FlatD: Protecting Deep Neural Network Program from Reversing Attacks

Jinquan Zhang
The Pennsylvania State University
University Park, USA
jxz372@psu.edu

Zihao Wang
The Pennsylvania State University
University Park, USA
zihao@psu.edu

Dinghao Wu
The Pennsylvania State University
University Park, USA
dinghao@psu.edu

Pei Wang
Individual Researcher
San Jose, USA
uraj@apache.org

Rui Zhong
Palo Alto Network
Santa Clara, USA
reversezr33@gmail.com

***Abstract*—The emergence of Deep Learning compilers provides automated optimization and compilation across Deep Learning frameworks and hardware platforms, which enhances the performance of AI service and benefits the deployment to edge devices and low-power processors. However, deep neural network (DNN) programs generated by Deep Learning compilers introduce a new attack interface. They are targeted by new model extraction attacks that can fully or partially rebuild the DNN model by reversing the DNN programs. Unfortunately, no defense countermeasure is designed to hinder this kind of attack.**

To address the issue, we investigate all of the state-of-the-art reversing-based model extraction attacks and identify an essential component shared across the frameworks. Based on this observation, we propose FlatD, the first defense framework for DNN programs toward reversing-based model extraction attacks. FlatD manipulates and conceals the original Control Flow Graphs of DNN programs based on Control Flow Flattening. Unlike traditional Control Flow Flattening, FlatD ensures the DNN programs are challenging for attackers to recover their Control Flow Graphs and gain necessary information statically. Our evaluation shows that, compared to the traditional Control Flow Flattening (O-LLVM), FlatD provides more effective and stealthy protection to DNN programs with similar performance and lower scale.

***Index Terms*—Software Engineering, Protection mechanisms, Artificial Intelligence**

I. Introduction

Due to the widespread success [1]–[4] of Deep Learning (DL) across various domains, the demand for DL-based services has surged in recent years. This has led service providers to deploy the Deep Neural Network (DNN) models across a wide range of hardware devices, from cloud servers to embedded devices [5], to meet diverse requirements. However, deployment across multiple platforms presents challenges due to the differences in on-chip memory architecture and compute primitives across CPU, GPU, and TPU-like accelerators [6]. Additionally, the rapid growth of DL frameworks [7]–[10] further complicates the situation.

The DL compilers [11]–[14] ease this process by automatically compiling the models into standalone DNN programs with decent optimization using multiple intermediate representations (IR) during compilation. Generally, a DL compiler can support various frameworks as input and generate programs for different hardware devices. Some also [11], [13] allow third-party toolchains such as LLVM [15] and CUDA [16] for further code generation. The powerful automated optimization provided by DL compilers suddenly attracted the attention of both academia and industry. Gaint AI providers such as Google, Amazon, and Facebook are all considering embedding the DL compilers into their AI infrastructure to enhance the performance of their AI services. [17]–[21].

While the DL compilers significantly impact the AI industry, they also introduce a new attack interface from the binary analysis side. Due to the lack of defense applied at the binary level, the DNN programs are vulnerable to reversing-based model extraction attacks. The targets of traditional model extraction attacks [22]–[31] can mainly be classified into three categories: side channel information, sniffing bus traffic, and prediction pairs from black box models. These information sources are limited and sometimes depend on strict assumptions. Unlike these sources, DNN programs always contain complete information that can be used to run in an isolated environment. To date, there are four state-of-the-art model extraction attack frameworks [32]–[35] that can fully or partially reconstruct models by reversing the DNN programs. However, to our knowledge, effective defense mechanisms still need to be developed to countermeasure these reversing attacks. Moreover, training DNN models at an industry scale often involves processing TB-sized datasets [36], [37] with high training costs. For example, using a v2 Tensor Processing Unit (TPU) in a cloud environment costs approximately $4.50 per hour, and completing an entire training cycle may exceed $400,000 [38], [39], which emphasizes the importance of protecting DNN models.

Unlike the typical binary program, the DNN program is generated directly from the model without any source code, which excludes source-code-level defense frameworks like Tigress [40]. Moreover, the DNN Program is more sensitive to

performance and scale than a typical binary program, making the time-consuming framework unavailable. Fortunately, most state-of-the-art DL Compilers [11], [13] support third-party code-gen tools (e.g., LLVM [15]) for users to apply the customized transformation, which leaves us the window to shield DNN programs.

We carefully investigated attacking frameworks' basic logic and workflow to gain more insight into the reversing-based model extraction attack. These frameworks include the same components to rebuild the model: operator-type recovery, topology recovery, and metadata recovery (including dimensions, parameters, and attributes). Although the methodologies vary from framework to framework, they share the idea of using the computation pattern to recover the operator type. Specifically, each kind of operator in the DNN model has a formula for transforming the input data to the next operator. For instance, the ReLU activation function uses the formula 1, and the Tanh activation function uses the formula 2. They exhibit entirely different syntax and semantics meanings when represented in the program. This feature helps attackers infer the operator type by using binary similarity comparison. On the other hand, it also guides the protection of DNN programs because we found that the Control Flow Graph (CFG) plays a vital role in all attack frameworks.

$$f(x) = x^+ = \max(0, x) = \frac{x + |x|}{2} \tag{1}$$

$$f(x) = \frac{e^x + e^{-x}}{e^x - e^{-x}} \tag{2}$$

Based on our observations, this paper proposes FlatD, an advanced defense framework based on Control Flow Flattening, for DNN programs to protect them from reversing-based model extraction attacks. Unlike the traditional Control Flow Flattening, we leverage the opaque predicate, one-way cryptographic hashing, and indirect jump to conceal the control flow further so that attackers cannot quickly recover the original CFG and apply more inference analysis. We also use several strategies to preserve the DNN program's performance and reduce the overall time overhead.

We implemented FlatD on the top of O-LLVM [41] and embedded it into the code generation part of TVM [11]. We used O-LLVM as the baseline and evaluated FlatD on eight real-world pre-trained models and one self-trained model from four frameworks. Our experiment results show that compared to the traditional Control Flow Flattening, FlatD can more effectively counterwork state-of-the-art reversing-based model extraction attacks while preserving the functionality of the original DNN programs. Moreover, the DNN program transformed by FlatD performs similarly to the one using traditional Control Flow Flattening in most cases and always has a lower scale.

In summary, we make the following contributions:

- We investigate four state-of-the-art reversing-based model extraction attacks and identify a key component shared across the attack frameworks. This component guides the provision of protection and contributes to future research on DNN program safety.

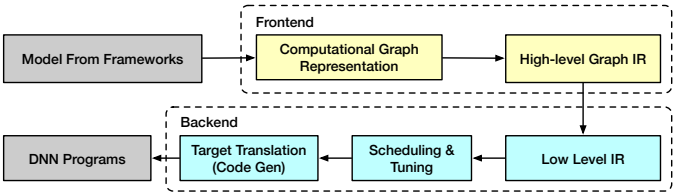


Fig. 1: Compilation Flow of the Deep Learning Compiler. The input of the DL compiler is a DNN model. The compiler frontend transforms the model description into a computational graph representation and further conveys it into graph IR to apply graph- and node-level optimizations. At the compiler backend, it does hardware-specific optimization on low-level IR. Finally, the compiler generates the DNN Programs for the target platform.

