DnD: A Cross-Architecture Deep Neural Network Decompiler

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

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

Abstract

The usage of Deep Neural Networks (DNNs) has steadily increased in recent years. Especially when used in edge devices, dedicated DNN compilers are used to compile DNNs into binaries. Many security applications (such as DNN model extraction, white-box adversarial sample generation, and DNN model patching and hardening) are possible when a DNN model is accessible. However, these techniques cannot be applied to compiled DNNs. Unfortunately, no dedicated decompiler exists that is able to recover a high-level representation of a DNN starting from its compiled binary code.

To address this issue, we propose DNDD, the first compiler- and ISA-agnostic DNN decompiler. DND uses symbolic execution, in conjunction with a dedicated loop analysis, to lift the analyzed binary code into a novel intermediate representation, able to express the high-level mathematical DNN operations in a compiler- and ISA-agnostic way. Then, DND matches the extracted mathematical DNN operations with template mathematical DNN operations, and it recovers hyper-parameters and parameters of all the identified DNN operators, as well as the overall DNN topology. Our evaluation shows that DND can perfectly recover different DNN models, extracting them from binaries compiled by two different compilers (Glow and TVM) for three different ISAs (Thumb, AArch64, and x86-64). Moreover, DND enables extracting the DNN models used by real-world micro-controllers and attacking them using white-box adversarial machine learning techniques.

1 Introduction

Deep Neural Networks (DNNs) have become a fundamental component for a number of usage scenarios, ranging from cloud services to IoT applications [13, 38]. While the training of DNNs is usually performed on GPUs or TPUs [17], DNNs can be deployed on general purposes CPUs and MCUs in a variety of devices, such as IoT devices and embedded systems. In these cases, DNNs are typically *compiled* into binary code using dedicated compilers, such as TVM [7] and Glow [30], exploiting local instruction set architectures' (ISAs) features and performance optimizations.

An increasing usage of DNN compilers is in edge devices, such as micro-controllers [1, 16, 38, 44, 45] and mobile phones [53]. For instance, micro-controllers use compiled DNN binaries, like the NXP Semiconductors EdgeReady Solution [43], to perform diverse tasks such as secure face recognition, speech recognition, and voice control. A recent forecast estimates that 98% of edge devices will have intelligence features, typically powered by DNNs, by 2025 [38].

While several tools are available to decompile binary code back to its original source code [5, 21, 33, 57], no existing approach can recover a high-level description of a compiled DNN. As the ability to decompile binary code has enabled several security applications (and decompilers have become part of the standard tool set of security analysts), likewise extracting DNN models from their compiled binary code enables a variety of security applications, including DNN model extraction and analysis (both manual and automatic), white-box adversarial sample generation, white-box DNN backdoor detection, and DNN model patching and hardening [6, 18, 50, 51]. Unfortunately, in all these usage scenarios, a source-level representation of the decompiled code obtained by applying traditional decompilers does not offer much help to an analyst, because such decompilers cannot capture the mathematical semantics of compiled DNN models.

In this paper, we propose DND, the first compiler- and ISA-agnostic deep neural network decompiler capable of extracting DNN models from compiled binaries. Specifically, working with a compiled DNN model, DND can precisely recover its parameters, hyper-parameters, and topology, and express the decompiled model in a high-level representation (as the one in Figure 1), encoded in the ONNX modeling language [9].

DND is based on the three following novel techniques. First, it uses symbolic execution in conjunction with a dedicated loop analysis to capture precise mathematical formulas representing how different DNN operators process the received data. Second, DND uses a novel intermediate representation (IR) to express the high-level mathematical DNN operations in a compiler- and ISA-agnostic way. Third, DND identifies the type and location of the DNN operators in a target binary by matching the extracted mathematical operations with template mathematical DNN operations, recovering hyper-parameters and parameters of all the identified DNN operators, as well as the overall network topology.

Our evaluation shows that DnD is both *generic* and *accurate*. It supports decompiling different DNN models compiled by two different compilers for three different ISAs, without requiring manual effort. Moreover, the decompiled DNN models are structurally equivalent to the original ones, and, after re-compiling the decompiled DNNs, the generated binaries classify samples exactly as the original binaries. We further demonstrate that DnD can successfully decompile a DNN binary used by a real-world micro-controller, and the recovered DNN model can be used to boost adversarial attacks against the original DNN, enabling the usage of white-box attacks, in place of less efficient black-box ones.

In summary, our main contributions are as follows:

- We design and implement DnD, the first compiler- and ISA-agnostic decompiler for compiled DNN models. DnD can decompile a (stripped) DNN binary to recover the full details of the compiled DNN model and represent them using the ONNX high-level modeling language.
- We design a dedicated IR to represent each DNN operator and develop a novel technique that uses symbolic execution to lift the DNN binary to IR expressions. The extracted IR expressions are then matched against template expressions to identify different operators used by a compiled DNN, recovering their hyper-parameters, parameters, and the overall network topology.
- We evaluate DnD on three ISAs (Thumb, AArch64, and x86-64), two DNN compilers (Glow and TVM), and three widely-used DNN models (MNIST [59], ResNet v1 [20], and MobileNets v2 [40]). The results show that DnD can fully and accurately recover DNN models from the compiled binaries across different ISAs, compilers, and models. We further showcase how the extracted DNN model, decompiled by DnD from a DNN binary running on a real-world micro-controller, can be used to boost adversarial attacks [6, 34], enabling more effective white-box attacks.

Our artifacts are available at https://github.com/purseclab/DnD.

2 Background and Motivation

2.1 Deep Neural Networks

Deep Neural Networks (DNNs) are a class of machine learning (ML) algorithms that use cascaded DNN operators for feature extraction and transformation. Figure 1 shows a snippet of ResNet v1 model [20] represented in the ONNX format [9], which is the open standard for ML interoperability developed by Linux Foundation. Terminologies used in this paper are described as follows:

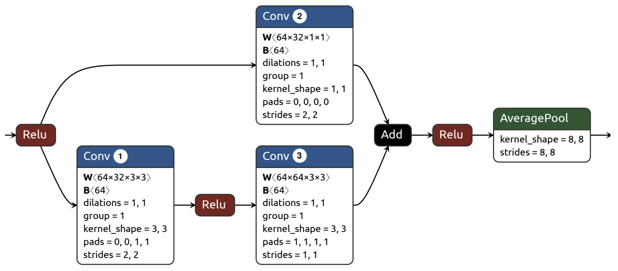


Figure 1: A snippet of ResNet v1 model in the ONNX format

Training and Inference. DNNs are used in two phases: training and inference phases. The training phase feeds labeled data to a DNN and updates its parameters based on the erroneous output results. This phase is computationally heavy and generally takes a long time; thus, it is usually running on DNN accelerators (e.g., GPUs and FPGAs). During the inference phase, a trained DNN predicts the labels of input data. Since the inference phase is typically not as computationally heavy as the training phase, they usually run on CPUs, especially on edge devices [53].

DNN Operators. DNN operators are the building blocks of DNNs. A DNN operator takes the output of previous operators as its input (or the input to the DNN model when there are no previous operators), and computes its output based on its operator type and its parameters. There are various types of DNN operators with different high-level semantics, as shown in Figure 1. For instance, there are feature extraction operators (e.g., `Convolution`), down-sampling operators (e.g., `Average-pool`), and activation operators (e.g., `Relu`). As DNN models have been advanced, new DNN operators have been introduced. There are 174 different DNN operators defined in ONNX, and this number is still growing [11].

DNN Hyper-parameters and Parameters. DNN hyper-parameters can be classified into two categories: (1) the *algorithm hyper-parameters*, which are only used during the training phase, and do not influence the inference phase. Examples of the *algorithm hyper-parameters* are learning rate and regularization factor. (2) the *model hyper-parameters* that define the network structure and how the operators function, which include the following:

1. Total number of operators and the type of each operator.
2. The DNN topology. The connections between operators can be either sequential or non-sequential. Sequential connection denotes that the operator takes only the output of the prior DNN operator as the input, while non-sequential connection includes shortcuts and branches [20, 58]. For example, shortcut connection [20] (the edges connecting the `Conv2` in Figure 1) feeds one DNN operator's output directly as the input of another DNN operator while skipping one DNN operator.
3. The attributes of each operator that define its detailed se-

mantics. For example, the attributes of a `Convolution` (i.e., `Conv`) operator, as shown inside each operator in Figure 1, include the shape of kernel (i.e., filter), the pad size, and the stride size.

The *parameters* are the variables that are learned during the training process (e.g., weights and bias).

We note that *model hyper-parameters* and *parameters* are targets that DND aims to recover, unlike *algorithm hyper-parameters* which are involved only in the training phase.

2.2 DNN Compilers

Different DNN compilers exist to ease the deployment of a DNN model on various devices. These compilers include Glow [30] by Facebook, TVM [7] by Apache, XLA [14] by Google, and NNFusion [27] by Microsoft. Nowadays, they have been adopted in many real-world products, including smart edge devices. For instance, NXP Semiconductors integrate a customized Glow compiler into their SDK to support DNN model deployment on low-power microcontrollers [45].

Frontend and Backend. DNN compilers commonly consist of a frontend and a backend component. The frontend transforms a DNN model into a high-level intermediate representation (IR) and performs hardware-independent optimizations, such as operator fusion [32]. Operator fusion combines the loop bodies of two adjacent operators leveraging the operator type and DNN topology, where the second operator is usually an activation operator (e.g., `Relu`). For instance, the `Conv1` and the following `Relu` in Figure 1 can be combined. Then, the backend transforms a high-level IR to a low-level IR and performs hardware-specific optimizations, including vectorization and loop-related optimizations (e.g., loop unrolling). Finally, the backend emits machine code from the optimized low-level IR.

