The Pennsylvania State University
The J. Jeffrey and Ann Marie Fox Graduate School

REVERSE AND ANTI-REVERSE ENGINEERING OF DEEP NEURAL NETWORK PROGRAMS

A Dissertation in
Information Sciences and Technology
by
Jinquan Zhang

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2025

The dissertation of Jinquan Zhang was reviewed and approved by the following:

Dinghao Wu
Professor of Information Sciences and Technology
Dissertation Advisor
Chair of Committee

Peng Liu
Professor of Information Sciences and Technology

Taegyu Kim
Assistant Professor of Information Sciences and Technology

Sencun Zhu
Associate Professor of Computer Science and Engineering

Carleen Maitland
Professor of Information Sciences and Technology
Associate Dean for Research and Graduate Affairs

Abstract

The remarkable performance of Deep Learning (DL) over the past decades has driven the population of DL-based services. On the other hand, as edge computing advances, Deep Neural Network (DNN) applications have expanded from cloud to edge devices, enabling low-latency and highly secure intelligent interactions. Hence, the need to deploy DNN models on different hardware devices has rapidly increased. Yet, as most frameworks prioritize GPU environments, deploying DNN models across diverse hardware requires significant manual effort and remains challenging. Deep Learning compilers address this challenge by automating model conversion and optimization, streamlining deployment across platforms. Leading AI providers, including Google and Amazon, have already integrated DL compilers into their production workflows. However, while DL compilers enhance efficiency, they also bring the security and privacy of DNN models to a new arms race. In this dissertation, we investigate the Deep Learning Compiler concerning the reverse and anti-reverse engineering of DNN programs.

Model extraction attacks rely on different information sources to extract information from DNN models. Unlike other attack vectors, DNN programs always carry all the necessary information to run as a standalone program, making them ideal for model extraction. We introduce LIBSTEAL, a novel framework that reconstructs DNN architectures by reversing DL compiler-generated DNN programs. Our empirical analysis demonstrates that LIBSTEAL can extract layer types, attributes, dimensions, and connectivity, enabling near-exact replication of the original model. We implemented the prototype of LIBSTEAL and evaluated its effectiveness against both sequential and non-sequential DNN models, showing it can efficiently recover victim architectures.

As demonstrated in LIBSTEAL, the security risks of reversing-based model extraction have drawn increasing attention. Other reversing-based model extraction attacks have been proposed at the same time. Unfortunately, no defense countermeasure is designed to hinder such kind of attack. To address this gap, we investigate the state-of-the-art reversing-based model extraction attacks and propose FLATD, an advanced defense framework that obfuscates DNN Control Flow Graphs using Control Flow Flattening (CFF). Unlike traditional CFF techniques (e.g., O-LLVM), FLATD provides more effective and stealthy protection to DNN programs with similar performance and lower scale.

Beyond extraction threats, the integrity of DNN models is another critical concern, which ensures the correctness and predictability of the model, prevents unauthorized modifications, and enhances trust in deployment. Several attacks, such as bit-flip attacks, have been proposed to target the integrity of the DNN model. Although no existing

attacks directly target DNN programs, we anticipate future risks and propose a prototype defense mechanism. Our evaluation confirms that the protected DNN program retains full functionality, demonstrating the effectiveness of our integrity-preserving approach.

Contents

List of Figures

List of Tables

Acknowledgments

First, I sincerely thank my advisor, Prof. Dinghao Wu, for his guidance, support, and thoughtful feedback throughout my Ph.D. study. His mentorship has been instrumental in shaping my research and helping me grow as a scholar. I honestly could not have reached this point without his encouragement and expertise.

Second, I want to express my heartfelt gratitude to the rest of my committee members, Prof. Taegyu Kim, Prof. Peng Liu, and Prof. Sencun Zhu, for their invaluable insights, thoughtful suggestions, and continuous support. Their expertise has significantly contributed to my research and guided me through the challenges of this journey. I would like to especially thank Prof. Peng Liu for the opportunity to work as his teaching assistant. That experience not only deepened my knowledge but also helped me improve my teaching and mentoring skills.

I also extend my appreciation to my peers, collaborators, and colleagues, Rui Zhong, Zihao Wang, Xiaoting Li, and Zitong Shang. Without their help, I would never overcome the challenges I faced. I would like to show my special appreciation to Pei Wang. Although he has already graduated, he spent his spare time giving me valuable guidance and advice to help me go through the milestones and challenges of my Ph.D. journey.

Finally, I want to thank my parents for always supporting and encouraging me during my academic journey. Their confidence in me has kept me motivated, and their sacrifices have been the foundation of everything I've achieved.

This dissertation is based on research supported in part by the National Science Foundation (NSF) grant CNS-1652790 and the Office of Naval Research (ONR) grant N00014-17-1-2894. The findings and conclusions do not necessarily reflect the view of the funding agencies.

Chapter 1 Introduction

Machine learning models, especially deep neural networks (DNNs), have made a massive impact in the past decade due to their ability to solve complex problems in various domains, including computer vision [50, 73, 119], speech recognition [47, 51], natural language processing [31], and autonomous driving [66]. The exceptional performance of DNNs on recognition and prediction tasks has led to their commercial adoption, resulting in a surge in demand for deep learning-based services. As a result, there is an increasing need to deploy DNN models on diverse hardware targets ranging from cloud servers to self-driving cars and embedded devices [84, 137]. However, the existing frameworks [1,21,26,93,108] depend on vendor-specific operator libraries and mainly focus on the optimization of a limited set of server-class GPUs. Moreover, Figure 1.1 shows that different hardware targets (e.g., CPU, GPU, and TPU-like accelerators) require different on-chip memory architectures and compute primitives. Therefore, deploying DNN models on devices like mobile devices or embedded systems with limited resources is challenging and requires significant manual effort. The appearance of Deep Learning Compilers eases the process by bridging the gap between DNN frameworks and the hardware backend, making deployment more efficient. Various DL compilers, such as TVM [22], Tensor Comprehension [125], Glow [116], nGraph [32], XLA [76], and NNfusion [91], have been proposed by both industry and academic players to address this issue.