- We design and implement FlatD, the advanced defense framework targeting compiled models toward reversing-based model extraction attacks. FlatD conceals the original Control Flow Graph of DNN programs based on Control Flow Flattening and ensures minimal information gained by attackers through statistical analysis.
- We successfully apply FlatD on DNN programs compiled from large-scale models using TVM to evaluate these DNN programs regarding functionality, performance, and resilience. We use O-LLVM as the baseline to compare the results. Our experiment demonstrates that DNN programs transformed by FlatD can prevent leaking information from reversing-based model extraction attacks more effectively than traditional Control Flow Flattening with similar performance and lower scale.

II. Background

A. Deep Learning Compiler

The objective of deep learning compilers, such as XLA [14], TVM [11], Intel nGraph [42], and Tensor Comprehension [43], is to simplify the process of deploying DNN models on different hardware platforms, by automating the optimization and transformation. These compilers can take models described within popular frameworks like TensorFlow [8], PyTorch [44], MXNet [10], Caffe2 [9], and Keras [45] as inputs and generate standalone DNN programs or kernel libraries that can be statically linked with executables for CPUs, GPUs, and TPU-like accelerators. As shown in Fig.1, the DL compiler architecture can be divided into two main phases: frontend and backend, each manipulating one or several Intermediate Representations (IR).

Frontend. DL compilers first transform high-level model descriptions into computational graph representations and convert them into graph IRs. These IRs, independent of the target hardware platform, define the graph structure, including the network topology and layer dimensions. They facilitate graph- and node-level optimizations, such as operator fusion, static memory planning, and layout transformation [11], [13].

Backend. From the graph IRs, hardware-specific low-level IRs are generated. These IRs serve as an intermediary step

for tailored optimizations, incorporating knowledge of DL models and hardware characteristics. The graph IR operators can be converted into low-level linear algebra operators, simplifying the support for high-level operators across various hardware targets. This stage's optimization includes hardware intrinsic mapping, memory allocation, loop-related optimizations, and parallelization [11], [46]–[48]. The backend also involves scheduling and tuning, where the compiler searches for optimal parameter settings, such as loop unrolling factors. Recent advancements [11], [43], [49], [50] introduce automated scheduling and tuning to improve optimization, reducing manual efforts.

Code Generation. Finally, these low-level IRs are compiled into code for different hardware targets. Before that, the DL compilers can also integrate with existing infrastructure like LLVM [15] and CUDA [16] to leverage third-party toolchains and further manipulate the generated code, which also provides the opportunity for us to apply defense mechanisms to protect DNN programs from reversing-based model extraction attack. We implement FlatD on the top of LLVM to protect DNN programs from reversing-based model extraction attacks.

Thanks to the appearance of the DL compilers, the DNN model can be deployed on edge devices and low-power processors [51]–[53] having limited hardware resources with low overhead, while the popular DL frameworks like Tensorflow [8] only provide optimization for a narrow range of server-class GPUs. Giant AI providers like Amazon and Google also include DL compilers in their AI services to boost performance [17]–[19], [21]. As the need for DL-based services has increased, DL compilers play a more critical role in deploying DNN models, and the safety of DNN programs becomes increasingly vital.

B. Control Flow Obfuscation

Obfuscation is a technique that software developers have used for a long time to protect their intellectual property. The basic idea behind obfuscation is to transform a program into a new version that retains its functionality and semantics but hides its high-level structures [54]–[56]. Obfuscation significantly increases the difficulty of static program analysis [57]–[59], reversing engineering [60], [61], and also higher the bar of dynamic program analysis [62]–[66]. As an essential branch of obfuscation, control flow obfuscation aims to conceal the proper control flow and make the control flow graph as complicated as possible to raise the bar of countermeasures.

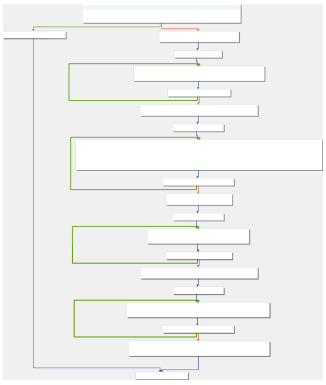
Opaque Predicates involve inserting conditional statements that always evaluate to true or false but appear complex and non-trivial to the analyzer, typically using mathematical or logical expressions that seem relevant but are redundant [67]. Opaque predicates can help insert bogus control flows, which seem viable but are never actually used, into the program, increasing the complexity of the CFG.

Control Flow Flattening is a technique that involves restructuring the Control Flow Graph of a program to make it appear as a single loop with a switch-case construct inside. This loop is controlled by a dispatcher, which decides which block of code to execute next based on the program's state [68]–[70]. Control Flow Flattening replaces the traditional hierarchical control flow structure, such as nested if-else statements and loops, with a single, linear structure. This significantly complicates understanding the program's execution flow, as it hides the original control structures and decision points within a seemingly linear flow.

III. Inspiration from Attacks

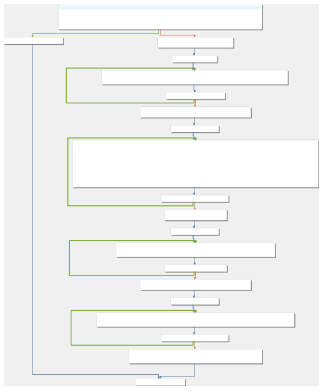
This section compares the methodology and design logic between state-of-the-art reversing-based model attacks (as shown in Table I) to determine their similarities and differences. Specifically, NNReverse [34] is a learning-based method that can fully recover the architecture from DNN programs compiled from TVM across platforms. DnD [32] implemented a cross-architecture DNN decompiler based on symbolic execution, which can fully recover the architecture and parameters of DNN programs compiled from TVM and Glow. Instead, BTD [33] focuses on the decompiling DNN Programs on x86 platforms. BTD can reconstruct models from DNN programs compiled from three DL compilers using the neural identifier model and dynamic analysis. LibSteal [35] partially recovers the DNN architecture using only a shared library. Although the target and methodology vary from frameworks, the logic and workflow align the same. The following characteristics must be recovered to rebuild the original DNN model: operator types, topology, and data (parameters and hyperparameters). We detail the review of each part below and summarize our findings at the end.

Operator type Recovery. The critical component of each attack framework is to recover operator types of DNN programs. The idea of operator-type recovery is also the most straightforward due to the unique computation pattern of each DNN operator. We take the softmax function as an example. As shown in Fig.2, VGG16 and ResNet50 are loaded from Keras Application Zoo and compiled by TVM with the configuration -O0. Fig.2a is the CFG and opcode sequence of the VGG16 softmax function, and Fig.2b is the CFG and opcode sequence of the ResNet50 softmax function. Both the CFGs and opcode sequences are obtained from Binary Ninja [71]. We can see that no matter which feature is compared, the VGG16 softmax function is almost identical to the ResNet50 softmax function, although they do not even have the exact hyperparameter. With this insight, all reversing attacks coincidentally infer operator type based on binary similarity. NNReverse combines the syntax representation (opcodes and operands) and topology representation (CFG) to train an embedding model and find the most similar function in the dataset based on the semantic representation. DnD uses symbolic execution to lift each operator function to an Abstract Syntax Tree (AST) and match it with a template AST to infer the operator type. BTD chose to train a model with a sequence of Atomic Opcodes from each DNN function and predict the operator type. LibSteal trained a representation model with loop structures extracted from each operator function and compared the victim operator function with functions in the dataset.



```
push push push push push
sub mov mov mov mov mov
mov mov mov call test
jne mov add pop pop pop
pop pop ret mov movss
mov maxss add jne shufps
movaps mov movaps subps
movaps shufps call
movaps movaps movhlps
call unpcklps movaps
movaps call movaps
movaps shufps call
movaps unpcklps unpcklpd
movaps add jne xorps mov
addss add jne shufps mov
movaps divps movaps add
jne mov mov mov call neg
sbb jmp
```

(a) VGG16 softmax function



```
push push push push push
sub mov mov mov mov mov
mov mov mov call test
jne mov add pop pop pop
pop pop ret mov movss
mov maxss add jne shufps
movaps mov movaps subps
movaps shufps call
movaps movaps movhlps
call unpcklps movaps
movaps call movaps
movaps shufps call
movaps unpcklps unpcklpd
movaps add jne xorps mov
addss add jne shufps mov
movaps divps movaps add
jne mov mov mov call neg
sbb jmp
```

(b) ResNet50 softmax function

Fig. 2: Comparison of CFG and Opcode sequence of softmax function between VGG16 (Fig.2a) and ResNet50 (Fig.2b) DNN Programs, compiled by TVM with optimization -O0.