Compilation Scheme. Compilation generates DNN binaries (i.e., the binary programs where a compiled DNN model is embedded). We can classify the compilation schemes of DNN compilers into two categories: interpreter-based and ahead-of-time (AOT) compilation schemes [26].

Interpreter-based compilers generate DNN binaries whose DNN models are configured at runtime. They usually produce two artifacts: a DNN configuration file describing the DNN model (e.g., TFLite [15]) and a runtime library that contains all the DNN operator implementations. At runtime, a generic interpreter reads the DNN configuration file, iterates through DNN operators, and invokes the corresponding DNN operator implementations [16]. This approach has two disadvantages: (1) it introduces space and time overhead because of the need to include the runtime library and parse the DNN configuration file dynamically; (2) it misses opportunities to optimize the invoked DNN operator implementations based on specific operator instances' attributes (e.g., dimensions, padding size).

On the contrary, during compilation, AOT compilers *specialize* the operator implementation for the specific compiled operator instance's context, such as its attributes (e.g., kernel shape), and they use general-purpose compiler backends (e.g., LLVM, GCC) to generate a self-contained executable. For instance, in Figure 1, although the `Conv1` and `Conv2` are both of the same operator type (i.e., `Conv`), they are compiled into different binary functions with different control flows and memory access patterns, due to their different attributes (e.g., kernel shape). Thanks to the low overhead, the popular DNN compilers (e.g., Glow, TVM, XLA, and NNFusion) support AOT compilation to deploy a DNN on embedded systems [14, 45] with limited hardware resources.

We note that AOT DNN compilers, such as Glow [30] and TVM [1], compile a DNN model to a binary module exposing an inference function. An application feeds the input data to this inference function and obtains the predicted label as output.

2.3 DNN Attacks

Model Extraction Attacks. Model extraction attacks, which reveal a DNN's model hyper-parameters and/or parameters, can be classified into two categories: algorithm-level extraction and architecture-level extraction attacks. The algorithm-level approaches [46, 49, 50] conduct model extraction by repeatedly querying a black-box DNN model and then retraining a DNN using the querying results. However, such approaches require prior knowledge of a DNN's model hyper-parameters and significant computational resources [49], prohibiting them from efficiently extracting DNN models, especially from the embedded systems. On the other end, the architecture-level approaches [3, 22, 29, 52, 58, 61] leverage the architecture-level information (e.g., cache side-channel) exposed by the hardware. The architecture-level approaches, however, either only obtain a partial DNN model, or require co-locating onto the same processor chip the victim process is running on.

Adversarial Attacks. Adversarial attacks, which induce a DNN to misclassify an input, can be classified into two categories: black-box attacks and white-box attacks. Black-box attacks [4, 34] first acquire a substitute model either from the DNN models with a similar task or from training with the querying data of the victim model, then generate the adversarial examples based on the substitute model. Although black-box attacks do not require attackers to have prior knowledge of the victim DNN (e.g., model hyper-parameters and parameters), they are ineffective and time-consuming. Compared with black-box attacks, white-box attacks [6, 48] are more effective and efficient [6, 34] by leveraging the prior knowledge of the victim DNN model.

2.4 Decompiler

Decompilers are designed to reconstruct a high-level language representation (typically in C pseudocode) from program binaries. There are many works on C/C++ decompilers from both academia [5, 12, 19, 25, 41, 56, 57] and industry [21, 33].

Despite the existing general-purpose decompilers can decompile the binary programs to (usually not easily understandable, nor fully correct) C statements, they lack the capabilities to lift the binary code of vectorized mathematical calculations into high-level DNN operators (e.g., ONNX operators [11]) and recover the DNN topology. These limitations hinder their usefulness for the security analysis of DNN binaries. On the contrary, DnD can decompile a DNN model embedded in the binary program and generate a high-level representation (i.e., in the ONNX format [9]), including both the model hyper-parameters and parameters of the embedded DNN model.

3 Scope

In this section, we describe the input/output of DnD, and the standard and realistic assumptions on which DnD relies.

Input. DnD supports (stripped) DNN binaries (i.e., the binary programs where a compiled DNN model is embedded) compiled by the AOT compilation scheme running on CPU without hardware accelerators. This configuration is common on edge devices [53]. DnD does not support DNN binaries compiled by interpreter-based compilation schemes because of the following reasons: (i) DNN binaries compiled by the interpreter-based compilation scheme usually accompany the DNN configuration files. We can easily infer DNN models from those files because they contain the information on the model hyper-parameters and parameters of the deployed DNN model. (ii) static analysis cannot extract DNN models from DNN binaries compiled by the interpreter-based compilation without the DNN configuration file because DNNs are configured dynamically. Furthermore, DnD does not support the DNN binaries running on DNN accelerators because DNN accelerators have very diverse ISAs, and they are not supported by the general-purpose disassemblers. Section 9 discusses more details why DnD does not support decompiling DNN binaries on accelerators.

Output. DnD can decompile a DNN model embedded in an input binary. The output is in the ONNX format [9] (e.g., Figure 1) including the DNN model's model hyper-parameters and parameters. We can use this output to reveal the DNN model's details and conduct security analysis, such as model extraction, adversarial examples discovery, and model hardening. DnD does not recover the algorithm hyper-parameters (defined in Section 2.1) because they neither affect the inference process nor are recoverable from the binary.

Assumptions. DnD relies on the following assumptions:

1. We have access to a DNN binary (e.g., dumping DNN binaries running on an embedded system).
2. The control-flow graph (CFG) recovery is reliable. Our evaluation shows that the recovered CFGs, though imprecise, are sufficient enough for our decompilation purpose.
3. DNN compilers do not use obfuscation technique. To the best of our knowledge, we are not aware of obfuscated DNN binaries, and de-obfuscating binaries is an orthogonal direction.

4 Challenges and Solutions

DNN binary decompilation imposes several unique challenges compared to C/C++ binary decompilation. We enumerate three major challenges and provide a summary of how DnD addresses them.

Challenge 1: Diverse Compilers and Architectures. DNN models are compiled with various DNN compilers and onto different instruction set architectures (ISAs), such as Arm Thumb, Arm AArch64, and x86-64. Each combination generates totally different binary code, regarding their control flows and data flows, as demonstrated in decompiled code samples in Appendix C. Therefore, simple pattern matching, using either binary code or decompiled code generated by generic decompilers, is not suitable to recover the DNN model hyper-parameters in a generic, compiler- and ISA-agnostic way.

Solution 1: We use a dedicated IR which is able to represent each DNN operator as an operator summary, including an AST of algebraic operations. DnD first identifies the location of each DNN operator in a DNN binary and then uses selective symbolic execution to generate an operator summary with an AST of algebraic operations of each DNN operator, which is represented with the IR we design. Because a DNN operator has the same mathematical semantic even with different DNN compilers and ISAs, and our IR and operator summary are able to capture the mathematical semantic, DnD can identify them in a compiler- and ISA-agnostic manner.

Challenge 2: Vectorized Mathematical Computation and Complicated Loop Structure. DNN operators, as tensor operations, are always implemented and compiled as nested loops with vectorized mathematical computations inside the loop bodies. The existing general-purpose decompilers (e.g., Hex-Rays) do not currently recognize vectorized mathematical computations, leading to decompiled code containing long loop bodies and excessive bitwise operations, as shown in Appendix C. Moreover, the complicated control flow and the huge loop index range of nested loops hinder the usage of symbolic execution from generating the operator summary of each DNN operator, because of the path explosion problem.

Solution 2: We use a dedicated symbolic execution to capture the semantics of vectorized mathematical computations, by keeping track of the symbolic constraints related to each DNN operator's input and output, and lifting such symbolic constraints to the operator summary represented with our IR. To solve the path explosion issue, DnD only executes one iteration of each loop, leveraging the fact that the loops in DNN operators usually implement tensor operations, and have no data dependencies between each iteration. Therefore, executing one iteration of each loop can sufficiently capture the semantics of a DNN operator. To enable such dedicated symbolic execution, we conduct a loop analysis to identify each loop's induction variables (i.e., loop index).

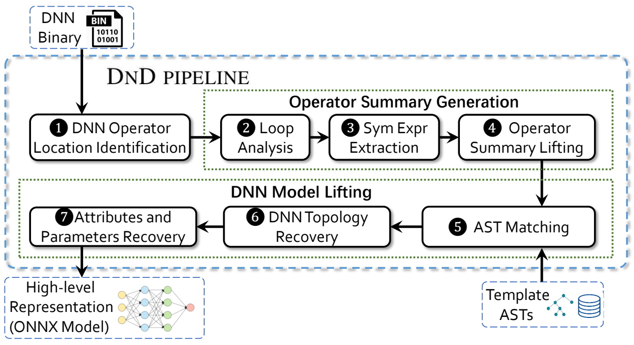

DNN Binary
DND PIPELINE
Operator Summary Generation
❶ DNN Operator Location Identification
❷ Loop Analysis
❸ Sym Expr Extraction
❹ Operator Summary Lifting
DNN Model Lifting
❼ Attributes and Parameters Recovery
❻ DNN Topology Recovery
❺ AST Matching
High-level Representation (ONNX Model)
Template ASTs

Figure 2: Pipeline of DND

Challenge 3: Scalable Support to DNN Operators. As mentioned in Section 2.1, there are many different DNN operators. Therefore, it is not scalable to manually design a heuristic for each DNN operator to identify its DNN operator type and extract its parameters.