1.1 Background

1.1.1 Deep Neural Networks

Deep Neural Networks (DNNs) are a sub-area of Deep Learning in Artificial Neural Networks (ANNs). They have multiple layers between the input and output, enabling

them to model complex, non-linear relationships. A DNN can be mathematically described as a function $y = f(x)$ where the input $x \in \mathbb{R}^n$ and the output $y \in \mathbb{R}^m$. Distinct features of models define their structure, training dynamics, and overall performance. Below are the fundamental aspects of DNNs:

Network Architecture: The architecture of a neural network includes the types of operators, operator dimensions, the connection topology between operators, and specific attributes of each operator. Broadly, DNN architectures can be categorized into sequential and non-sequential. Operators are connected linearly in sequential architectures, with each operator having one input and one output. On the other hand, the non-sequential architecture may include the operators with multiple inputs and the operators sharing the same input.

Hyper-parameters: Hyper-parameters are configuration settings that govern the training process of the model and significantly influence its efficiency and effectiveness. These include learning rate, which determines the step size during optimization; batch size, which specifies the number of samples processed before updating parameters of the model; and the number of epochs, which defines how many times the entire training dataset is used during training. Selecting appropriate hyper-parameter values is crucial, as they directly affect the convergence speed, stability, and final accuracy of the model.

Parameters: The function f of a DNN is defined by its learnable parameters, which include weights, biases, and additional parameters associated with specific operators, such as Batch Normalization (BN) parameters. Weights represent the strength of connections between operators, biases allow the model to shift activation thresholds, and BN parameters normalize operator outputs to stabilize training. These parameters are updated iteratively during the training process through optimization algorithms, such as stochastic gradient descent (SGD) [4] or Adam [71]. The quality of these parameter updates determines the ability of the model to generalize to unseen data.

In our dissertation, we focus on the network architecture information of the DNN model through our framework, which is the most fundamental model characteristic for neural network security because, with the knowledge of the network architecture, it is possible to infer parameters and hyper-parameters. [55, 124, 126].

1.1.2 Deep Learning Compiler

The objective of deep learning compilers, such as XLA [76], TVM [22], Intel nGraph [32], and Tensor Comprehension [125], is to simplify the process of deploying DNN models on different hardware platforms, by automating the optimization and transformation. These

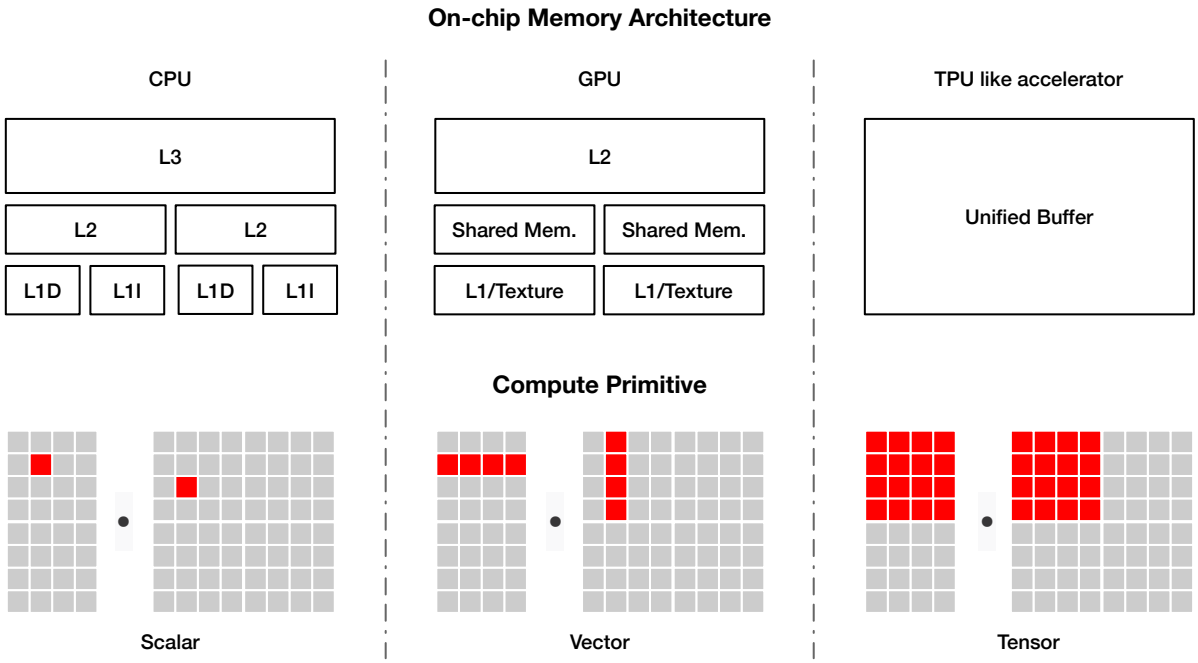


Figure 1.1: Comparison of compute primitives and on-chip memory architectures across CPUs, GPUs, and TPU-like accelerators. CPUs use scalar units with multi-level caches (L1I/L1D, L2, L3). GPUs employ vector parallelism via Streaming Multiprocessors with shared memory, L1/Texure, and L2 caches. TPU-like accelerators specialize in tensor operations using Unified Buffer, a large on-chip memory, for activations and weights. This divergence requires significant manual efforts when deploying the DNN model on different hardware devices.

compilers can take models described within popular frameworks like TensorFlow [1], PyTorch [108], MXNet [21], Caffe2 [93], and Keras [69] as inputs and generate standalone DNN programs or kernel libraries that can be statically linked with executables for CPUs, GPUs, and TPU-like accelerators. As shown in Figure 1.2, the DL compiler architecture can be divided into two main phases: frontend and backend, each manipulating one or several Intermediate Representations (IR).