TABLE I: Reversing-based Model Extraction Attacks. (★ stands for fully support and fully recover ☆stands for partial recover.)

Tool Name	Target Compilers			Platform Support			Results	
	TVM [11]	Glow [13]	NNFusion [12]	x86-64	ARM	AArch64	Architecture	Parameters
NNReverse [34]	★			★	★	★	★	
DnD [32]	★	★		★	★	★	★	★
BTD [33]	★	★	★	★			★	★
LibSteal [35]	★			★			☆	

Topology and Data Recovery. Attack frameworks have started to use different methodologies to recover the topology, parameters, and hyperparameters. NNReverse directly leverages the graph and parameter files generated along with the shared library file to retrieve the topology of DNN architecture. DnD first reconstructs the DNN topology structure by utilizing the sequence of DNN operator executions within the inference function and the data dependencies among the DNN operators. Then, it recovers the attributes and parameters according to the operator type and topology structure. BTD uses the Intel PinTool [72] to hook every call site as the operator function's inputs and outputs are transmitted via memory pointers in the function arguments. Subsequently, BTD seamlessly links the operator function using the identical memory address. As for data recovery, BTD applies taint analysis and symbolic execution to collected execution traces to infer the parameters and hyperparameters. LibSteal heuristically searches for possible topology combinations to link all the inferred operators. It also extracts the dimensions and partially recovers hyperparameters by analyzing the data flow of the DNN program.

Inspiration Our analysis of reverse-engineering-based extraction attacks reveals that the type of operator is a crucial element to protect. As shown in Fig.2, the unique computational patterns of each operator lead to specific features in the Control Flow Graph (CFG), particularly loop structures. It is widely acknowledged that operations within DNN models mainly involve matrix computations, which result in a nested loop structure. Such structures are significant features that attackers can exploit. Therefore, concealing the original CFG structure could be vital to defending against these attacks.

Algorithm 1 Flattening Algorithm

Input: P {DNN Program}

S_F = getAllFunctions(P)
for $F \in S_F$ **do**
  if F is necessary to be flattened **then**
    S_{BB} = getAllBasicBlocks(F)
    breakCFG(F)
    BB_{old} = getOldEntry(S_{BB})
    BB_{new} = createNewEntry(BB_{old})
    T = createDispatcher(BB_{new}) {Return the switch table that guides the control flow in new CFG}
    attachBBToDispatcher(T, S_{BB})

    $Salt$ = initializeSalt()
    F_{hash} = initializeHashFunc()
    updateSwitchVar(D, $Salt$, F_{hash})

    createBBAddrTable(S_{BB})
    encodeSwitchTable(T)
    F_{decode} = initializeDecodeFunc()
    updateDispatcher(T, F_{decode})
    inlineDispatcher(T)
  end if
end for

Output: Program with flattened operator functions.

IV. Design

A. Overview

Algorithm 1 uses pseudo code to represent the basic workflow of our defense framework, FlaD. First, for the given DNN

Program P, we extract and iterate all the functions (Line 1-2). Before applying the flattening, we check the specific function's necessity to increase the performance (Line 3). The rules are discussed in Section IV-E. Then, we break the original CFG and rebuild a new one using the dispatcher and switch table (Line 4-9), which follows the implementation steps of traditional Control Flow Flattening [70]. However, as shown in Section IV-B, traditional Control Flow Flattening is still vulnerable to static analysis.

To increase the robustness of the resulting program, we continue transforming using the following strategies. First, we hide the visible label by introducing the hashing method and a randomly chosen secret, `Salt` (Line 10-12). We provide more details in Section IV-C. Second, to further hide the loop structure, we create a table containing the address of each Basic Block and encode the switch table, which are both stored in the global variable list (Line 13-14). Then, we create a decode function (Line 15-16) to accomplish the indirect control flow. The function decodes the address of the corresponding basic block from the encoded switch table and switch variable. Finally, we inline the dispatcher to each basic block to hide the loop structure completely. Section IV-D takes an example of the final operator function to illustrate the whole process.

B. Init Flatten

Fig.3 shows the original CFG of the ReLu operator function from a simple MNIST convnet model [73] compiled from TVM [11] with optimization configuration -O0, and Fig.4 demonstrates the CFG after flattening. We omit unnecessary instructions for both figures and only retain the control flow-related instructions to make the figures clear and tidy. Moreover, the code in each basic block is represented in LLVM assembly language format (LLVM IR) because our defense mechanism is implemented on top of LLVM, and the big picture of the control flow structure stays the same across the compilation. As we can see in Fig.4, after flattening, all basic blocks in the original implementation share the same dominator, `LoopEntry`, and post-dominator, `LoopEnd` (except for Basic Block `BB.5`). Instead of the comparison result variable like `%cmp1`, the dispatch variable `%switchVar` takes over the functionality of manipulating the control flow. Although the CFG is already complex at this step, attackers can reconstruct the CFG by analyzing the operand value of the selection instruction in `BB.0` with the operand value of the switch instruction in `LoopEntry` to get the successors of each basic block. For example, as shown in Fig.4, we can infer that either `BB.1` or `BB.5` can be the next execution target after the execution of `BB.0` (marked as red).

C. Hide Visible Label

The first strategy to increase the resilience of traditional Control Flow Flattening is to hide the statically visible dispatch labels by introducing secret information and employing one-way cryptographic hashing. Fig.5 shows the CFG after hiding the visible label (marked as blue) of Fig.4. The value

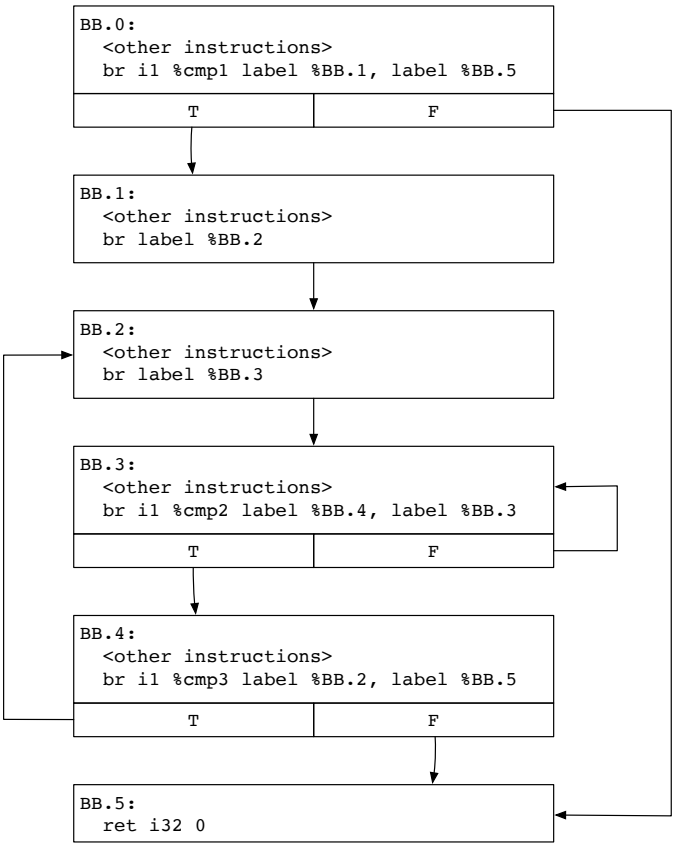


Fig. 3: Original Control Flow Graph of the ReLU operator function from MNIST compiled by TVM -O0. To simplify the graph, the figure only shows the control flow-related instructions in LLVM IR format.