Solution 3: For an unknown DNN operator, we match the AST included in its generated operator summary with operators' template ASTs to identify its operator type. DND first builds a template AST database, which maps each DNN operator to its corresponding AST. Specifically, DND leverages an up-to-date DNN compiler to compile each DNN operator and generate the template AST of each compiled DNN operator, represented with the IR we design. Then, given the previously generated operator summary of an unknown DNN operator, DND matches its AST to one of the AST in the template AST database, and determines the type of the unknown DNN operator.

5 System Design

DND's workflow is composed of three components, as illustrated in Figure 2. Specifically, these three components are (1) *DNN Operator Location Identification*, (2) *Operator Summary Generation*, and (3) *DNN Model Lifting*.

In the first stage, DND recovers the control flow graph (CFG) and identifies the location of inference function and DNN operators from the input (stripped) DNN binary (Step ❶in Figure 2, details in Section 5.1).

Next, DND generates operator summary of each DNN operator (Section 5.2). To do so, DND first conducts loop analysis (Step ❷) to identify loops' information. Such information is essential for further analysis. Then, DND leverages loop's information to perform selective symbolic execution that extracts the output of a DNN operator as symbolic expressions of its input and parameters (e.g., $output[i] = input[i] * param[i]$), which capture the mathematical semantic of a DNN operator (Step ❸). The extracted symbolic expressions are then lifted to the operator summary in our IR format (Step ❹). The operator summary of a DNN operator includes the ASTs and other information extracted from Step ❷and Step ❸. Note that DND also generates template ASTs through the aforementioned operator summary generation procedure (Step ❷, ❸ and ❹) that will be used in the next step (Section 5.3).

After the operator summary generation, the next step is to lift each operator summary to a DNN operator and convert it to a high-level DNN representation (i.e., an ONNX model [11]) (Section 5.4). Specifically, DND first matches the AST in each operator summary with a template AST to determine its DNN operator type (Step ❺). Then, DND recovers the DNN topology by identifying the data dependencies between DNN operators (Step ❻). Finally, DND recovers each DNN operator's attributes and parameters leveraging the identified DNN operator type and DNN topology, and converts the fully-recovered DNN model to an ONNX model (Step ❼).

5.1 DNN Operator Location Identification

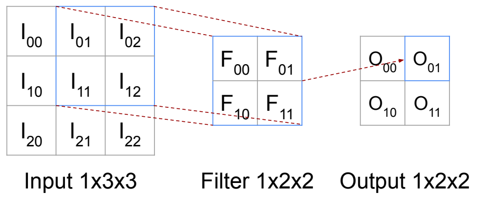
In this step, DND identifies the locations of the inference function and the DNN operators. Since DNN operators are essentially tensor computations, they are implemented and compiled as multiple nested loops with a number of numerical computations inside. Furthermore, DNN operators reside in either the inference function or its callee functions. DND leverages these two properties to identify the locations of DNN operators and the inference function.

Specifically, DND first identifies the locations of the functions with possible tensor computation (i.e., containing two or more nested loops or invoking math functions in the standard library) as DNN operator candidates. Then, DND collects the caller functions of each function in the candidate list. Among these caller functions, the one calling most candidates is considered as the "inference function" (i.e., acting as the DNN binary's dispatch function). Finally, DND filters out the candidate functions that are not the callees of the inference function.

5.2 Operator Summary Generation

After identifying the locations of DNN operators, DND extracts the symbolic expressions from each DNN operator and lifts them to operator summary in the IR we design. To do so, DND first conducts loop analysis for each DNN operator (Step ❷ described in Section 5.2.1). Next, leveraging the loop analysis results, DND extracts the output of each DNN operator as symbolic expressions of its input and parameters, using our customized selective symbolic execution algorithm (Step ❸ described in Section 5.2.2). Then, DND lifts the symbolic expressions, which contain a semantic of each DNN operator, to the operator summary (Section 5.2.4), in the IR we design (Step ❹ described in Section 5.2.3).

In the rest of this section, we refer, as an example, to the `Conv` operator depicted in Figure 3 to explain how DND generates operator summary. For simplicity, we assume the example `Conv` operator is not compiled with optimizations (e.g., loop unrolling and operator fusion) and without advanced attributes (i.e., padding and striding). We will explain how we handle common compiler optimizations and advanced attributes in



(a) `Conv` operator

```
1 void Conv(PTR* input, PTR* filter, PTR* output){
2   for(i=0;i<2;i++) // output width index
3     for(j=0;j<2;j++) // output length index
4       for(u=0;u<2;u++) // filter width index
5         for(v=0;v<2;v++) // filter length index
6           output[i][j]+=input[i+u][j+v]*filter[u][v]
7 }
```

(b) Simplifed decompiled code of `Conv`

```
output[i][j]+=input[i+u][j+v]*filter[u][v]
```

(c) Extracted symbolic expression of `Conv`

```
addr: output[i][j]
expr: Sum(element=Mul(input[i+u][j+v],
                      filter[u][v]),
         index=(u, v))
IVs: (u, init=0,inc=1,count=2)
     (v, init=0,inc=1,count=2)
     (i, init=0,inc=1,count=2)
     (j, init=0,inc=1,count=2)
```

(d) Generated operator summary

Figure 3: Operator summary generation of `Conv`

Sections 5.2.4 and 5.4.3, respectively.

5.2.1 Loop Analysis

The main goal of loop analysis is to identify the information on the basic induction variable (IV), or informally loop index variable of each loop in a DNN operator. For example, the `i`, `j`, `u`, `v` in Figure 3b are the IVs. The output of this analysis includes IVs, their initial values, their step sizes (i.e., the increment constant) and their loop count (i.e., how many iterations the loop is supposed to be repeatedly executed). Then, DND will use these IVs information to extract the symbolic expressions in Section 5.2.2.

To do so, DND first recovers loops' structural information in each DNN operator, including the entry points, the exit points and the break edges (denoting the edges in CFG that jump out of the current loop). Then, DND leverages the IVs' properties in the binary program that we observe to recover the IVs, which are the following: (i) IVs are initialized with constants and loaded into general registers in the loop's entry block (e.g., $i = 0$ in Line 2 in Figure 3b); (ii) IVs determine the conditions of the break edges; (iii) a constant increments the value of an IV (i.e., step size) within the execution of the loop's body (e.g., the step size is 1 for Line 2 in Figure 3b).

Let us show in Algorithm 1 how to identify IVs using the aforementioned three properties by symbolically executing each DNN operator from its entry point to its exit point (Line 2, 16-17). Leveraging the property (i), DND symbolizes every general register initialized by a constant in the loop's entry block (Line 10-12). These symbolic variables are the IV candidates, which are further filtered out by checking the property (ii) and (iii). To check the property (ii), whenever encountering the conditional statements related to the loop's break edges during the symbolic execution, DND records these conditional statements as loop exit conditions, along with the IV candidates that are being accessed (Line 14). At the same time, the conditional statements related to break edges will make the symbolic execution engine diverge into two execution paths: one that satisfies the condition leading to exiting the loop, and the other does not satisfy the condition leading to continuing executing the loop. To avoid path explosion caused by the complicated nested loops, DND only keeps the execution path that exits the loop and discards the other execution path (Line 15). To check the property (iii), after reaching the DNN operator's most outer loop's exit point (e.g., Line 2 in Figure 3b), DND inspects each IV's corresponding register to check if its register is updated with a constant (i.e., step size). At last, DND considers an IV candidate as an IV when it satisfies both properties (ii) and (iii) (Line 18).

Finally, DND recovers IVs' initial values, step sizes, and loop counts (Line 19). In particular, loop counts are computed from initial values, step sizes, and the collected loop exit conditions. For example, the initial value, step size and loop exit conditions of `i` in Figure 3b are 0, 1, and `i<2`, respectively. Then, the loop count is derived by inquiring the solver with these information.

Algorithm 1 Loop analysis

```
 1: procedure LOOPANALYSIS(op: operator)
 2:     symEngine ← SymbolicEngine(op.entryAddr)
 3:     candidates ← ∅
 4:     while symEngine.hasActive() do
 5:         symEngine.step()
 6:         for each state ∈ symEngine.states do
 7:             inst ← state.lastInst
 8:             addr ← state.addr
 9:             if addr ∈ op.entryBlocks then
10:                 if isRegWrite(inst) and isConstant(inst.writeVal) then
11:                     inst.writeVal ← createSym()
12:                     candidates.add((addr, inst.writeReg))
13:             if addr ∈ op.breakEdgeSrcAddr then
14:                 symEngine.record(state.branch.condition.get_IV())
15:                 symEngine.keep(getBreakState(state.succ))
16:             if addr ∉ op.addrRange then
17:                 symEngine.stash(state)
18:     IVs ← checkConditionAndUpdate(candidates)
19:     IVs.getLoopCount()
20:     Return IVs
```

5.2.2 Symbolic Expression Extraction

A DNN operator typically performs tensor computation, which takes its input and parameters, and generates the computed output transferring to its successor DNN operators as the input. As such, we can represent the output of a DNN operator as symbolic expressions of the operator's input and the operator's parameters. These expressions contain the mathematical semantics DND needs to recover. To extract such symbolic expressions, DND performs customized selective symbolic execution with the IVs (identified in Section 5.2.1) as symbolic variables. This is because making IVs as symbolic variables brings the two following benefits: (1) it enables DND to symbolize the mathematical expressions of the DNN operator's output as symbolic expressions. (2) it allows DND to efficiently extract the symbolic expressions of a DNN operator's output by only executing one iteration of each loop, as discussed in *Solution 2* of Section 4. We will explain those benefits using Figure 3b.