Frontend. DL compilers first transform high-level model descriptions into computational graph representations and further convert them into graph IRs. These IRs, independent of the target hardware platform, define the graph structure, including the network topology and layer dimensions. They facilitate graph- and node-level optimizations, such as operator fusion, static memory planning, and layout transformation [22, 116].

Backend. From the graph IRs, hardware-specific low-level IRs are generated. These IRs serve as an intermediary step for tailored optimizations, incorporating knowledge of DL models and hardware characteristics. The graph IR operators can be converted into

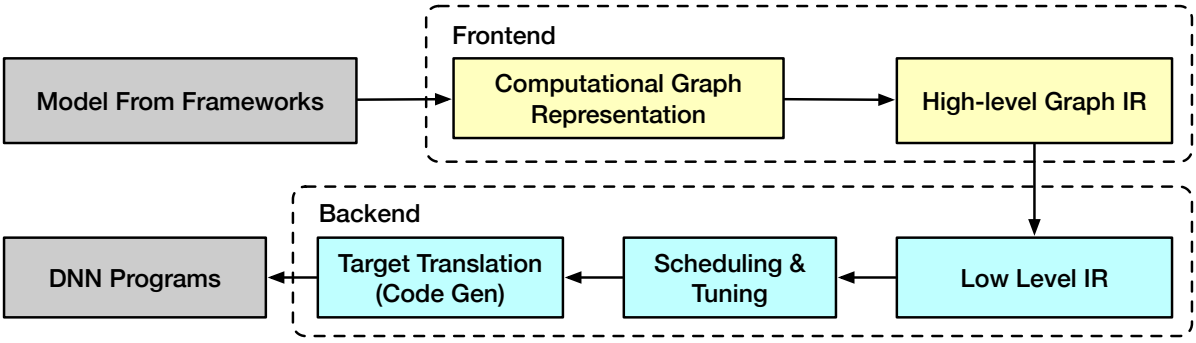


Figure 1.2: Compilation Flow of the Deep Learning Compiler. The input of the DL compiler is the model from different frameworks. The compiler frontend transforms the model description into the computational graph representation and further conveys it into graph IR to apply graph- and node-level optimizations. At the compiler backend, it does the hardware-specific optimization on low-level IR and also involves scheduling and tuning. Finally, the compiler uses the corresponding target executable format (e.g., LLVM) to generate the DNN Programs.

low-level linear algebra operators, simplifying the support for high-level operators across various hardware targets. This stage's optimization includes hardware intrinsic mapping, memory allocation, loop-related optimizations, and parallelization [9, 22, 111, 145]. The backend also involves scheduling and tuning, where the compiler searches for optimal parameter settings, such as loop unrolling factors. Recent advancements [3, 22, 23, 99, 125, 149, 150] introduce automated scheduling and tuning to improve optimization, reducing manual efforts.

Code Generation. Finally, these low-level IRs are compiled into code for different hardware targets. Before that, the DL compilers can also integrate with existing infrastructure like LLVM [75] and CUDA [136] to leverage third-party toolchains and further manipulate the generated code.

Thanks to the DL compilers, the DNN model can be deployed on edge devices and low-power processors [59, 101, 102, 109] with limited hardware resources and low overhead. Giant AI providers like Amazon and Google also include DL compilers in their AI services to boost performance [5, 60, 88, 134]. As the need for DL-based services has increased, DL compilers play a more critical role in deploying DNN models, and the safety of DNN programs becomes increasingly vital.

1.2 Motivation

The remarkable achievements of deep learning make the DNN models from giant AI providers attractive to malicious users and attackers because a DNN model with high accuracy always needs a large dataset [33, 141] and high training costs. For instance, training a model using a v2 Tensor processing unit (TPU) in the cloud would cost $400K or higher [35, 110]. Additionally, these AI providers offer service privatization, allowing them to sell their high-quality DNN models to other companies and organizations for a license fee. The high value of DNN models raises the arms race between the attacks and defenses toward the DNN models. The appearance of the Deep Learning Compiler brings the war to another battlefield.

Existing model extraction works target different attack surfaces [41, 55–57, 65, 103, 104, 114, 124, 126, 135, 139, 143, 152]. Most of the previous research [41, 56, 57, 135, 139, 143] utilize side-channel information to infer the model characteristics — [55, 152] used information leaked by PCIe bus traffic and memory bus traffic to extract partial or whole DNN model; [103, 124] relied on the query-prediction pairs generated from the target model to predict the model type. However, these information sources are limited and sometimes depend on strict assumptions. Due to the emergence of the DL compilers, DNN models have also been exposed to binary reversing engineering. Unlike the above information sources, DNN programs compiled from DNN models always contain complete information that can be used to run in an isolated environment. To take advantage of the information contained by DNN programs, we propose a new model extraction attack framework to leak the information from DNN models.

