times during inference. On the other hand, Figure 6a shows that as long as each layer of the DNN model holds an unique layer function, we are able to fully recover the whole network architecture.

### 5.3 Accuracy of Extracted Models

We compared other statistics information between original DNN models with the extracted ones as shown in Table 2. In order to evaluate the functionality of the extracted model, we re-trained the extracted models and compared their accuracy with the original models. As for the training settings, we use the same hyper-parameter as the original one for a more precise comparison. For the record, we re-train the model to demonstrate the accuracy of extracted model architecture which does not assume that our framework requires the training dataset and the hyper-parameters. As shown in the Table 2, with the missing layers, it is not surprising that the accuracy of VGG16, ResNet20, and MobileNet is worse than the original ones. However, we can see that the accuracy drop of ResNet20 and MobileNet is more significant than that of VGG16, although we recover more layers for ResNet20 and MobileNet. We guess that the importance of layers varies inside the neural network architecture. Although VGG16 missed more layers, the key skeleton still remains. As for the layer types, so far, the only layer we are not able to recover is the Dropout layer, which will disappear during the inference process.

## 6 DISCUSSION

The limitation of our attack framework is due to the connection between layers. The shared library compiled from DNN models only contains the distinct layer functions, so we need to know the exact used number of each layer function. One solution is that we can enlarge our search space and use meta-learning to make us closer to the original network architecture. For example, we do not limit the usage of any function layers and allow the search engine to explore the possible combination. We train and test each explored architecture's accuracy and leave the best result. However, time-consuming will be out of imagination. On the other hand, if we can access the JSON file or even the parameter file, we can recover the precise DNN models, which makes our work more meaningful.

## 7 RELATED WORK

The basic logic of the model extraction attack is leveraging the information gathered from a different source to leak the vital features of the machine learning model. The most commonly used source is the side channel (Hua et al., 2018; Yan et al., 2020; Wei et al., 2018; Xiang et al., 2020; Duddu et al., 2018; Hunt et al., 2020; Batina et al., 2019). For example, by exploiting the memory and timing side-channel, (Hua et al., 2018) presented a model extraction attack to infer the network architecture and identify the value of parameters of the convolutional neural network (CNN) running on a hardware accelerator. Bus traffic is also an important information source (Zhu et al., 2021; Hu et al., 2019). (Zhu et al., 2021) identified a new attack surface based on encrypted PCIe traffic and fully extracted the DNN model with the exact model characteristics, and achieved the same inference accuracy as the target model. Some exciting work also relied on the query-prediction pairs from the target model (Tramèr et al., 2016; Oh et al., 2019; Orekondy et al., 2019; Kariyappa et al., 2021). (Tramèr et al., 2016) relied on the information carried by the output from the ML prediction APIs to generate a similar or the same model and successfully applied the attack against the online services.

# 8 CONCLUSIONS

The rising of DL compilers and the privatization situation introduce a new threat to the ML community. Several novel attack framework has been proposed. However, all of them assume they have full access to the DNN binaries. In this paper, we narrow down the threat model and demonstrate that with only the DNN binary library, we can leak the network architecture information of DNN models. We propose a framework, namely LibSteal, using only the DNN library to get the layer types, attributes, dimensions, and even compatible topology connections. We implemented a prototype of LibSteal and evaluated it on four DNN models compiled from TVM. The evaluation results indicate that our framework can reconstruct a similar or even equivalent model architecture compared to the original one, achieving comparable accuracy after training with the public datasets with only the library files.

# ACKNOWLEDGEMENTS

This research was supported in part by the National Science Foundation (NSF) grant CNS-1652790 and the Office of Naval Research (ONR) grant N00014-17-1-2894.

# REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *OSDI 16*, pages 265–283.

Batina, L., Bhasin, S., Jap, D., and Picek, S. (2019). CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel. In *USENIX Security 19*, pages 515–532.

Brumley, D., Jager, I., Avgerinos, T., and Schwartz, E. J. (2011). Bap: A binary analysis platform. In *CAV 2011*, pages 463–469. Springer.

Chen, S., Khanpour, H., Liu, C., and Yang, W. (2022). Learning to reverse dnns from ai programs automatically. *arXiv preprint arXiv:2205.10364*.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. (2015). Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al. (2018). TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI 18*, pages 578–594.

Chollet, F. et al. (2015). Keras. https://keras.io.

Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *ICML 2008*, pages 160–167.

Cyphers, S., Bansal, A. K., Bhiwandiwalla, A., Bobba, J., Brookhart, M., Chakraborty, A., Constable, W., Convey, C., Cook, L., Kanawi, O., et al. (2018). Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *arXiv preprint arXiv:1801.08058*.

Ding, S. H., Fung, B. C., and Charland, P. (2019). Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *IEEE S&P 2019*, pages 472–489. IEEE.

Duddu, V., Samanta, D., Rao, D. V., and Balas, V. E. (2018). Stealing neural networks via timing side channels. *arXiv preprint arXiv:1812.11720*.