Regarding the first benefit, the expression at Line 6 in Figure 3b is a symbolic expression of the operator's output, where the `i`, `j`, `u`, `v` are symbolized during the symbolic execution. Along with the information carried by the symbolic variables, this symbolic expression can represent the semantics of this `Conv` operator. Specifically, for each value of the `i` and `j`, `output[i][j]` (i.e., the `Conv` operator's output) is the accumulation of `input[i+u][j+v]*filter[u][v]` over all the possible values of `u` and `v`.

Regarding the second benefit, DND can symbolically execute only one iteration of the loop in Line 5-6 with the symbolized `v`. Specifically, DND exits the loop during the symbolic execution by assigning 2 to the symbolic variable `v` to satisfy the `v==2` predicate.

Furthermore, in order to keep track of the symbolic constraints related to memory reads and writes, DND's customized concretization strategy does not concretize memory addresses. Instead, when reading from symbolic memory, DND returns the symbolic memory address together with a proper annotation. For instance, when reading from address `input+i`, DND returns `input+i` with `MemReadVal` annotation, denoting where the value is read from. Using this annotation, DND keeps track of memory read values, and records the written expressions when the code write to symbolic memory.

We explain the detailed procedure to extract symbolic expressions in Algorithm 2. In particular, DND symbolically executes each DNN operator starting from its entry point (Line 3). When reaching the identified IV initialization code, DND symbolizes IVs' corresponding registers instead of initializing them with a constant (Line 9-10). Whenever encountering a conditional statement that can exit a loop, DND follows the path exiting the loop (Line 15-16). Furthermore, when reading an operator input or parameter with the symbolic address, DND returns the expression of symbolic address itself (e.g., the address of `filter[u][v]`) (Line 11-12). In this way, DND can keep track of each symbolic expression's provenance (i.e., the symbolic address where it is read from). To extract the symbolic expressions of DNN operator output, when the DNN operator output is updated, DND collects the symbolic address of the DNN operator output and its corresponding symbolic expressions (i.e., `+= input[i+u][j+v]*filter[u][v]`) (Line 13-14). Figure 3b shows the output example. In Line 6, `output[i][j]` is the symbolic address of operator output, and `+= input[i+u][j+v]*filter[u][v]` is its corresponding symbolic expressions. We show the example result of this symbolic expression extraction in Figure 3b.

Note that there are conditional statements that are not related to loops' break edges (e.g., the statement in `Conv` checking if padding is necessary). For example, if the padding size of the `Conv` operator in Figure 3a is 1, there will be conditional statements checking if the computation is in the padding zone (i.e., `i+u<1` and `i+u>4`). When DND encounters these conditional edges, DND forks multiple execution states and records the corresponding conditions. Such state and condition information can be later used to infer a DNN operator's attributes (e.g., padding size of `Conv` as described in Section 5.4.3).

Algorithm 2 Symbolic expression extraction

```
1:  procedure LOOPANALYSIS(op: operator, IVs: loop analysis results)
2:      memWrite ← ∅
3:      symEngine ← SymbolicEngine(op.entryAddr)
4:      while symEngine.hasActive() do
5:          symEngine.step()
6:          for each state ∈ symEngine.states do
7:              inst ← state.lastInst
8:              addr ← state.addr
9:              if addr ∈ IVs.definitionAddr then
10:                 inst.writeVal ← createSym(IVs.getIV(addr))
11:             if isMemRead(inst) and isSymbolic(inst.readAddr) then
12:                 inst.readVal ← inst.readAddr.annotate(MemRead)
13:             if isMemWrite(inst) and isSymbolic(inst.writeAddr) then
14:                 memWrite.add((inst.writeAddr, inst.writeVal))
15:             if addr ∈ op.breakEdgeSrcAddr then
16:                 symEngine.keep(getBreakState(state.succ))
17:             if addr ∉ op.addrRange then
18:                 symEngine.stash(state)
19:     Return memWrite
```

5.2.3 IR Design

We introduce our IR that can abstract an extracted symbolic expression of a DNN operator into a tensor computation. Unlike the extracted symbolic expressions, our IR is suitable to represent operator type and recover operator parameters of optimized DNN binaries (discussed in Section 5.2.4). We define two types of mathematical functions used in our IR: *reducing functions* and *transforming functions*. *Reducing functions* produce a single output result from a variable number of inputs (e.g., summation (`Sum`), average (`Avg`), and maximum (`Max`)). Intuitively, we use an expression to represent each element of the inputs (i.e., *element*), and use index variables (i.e., *index*) to represent loops' indexes with the range and step size. For instance, our IR represents $a_0 * b_0 + a_1 * b_1 + ... + a_n * b_n$ as `Sum`$(a_i * b_i, i)$, where $a_i * b_i$ is an *element*, and i is the *index* variable, ranging from 0 to n, with 1 as the step size. On the contrary, *transforming functions* take a fixed number (e.g., one or two) of inputs. `Mul`(a, b) multiplying a by b is one example of transforming functions. The reducing functions and transforming functions can be used together to represent a single IR expression. For instance, the IR expression in Line 2-8 in Figure 3d denotes a summation expression, where each element is a multiplication expression, and the index variables' information is listed in Line 5-8. We show the grammar of our IR in Table 4 in Appendix B.

5.2.4 Operator Summary Lifting

In this step, DND lifts the extracted symbolic expressions of each DNN operator to the operator summary in the IR we design. Each operator summary contains three parts: `addr`, `expr` and `IVs`, denoting the symbolic addresses of a DNN operator output, the AST of a DNN operator output, and the IVs information (i.e., initialization value, step size, and loop count), respectively. We show an example of generated operator summary in Figure 3d.

To lift `addr`, DND simply uses the DNN operator output address in the extracted symbolic expression (e.g., `output[i][j]` in Figure 3c). For `expr`, DND recursively parses the extracted symbolic expressions and then builds the AST in our IR format. Specifically, during the parsing process, DND analyzes the extracted symbolic expressions to identify their corresponding *reducing function* and its *elements* and *index*. For example, in Figure 3c, DND first identifies the accumulation operation (i.e., +=) as a reducing function, more specifically a summation function (i.e., `Sum`), and `input[i+u][j+v]*filter[u][v]` as the *element* from the extracted symbolic expressions. DND identifies the *index* variables as the difference between the IVs set included in the symbolic expressions of operator parameters and operator input (i.e., `i`, `j`, `u`, `v`) and the IVs set included in the symbolic address of an operator output (i.e., `i`, `j`), which is `u` and `v` in our example. Then, DND parses and identifies the *element* as `Mul` operation, with two arguments `input[i+u][j+v]` and `filter[u][v]`. Finally, DND stops parsing `input[i+u][j+v]` and `filter[u][v]` because they are the symbolic addresses of the values that are read from the memory (i.e., operator input or parameters). Figure 3d shows the generated operator summary.

Lifting operator summary with compiler optimizations. DND can correctly generate operator summary even with compiler optimizations. Figure 4a shows the simplified decompiled code of a `Conv` operator followed by a `Relu` operator, a combination used in many DNN models. The DNN compilers can optimize this combination, and Figure 4b shows the simplified decompiled code after being optimized using the operator fusion optimization and the loop unrolling optimization.

In this example, on one hand, the loop unrolling optimization unrolls the loop iterating over the filter length (Line 6-7 in Figure 4a), resulting in two update assignments (Line 5-7 in Figure 4b). On the other hand, the DNN operator fusion optimization embeds the `Relu` operator (Line 9-12 in Figure 4a) into the loop body of the `Conv` operator (Line 2-7 in Figure 4a).

In this context, `Relu` is applied to `output[i][j]` (Line 10-11 in Figure 4b) when the loop with the IV `u` (Line 4-8) is finished, and the loop with `i`, `j` as the IV (Line 4-8) are still ongoing. In this way, given a certain `i` and `j`, `Relu` is applied *after* the accumulation of `output[i][j]` is finished.

To lift the generated symbolic expression of the aforementioned heavily-optimized binary code in Figure 4c, DND introduces two techniques. First, in order to make the `expr` (i.e., AST) in the result operator summary succinct, DND conducts a loop rerolling analysis [39] on the extracted symbolic expression to handle the loop unrolling optimization. Specifically, DND recognizes a similar pattern among the symbolic expressions representing each one of the rolled iterations (e.g., Line 1 and Line 2 in Figure 4c), and recovers the rolled loop (e.g., the loop iterating over the filter length in Line 5 in Figure 4a) by creating a loop index (e.g., `v_reroll` in Figure 4d). Second, to divide a combined DNN operator into two separate DNN operators, DND leverages the heuristic that the combined second operator is usually an activation operator (e.g., `Relu`). Therefore, DND first identifies the activation operator in the extracted symbolic expressions (Line 3 in the Figure 4c), and then divides and lifts the expressions in Line 1-2 and Line 3 separately, resulting in two `expr` (Line 2 and 5 in Figure 4d).

5.3 Template ASTs Generation

The template ASTs are the references that are matched with the AST in an unknown DNN operator's operator summary, to determine its operator type. To generate a template AST of a DNN operator, we first manually construct an instance of the operator in the ONNX format, leveraging the usage examples of each ONNX operator [11]. Then, DND uses a DNN compiler to compile this ONNX operator instance to a binary. At last, DND generates the operator summary from the compiled binary, using the same operator summary generation procedure (described in Section 5.2), and takes its `expr` as the template AST. We will show the DNN operators from which we are able to generate the template ASTs in Section 7.1.