On the other hand, it is always vital to protect the DNN models. Besides our attacking framework, there are another three attacking frameworks proposed at the same time [20, 89, 138] that can reconstruct DNN models by reversing the DNN programs, which is a significant threat to the security and privacy of DNN models. There exist several defense mechanisms to defend against the attack from different perspectives. For example, oblivious RAM (ORAM) [86, 87, 122] can prevent information leakage on the bus by encrypting the data addresses to hide the memory access patterns. Therefore, attackers are not able to identify two operations even with the same physical address [122]. Another potential defense method is to obfuscate the identification of operator dependencies by inserting dummy read/write operations as fake memory traffic to disturb the tracing of memory events. The graph-level optimization applied by the DL compiler [22] increases the difficulty of inferring the computation graph at the system level. However, none

of these defense frameworks can hinder the DNN programs from reverse engineering. Fortunately, after carefully investing in all the existing reverse-based model extraction attacks, we found a key feature adopted by all the attack frameworks. Based on the observation, we propose an advanced defense framework for protecting DNN programs from reverse model extraction attacks.

Last but not least, we notice the importance of protecting the integrity of the DNN programs to ensure their reliability and security. Due to their magnificent achievement in many decision-making tasks, DNN models face threats not only from information leakage but also from model compromising, especially fault injection attacks [132]. By introducing faults into hardware, software, or DNN models, attackers can lead the model to produce incorrect or unpredictable outputs and break the alignment between the models' training behavior and deployment behavior, undermining trust in safety-critical systems like autonomous driving or financial systems. Actually, several research projects have been proposed from both the attack and defense sides [7, 53, 82]. Aramoon et al. [7] proposed AID to verify the integrity of DNN models by generating a set of test cases called edge points to detect any unauthorized modifications that could compromise their functionality. Li et al. [82] presented a novel attack framework called FrameFlip to deplete DNN model inference with runtime code fault injections. However, none of the research pays attention to the integrity of DNN programs, which provide a new interface for attackers to manipulate DNN models. We take the first step and implement a prototype to protect the integrity of DNN programs by ensuring the control flow integrity of DNN programs.

1.3 Research Goals

In this section, we summarize our research goals and then present an overview of three research studies in this dissertation. We present our research in three parts:

1. Model Extraction Attack towards Deep Learning Compilers by Reversing DNN Program

2. Protecting Deep Neural Network Program from Reversing Attacks.

3. Securing Integrity of Deep Neural Network Program from Instrumentation.

With the emergence of Deep Learning Compilers, the threats and defenses of DNN models have been introduced to a new battlefield. We aim to explore the possibility

of reverse engineering DNN programs and anti-reverse engineering them. As shown in the first project, we want to leak the information from DNN programs. In this project, we statically analyze the DNN programs compiled from DL compilers and identify several challenges to stealing information from DNN programs. We propose a novel attack framework to extract the neural network architecture from DNN programs and reconstruct the DNN models accordingly. At the same time, several projects focusing on decompiling the DNN programs have been proposed. After carefully investigating these model extraction attacks, we provide our solution to protect the DNN programs from reversing based on control flow flattening while preserving efficiency. Finally, we raise the importance of securing the integrity of the DNN program. We attempt to prevent the DNN program from fault injection attacks by securing the control flow integrity of the DNN program.

1.3.1 Model Extraction Attack towards Deep Learning Compilers by Reversing DNN Program

The importance of the Deep Neural Network model is self-evident. The great value of high-quality DNN models from giant AI providers always attracts the interest of malicious users and attackers who want to steal them, as many existing model extraction attacks try to leak essential information from different attacking surfaces.

Meanwhile, the increasing need to deploy Deep Learning-based services to different hardware devices, especially edge devices, led to the birth of the Deep Learning (DL) compiler, which can ease the deployment process by optimizing and compiling the DNN model into a DNN program. However, it also provides a new attack interface for attackers to extract information using reverse engineering. Our goal is to explore the possibility of such a model extraction attack.

Analyze Deep Neural Network Program. Each DNN model layer has a unique computational pattern, which is reflected in the code generated by DL compilers. These patterns result in identifiable characteristics within DNN programs, making them more vulnerable to reverse engineering than general-purpose binary programs. Moreover, unlike traditional software, the DNN program contains nested loop structures that correspond directly to specific layer operations.

Furthermore, these computational patterns are consistent across a wide range of DNN models, regardless of their architecture or application domain. This consistency allows us to systematically analyze and extract meaningful patterns from a set of well-understood

white-box models, even if they differ from the target model. By leveraging these patterns, we can infer key structural components of an unknown model, significantly reducing the complexity of reverse engineering tasks.

Extract architecture and rebuild DNN Model. To reconstruct a DNN model, key information about neural network architecture must be extracted, including the types, attributes, dimensions, and connectivity between the layers. Therefore, we designed our framework with three main components: binary analyzer, layer identification, and search engine. These components allow us to obtain this critical information and accurately rebuild the network architecture.

The binary analyzer disassembles DNN programs and extracts their input/output (I/O) dimensions. The layer identification step operates on the principle that layers with the same type, I/O dimensions, and attributes—even if they belong to different models—are compiled into exact layer functions. This observation lets us systematically explore possible layer attributes and generate layer functions with the exact I/O dimensions as those extracted from the binary. These generated functions are stored in a layer repository, which serves as a reference for identifying and matching the unknown layer functions of the victim model. We can infer the correct layer types and attributes by comparing the extracted layer functions against those in the repository. We employ a search engine to reconstruct the model, organizing the extracted layers into a directed data flow graph based on their I/O dimensions. This structured representation allows us to efficiently identify a valid computational path that spans all layers, ensuring that the reconstructed network accurately reflects the original architecture.