of `%salt` is initialized at the Block `%Entry` (marked as red). Then, each time a new basic block needs to be dispatched, we compute the new label (`%hash`) with the `%salt` and the old label (`%switchVar`) through the hashing function (`@OBF_HASH()`) (marked as red). Finally, the code decides its successor basic block according to the value of `%hash`. As we can see, the value assigned in Block `%Entry` and `%BB.0` no longer appear in the dispatcher (`%LoopEntry`) (marked as blue). More specifically, we pick the hash function and compute the value of `%salt` wisely.

Hash function. We want to ensure the hash function has preimage resistance, which means finding any input that maps to a given output hash is computationally infeasible. In other words, given a hash value h, it should be tough to find any original input x such that $hash(x) = h$. This property is crucial for security, as it prevents attackers from reverse-engineering the hash to discover the original data. Preimage resistance keeps the original data secure and practically impossible to deduce even if the hash value is exposed. This feature perfectly fits our requirements since the hash values are used in the switch table to determine the following control flow, and we do not want the attackers to match it with the original data in each basic block. Moreover, the overhead of the hash

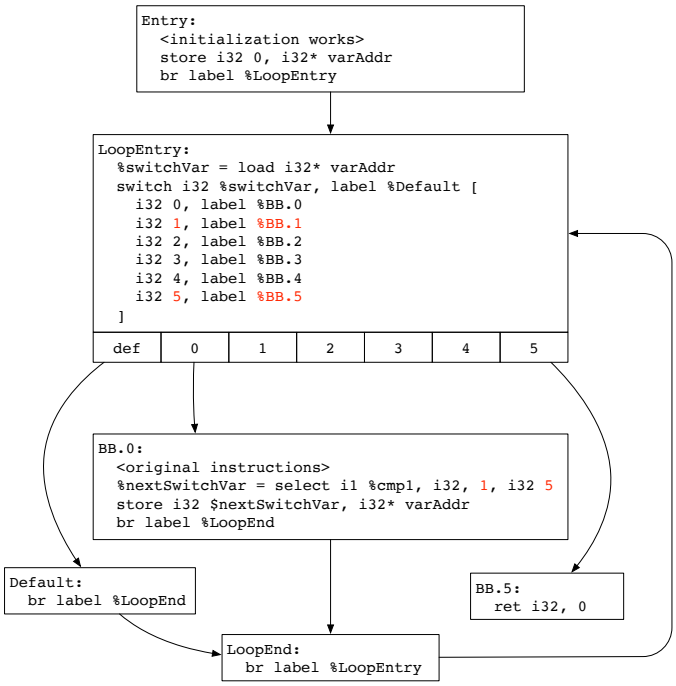


Fig. 4: CFG after initially apply Control Flow Flattening to Fig.3. The figure only shows part of the resulting CFG because the modifications of all basic blocks are similar except for basic blocks with label `BB.5` and `Default`, where `BB.5` is the exit block of this function, and `Default` is added by switch instruction to avoid assertion. Although we only include flows to `BB.0`, `BB.5` and `Default`, flows to other basic blocks still exist.

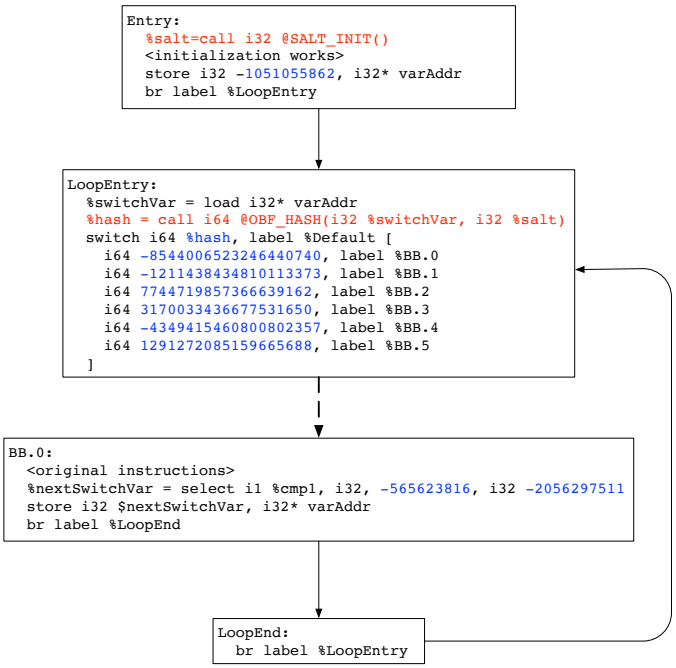


Fig. 5: CFG after hiding the visible label of Fig.4. To achieve the goal, we introduce a 32-bit secret number, `%salt` and initialize it (red) at the Basic Block `Entry`. Then we compute `%hash` using a one-way cryptographic hashing function (red) based on the old `%switchVar` value and `%salt`. Finally, we use the value of `%hash` to determine the control flow. In this case, the value assigned to `%switchVar` does not show in the switch table anymore (blue).

function is relatively low and does not affect the performance of DNN programs.

Salt computation. Although preimage resistance prevents attackers from inferring the original data from the hash value, they can directly compute the hash value from the original data because regardless of which hash function we choose, its body is included in the DNN program file. Therefore, we must introduce a secret value, `%salt`, and keep it unknown from attackers. The `%salt` is computed at run time and should not be easily statically revealed to make attackers unable to recover the original CFG statically. To achieve the goal, we leverage the concept of opaque predicate [67], [74]. An adequate opaque predicate should be resilient to static analysis. Therefore, for the computation of `%salt`, the compiler first randomly generates for each function. At run time, each bit of the `%salt` is computed by a "query," which can be an opaque predicate, making `%salt` computation statically obscure and dynamically confidential. Since the computation of `%salt` is a one-time job for each function, it also does not affect the time overhead of the DNN programs.

D. Hide Loop Structure

After hiding the statically visible dispatcher label, attackers have already struggled to recover the original CFG or gain important information from CFG. However, as the loop structure is retained in each function, the control flow still explicitly goes to each basic block and comes back to the dispatcher (`%LoopEntry` in Fig.5), which leaves a window for attackers to leak essential information of DNN programs. For example, attackers at least know all the original basic blocks contained inside a function. To conceal the CFG further, we aim to hide the loop structure by turning all the explicit flow into the implicit flow (i.e., indirect jump). The first thing we need to do is to remove the switch table. Thus, we encoded and embedded the original switch table in the global variable list. Besides, we create a table for each function to store the addresses of basic blocks. Then, we create a decode function that uses the hashed dispatcher label `%hash` to get the address of the following basic block. This way, the flows from the dispatcher (`%LoopEntry`) to each basic block are removed. For the last step, we remove the flows from basic blocks to the dispatcher (`%LoopEntry`) by inlining the dispatching part (mark as blue) into every flattened basic block. Fig.6 shows the part of the final result transformed from the original CFG from Fig.3. Attackers do not see a loop-like structure (or even part of it) in such a CFG because all basic blocks are floated in the DNN programs.

