# DnD: Decompiling Deep Neural Network Compiled Binary

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 and embedded systems, dedicated DNN compilers are used to compile DNNs into binaries for the best performance. Security applications such as DNN model extraction, white-box adversarial sample generation, and DNN model patching become possible when a DNN model is accessible. However, these techniques cannot be applied to compiled DNN binaries. No decompilers can recover a high-level representation of a DNN model from its compiled binary code.

In this paper, we introduce DnD, the first ISA- and compiler-agnostic DNN decompiler. DnD uses the customized symbolic execution to lift the DNN binary into symbolic expressions represented in a novel intermediate representation (IR), which abstracts the high-level mathematical DNN operations in an ISA- and compiler-agnostic fashion. Then, DnD matches the lifted mathematical DNN operations with template mathematical DNN operations and recovers hyper-parameters and parameters of all the identified DNN operators, as well as the overall DNN topology. We evaluate DnD on different DNN binaries compiled by two different compilers (Glow and TVM) and for three different ISAs (Thumb, AArch64, and x86-64). and show that DnD can perfectly recover their model information. Moreover, using DND, we successfully reverse engineer the DNN models from the binary deployed on real-world micro-controllers and attack them with the white-box adversarial machine learning techniques.

## 1 Background

### 1.1 DNNs

Figure 1 shows a snippet of a DNN model. As shown in the figure, DNNs are composed of interconnected DNN operators. Each DNN operator takes the output of previous operators as its input and conducts mathematical operations based on its operator type and parameters.

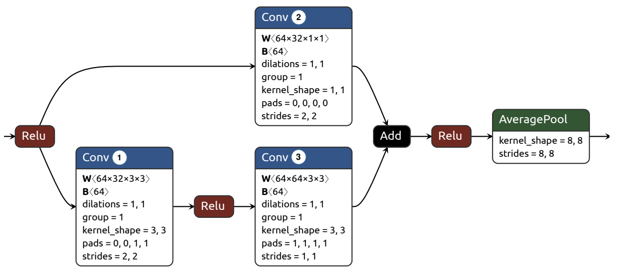

Figure 1: A snippet of ResNet v1 model in the ONNX format

There are numerous types of DNN operators with different high-level mathematical semantics. For example, there are activation operators (e.g., Relu), feature extraction operators (e.g., Convolution), and down-sampling operators (e.g., Average-pool). There are 174 different DNN operators defined in ONNX [1], an open standard for ML interoperability developed by Linux Foundation.

A DNN is defined by its hyper-parameters and parameters. DNN's hyper-parameters define the network architecture and how each operator functions, including the DNN topology, the type of each operator, and the attributes of each operator. The attributes of operators define their detailed configurations. For instance, as shown inside each operator in the figure above, Convolution (i.e., Conv) operator's attributes include the kernel shape, the pad size, and the stride size. The parameters are the variables that are learned during the training process (e.g., weights and bias).

### 1.2 DNN compilers

Multiple dedicated DNN compilers (e.g., Glow by Facebook, TVM by Apache, XLA by Google) exist to ease the deployment of DNN models on various devices. These DNN compilers are used in many real-world products, including edge devices and embedded systems. For example, to support the DNN model deployment on micro-controllers, NXP Semiconductors integrate a customized Glow compiler into its MCUXpresso SDK [10].

DNN compilers can be classified into two categories based on the compilation scheme: interpreter-based and ahead-of-time (AOT) schemes. Interpreter-based compilers (e.g., TFLM [4]) generate a DNN configuration file describing the DNN model. At runtime, a generic interpreter parses the DNN configuration file and invokes the corresponding DNN operator implementations from a generic runtime library. While generic, this scheme introduces space and time overhead since it requires interpreting the configuration file and loading the DNN runtime library. Moreover, it relies on a generic DNN operator library; thus, missing optimization opportunities of specific operator instances (e.g., kernel size equals 1,1 for the Conv-1 shown in the figure).

On the contrary, AOT compilers generate the operator instances specialized for their context information, such as attributes. For instance, Conv-1 and Conv-2 in the above figure are both Convolution operators, but they are compiled into distinct binary functions with different control/data flows, because of their different attributes. Thanks to the AOT scheme's low overhead, most DNN compilers support this scheme to deploy a DNN on edge devices whose hardware resources are limited.

### 1.3 Adversarial attacks

Adversarial attacks, inducing a DNN to misclassify an input, can be classified into two categories: black-box attacks and white-box attacks. Black-box attacks do not require attackers to have prior knowledge of the victim DNN (e.g., model hyperparameters and parameters). While white-box attacks require such knowledge to generate adversarial examples, they are more effective and efficient than black-box attacks.

## 2 DnD Design and Implementation

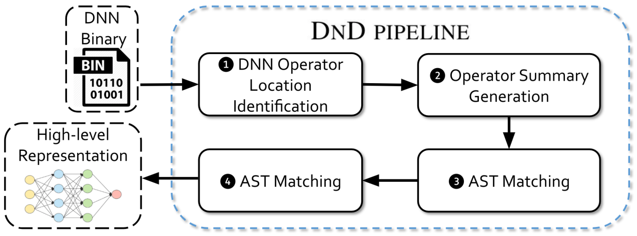
DNN Binary
BIN
10110
01001
DnD PIPELINE
❶ DNN Operator Location Identification
❷ Operator Summary Generation
❸ AST Matching
❹ AST Matching
High-level Representation

Figure 2: Pipeline of DnD

DnD is designed to decompile DNN binaries where a compiled DNN model is embedded. DnD targets DNN binaries compiled by the AOT compilation scheme running on CPU/MCU without hardware accelerators (e.g., NPU, GPU). The figure above depicts the overview of DnD. Specifically, DnD is composed of four components: (1) DNN Operator Function Location Identification, (2) Operator Summary Generation, (3) Abstract syntax tree (AST) matching, and (4) DNN Model Lifting.