1.3.2 Protecting Deep Neural Network Program from Reversing Attacks.

Given the high risk of information leakage from reversing DNN programs, our goal is to propose an effective defense mechanism compatible with the DL compiler to prevent attackers from inferring essential information from them.

Investigating the reversing-based model extraction attacks. We analyzed the underlying logic and workflow of various attack frameworks to gain deeper insights into reverse engineering-based model extraction attacks. First, these frameworks all include an essential component: operator-type (layer-type) recovery. While the specific techniques employed for this component may vary across frameworks, they all rely on leveraging computation patterns to infer operator types, which means that each

operator in a DNN model follows a distinct mathematical transformation to process input data before passing it to the next layer. This characteristic enables attackers to identify operator types through binary similarity analysis, matching observed computation patterns against known references. At the same time, our study reveals that this reliance on computation patterns also presents an opportunity for strengthening defenses. In particular, we observed that the Control Flow Graph (CFG) plays a crucial role in all attack frameworks, suggesting that CFG-based obfuscation could protect DNN programs against reverse engineering-based extraction attacks.

Improve the robustness of the DNN program. Based on our observations, we present FlatD, a defense framework that safeguards DNN programs from model extraction attacks that rely on reverse engineering. FlatD is based on Control Flow Flattening (CFF), a technique that disrupts the structured execution flow of a program, making it harder to analyze. However, traditional CFF approaches still leave identifiable patterns that attackers can leverage. To address this, FlatD enhances obfuscation by incorporating additional security mechanisms, further complicating efforts to reconstruct the Control Flow Graph (CFG) and extract meaningful insights. Moreover, FlatD integrates opaque predicates, one-way cryptographic hashing, and indirect jumps to achieve stronger protection. Opaque predicates introduce misleading conditional statements that make static analysis unreliable, preventing attackers from quickly mapping out execution paths. One-way cryptographic hashing ensures that even if certain control flow elements are exposed, they cannot be reversed or interpreted meaningfully. Additionally, indirect jumps obscure function call sequences, making identifying layer operations within the compiled binary challenging.

1.3.3 Securing Integrity of Deep Neural Network Program.

The major goal of this work is to safeguard the integrity of DNN programs and defend them against potential fault injection attacks, which are a growing threat in the AI community. As the number of DNN programs deployed on diverse hardware targets has rapidly increased, ensuring their security, integrity, and reliability has become a critical concern. Protecting DNN programs from malicious manipulation is essential to maintaining their performance, trustworthiness, and overall resilience in practice.

Investigate the potential fault injection attacks. In addition to model extraction attacks, fault injection attacks pose another security risk to DNN programs. A particularly concerning example is the FrameFlip [82] attack, which leverages RowHammder [70] to

induce bit flips in the memory regions where DNN parameters or control logic are stored, which leads to severe consequences, including misclassification of inputs, degradation of model accuracy, and critical security breaches. RowHammer exploits physical weaknesses in modern DRAM memory modules by repeatedly activating specific memory rows, causing electrical interference that can flip bits in adjacent rows without direct access to those memory locations. The attack can bypass most traditional code integrity protection mechanisms, particularly those that operate solely at the software level, which often assume the underlying hardware to be trustworthy.

Protect DNN program code integrity. To counter such a risk, the third work proposes and develops a framework designed to secure the code integrity of DNN programs by detecting unauthorized code modifications at runtime. The framework proposes a crash-based security mechanism, which guarantees that any attempt to tamper with the DNN program triggers an immediate system termination. This aggressive response effectively prevents an attacker from gaining further control or causing additional harm to the system. To implement this protection strategy, we introduce a two-stage prototype framework consisting of compilation and post-compilation phases, leveraging hashing and indirect control flow to check the code integrity at runtime. By combining these methods, the framework enhances runtime integrity enforcement and provides a defense capable of addressing fault injection attacks, including those leveraging RowHammer.

1.4 Thesis Organization

In this thesis, we aim to explore the reverse and anti-reverse engineering of DNN programs. In particular, we leverage the new attack interface introduced by Deep Learning Compilers and attempt to leak information from the DNN programs. Then, we propose an advanced defense mechanism to protect the DNN programs from decompiling. Additionally, we secure the integrity of the DNN programs to prevent them from compromising. The first work presents a novel attack framework that targets the DNN program library. The framework extracts neural network architecture information from the DNN programs and rebuilds the DNN models. In the second work, we investigate all the reversing-based model extraction attacks and propose an advanced defense mechanism to protect the DNN program from reversing. The third work emphasizes the importance of DNN program integrity and implements a prototype to secure the control flow integrity of DNN programs.

The rest of the thesis is organized as follows. We present related works of this thesis in Chapter 2. In Chapter 3, we outline our novel attack framework for model extraction. In Chapter 4, we investigate all the existing reversing-based model extraction attacks and present an advanced defense mechanism. Chapter 5 focuses on the importance of the integrity of DNN programs and implements a prototype to secure the integrity. Chapter 6 discusses current limitations and proposes potential improvements. In Chapter 7, we give out a conclusion of the entire thesis.

Chapter 2
Related Work

In this chapter, we present recent research related to this thesis. First, we introduce the research focusing on the model extraction attacks. Then, we present the countermeasure to prevent the model information from leaking.