Additional Effort. While creating the table for basic block

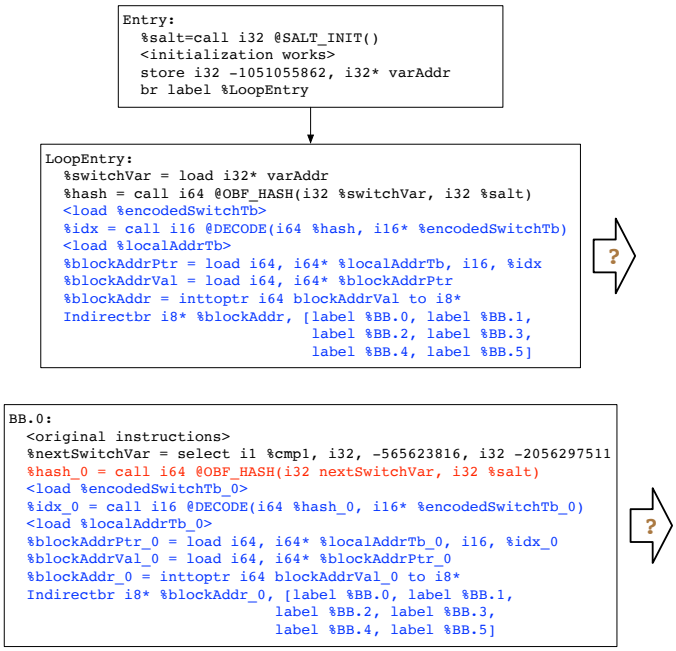


Fig. 6: CFG after hiding the loop structure of Fig.5. Instead of directly using hashed dispatcher label `%hash` to determine the subsequent control flow, we use it to decode the switch table and get `%idx` to retrieve the address of the target basic block in the basic block address table so that we can implicitly go to the following basic block. The potential candidate can be all the original basic blocks. Moreover, the main body of the dispatcher is inlined into each basic block, and the loop structure is completely removed.

TABLE II: A classification of the number of Basic Blocks in each operator function. Here **BB** refers to Basic Block.

# of **BB**s	**Operators**
less than 3	BatchFlatten
3	ReLu; BiasAdd
4	ReLu; BiasAdd; Add; Divide; Sqrt; Multiply; Negative;
more than 4	Dense; Pooling; Conv; ...

addresses, we added a mask to each address to prevent immediate disclosure of block addresses. The mask is a random noise randomly generated for each function. It will be deduced from the retrieved entry before it is used as the target address of the indirect jump.

E. Optimization

After transforming the original CFG from Fig.3 to the one shown in Fig.6, the time overhead definitely increases due to the increasing of instructions, indirect jump, and the function calls. Especially for the inner loop of a nested loop, one added instruction may be executed tens of hundreds of times during the execution, which leads to extreme additional overhead. To weaken the overhead introduced by the transformation, we propose two optimization methods: reduce the flattened functions and inline added function calls.

Reduce the Transformed Functions. Apparently, function transformation introduces additional time overhead. Reducing the number of transformed functions can improve performance. Applying the transformation to all functions is unnecessary because some lack helpful information. To determine the necessity of transformation of the functions, we provide a set of rules based on the knowledge of DNN programs:

- We only consider transforming the function with the computation. For example, the DNN programs that TVM generates contain functions that check the input and output data layout constraint before starting the computation. These functions typically have more basic blocks than the actual computation function. Applying transformation to such a function increases not only the time overhead but also the scale of the DNN program.
- We want to ignore the functions with few blocks because the transformation would be trivial in this case. However, if we refer to the number of such blocks as N, determining the value of N is tricky since we do not want to exclude the vital operator functions. Thus, we collect information about the number of basic blocks in each operator function as shown in Table II. We find that only activation functions like `ReLU`, element-wise arithmetic operators like `Add`, and operator function `BatchFlatten` have less than five basic blocks. Since these operators are unimportant, we ignore the functions with less than five basic blocks. Note that the DNN program with a high optimization level rarely contains such a function because operators can be fused into one operator function.

Inline Function Call. As described in Section IV-C and Section IV-D, our algorithm integrates two specialized function calls within the dispatch segment of the program. Recognizing that additional function calls may incur a notable runtime overhead is essential. This overhead primarily stems from the potential for extensive jumps between function calls, alongside the requisite establishment of stack frames, each of which demands considerable processing time and can impede overall system performance.

One effective strategy we employed to mitigate this overhead is inlining these function calls. However, while beneficial in reducing function call overhead, excessive inlining can also substantially increase the size of the function body. This expansion can negatively impact the time overhead, as larger function bodies may lead to increased compilation times and potentially hinder execution efficiency due to factors such as cache misses. Therefore, we want to inline function calls selectively. In our case, inlining is particularly beneficial in scenarios where the function calls are situated within a block that was the inner loop of a nested looping construct before optimization because they can invoked repeatedly during the runtime. To locate these function calls, we analyzed the loop structure of CFG during the compilation phase. This analysis enabled us to identify all the inner loops within the CFG accurately and inline them.

TABLE III: Statistics of DNN models. ResNet18 is loaded from three different frameworks: PyTorch (**P**), ONNX (**O**), and MXNet (**M**). All other models are loaded from Keras Application Zoo.

Model		# of Parameters	# of Operators
MNIST [73]		34,826	12
VGG16 [75]		138,357,544	23
VGG19		143,667,240	26
Xception [76]		22,910,480	134
ResNet50 [2]		25,636,712	177
ResNet101		44,707,176	347
ResNet152		60,419,944	517
MobileNet [77]		4,253,864	91
ResNet18	**P**	11,689,512	51
	O	11,699,112	69
	M	11,699,112	171

V. Evaluation

In this section, we evaluate FlatD by answering the following research questions (RQs) through empirical evaluation.

- ***RQ1: (Correctness)*** *After applying* FlatD *to the DNN programs from different DL frameworks, can they still apply the inference functionality properly?*
- ***RQ2: (Resilience)*** *Does* FlatD *effectively counterwork against the state-of-the-art reversing-based extraction attack?*
- ***RQ3: (Performance and Scale)*** *How does* FlatD *affect the performance and scale of the DNN programs?*

To explore the above RQs and provide a comprehensive evaluation, we evaluate FlatD with eight real-world pre-trained models and one self-trained model from four different frameworks and use the well-known obfuscator, O-LLVM [41] as the baseline. We only use the control flow flattening (-fla) obfuscation of O-LLVM to transform the program. All models are optimized and compiled by TVM [11] to generate DNN programs. FlatD and O-LLVM are both applied during the compilation and optimization.

A. Experimental Setup

We implement FlatD on the top of O-LLVM [15], [41] (version 8.0), primarily written in C++ with about 5K LOC. The current implementation obfuscates and evaluates DNN Programs in the ELF format on x86 platforms.

In the evaluation, we use TVM [11] as the state-of-the-art DL compiler to compile the models into DNN programs. For most of our evaluation, we used TVM v0.13.0 with the highest optimization level (O3) to compile the model. However, for the resilience evaluation, since the TVM version's iteration is relatively fast, in order to align the attack environment, we chose TVM v0.9.0 to generate the victim DNN programs with both the lowest optimization level (O0) and the highest optimization level (O3).