DnD first takes a DNN binary and then locates the functions that implement all the DNN operators. To locate DNN operators, DnD relies on functions' patterns, such as heavy usage of nested loops and vectorized computations.

In the second step, for each identified DNN operator function, DnD captures and summarizes its mathematical semantics in operator summaries. Each operator summary is encoded in intermediate representation (IR) that can represent the mathematical semantics of DNN operators. Specifically, DnD conducts selective symbolic execution to extracts symbolic expressions from each DNN operator (e.g., output[i] = input[i] * params[i]), and stores them in an operator summary in our IR format.

Next, DnD identifies the type (e.g., convolution) of each DNN operator through AST matching. Before DnD analyzes DNN binaries, we pre-generate reference ASTs of all the DNN operators supported by DNN compilers. After operator summaries are generated, DnD converts every symbolic expression in each operator summary into an AST, and recognizes the operator type of this AST using the reference ASTs if a match is found!

Finally, DnD recovers the entire DNN model in the ONNX format. Specifically, DnD first recovers each DNN operator's attributes and parameters by analyzing its operator summary and recovered operator type. Then, DnD recovers the DNN topology (i.e., the interconnection between DNN operators) by identifying the data dependencies between DNN operators using data flow analysis.

We implement the DnD prototype using Python on top of angr [12]. DnD relies on angr to disassemble binary and recover the control flow graph (CFG) for the following symbolic execution analysis. DnD generates the extracted DNN model in ONNX format using the onnx library [2].

## 3 Evaluation

### 3.1 Generality Evaluation

We evaluate the generality of DnD by counting how many widely-used DNN models can be supported. We consider a DNN model supported when all of its DNN operators are supported. For this purpose, we evaluate how many DNN operators DnD can create reference ASTs for.

We collect 37 DNN models from the ONNX Model Zoo [3] and the MLPerf Tiny Benchmarks [7], covering many application scenarios such as image classification (e.g., ResNet v1), object detection (e.g., YOLO), and natural language processing (e.g., GPT-2). These 37 DNN models use 70 different DNN operators in total. Our evaluation shows that DnD supports 59 DNN operators out of the used 70 DNN operators (84%), which enables DnD to fully support 30 DNN models out of the collected 37 DNN models (81%).

## 3.2 Correctness Evaluation

We evaluate the correctness of DnD by checking if a DnD-decompiled DNN model is equivalent to the original model. In our experiments, both the decompiled and the original models are considered equivalent if their inference results are identical given the N=10000 test inputs. We use MNIST [15], ResNet v1 [5] and MobileNets v2 [8] as our test DNN models. To generate a diverse set of binaries (in terms of ISAs and compilers), we compile these three DNN models for three ISAs (i.e., Thumb, AArch64, and x86-64) and with two different popular DNN compilers (i.e., Glow and TVM). Our evaluation shows that all the decompiled DNN models are identical to those of the original DNN models for all the tested inputs, achieving 100

## 4 Case Study

We show two case studies: (1) how DnD can be used to steal DNN model hyper-parameters and parameters from a DNN application running on a real-world micro-controller, and (2) how the decompiled DNN model can be leveraged to boost adversarial attacks.

We apply DnD to the NXP i.MX RT1050-EVK board [11] as the real-world micro-controller. This board uses an Arm Cortex-M7 processor and Thumb ISA and supports the DNN model deployment through NXP's eIQ ML software development environment [9]. The ResNet v1 DNN model is deployed on the board in our case study.

To steal the DNN model, we first obtain the DNN binary by dumping the memory image using GDB. Then, we use DnD to decompile the dumped binary and recover its DNN model. We verify the correctness of the recovered DNN model by comparing the inference results over the same inputs (N=1000) between the decompiled DNN model running on the desktop and the original DNN model running on the board. Our evaluation shows DnD can perfectly recover the ResNet v1 model.

With the DNN model decompiled by DnD, we can boost adversarial attacks by launching white-box adversarial attacks against the ResNet v1 DNN model, and reduce the inference accuracy from 88.4% with original inputs to 11.4% with adversarial inputs, whereas black-box adversarial attacks can only degrade the inference accuracy to 57.3%.

## 5 Related Work

There are some works on reverse engineering domain-specific binary. For instance, researchers proposed reverse engineering techniques targeting smart contract [16], control firmware [6] and Bluetooth firmware [13]. However, those techniques cannot capture the mathematical semantics of compiled DNN models. This work is also presented in [14]

## References

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

[2] Linux Foundation. ONNX. https://github.com/onnx/onnx.

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

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

[5] 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.

[6] 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.

[7] MLCommons. Mlperf™ tiny deep learning benchmarks for embedded devices. https://github.com/mlcommons/tiny.

[8] 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.

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

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

[11] NXP Semiconductors. i.mx rt1050 evaluation kit. https://www.nxp.com/design/development-boards/i-mx-evaluation-and-development-boards/i-mx-rt1050-evaluation-kit:MIMXRT1050-EVK.

[12] 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.

[13] 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.

[14] Ruoyu Wu, Taegyu Kim, Dave (Jing) Tian, Antonio Bianchi, and Dongyan Xu. DnD: A Cross-Architecture deep neural network decompiler. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2135–2152, Boston, MA, August 2022. USENIX Association.

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

[16] 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.