Eagle, C. (2011). *The IDA pro book*. no starch press.

Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *ICASSP 2013*, pages 6645–6649. Ieee.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *CVPR 2016*, pages 770–778.

Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., et al. (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*.

Hu, X., Liang, L., Deng, L., Li, S., Xie, X., Ji, Y., Ding, Y., Liu, C., Sherwood, T., and Xie, Y. (2019). Neural network model extraction attacks in edge devices by hearing architectural hints. *arXiv preprint arXiv:1903.03916*.

Hua, W., Zhang, Z., and Suh, G. E. (2018). Reverse engineering convolutional neural networks through side-channel information leaks. In *DAC 2018*, pages 1–6. IEEE.

Hunt, T., Jia, Z., Miller, V., Szekely, A., Hu, Y., Rossbach, C. J., and Witchel, E. (2020). Telekine: Secure computing with cloud GPUs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 817–833.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. (2017). In-datacenter performance analysis of a tensor processing unit. In *ISCA '17*, pages 1–12.

Kariyappa, S., Prakash, A., and Qureshi, M. K. (2021). Maze: Data-free model stealing attack using zeroth-order gradient estimation. In *CVPR 2021*, pages 13814–13823.

Kato, S., Takeuchi, E., Ishiguro, Y., Ninomiya, Y., Takeda, K., and Hamada, T. (2015). An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68.

Ketkar, N. (2017). Introduction to keras. In *Deep learning with Python*, pages 97–111. Springer.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25.

Lattner, C. and Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO 2004.*, pages 75–86. IEEE.

Leary, C. and Wang, T. (2017). Xla: Tensorflow, compiled. *TensorFlow Dev Summit*.

LeCun, Y. (1998). The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436–444.

Li, M., Liu, Y., Liu, X., Sun, Q., You, X., Yang, H., Luan, Z., and Qian, D. (2020). The deep learning compiler: A comprehensive survey. *arXiv preprint arXiv:2002.03794*.

Liang, Y., Cai, Z., Yu, J., Han, Q., and Li, Y. (2018). Deep learning based inference of private information using embedded sensors in smart devices. *IEEE Network*, 32(4):8–14.

Liu, Z., Yuan, Y., Wang, S., Xie, X., and Ma, L. (2022). Decompiling x86 deep neural network executables. *arXiv preprint arXiv:2210.01075*.

Markham, A. and Jia, Y. (2017). Caffe2: Portable high-performance deep learning framework from facebook. *NVIDIA Corporation*.

Oh, S. J., Schiele, B., and Fritz, M. (2019). Towards reverse-engineering black-box neural networks. In *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, pages 121–144. Springer.

Orekondy, T., Schiele, B., and Fritz, M. (2019). Knockoff nets: Stealing functionality of black-box models. In *CVPR 2019*, pages 4954–4963.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in pytorch. In *NIPS-W*.

Rotem, N., Fix, J., Abdulrasool, S., Catron, G., Deng, S., Dzhabarov, R., Gibson, N., Hegeman, J., Lele, M., Levenstein, R., et al. (2018). Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural networks*, 61:85–117.

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P. (2008). Bitblaze: A new approach to computer security via binary analysis. In *ICISSP 2008*, pages 1–25. Springer.

Team, K. (2022a). Keras applications.

Team, K. (2022b). Keras examples.

Tramèr, F., Zhang, F., Juels, A., Reiter, M. K., and Ristenpart, T. (2016). Stealing machine learning models via prediction apis. In *USENIX Security 16*, pages 601–618.

Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A. (2018). Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*.

Wang, F. and Shoshitaishvili, Y. (2017). Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9. IEEE.

Wang, S., Wang, P., and Wu, D. (2015). Reassembleable disassembling. In *USENIX Security 15*, pages 627–642.

Wei, L., Luo, B., Li, Y., Liu, Y., and Xu, Q. (2018). I know what you see: Power side-channel attack on convolutional neural network accelerators. In *ACSAC '18*, pages 393–406.

Wu, C.-J., Brooks, D., Chen, K., Chen, D., Choudhury, S., Dukhan, M., Hazelwood, K., Isaac, E., Jia, Y., Jia, B., et al. (2019). Machine learning at facebook: Understanding inference at the edge. In *HPCA 2019*, pages 331–344. IEEE.

Wu, R., Kim, T., Tian, D. J., Bianchi, A., and Xu, D. (2022). DnD: A cross-architecture deep neural network decompiler. In *USENIX Security 22*, pages 2135–2152.

Xiang, Y., Chen, Z., Chen, Z., Fang, Z., Hao, H., Chen, J., Liu, Y., Wu, Z., Xuan, Q., and Yang, X. (2020). Open dnn box by power side-channel attack. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 67(11):2717–2721.

Yan, M., Fletcher, C. W., and Torrellas, J. (2020). Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *USENIX Security 20*, pages 2003–2020.

Zhu, Y., Cheng, Y., Zhou, H., and Lu, Y. (2021). Hermes attack: Steal DNN models with lossless inference accuracy. In *USENIX Security 21*.