2.1 Model Extraction Attacks

The basic logic of the model extraction attack is leveraging the information gathered from different sources to leak the vital features of DNN models. Two primary sources are the side-channel information and query results from prediction API. The emergency of DL Compilers also makes it possible for attackers to extract models from reverse engineering.

2.1.1 Side-channel information

Among the various techniques employed in model extraction attacks, exploiting side-channel information is one of the most common strategies. Side-channels refer to indirect information leakage that can reveal sensitive data about the internal behavior of machine learning models without requiring direct access to their parameters or architectures.

For example, by exploiting the memory and timing side-channel, Hua et al. [56] presented a model extraction attack targeting the convolutional neural network (CNN) running on a hardware accelerator, which can infer the network architecture and identify the value of parameters. Similarly, Duddu et al. [41] and Hunt et al. [57] have also leveraged timing side-channel to perform model extraction. Duddu et al. focused on measuring execution time to deduce the depth of a DNN model, providing valuable insights into its layer composition. Hunt et al., on the other hand, introduced Telekine, a novel attack that observes the GPU kernel execution timing to classify input images,

demonstrating that timing-based leakage can be exploited not only to reveal model architecture but also to infer model inputs. Beyond timing channels, researchers have also exploited cache-based side-channels. Yan et al. [143] showed that by analyzing cache access patterns, it is possible to reconstruct the architectural layout of DNN models, including details about the number and types of layers. Another form of side-channel attack involves power analysis. Wei et al. [135] performed an attack on an FPGA-based convolutional neural network accelerator to recover the input image without knowing the parameter. Similarly, Xiang et al. [139] exploited power side-channel data in embedded systems, successfully extracting critical information about the internal network architecture and making reliable estimations of model parameters. Additionally, bus traffic is also an important information source. Zhu et al. [152] introduced Hermes, a novel attack framework that capitalizes on information leakage from encrypted PCIe traffic. Their method successfully reconstructed the entire DNN model, including its architecture and weights, achieving identical inference accuracy to the original target model. Hu et al. [55] also investigated the vulnerability of bus traffic, utilizing side-channel leakage from both PCIe and memory buses. By adopting techniques from speech recognition, they developed a method to predict the architecture of the target DNN accurately.

2.1.2 Prediction API

Many ML-as-a-service platforms provide not only prediction labels but also confidence scores. Although users benefit from the rich results, they leave a window for attackers or malicious users to apply model extraction attacks. Many studies have shown that this information can be exploited to enhance the efficiency of model extraction attacks, enabling attackers to reconstruct models with fewer queries.

Tramèr et al. [124] focused on scenarios where attackers, without prior knowledge of parameters or training dataset, can duplicate functionality of the model by leveraging its prediction API. They demonstrated that attackers can efficiently extract target ML models with high fidelity by making strategic queries to the prediction API. The attack is particularly effective for popular model classes, including logistic regression, neural networks, and decision trees. The attack was evaluated against online services such as BigML and Amazon Machine Learning. The results indicate that it is possible to extract models from these services with a limited number of queries. Papernot et al. [107] proposed a method where an attacker trains a local substitute model to mimic the target DNN model, which is achieved by generating synthetic inputs and using outputs of the target model to label them. The substitute model learns to approximate the

decision boundaries of the target model. The paper demonstrated the effectiveness of this approach by attacking a remote DNN classifier hosted by MetaMind. The crafted adversarial examples caused the target model to misclassify 84.24% of the inputs. Oh et al. [103] presented that they can infer internal attributes of neural networks that are accessible only through their input-output behavior. They train a secondary model to predict specific characteristics of a target black-box neural network. The dataset consists of diverse known neural networks to learn the relationship between observable outputs and internal characteristics.

2.1.3 Reverse Engineering

With the appearance of the Deep Learning Compiler, DNN programs have become an essential source for attackers to leak information about DNN models. Compared to other information sources, DNN programs always contain complete information about DNN models to perform the inference task. In addition to the work in our thesis, three other works focus on decompiling DNN programs and reconstructing DNN models.

Chen et al. [20] propose NNReverse, a method for reconstructing the architecture and parameters of DNN models from compiled DNN programs using a learning-based approach. It employs a fine-grained embedding model to represent assembly functions by combining syntax and topology semantics. The authors demonstrate that NNReverse can accurately infer the type of DNN layers across various hardware platforms, leveraging a pre-trained database of binary functions. Wu et al. [138] introduce DnD, the first compiler- and ISA-agnostic decompiler for DNN programs. The method combines symbolic execution with loop analysis to lift compiled binaries into an intermediate representation (IR) that captures high-level mathematical operations. DnD identifies DNN operators, hyper-parameters, and topology, representing the decompiled model in the ONNX format. The evaluation reveals that DnD can recover models from binaries generated by different compilers (Glow, TVM) and architectures (Thumb, AArch64, x86-64). Liu et al. [89] present BTD (Bin to DNN), a decompiler designed for x86 DNN executables. BTD employs a hybrid approach involving representation learning, dynamic analysis, and symbolic execution to fully reconstruct the DNN model, including operator types, topology, dimensions, and parameters. BTD also demonstrates practical applications by boosting adversarial attacks and supporting model migration across platforms.

2.2 Countermeasure

While the widespread adoption of DNN models has made remarkable achievements in various fields, model extraction attacks raise a significant concern in the community because the leaked information from DNN models can lead to more severe attacks, like adversarial attack [45, 83]. For the security and privacy of DNN models, defense mechanisms have been developed by academia and industry to protect DNN models from such attacks from different surfaces.

2.2.1 Side Channel

Protecting memory access patterns is key to defending against side-channel-based model extraction attacks, particularly in architectures that process sensitive computations, like GPUs executing DNN models. Several methods have been proposed to mitigate model extraction attacks.