Table III shows all DNN models used for evaluation. All these models, except MNIST-convnet, are pre-trained models loaded from different frameworks. Among them, MNIST-convnet is a self-construct and self-trained model following the guide from [73], and ResNet18 is used to evaluate the effect of FlatD across the frameworks, so we loaded it from PyTorch [44], ONNX [78], and MXNet [10] respectively. The rest of the models are all loaded from the Keras application zoo [7], [79].

We perform all the evaluations on an Ubuntu 22.04 system on a machine with an Intel(R) Xeon(R) Silver 4114 CPU (2.20 GHz), 40 cores, and 219GB RAM.

B. (RQ1) Correctness

To evaluate the impact of FlatD on preserving the inference accuracy of DNN programs, we compare the inference outcomes of the original DNN programs with the counterparts transformed from FlatD and O-LLVM. The primary metric for this comparison is to check if the prediction results of the two models are identical. We assumed the original DNN program results were the ground truth and computed the identical percentage for the programs generated from FlatD and O-LLVM. To ensure a diverse and representative sample of test inputs, we sourced the test dataset from the ImageNet obtained from TorchVision [80]. We randomly select $10,000$ test inputs from this dataset as our evaluation set. Moreover, to illustrate the adaptability of FlatD, we evaluated ten versions. Within this set, three versions of ResNet18 were sourced from three distinct frameworks: PyTorch, ONNX, and MXNet. The remaining seven models were obtained from the Keras Application Zoo.

The summarized results are presented in Table IV. The findings from this table indicate that the inference results of the transformed DNN programs generated by FlatD align perfectly with those of the original programs across all the sampled inputs, which is under the expectation. However, we surprisingly found that after being obfuscated by O-LLVM, the MobileNet Program lost functionality. This outcome underscores the effectiveness of FlatD in preserving the original functionality and prediction accuracy of the DNN models.

C. (RQ2) Resilience

In this section, we evaluate the resilience of our defense framework. We first describe our evaluation setup (Section V-C1). Then, we show how FlatD influences the operator-type inference to the reversing-based extraction attacks (Section V-C2) by comparing the result between FlatD and O-LLVM.

1) Evaluation Setup: To align with the attack environment of prior reversing-based model extraction attacks [33], [35], we choose the TVM with released version v0.9.0 and use MNIST [81] and VGG16 [75] as two of our test models. We acquire MNIST by following the guide from [73] and VGG16 from Keras Application Zoo [79]. To test the effect of our defense mechanism on a more diverse set of DNN programs, we compile the two models above with two different optimizations (O0 and O3).

2) Operator Type Inference: As mentioned in Section III, all the reversing-based model extraction attacks fully or partially include four parts: Operator-type recovery, Topology recovery, Parameter recovery, and Dimensions (Attributes) Recovery. Since FlatD mainly focuses on manipulating the CFG

TABLE IV: Comparison of the inference results of the obfuscated DNN programs from O-LLVM and FlatD to the original DNN programs. Here **P** refers to PyTorch, **O** refers to ONNX, **M** refers to MXNet

	VGG16	VGG19	Xecption	ResNet50	ResNet101	ResNet152	MobileNet	ResNet18		
								P	**O**	**M**
FlatD	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
O-LLVM	100%	100%	100%	100%	100%	100%	0%	100%	100%	100%

TABLE V: The accuracy change in DNN operator inference before and after applying FlatD and O-LLVM. "N/A" means the attack framework does not support the DNN programs with the settings.

Attack Framework	MNIST						VGG16					
	TVM -O0			TVM -O3			TVM -O0			TVM -O3		
	Orig	O-LLVM	FlatD	Orig	O-LLVM	FlatD	Orig	O-LLVM	FlatD	orig	O-LLVM	FlatD
BTD [33]	100%	91.67%	50%	100%	92.31%	38.46%	100%	96.88 %	40.63%	100%	91.23%	64.91%
LibSteal [35]	100%	58.34%	25.00%	N/A			100%	40.63%	9.38%	N/A		

TABLE VI: Comparison between the performance of the transformed DNN programs generated by FlatD and O-LLVM and the original DNN programs. We use the time overhead of the original program as the baseline (100%). This table reports the time overhead of each DNN program running the inference to one specific picture from ImageNet and compares it to the original version to indicate the increasing time overhead. Here, **P** refers to PyTorch, **O** refers to ONNX, and **M** refers to MXNet.

	VGG16	VGG19	Xecption	ResNet50	ResNet101	ResNet152	MobileNet	ResNet18		
								P	**O**	**M**
O-LLVM	162%	166%	181%	170%	176%	176%	228%	147%	149%	146%
FlatD	188%	183%	280%	169%	163%	161%	231%	138%	138%	134%

TABLE VII: Comparison between the scale of the transformed DNN programs generated by FlatD and O-LLVM and the original DNN programs. We use the original DNN program size as the baseline (100%). This table shows the increased percentage between transformed DNN programs and original DNN programs. Here, **P** refers to PyTorch, **O** refers to ONNX, and **M** refers to MXNet.

Diff (%)	VGG16	VGG19	Xecption	ResNet50	ResNet101	ResNet152	MobileNet	ResNet18		
								P	**O**	**M**
O-LLVM	27.90	27.57	34.71	28.59	28.44	28.88	34.24	36.73	35.53	35.72
FlatD	17.43	17.48	18.76	12.91	12.91	12.90	17.04	22/45	21.93	21.94

of Operator functions, which is only related to the Operator Type recovery, we only evaluate our defense mechanism on how it can affect the inference of Operator Type of each reversing-based model extraction attack. Moreover, Operator-type recovery is the most essential and fundamental step in reconstructing the final models because, in some attacks [32], [34], [35], the recovery of other parts highly depends on the recovery of Operator-type.

We report the difference in the accuracy of DNN operator inference between the original version and the transformed version generated from O-LLVM and FlatD in Table V. We apply the same metrics to compute the accuracy of each attack used, where the prediction of operator type is regarded as correct only when the predicted result describes precisely the same operation as the ground truth. Since LibSteal [35] cannot deal with the situation when multiply operators are fused into one operator function, we only evaluate the DNN programs compiled with configuration -O3 on BTD [33]. As we can see, compared to the O-LLVM, FlatD can effectively reduce accuracy in DNN operator inference for each attack framework. Notably, while BTD can still achieve over 90% accuracy decompiling the program transformed by O-LLVM, FlatD decreases the accuracy to around 60% and even lower. Specifically, for MNIST with TVM -O0, the accuracy of BTD reduces to 50.00%; for VGG16 with TVM -O0, the accuracy of BTD reduces to 40.63%; for the optimization level -O3, BTD only gets 38.46% accuracy when targeting the transformed MNIST program compared to 92.31% targeting the MNIST program obfuscated by O-LLVM. BTD can achieve 61.54% accuracy when targeting the VGG16 program transformed by FlatD. When facing the LibSteal Attack, although O-LLVM has already significantly reduced the accuracy, FlatD can still outperform it (58.34% compared to 25.00% for MNIST TVM -O0 and 40.63% compared to 9.38% for VGG TVM -O0).

LibSteal and BTD rely heavily on the complete CFG information to infer the operator type. However, FlatD completely conceals the CFG by breaking the visible control flow between basic blocks. Even IDA Pro cannot extract the complete CFG without manual effort. On the other hand, although O-LLVM changes the control flow structure, the basic blocks are still visibly connected in the same chunk, which is still risky for the operator type to be inferred. We did not evaluate all the attacks mentioned in Section III due to the failure of setting up the attack, which is discussed in Section VI.