5.4 DNN Model Lifting

In this section, we describe how to further lift the operator summary of each DNN operator to the high-level representation of a DNN model (i.e., ONNX format). DND first recovers types of DNN operators using AST matching (Section 5.4.1). Then, DND recovers the DNN topology leveraging the inter-operator data dependencies (Section 5.4.2). Finally, DND recovers DNN operators' attributes and parameters leveraging both the DNN operator type and DNN topology, and converts the fully-recovered model into ONNX format (Section 5.4.3).

5.4.1 AST Matching

For each identified DNN operator, DND matches the ASTs (i.e., `expr`) in its operator summary with one of the template ASTs to determine its DNN operator type. Specifically, DND

```
void Conv(PTR* input, PTR* filter, PTR* output){
  // Conv operator
  for(i=0;i<2;i++) // output width index
    for(j=0;j<2;j++) // output length index
      for(u=0;u<2;u++)  // filter width index
        for(v=0;v<2;v++) // filter length index
          output[i][j]+=input[i+u][j+v]*filter[u][v]

  // Relu operator
  for(i=0;i<2;i++) // output width index
    for(j=0;j<2;j++) // output length index
      output[i][j]=Relu(output[i][j])
}
```

(a) Simplifed decompiled code *before* optimization

```
void Conv(PTR* input, PTR* filter, PTR* output){
  for(i=0;i<2;i++) // output width index
    for(j=0;j<2;j++) { // output length index
      for(u=0;u<2;u++) { // filter width index
        // filter length loop unrolled
        output[i][j]+=input[i+u][j]*filter[u][0]
        output[i][j]+=input[i+u][j+1]*filter[u][1]
      }

      // operator fusion optimization applied
      output[i][j]=Relu(output[i][j])
    }
}
```

(b) Simplifed decompiled code *after* optimization

```
output[i][j]+=input[i+u][j]*filter[u][0]
output[i][j]+=input[i+u][j+1]*filter[u][1]
output[i][j]=Relu(output[i][j])
```

(c) Extracted symbolic expressions

```
addr: output[i][j]
expr: Sum(element=Mul(input[i+u][j+v_reroll],
                  filter[u][v_reroll]),
          index=(u, v_reroll))
expr: Relu(output[i][j])
IVs: (u, init=0,inc=1,count=2)
     (v_reroll, init=0,inc=1,count=2)
     (i, init=0,inc=1,count=2)
     (j, init=0,inc=1,count=2)
```

(d) Generated operator summary

Figure 4: Operator summary generation of optimized `Conv` and `Relu`

performs a breadth-first search (BFS) to check if a DNN operator's AST and a template AST have the same tree structure, and both ASTs have the same mathematical functions (e.g., `Mul`) in each node. Note that DnD checks all the possible orderings of the compared nodes' sub-ASTs to match equivalent expressions with different ordering (e.g., `a+b` versus `b+a`). Furthermore, in order to match equivalent but optimized expressions (e.g., `a«1` versus `a*2`), DnD leverages angr's expression simplifier to determine the equivalence of two expressions.

This AST matching algorithm works for most of the template ASTs. However, this algorithm sometimes cannot distinguish DNN operators having the same tree and mathematical functions. For example, this algorithm cannot distinguish between `Conv` operator (shown in Figure 3d) and `Fully Connected` operator (i.e., `FC`, shown in Figure 5b) because of their same tree structure and mathematical functions in each

```
void FC(PTR* input, PTR* weight, PTR* output){
  // input shape (1, 256), weight shape (256, 10)
  for(i=0;i<256;i++) // number of input neuron
    for(j=0;j<10;j++) // number of output neuron
      output[j]+=input[i]*weight[j][i]
}
```

(a) Simplifed decompiled code

```
addr: output[j]
expr: Sum(element=Mul(input[i],
                      weight[j][i]),
          index=(i, j))
IVs: (i, init=0,inc=1,count=256)
     (j, init=0,inc=1,count=10)
```

(b) Generated operator summary (for simplicity, this demonstration `FC` does not include bias)

Figure 5: `FC` operator

node (i.e., `Sum` as the root node, with `Mul` as its *element*).

To correctly match the ASTs of `Conv` and `FC`, we further leverage the number of IVs they use in their ASTs to distinguish them: The `FC` uses two IVs because `FC` is a two-dimensional matrix product operation, which only involves two loops. Meanwhile, the `Conv` uses four or more IVs in their ASTs, because it conducts convolution operation over a (multi-dimensional) matrix, which involves at least four loops.

5.4.2 DNN Topology Recovery

As described in Section 2.1, there are two types of connections between DNN operators: sequential connections (e.g., the edges connecting the `Conv1` and the `Conv3` in Figure 1) and non-sequential connections (e.g., the edges connecting the `Conv2` at the top of Figure 1). DnD recovers the DNN topology by leveraging the DNN operator execution sequence in the inference function and the data dependencies between individual DNN operators.

Specifically, DnD first extracts the DNN operator execution sequence from the identified inference function (e.g., `Conv1 -> Conv2 -> Conv3`). Then, DnD calculates each DNN operator's (concrete) input/output address range by concretizing their input/output expressions in the operator summary with the corresponding ranges of IVs. For example, the operator output address range of the `Conv` in Figure 3d are calculated by concretizing its `addr` (i.e., `output[i][j]`) with the minimum and maximum of the used IVs (i.e., `i=0`, `j=0` and `i=1`, `j=1`, respectively). DnD determines there is a connection from one operator (e.g., `Conv1`) to the other operator (e.g., `Conv3`) if one operator's output address range overlaps with the other operator's input address range (e.g., `Conv1`'s output address range overlaps with the `Conv3`'s input address range). At last, DnD iterates the DNN operator execution sequence from the first DNN operator to the last DNN operator, identifies the data dependencies between adjacent operators, and connects them accordingly.

Furthermore, from the data dependencies, DnD can also recognize the *input term* (i.e., the term which is the output of the previous DNN operator) and *parameter term* (i.e., the term which is the parameters of the DNN operator) in the operator summary's `expr`, which can be leveraged for attributes and parameters recovery in the next step. For example, for the `Mul` function in the `FC` operator's summary (Line 2-3 in Figure 5b), DND identifies that the `input[i]` is the output of the previous DNN operator (i.e., its address range overlaps with previous DNN operator's output range), and that the `weight[j][i]` is the parameter (i.e., its address range does not overlap with any previous DNN operator's output range).

5.4.3 Attributes and Parameters Recovery

In the last step, DND recovers the attributes and parameters of each DNN operator by leveraging the generated operator summary and recovered DNN topology, and it then generates a high-level DNN representation in the ONNX format.

Attribute Recovery. For DNN operators with only shape-related attributes (e.g., filter length of `AveragePool`), DND recovers their attributes by checking the nesting structure of their loops and the loops' counts (e.g., the filter length is the loop count of the loop that iterates over the inputs).

For DNN operators with other attributes (e.g., padding size of `Conv`), DND uses heuristics to recover those attributes. We demonstrate how DND recovers the attributes of the `Conv` operator in Figure 3d. `Conv` is one of the most complicated operators, with three commonly-used attributes: filter shape, padding size, and striding size.

First, DND leverages the fact that the `Conv` applies the same filter to every input in order to infer the filter shape from the IVs of its *parameter term*. For example, there are only two IVs instead of three IVs (i.e., the number of filters, filter width, and filter length) in *parameter term* of the `Conv`: `u` and `v`, whose loop counts are both three. Therefore, DND infers that the filter shape is $(1,2,2)$, with the number of the filter as 1, and filter width/length as 2.

Meanwhile, DND also determines the output dimensions from the ranges of IVs used in operator summary's `addr` (i.e., `i` and `j`), which is $(2,2)$. Then, DND determines the padding size by identifying the padding checking conditions recorded during the symbolic expressions extraction step (Section 5.2.2). If there is no recorded padding checking condition, DND considers the padding size zero. At last, DND calculates the striding size using the formula below that constrains the valid `Conv`:

$$S = (In - F + 2 * P + 1)/Out$$

where S, In, F, P and Out denote the striding size, input dimension size, filter size, padding size, and output dimension size, respectively.

Parameter Recovery. To recover the parameters, DND leverages the recovered ranges of IVs to concretize the *parameter term* (identified in Section 5.4.2) as the concrete addresses, then the parameters are extracted from these concrete addresses. For example, for the `Conv` in Figure 3d, DND concretizes the *parameter term* (i.e., `filter[u][v]`) with all the possible values of `u` and `v` (both ranging from 0 to 1) to generate a list of concrete addresses, and the parameters are extracted from these memory addresses. Finally, with all the recovered information, DND generates a high-level DNN model description file in ONNX format.

6 Implementation

We implement DND with over 7.5K lines of Python code on top of `angr` [47].

DNN Operator Location Identification DNN operator location identification requires recovering CFGs and identifying loop locations. DND uses angr's to recover CFG, which is essential to find the locations of DNN operators.

Loop Analysis. DND requires finding all the loops and their nested loops in each DNN binary to perform loop analysis in Section 5.2.1. For that, we use angr's loop finder.

Operator Summary Generation. We implement the customized symbolic execution on top of angr simulation manager and angr under-constrained symbolic execution functionality. This symbolic execution engine is responsible for symbolizing variables (e.g., IVs) and collecting the symbolic expressions of each DNN operator output. In some cases, some DNN operators (e.g., `Softmax`) call specific mathematical functions (`exp`, `pow`, `sqrt`, `tanh`, `log`) in standard libraries (`libc` and `libm`). In these cases, DND needs to identify the called mathematical functions. To this aim, DND can use a function signature-based approach [21] if those functions are statically linked. Because those functions are pre-built, compilers insert those pre-built functions into DNN binaries without being changed. Alternatively, an analyst can search for such functions by checking called functions' names if function names are not stripped from the binary or those functions are dynamically linked.