Oblivious RAM (ORAM) is a well-established method for protecting memory access patterns by encrypting data addresses and reshuffling memory accesses. Stefanov et al. [122] introduced Path ORAM, a simplified ORAM protocol that significantly reduces client storage requirements. Alternatively, Liu et al. [87] introduce compiler-assisted techniques that obfuscate execution flow while minimizing overhead. Furthermore, Liu et al. [86] proposed GhostRider, a hardware-software co-design system that aims to hide memory access patterns at a higher level and enables secure computation with minimal leakage of memory traces. GhostRider achieves this by leveraging an execution environment that balances security with performance, ensuring an attacker cannot infer program execution details from memory access patterns. Moreover, Hu et al. [55] discuss a defense strategy about introducing dummy memory operations to hide the actual memory access sequence, preventing attackers from accurately identifying layer dependencies in DNN models.

2.2.2 Cloud

Attackers who target the model on the cloud usually use the input data sequence and prediction output pairs to infer the model information. In this case, Juuti et al. [63] propose PRADA, the first generic and effective tool to detect such a DNN model extraction attack. PRADA analyzes the sequence of API queries and raises the alarm if it deviates from benign behavior. They claim that PRADA can detect all prior model extraction

attacks without false positives. Orekondy et al. [105] propose another defense against this model extraction attack by actively perturbing predictions to poison the attacker's training objective. The defense is effective across many datasets and outperforms the same type of defenses. It can resist model extraction attacks with high accuracy for thousands of queries, increasing the attacker's error rate up to 85 times, with minimal impact on benign users. Besides, Li et al. [80] propose a protection scheme against black-box model extraction attacks that uses a physical unclonable function (PUF) obfuscation technique. The scheme involves building a PUF on the user side and a corresponding PUF model on the service provider side. The proposed scheme allows legitimate users to accurately restore the model predictions while preventing attackers from extracting helpful information. Karchmer [64] discusses the possibility of providing provable security against model extraction attacks. To detect such an attack, the author proposes a theoretical framework for analyzing observational model extraction defenses (OMEDs) that examine the distribution of queries made by adversaries. They introduce the concepts of complete and sound OMEDs and show that achieving provable security against model extraction through these defenses is possible using average-case hardness assumptions for PAC learning. The framework provides a way to abstract current techniques used in the literature to achieve provable security.

2.2.3 Obfuscation

To our knowledge, no defense mechanism has yet been designed for reversing-based model extraction attacks. Therefore, this section shows a potential countermeasure solution: obfuscation. Obfuscation is a technique that software developers have used for a long time to protect their intellectual property. The basic idea behind obfuscation is to transform a program into a new version that retains its functionality and semantics but hides its high-level structures [11, 19, 142]. Obfuscation significantly increases the difficulty of static program analysis [130, 133, 140], reversing engineering [128, 131], and also higher the bar of dynamic program analysis [12, 13, 24, 25, 151]. As an essential branch of obfuscation, control flow obfuscation aims to conceal the proper control flow and make the control flow graph as complicated as possible to raise the bar of countermeasures. Although implementing control flow obfuscation is flexible and has many variants [37, 39, 129], specific techniques can efficiently obscure the control flow graph and are frequently used in practice.

```
int x, y;
x = 0;
try {
    y = x-2/x;
    // Bogus Code
} catch (ArithmeticException e) {
    // Original Code
}
```

(a) Programming schemes

```
int x;
while (x>1) {
    if (x%2==1) x=x*3+1
    else x=x/2
    if (x==1) {
        // Original Code
    }
}
```

(b) Contextual schemes

Figure 2.1: Opaque Predicates Schemes. Programming schemes (Fig 2.1a) leverage exception handling as opaque predicates to mislead analysis by prioritizing bogus paths. Contextual schemes (Fig 2.1b) utilize loops and state-dependent conditions (e.g., Collatz-like sequences) to embed opaque predicates to deter reverse engineering.

2.2.3.1 Opaque Predicates

Opaque predicates involve inserting conditional statements that always evaluate to true or false but appear complex and non-trivial to the analyzer, typically using mathematical or logical expressions that seem relevant but are redundant [30]. Replying on opaque predicates, bogus control flows are inserted into code sections that pretend to be regular code to confuse the static analyzer. Typical schemes to create opaque predicates include numerical, programming, and contextual schemes. Figure 2.1 has demonstrated examples for each scheme.

Numerical schemes use mathematical expressions to achieve opaque predicates. For example, $(x^3 - x)\ \%\ 3 = 1$ is always valid for all x. Nevertheless, this format of opaque predicates is easy to detect. Several other mathematical approaches use more complex schemes to reach the same goal. The second scheme is a programming scheme, which uses some program tricks as an opaque predicate, such as comparing two references pointing to the same address. However, pointer analysis is a severe challenge, even for state-of-the-art static analysis. Besides, Dolz and Parra [38] presented a method using `try-catch` mechanism to compose opaque predicates as shown in Figure 2.1a. In this sample, `ArithmeticException` will be triggered by division by zero, and there are some exception events similar to division by zero, which are easy to trigger. The final one is contextual schemes utilizing the context of the program and proposing suitable opaque predicates. Figure 2.1b indicates an example using a program verifying $3n + 1$ problem(i.e., Collatz problem or the hailstone problem); the result will always be one, therefore the original code under condition $x == 1$ will always be executed, the only problem is waiting for how long.

```
int s=0;
int x=1;
while (x<10) {
    s=s+x;
    x++;
}
printf("%d",s)
```