D. (RQ3) Performance and Scale

Runtime performance and scale are critical to a DNN program, especially when deploying the model on devices with

limited resources, like edge devices or low-power processors. Therefore, in this section, we compare the scale change between the original DNN programs and transformed versions (Section V-D2), as well as their performance of inference tasks (Section V-D1). Compared to O-LLVM, the programs generated by FlatD have a lower scale while maintaining a similar performance.

1) Inference Time Overhead: Since the time overhead is sensitive to the runtime environment and can fluctuate wildly due to unexpected reasons, we run each DNN program, including the programs generated by O-LLVM and FlatD and the original program, in an isolated environment to mitigate the influence of the runtime environment and reduce such fluctuation. Moreover, we randomly chose one picture from ImageNet and used it as input for all the inference tasks. For each DNN program, we run the inference 100 times and record the mean value as the evaluation result. As shown in Table VI, the results demonstrate that the time overhead introduced by FlatD is similar to O-LLVM for most DNN models except the Xecption model.

2) Program Scale: Table VII shows the scale change between the transformed DNN programs generated by FlatD and O-LLVM and the original DNN programs. Since the transformation process does not affect the parameter part of the DNN program, we only compare the scale change of the shared library files, which only contain the operator functions. To note, **Diff** $= (\frac{S_t}{S_o} - 1) * 100(\%)$ where S_t refers to the scale of a transformed DNN program and S_o refers to the scale of its original version. As we can see, the final percentages increased by FlatD to the DNN programs are much less than O-LLVM. While the size increased by FlatD can range less than 20%, the program generated from O-LLVM may increase over 30%.

VI. Related Work and Discussion

Existing defense framework. Besides the reversing-based model extraction attack, DNN models also need to face the threat from different interfaces. [23], [24], [26], [30], [82]. For example, attackers who target the model on the cloud usually use the input data sequence and prediction output pairs to infer the model information. In this case, Juuti et al. [83] propose PRADA, the first generic and effective tool to detect such a DNN model extraction attack. PRADA analyzes the sequence of API queries and raises the alarm if it deviates from benign behavior. Besides, Li et al. [84] propose a protection scheme against black-box model extraction attacks that uses a physical unclonable function(PUF) obfuscation technique. The scheme involves building a PUF on the user side and a corresponding PUF model on the service provider side. The proposed scheme allows legitimate users to accurately restore the model predictions while preventing attackers from extracting helpful information. Karchmer [85] discusses the possibility of providing provable security against model extraction attacks. To detect such an attack, the author proposes a theoretical framework for analyzing observational model extraction defenses (OMEDs) that examine the distribution of queries made by adversaries. They introduce the concepts of complete and sound OMEDs and show that achieving provable security against model extraction through these defenses is possible using average-case hardness assumptions for PAC learning. The framework provides a way to abstract current techniques used in the literature to achieve provable security.

Protect other characteristics. The primary purpose of FlatD is to protect the Control Flow Graph of the operator function so that attackers cannot infer the operator types accordingly. However, we do not protect other model characteristics, such as graph topology, operator attributes, and parameters, which should be protected from different views. For example, the data flow of DNN programs is also an essential feature attackers use to extract model information, like graph topology. Attackers [32], [33] utilize the data dependency between operator functions to determine the graph topology structure because the input data of the successor operator function and the output data of the predecessor operator function share the same memory address.

DNN program integrity and dynamic analysis In section IV-C, we introduce the secret information, $salt$, and one-way cryptographic hashing to secure the dispatcher label. We also consider the probability of such a strategy to secure the DNN program integrity (i.e., tamper-proof) to prevent the DNN program from program analysis with dynamic instrumentation (e.g., PinTool [72]). Theoretically, we can make another secret value related to each basic block's code, like the code chunk's hash value, and make it relevant to the indirect jump. This value can be calculated after executing each basic block. Thus, If the code is compromised, the value will not match the original one, and the execution sequence of the basic block will not follow the original control flow. Most likely, the code crashes.

Other attacks Although we failed to evaluate NNreverse [34] and DnD [32], the defense effect of FlatD towards these two attacks should be more significant than the result from BTD because they both highly rely on accurate CFG. The advantage of NNReverse compared to other binary mapping tools [86] is that it combines syntax and topology structure representation. However, after the transformation, the loop structure of CFG will be hidden, and all the CFG structures will look similar. In other words, the proposed advantage is cut off. DnD implemented its framework on the top of anger [61] and utilized symbolic execution to recover all the essential information for reconstructing the model. Nevertheless, after FlatD turns all branch instructions into indirect jumps, each operator function is transformed into a state-machine-like format, which is fatal to symbolic execution-based tools. Overall, FlatD can also effectively hinder these attacks.

VII. Conclusion

In this paper, we design and implement FlatD, an advanced defense framework for protecting DNN programs from reversing-based model extraction attacks based on control flow flattening. FlatD makes it challenging for attackers to recover the CFG statically and gain necessary information from DNN programs. Compared to the traditional Control Flow

Flattening, our evaluation shows that FlatD is an effective, adequate, and practical defense framework that prevents DNN programs from leaking essential information while ensuring their performance and program scale.

REFERENCES

[1] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013, pp. 6645–6649.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[3] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada, "An open approach to autonomous vehicles," *IEEE Micro*, vol. 35, no. 6, pp. 60–68, 2015.

[4] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[5] Google. Google ai and machine learning products. [Online]. Available: https://cloud.google.com/products/ai

[6] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[7] F. Chollet *et al.* (2015) Keras. [Online]. Available: https://keras.io

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

[9] A. Markham and Y. Jia, "Caffe2: Portable high-performance deep learning framework from facebook," *NVIDIA Corporation*, 2017.

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

[11] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[12] L. Ma, Z. Xie, Z. Yang, J. Xue, Y. Miao, W. Cui, W. Hu, F. Yang, L. Zhang, and L. Zhou, "Rammer: Enabling holistic deep learning compiler optimizations with rtasks," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 881–897.

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

[14] C. Leary and T. Wang, "Xla: Tensorflow, compiled," *TensorFlow Dev Summit*, 2017.

[15] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[16] N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.

[17] Amazon. (2021) Amazon sagemaker neo uses apache tvm for performance improvement on hardware target. [Online]. Available: https://aws.amazon.com/sagemaker/neo/

[18] A. Jain, S. Bhattacharya, M. Masuda, V. Sharma, and Y. Wang, "Efficient execution of quantized deep learning models: A compiler approach," *arXiv preprint arXiv:2006.10226*, 2020.

[19] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang, "Optimizing {CNN} model inference on {CPUs}," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 1025–1040.

[20] T. P. Morgan. Inside facebook's future rack and microserver iron. [Online]. Available: https://www.nextplatform.com/2020/05/14/inside-facebooks-future-rack-andmicroserver-iron/

[21] S. Ward-Foxton. (2021) Google and nvidia tie in mlperf; graphcore and habana debut. [Online]. Available: https://www.eetimes.com/google-and-nvidia-tie-inmlperf-graphcore-and-habana-debut

[22] T. Orekondy, B. Schiele, and M. Fritz, "Prediction poisoning: Towards defenses against dnn model stealing attacks," *arXiv preprint arXiv:1906.10908*, 2019.

[23] V. Duddu, D. Samanta, D. V. Rao, and V. E. Balas, "Stealing neural networks via timing side channels," *arXiv preprint arXiv:1812.11720*, 2018.

[24] L. Wei, B. Luo, Y. Li, Y. Liu, and Q. Xu, "I know what you see: Power side-channel attack on convolutional neural network accelerators," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 393–406.