7 Evaluation

We first demonstrate the generality of DND by showing how many commonly-used DNN operators and models can be supported in Section 7.1. Then, we show the correctness of DND across different DNN compilers, ISAs, and DNN models. To this aim, in Section 7.2, we compare the original DNN models and their corresponding decompiled DNN models, to verify the equivalence of model architecture, which means compared models have the identical DNN operators and topology, as well as the equivalence of their inference results (i.e., given the same input, both models output the same label).

7.1 Generality Evaluation

We evaluate DND's generality by evaluating how many widely-used DNN models DND can support. We consider a DNN as supported when all of its used operators are supported by DND. To this aim, we evaluate how many DNN operators DND can create template ASTs for.

We download an instance of every available DNN model

Table 1: Detailed statistics of the evaluated DNN models

	MNIST	ResNet v1	MobileNets v2
# of DNN Operators	12	25	105
# of DNN Operator Types	6	8	11
# of Parameters	6K	80K	3.4M
# of Connections	12	27	115

from the ONNX Model Zoo [10] and the MLPerf Tiny Benchmarks [2]. In this way, we collect 37 widely-used DNN models, covering many application scenarios such as image classification (e.g., ResNet v1 [20]), object detection (e.g., YOLO [37]), and nature language processing (e.g., GPT-2 [35]).

These 37 DNN models use 70 different DNN operators in total. As the first step, to generate the template AST of these 70 operators, we manually construct the DNN operator instances, use Glow to compile them to binaries, and lift the compiled binaries to the template ASTs (as described in Section 5.3). Note that we choose to use Glow because it supports all of our experimental target ISAs.

Our evaluation shows that DnD successfully generates the template ASTs of 59 DNN operators (i.e., our supported DNN operators) out of the collected 70 DNN operators. Those 59 DNN operators include the most commonly-used DNN operator, such as `Conv`, `MatMul`, and `MaxPool`. We show the complete list of the evaluated DNN operators in Table 3 in Appendix A. While there are 11 DNN operators that DnD cannot support because of failures in the template AST generation (Section 9 provides more details about these failures and propose possible solutions), the supported 59 (84%) DNN operators enable us to fully support 30 (81%) DNN models out of the collected 37 DNN models.

7.2 Decompilation Correctness Evaluation

We first describe our evaluation setup in Section 7.2.1. Then, we demonstrate the correctness of DnD by showing that the decompiled DNN models have equivalent model architectures (Section 7.2.2) and equivalent inference results as the original models, across different ISAs and compilers (Section 7.2.3).

7.2.1 Evaluation Setup

To align with prior model extraction attacks [58, 61], we use MNIST [59] and ResNet v1 [20] as two of our test DNN models. We also include MobileNets v2 [40], a DNN model designed for mobile and embedded systems. We show their statistics in Table 1. We acquire MNIST and MobileNets v2 from ONNX Model Zoo [10] and ResNet v1 from MLPerf Tiny Benchmarks [2]. Table 1 shows details of these three DNN models.

To generate a diverse set of binaries (in terms of ISAs and compilers), we compile the above three DNN models for three different ISAs (i.e., Thumb, AArch64, and x86-64) and with two different compilers (i.e., Glow and TVM).

Using Glow, we compiled the three DNN models with three different ISAs into nine DNN binaries. On the other hand, using TVM, we compiled the three DNN models only with *two* different ISAs (Thumb and x86-64) into six DNN binaries because TVM's AOT feature (i.e., microTVM [1]) does not support AArch64 to the best of our knowledge. Therefore, we evaluated 15 DNN binaries in total.

7.2.2 DNN Model Architecture Equivalence

Next, we demonstrate the model architecture equivalence between the decompiled models and the original models. For that, we compare the neural network architecture (i.e., the network topology, number of operators, and type of each operator) of 15 generated decompiled DNN models and their corresponding original DNN models. We report that all 15 decompiled DNN models are identical to their corresponding original models. An example of the original ResNet v1 model and the decompiled ResNet v1 model from a DNN binary (compiled with Glow and Arm Thumb ISA) is shown in Figure 6. Note that `Conv` operators and `Relu` operators are combined together in the target binaries, due to operator fusion (as explained in Section 2.2). Regardless of this optimization, DnD can correctly divide them into two operators, as shown in Figure 6b, using the approach described in Section 5.2.4.

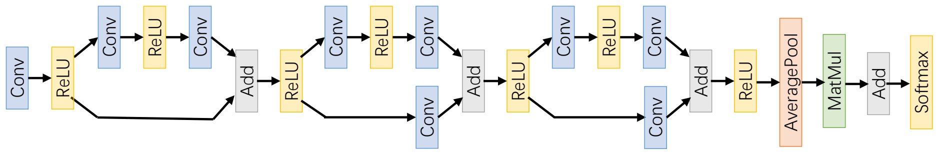
7.2.3 Inference Result Equivalence

To evaluate the inference result equivalence between the original DNN models and the decompiled DNN models, we check if the prediction results of the two models are identical. Specifically, given N test inputs, we measure the inference result equivalence as the percentage of how many inputs are identically predicted by the original models and the decompiled models. To this aim, we randomly sample the test inputs to the MNIST, ResNet v1, and MobileNets v2 models from the MNIST, CIFAR-10, and ImageNet datasets, respectively. We obtain the datasets from TorchVision [31]. Table 2 summarizes the results regarding the inference equivalence between the decompiled models and the original ones, using $N = 10,000$ test inputs. As shown, the inference results of both the decompiled DNN models and the original ones are identical for all the samples.

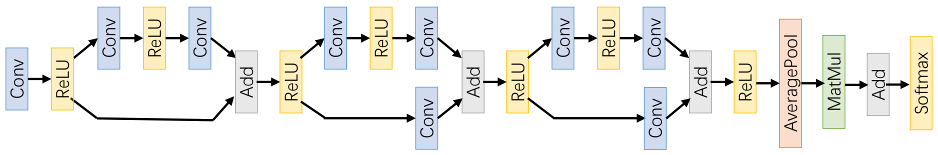
8 Case Study

In this section, we show how DnD can extract DNN model hyper-parameters and parameters from a DNN application running on a real-world micro-controller and demonstrate how we can leverage the decompiled DNN model to boost adversarial attacks.

As a real-world micro-controller, we use the NXP i.MX RT1050-EVK board, using an Arm Cortex-M7 processor and Thumb ISA. Its firmware development is supported by the eIQ ML software development environment [44], the ML development and deployment tool released by NXP Semiconductors. We use eIQ (which, in turn, uses the Glow DNN compiler) to



(a) Original ResNet v1 model



(b) Decompiled ResNet v1 model

Figure 6: Model architecture comparison. This figure shows that the architecture of the decompiled ResNet v1 model (obtained from a binary compiled with Glow using the Arm Thumb ISA) is identical to the original ResNet v1 model.

Table 2: Comparison between inference results of the DNN models decompiled from the test DNN binaries and their original DNN models. This table shows that the predicted labels of all the decompiled DNN models are identical to those of the original DNN models for all the tested inputs. N/A means that the ISA is not supported by the compiler.

	Thumb (Arm)			AArch64 (Arm)			x86-64		
	MNIST	ResNet v1	MobileNets v2	MNIST	ResNet v1	MobileNets v2	MNIST	ResNet v1	MobileNets v2
Glow	100%	100%	100%	100%	100%	100%	100%	100%	100%
TVM	100%	100%	100%	N/A	N/A	N/A	100%	100%	100%

build an application classifying images from the CIFAR-10 dataset using the ResNet v1 DNN model, and we install this application on the board.

8.1 Extraction Attack

To conduct a DNN extraction attack, we use DND to decompile the DNN model embedded in the DNN binary installed on the board. In our case, we obtained the DNN application by connecting the GDB debugger to the board's GDB port and then dumping the DNN application.

After obtaining the binary, DND locates the inference function in the DNN binary, decompiles its DNN model, and recovers its model hyper-parameters and parameters. To prove that the DNN extraction attack is successfully launched, we verify the correctness of the decompiled DNN model by comparing the inference results over the same inputs between the decompiled DNN model and the original DNN model. Specifically, we first randomly select 1,000 images from the CIFAR-10 dataset as the sample inputs. Then, we feed those chosen input images to both the decompiled DNN model and the original DNN model on the NXP i.MX RT1050-EVK board. Our evaluation results show that the decompiled model's inference results are identical to those of the original DNN model, proving the DNN extraction attack conducted by DND is successful.

8.2 Boosting Adversarial Attacks

The decompiled ResNet v1 DNN model can be used to conduct further attacks to the DNN model running on the NXP i.MX RT1050-EVK board. In this section, we demonstrate that, by using the decompiled model, we can launch effective adversarial attacks.

In DNN adversarial attacks, the adversary maliciously influences the predicted labels of the DNN model by adding minimal (typically imperceptible) perturbations to the input images [48]. The goal of these attacks is to find minimal perturbations on input images that can induce the DNN model to predict the wrong label. As described in Section 2.3, there are two categories of adversarial attacks: black-box attacks and white-box attacks. While white-box attacks are more effective than black-box attacks, they require an adversary to have prior knowledge of the victim DNN model (i.e., model hyper-parameters and parameters).

We demonstrate that, with the DNN model decompiled by DND, we can launch white-box adversarial attacks against the ResNet v1 DNN model running on the NXP i.MX RT1050-EVK board, and achieve better effectiveness than that achieved by black-box adversarial attacks.

To perform this evaluation, we first randomly select 1,000 images from the CIFAR-10 dataset as sample inputs, and we infer their labels on the NXP i.MX RT1050-EVK board to evaluate the accuracy of the DNN model. To launch the white-box attack, we use the projected gradient descent (PDG) algorithm [28] implemented in Foolbox [36] to generate adversarial images based on the sample inputs and the decompiled DNN model. As for the black-box attack, we utilize the black-box attack on CIFAR-10 implemented in Torchattacks [23] to generate adversarial images.

For both the attacks, we set the parameter `epsilon` to the same value (0.03, the default value). This value represents the maximally used perturbation between the adversarial images and their corresponding original images. Hence, both attacks perturb the images to the same extent. However, as expected, the white-box attack's perturbations are more effective, i.e., they are more capable of causing misclassifications in the victim DNN. In particular, the white-box and black-box attacks reduce the accuracy of the ResNet v1 model from 88.4% to 11.4% and 57.3%, respectively. We highlight that an adversary cannot launch the white-box attack without having access to a high-level representation of the DNN model, such as the one generated by DnD.

9 Discussion and Limitations

Existing General-Purpose Decompiler Existing general-purpose decompilers (e.g., Hex-Rays) have the following limitations when dealing with DNN binaries: (1) they do not recognize vectorized mathematical computations, leading to decompilation representations containing long loop bodies and excessive bitwise operations; (2) their decompilation representations differ significantly depending on the compilers or ISAs; (3) even with the same compiler and ISAs, DNN operators of the same type but with different attributes have different decompilation representations, because they are *specialized*. We demonstrate these limitations in Appendix C. These limitations hinder using simple pattern matching to recover the DNN high-level representation.

Correctness of the Recovered CFG and Binary Code. As other decompilation works [19, 56, 57], DnD assumes that the recovered CFG provided by the disassembler is reliable. Fortunately, our evaluation shows that the recovered CFGs of our test DNN binaries are reliable enough for our purpose.

Similarly, we assume that binary code in our target binaries is available before analysis because DnD works on top of the disassembled binary code. Therefore, obfuscated, encrypted, and packed binary code, available only after unpacking, deobfuscation, or decryption, is out of our scope.

Traditional Binary Analysis Challenges. DNN binaries, compared with the binaries compiled by general-purpose compilers, are more structured (e.g., nested loops), have less complicated control flows (e.g., no indirect control-flow transfer), and have no interaction with the environment (e.g., system calls). For these reasons, the information extracted by `angr` is accurate, which helps DnD focus on the challenges specific to DNN binaries, discussed in Section 4.

Handling Failure. DnD detects failures by checking if the output of each stage of the pipeline is expected (e.g., AST matching returns a matched AST). When failures happen (e.g., when dealing with unsupported operators), DnD skips decompiling the errored operator and proceeds with the other operators. In the end, DnD generates a report consisting of the decompilation results of the successfully decompiled operators and the binary function locations of the skipped operators.

Supporting Additional Operators. Out of the 70 operators used by the DNN model in our dataset, DnD currently does not support 11 of them for the following reasons: (1) our symbolic-execution-based approach is designed to capture the semantics of vectorized mathematical operations, and cannot capture the semantics of some operations such as sorting and searching; (2) our algebraic IR cannot represent recurrent structures; and (3) Glow does not support some operators defined in ONNX.

As future work, we plan to overcome these issues by: (1) extending our symbolic-execution-based approach and the expressiveness of our IR to support higher-level mathematical expressions involving sorting and searching operations; (2) transforming recursive code into equivalent iterative code, improving our current loop analysis, and extending our IR to support recurrent structure; and (3) using multiple DNN compilers to create additional DNN operations' templates.

Supporting Other DNN Compilers. Other DNN compilers supporting compiling DNN models to CPU exist (e.g., XLA [14] and NNFusion [27]). Unlike Glow [30] and TVM [1, 7], which compile DNN models into standalone binaries, XLA and NNFusion generate DNN binaries linked with open-source mathematical libraries to leverage the tensor operations of these libraries. For instance, XLA's generated binaries rely on `MatMul` implemented in Eigen.

As future work, to support these additional compilers, we will need to implement a dedicated analysis to identify these tensor-specific library functions. This analysis could take advantage of function matching approaches [55].

Decompiling Binary on DNN Accelerators. DnD does not support decompiling DNN binaries running on DNN accelerators (e.g., GPUs, FPGAs). This limitation is caused by the fact that DNN accelerators have very diverse ISAs that are usually not supported by the general-purpose disassemblers and the symbolic execution framework, which DnD relies on. For instance, although Nvidia provides closed-source disassemblers *cuobjdump* and *nvidiaasm*, which translate the CUDA binary into SASS assembly code, most details of the SASS assembly code are kept secret, which hinders further analysis.

10 Related Work

Decompiler. Many decompilation techniques have been proposed to improve the C/C++ decompilation performance through control-flow recovery [19, 56, 57] and decompiled code's readability enhancements [8, 25, 42]. Furthermore, Neural Machine Translation (NMT) has also been introduced [12] to improve the quality of the decompiled code. Other than the aforementioned academic researches, some open-source or commercial C/C++ decompilers have been introduced [5, 21, 33, 57]. In addition to decompiling C/C++ binary, researchers proposed reverse engineering techniques targeting smart contract [60], control firmware [24] and Bluetooth firmware [54]. However, those techniques cannot capture the mathematical semantics of compiled DNN models.

Extraction Attacks. Security researchers have proposed two types of DNN extractions attacks. The first type is algorithm-level extraction attacks. This type of attack is generally conducted by querying a black-box DNN model and then retraining a model using the querying results [46, 49, 50]. The other type of attack is architecture-level extraction attacks. They exploit hardware or side channels, such as PCIe traffic [22, 61], cache-based side-channel [58], power side-channel [29, 52], and even electromagnetic [3], to launch this type of attack. However, they either require prior knowledge of a DNN model (e.g., model hyper-parameters and parameters) or significant computational resources. Otherwise, they can recover only an incomplete DNN model.

Adversarial Attacks. Szegedy proposed the first adversarial attacks [48], which led DNNs to misclassify images. In follow-up studies, there are two types of attack schemes: the white-box and black-box attack approaches. However, the former approach [6, 48] assumes an attacker has prior knowledge of a victim DNN model, such as model hyper-parameters and parameters. On the other hand, the latter approach [4, 34] is less effective than the white-box attack because the black-box approach does not leverage model hyper-parameters and parameters that help improve attack effectiveness.

11 Conclusions

In this work, we designed and implemented DnD, the first compiler- and ISA-agnostic DNN decompiler. Our evaluation shows that DnD can perfectly recover different DNN models compiled by two different compilers for three different ISAs. As a potential real-world usage of DNN decompilation, we show how DnD can be used to extract a compiled DNN model from a real-world micro-controller, and to enable white-box adversarial ML techniques.

As traditional decompilers are the foundation for many security applications, we envision that, in the future, the capability of decompiling DNN binaries will bootstrap further security research in attacking and defending DNNs, in all those scenarios in which their original models are unavailable.

Acknowledgments

We thank the reviewers and our shepherd, Vasileios Kemerlis, for their valuable comments and suggestions. This project was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contracts number N6600120C4031 and number FA875019C0003, and the Office of Naval Research (ONR) under grant N00014-21-1-2328. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of United States Government or any agency thereof.

References

[1] Apache. microTVM. https://tvm.apache.org/docs/topic/microtvm/index.html.

[2] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, et al. Mlperf tiny benchmark. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.

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

[4] Arjun Nitin Bhagoji, Warren He, Bo Li, and Dawn Song. Exploring the space of black-box attacks on deep neural networks. *arXiv preprint arXiv:1712.09491*, 2017.

[5] David Brumley, JongHyup Lee, Edward J Schwartz, and Maverick Woo. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the USENIX Security Symposium (Usenix SEC)*, 2013.

[6] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated End-to-End optimizing compiler for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[8] Yaniv David, Uri Alon, and Eran Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. In *Proceedings of the ACM on Programming Languages (OOPSLA)*, 2020.

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

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

[11] Linux Foundation. ONNX operators. https://github.com/onnx/onnx/blob/master/docs/Operators.md.

[12] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. *Advances in Neural Information Processing Systems*, 2019.

[13] Google. Google AI and machine learning products. https://cloud.google.com/products/ai.

[14] Google. Tensorflow AOT compilation. https://www.tensorflow.org/xla/tfcompile.

[15] Google. Tensorflow lite. https://www.tensorflow.org/lite.

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

[17] Google. TPU. https://cloud.google.com/tpu.

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

[19] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. A comb for decompiled c code. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2020.

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

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

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

[23] Hoki Kim. Torchattacks: A pytorch repository for adversarial attacks. *arXiv preprint arXiv:2010.01950*, 2020.

[24] Taegyu Kim, Aolin Ding, Sriharsha Etigowni, Pengfei Sun, Jizhou Chen, Luis Garcia, Saman Zonouz, Dongyan Xu, and Dave (Jing) Tian. Reverse engineering and retrofitting robotic aerial vehicle control firmware using dispatch. In *Proceedings of the ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2022.

[25] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A neural approach to decompiled identifier naming. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[26] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 2020.

[27] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[28] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[29] Saurav Maji, Utsav Banerjee, and Anantha P Chandrakasan. Leaky nets: Recovering embedded neural network models and inputs through simple power and timing side-channels–attacks and defenses. *IEEE Internet of Things Journal*, 2021.

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

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

[32] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2021.

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

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

[35] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 2019.

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

[37] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

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

[39] Rodrigo CO Rocha, Pavlos Petoumenos, Björn Franke, Pramod Bhatotia, and Michael O'Boyle. Loop rolling for code size reduction. In *IEEE/ACM International Symposium on Code Generation and Optimization*, 2022.