[25] Y. Xiang, Z. Chen, Z. Chen, Z. Fang, H. Hao, J. Chen, Y. Liu, Z. Wu, Q. Xuan, and X. Yang, "Open dnn box by power side-channel attack," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 11, pp. 2717–2721, 2020.

[26] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn DNN architectures," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2003–2020.

[27] Y. Zhu, Y. Cheng, H. Zhou, and Y. Lu, "Hermes attack: Steal DNN models with lossless inference accuracy," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

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

[29] T. Orekondy, B. Schiele, and M. Fritz, "Knockoff nets: Stealing functionality of black-box models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4954–4963.

[30] W. Hua, Z. Zhang, and G. E. Suh, "Reverse engineering convolutional neural networks through side-channel information leaks," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[31] S. J. Oh, B. Schiele, and M. Fritz, "Towards reverse-engineering black-box neural networks," in *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*. Springer, 2019, pp. 121–144.

[32] R. Wu, T. Kim, D. J. Tian, A. Bianchi, and D. Xu, "DnD: A cross-architecture deep neural network decompiler," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2135–2152.

[33] Z. Liu, Y. Yuan, S. Wang, X. Xie, and L. Ma, "Decompiling x86 deep neural network executables," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7357–7374.

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

[35] J. Zhang, P. Wang, and D. 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 (ENASE)*, 2023.

[36] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. IEEE, 2009, pp. 248–255.

[37] J. Xiao, J. Hays, K. A. Ehinger, A. Oliva, and A. Torralba, "Sun database: Large-scale scene recognition from abbey to zoo," in *2010 IEEE computer society conference on computer vision and pattern recognition*. IEEE, 2010, pp. 3485–3492.

[38] J. D. M.-W. C. Kenton and L. K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of naacL-HLT*, vol. 1. Minneapolis, Minnesota, 2019, p. 2.

[39] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of machine learning research*, vol. 21, no. 140, pp. 1–67, 2020.

[40] C. Collberg. (2014) Tigress. [Online]. Available: https://tigress.wtf/

[41] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-llvm–software protection for the masses," in *2015 IEEE/ACM 1st International Workshop on Software Protection*. IEEE, 2015, pp. 3–9.

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

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

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

[45] N. Ketkar, "Introduction to keras," in *Deep learning with Python*. Springer, 2017, pp. 97–111.

[46] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 193–205.

[47] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[48] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye, "Alphaz: A system for design space exploration in the polyhedral model," in *Languages and Compilers for Parallel Computing: 25th International Workshop, LCPC 2012, Tokyo, Japan, September 11-13, 2012, Revised Selected Papers 25*. Springer, 2013, pp. 17–31.

[49] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen *et al.*, "Ansor: Generating {High-Performance} tensor programs for deep learning," in *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, 2020, pp. 863–879.

[50] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng, "Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 859–873.

[51] NXP. (2020) Nxp uses glow to optimize models for low-power nxp mcus. [Online]. Available: https://www.nxp.com/company/blog/glow-compiler-optimizesneural-networks-for-low-power-nxp-mcus:BL-OPTIMIZES-NEURAL-NETWORKS

[52] OctoML. (2021) Octoml leverages tvm to optimize and deploy models. [Online]. Available: https://octoml.ai/features/maximize-performance/

[53] Qualcomm. (2020) Qualcomm contributes hexagon dsp improvements to the apache tvm community. [Online]. Available: https://developer.qualcomm.com/blog/tvm-open-source-compiler-now-includesinitial-support-qualcomm-hexagon-dsp

[54] A. Balakrishnan and C. Schulze, "Code obfuscation literature survey," *CS701 Construction of compilers*, vol. 19, 2005.

[55] J. Cappaert, "Code obfuscation techniques for software protection," *Katholieke Universiteit Leuven*, pp. 1–112, 2012.

[56] H. Xu, Y. Zhou, Y. Kang, and M. R. Lyu, "On secure and usable program obfuscation: A survey," *arXiv preprint arXiv:1710.01139*, 2017.

[57] Z. Wang, P. Wang, Q. Bao, and D. Wu, "Source code implied language structure abstraction through backward taint analysis," in *Proceedings of the 18th International Conference on Software Technologies (ICSOFT)*. SCITEPRESS-Science and Technology Publications, 2023.

[58] P. Wang, J. Zhang, S. Wang, and D. Wu, "Quantitative assessment on the limitations of code randomization for legacy binaries," in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2020, pp. 1–16.

[59] F. Xiao, J. Zhang, J. Huang, G. Gu, D. Wu, and P. Liu, "Unexpected data dependency creation and chaining: A new attack to sdn," in *2020 IEEE symposium on security and privacy (SP)*. IEEE, 2020, pp. 1512–1526.

[60] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 627–642.

[61] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 8–9.

[62] R. Zhong, Y. Chen, H. Hu, H. Zhang, W. Lee, and D. Wu, "Squirrel: Testing database management systems with language validity and coverage feedback," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 955–970.

[63] Y. Chen, R. Zhong, Y. Yang, H. Hu, D. Wu, and W. Lee, "μfuzz: Redesign of parallel fuzzing using microservice architecture," in *Proceedings of the 32nd USENIX Security Symposium (USENIX Security'23)*, 2023.

[64] Y. Chen, R. Zhong, H. Hu, H. Zhang, Y. Yang, D. Wu, and W. Lee, "One engine to fuzz'em all: Generic language processor testing with semantic validation," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 642–658.

[65] Q. Bao, Z. Wang, X. Li, J. R. Larus, and D. Wu, "Abacus: Precise side-channel analysis," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 797–809.

[66] Q. Bao, Z. Wang, J. R. Larus, and D. Wu, "Abacus: a tool for precise side-channel analysis," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2021, pp. 238–239.

[67] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

[68] C. Wang, J. Hill, J. Knight, and J. Davidson, "Software tamper resistance: Obstructing static analysis of programs," Technical Report CS-2000-12, University of Virginia, 12 2000, Tech. Rep., 2000.

[69] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 290–299.

[70] T. László and Á. Kiss, "Obfuscating c++ programs via control flow flattening," *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, vol. 30, no. 1, pp. 3–19, 2009.

[71] V. . Inc. Binary ninja. [Online]. Available: https://binary.ninja/

[72] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[73] F. Chollet. Simple mnist convnet. [Online]. Available: https://keras.io/examples/vision/mnist_convnet/

[74] D. Dolz and G. Parra, "Using exception handling to build opaque predicates in intermediate code obfuscation techniques," *Journal of Computer Science & Technology*, vol. 8, 2008.

[75] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[76] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1251–1258.

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

[78] J. Bai, F. Lu, K. Zhang *et al.* (2019) Onnx: Open neural network exchange. [Online]. Available: https://github.com/onnx/onnx

[79] Keras. Keras applications. [Online]. Available: https://keras.io/api/applications/

[80] Meat. Torchvision datasets. [Online]. Available: http://pytorch.org/vision/main/datasets.html

[81] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[82] L. Batina, S. Bhasin, D. Jap, and S. Picek, "CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 515–532.

[83] M. Juuti, S. Szyller, S. Marchal, and N. Asokan, "Prada: protecting against dnn model stealing attacks," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 512–527.

[84] D. Li, D. Liu, Y. Guo, Y. Ren, J. Su, and J. Liu, "Defending against model extraction attacks with physical unclonable function," *Information Sciences*, vol. 628, pp. 196–207, 2023.

[85] A. Karchmer, "Theoretical limits of provable security against model extraction by efficient observational defenses," in *2023 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*. IEEE, 2023, pp. 605–621.

[86] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 472–489.