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

[41] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. Evolving exact decompilation. In *Workshop on Binary Analysis Research (BAR)*, 2018.

[42] Edward J Schwartz, Cory F Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S Havrilla, and Charles Hines. Using logic programming to recover c++ classes and methods from compiled executables. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[43] NXP Semiconductors. Edgeready. https://www.nxp.com/applications/enabling-technologies/edge-computing/edgeready:EDGEREADY.

[44] NXP Semiconductors. eiq® glow ahead of time user guide. https://www.nxp.com/docs/en/user-guide/EIQGLOWAOTUG.pdf.

[45] NXP Semiconductors. Glow compiler optimizes neural networks for low-power nxp mcus. https://www.nxp.com/company/blog/glow-compiler-optimizes-neural-networks-for-low-power-nxp-mcus:BL-OPTIMIZES-NEURAL-NETWORKS.

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

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

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

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

[50] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

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

[52] Lingxiao Wei, Bo Luo, Yu Li, Yannan Liu, and Qiang Xu. I know what you see: Power side-channel attack on convolutional neural network accelerators. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2018.

[53] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[54] Jianliang Wu, Ruoyu Wu, Daniele Antonioli, Mathias Payer, Nils Ole Tippenhauer, Dongyan Xu, Dave Jing Tian, and Antonio Bianchi. LIGHTBLUE: Automatic Profile-Aware debloating of bluetooth stacks. In *Proceedings of the USENIX Security Symposium (Usenix SEC)*, 2021.

[55] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[56] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2016.

[57] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2015.

[58] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource

attacks to learn DNN architectures. In *Proceedings of the USENIX Security Symposium (Usenix SEC)*, 2020.

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

[60] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. Erays: reverse engineering ethereum's opaque smart contracts. In *Proceedings of the USENIX Security Symposium (Usenix SEC)*, 2018.

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

A List of Evaluated DNN Operators

Table 3: The list of the evaluated DNN operators. 'Y' denotes the operators which DND can generate the template ASTs, and 'N' denotes the operators which DND currently cannot generate the template ASTs. For these unsupported operators, we marked, using the numbers 1, 2, 3, the reason why they are not supported. Specifically, (1) means that DND cannot capture the semantics of some operations such as sorting and searching, (2) means that DND cannot represent recurrent structures, (3) means that our considered DNN compilers do not support some ONNX operators. These limitations are discussed in Section 9.

Abs (Y)	Add (Y)	And (Y)
ArgMax (N1)	AveragePool (Y)	BatchNormalization (Y)
Cast (Y)	CategoryMapper (N3)	Ceil (Y)
Clip (Y)	Compress (N1)	Concat (Y)
Constant (Y)	ConstantOfShape (Y)	Conv (Y)
ConvTranspose (Y)	CumSum (N1)	Div (Y)
Equal (Y)	Erf (N3)	Exp (Y)
Expand (Y)	Flatten (Y)	Floor (Y)
Gather (Y)	Gemm (Y)	GlobalAveragePool (Y)
Greater (Y)	Hardmax (N1)	Identity (Y)
LRN (Y)	LSTM (N2)	LeakyRelu (Y)
Less (Y)	Log (Y)	Loop (N2)
MatMul (Y)	MaxPool (Y)	Min (Y)
Mul (Y)	NonMaxSuppression (N1)	NonZero (Y)
Not (Y)	PRelu (Y)	Pow (Y)
Reciprocal (Y)	ReduceMax (Y)	ReduceMean (Y)
ReduceMin (Y)	ReduceSum (Y)	Relu (Y)
Reshape (Y)	Resize (Y)	RoiAlign (N1)
Scan (N1)	Scatter (Y)	Shape (Y)
Sigmoid (Y)	Slice (Y)	Softmax (Y)
Split (Y)	Sqrt (Y)	Squeeze (Y)
Sub (Y)	Sum (Y)	Tanh (Y)
Tile (N1)	TopK (N1)	Transpose (Y)
Unsqueeze (Y)		

B DND IR's grammar

Table 4: The grammar of DnD's IR. 'MemReadVal' denotes the value read from a memory location, and 'IV' denotes the symbolized loop induction variable.

Symbolic Expression	SymExpr	:=	RF | TF | Var
Reducing Function	RF	:=	Sum(SymExpr, IV) | Avg(SymExpr, IV) | Max(SymExpr, IV) | Min(SymExpr, IV) |
Transforming Function	TF	:=	Abs(SymExpr) | Add(SymExpr1, SymExpr2) | Div(SymExpr) | Mul(SymExpr1, SymExpr2) | Exp(SymExpr) | Log(SymExpr) | Sqrt(SymExpr) | Tanh(SymExpr) | Pow(SymExpr)
Variable	Var	:=	Constant | MemReadVal

C Decompiled Code Samples

We demonstrate snippets of decompiled code (i.e., by Hex-Rays decompiler) of three binary functions in Listing 1, Listing 2, and Listing 3. We elaborate on how we generate these three binary functions as follows:

- Listing 1: A convolution operator instance compiled with TVM and on Arm Thumb.
- Listing 2: The same convolution operator instance as the one used in Listing 1, compiled with Glow and on x86.
- Listing 3: A convolution operator instance with different attributes (e.g., kernel size, padding size), compiled with Glow and on x86.

We show the complete decompiled code at https://github.com/purseclab/DnD/tree/main/samples.

```
for ( j = 0; j != 64; ++j )
{
  *v17++ = 0.0;
  v19 = 0;
  v20 = 0.0;
  v21 = (i & 7) + 10 * (i >> 3);
  do
  {
    v22 = v19 << 6;
    v23 = (float *)&v6[256 * v21];
    v19 += 3;
    v24 = v22 + 192;
    while ( 1 )
    {
      v26 = (float *)((char *)arg1 + 256 * v22 + 4 * j);
      v27 = v23 + 64;
      do
      {
        v25 = *v23++;
        v20 = v20 + (float)(v25 * *v26);
        // 21 lines of code omitted
while ( (char *)arg2 + 256 != (char *)v28 );
```

Listing 1: Snippet of a decompiled convolution operator function of TVM/Arm Thumb binary function

```
do
{
  // 4 lines of code omitted
  do
  {
    // 4 lines of code omitted
    do
    {
      // 13 lines of code omitted
      do
      {
        // 6 lines of code omitted
        do
        {
          if ( (v130 | v27) < 0x10 )
          {
            // 8 lines of code omitted
            v51 = _mm_add_ps(
                    _mm_mul_ps(*(__m128 *)(a3 + 4 * v35),
                          v50),
                    _mm_add_ps(
                      _mm_mul_ps(*(__m128 *)(a3 + 4 * v34
                           ), v49),
                      _mm_add_ps(
                        _mm_mul_ps(*(__m128 *)(a3 + 4 *
                             v33), v48),
                        _mm_add_ps(
                          _mm_mul_ps(*(__m128 *)(a3 + 4 *
                               v32), v47),
                          _mm_add_ps(
                            _mm_mul_ps(*(__m128 *)(a3 + 4
                                 * v31), v46),
                            _mm_add_ps(
                              _mm_mul_ps(*(__m128 *)(a3 +
                                   4 * v30), v45),
                              _mm_add_ps(
                                _mm_mul_ps(*(__m128 *)(a3
                                     + 4 * v29), v44),
                                _mm_mul_ps(*(__m128 *)(a3
                                     + 4 * v28), v43))))
                                   ))));
            // 89 lines of code omitted
            v85 = (__m128)_mm_unpackhi_pd((__m128d)v82, (
                 __m128d)v82);
            v86 = (float)(_mm_shuffle_ps(v55, v55, 229).
                 m128_f32[0] + v55.m128_f32[0]) + *(v36 -
                  7);
            v42 = (float *)(a1 + ((4 * (v120 + ((v39 +
                 v129) << 6))) | 0x1C));
            // 91 lines of code omitted
}
while ( v120 < 0x38 );
```

Listing 2: Snippet of a decompiled convolution operator function of Glow/x86 binary function

```
  do
  {
    // 7 lines of code omitted
    do
    {
      // 6 lines of code omitted
      do
      {
        v84 = v32;
        if ( (unsigned __int64)(v32 + v103) <= 7 )
        {
          v33 = 8 * v32;
          v34 = 0LL;
          while ( (__int64)(v34 + v31) < 3 )
          {
            if ( (__int64)(v34 + v31) < 0 )
            {
              v35 = 1LL;
            }
            else
            {
              // 5 lines of code omitted
              do
              {
                // 21 lines of code omitted
                do
                {
                  v51 = _mm_shuffle_ps(
                          (__m128)*(unsigned int *)(v37 +
                                v41 - 768),
                          (__m128)*(unsigned int *)(v37 +
                                v41 - 768),
                          0);
                  // 38 lines of code omitted
                }
                while ( v41 );
                v60 = v40 + v36;
                *(__m128 *)(a1 + 4 * v60 + 16) =
                     _mm_add_ps(v50, *(__m128 *)(a1 + 4 *
                      (v40 + v36) + 16));
                *(__m128 *)(a1 + 4 * v60) = _mm_add_ps(
                     v49, *(__m128 *)(a1 + 4 * (v40 + v36
                     )));
                // 36 lines of code omitted
          }
          v67 = v83 + (v34 << 8);
          do
          {
            if ( v34 + v31 <= 7 )
            {
              // 4 lines of code omitted
              do
              {
                // 5 lines of code omitted
                do
                {
                  // 52 lines of code omitted
  }
```

Listing 3: Snippet of a decompiled convoluation operator function of Glow/x86 binary function