(a) Original Code

```
    int s=0;
    int x=1;
L1: if (x>=10) goto L3;
    s=s+x;
L2: x++;
    goto L1;
L3: printf("%d",s);
```

(b) Using `if-then-go`

```
    int swVar=1;
L1: switch (swVar) {
        case 1:
            s=0;
            x=1;
            swVar=2;
            break;
        case 2:
            if (x>=10) swVar=4;
            else swVar=3;
            break;
        case 3:
            s=s+x;
            x++;
            swVar=2;
            break;
        case 4:
            printf("%d",s);
            break;
    }
    goto L1;
```

(c) Using `switch-case`

Figure 2.2: Simplified Example of Control Flow Flattening. Figure 2.2a shows a simple code to sum up integers from 1 to 9. Figure 2.2b shows the result obfuscated by using `if-then-goto`. The loop is decomposed into labeled blocks with explicit jumps (`goto`), disrupting the linear flow. Figure 2.2c demonstrates the control flow flattening result using `switch-case`: A state variable, `swVar`, directs execution through a flattened control flow graph, replacing the loop with a state machine. This technique obscures the original loop structure, complicating reverse engineering by introducing artificial complexity and indirect control transfers.

2.2.3.2 Control Flow Flattening

Control Flow Flattening is a widely adopted technique in the domain of control flow obfuscation, primarily due to its ability to dynamically determine the sequence of basic blocks or instructions during program execution. By doing so, it effectively conceals the original control flow structure from static analysis tools and adversaries attempting reverse engineering. As illustrated in Figure 2.2, a simplified example of the approach introduced by Wang et al. [127], demonstrates how the technique leverages dispatcher mechanisms, such as the `switch-case` construct (Figure 2.2c) and the `if-then-goto` pattern (Figure 2.2b), to flatten the control flow graph. These dispatcher constructs enable the redirection of control flow to various basic blocks at runtime, thereby masking the static layout of the control flow graph and complicating any attempts to analyze

or reconstruct the original program logic. In a typical control flow flattening scheme, once a basic block finishes execution, program control automatically returns to the dispatcher, which then determines the next block to execute based on a control variable or index. This cycle repeats until the program concludes. As an alternative, Linn and Debray [85] proposed a method that manipulates stack-based control information to manage program execution, thereby adding further complexity by incorporating branch functions to obfuscate executable binaries. Control flow flattening replaces conventional, structured control flow mechanisms with a singular, linear dispatching system, such as nested conditional statements and iterative loops. This transformation significantly increases the difficulty of understanding the program's execution path, as it hides decision points and the logical structure within an opaque, uniform sequence of dispatch calls.

Building upon these traditional concepts, our second proposed framework, FLATD, is designed based on control flow flattening to DNN programs. This framework enhances the robustness of DNN programs against modern reversing-based model extraction attacks, which often rely on analyzing control flow to reconstruct model architecture and parameters.

2.2.3.3 Code Virtualization

Code virtualization is an advanced control flow obfuscation technique that transforms the original code into a form significantly different from the source yet functionally equivalent [8]. This method typically involves translating original native instructions into a custom set of operations designed for execution on a specialized virtual machine (VM). Unlike conventional virtual machines, such as the Java Virtual Machine (JVM), this VM is specifically constructed to interpret the obfuscated program. It operates with a unique instruction set architecture that can differ significantly from standard processor instructions, further complicating reverse engineering and analysis. However, code virtualization always introduces high time overhead, which is unsuitable for efficiency-sensitive programs like DNN programs.

2.2.3.4 Other

In addition to the control flow obfuscation strategies discussed earlier, several innovative techniques have been introduced from different perspectives. One of the most prominent examples is the work by Domas [39], who created an obfuscation tool known as *movfuscator*. This approach converts an entire program into a form that only uses `MOV` instruction set. The effectiveness of this method relies on the fact that the `MOV` instruction set is

Turing complete [37]. This transformation significantly complicates reverse engineering efforts by eliminating the use of common control and arithmetic instructions, leaving only `MOV` instructions to implement the program's logic. Wang et al. [129] introduced the concept of translingual obfuscation, which they implemented in their tool called BABEL. Their approach involves translating selected portions of a C program into Prolog, a language that follows a distinct programming paradigm and execution model. Since Prolog operates on a declarative paradigm, in contrast to the imperative nature of C, the resulting binary becomes substantially more difficult to analyze and reverse engineer. Additionally, Majumdar et al. [92] proposed a unique obfuscation strategy based on program slicing techniques. Their method leverages information extracted from slicing the code to construct obfuscated versions of the program. Despite their originality and effectiveness in specific scenarios, these methods generally lack broad applicability and universality across different platforms and programming environments.

Chapter 3
Model Extraction Attack towards Deep Learning Compilers by Reversing DNN Program

In this chapter, we propose LibSteal [146], a novel attacking framework that can rebuild DNN architecture by reversing the binary generated from the Deep Learning Compiler. We demonstrate that by empirically examining the characteristics of the DNN binary library, we can extract the layer types, attributes, dimensions, and connectivity between layers. We implemented the prototype of LibSteal and also conducted a set of evaluations to show LibSteal has the ability to extract architecture information and rebuild DNN models.

3.1 Introduction

Machine learning models, especially deep neural networks (DNNs), have been widely deployed to tackle challenging problems in computer vision [50,73,119], speech recognition [47, 51], natural language processing [31], and autonomous driving [66]. Compared to other machine learning technologies, the outstanding performance of DNNs on recognition and prediction tasks [78, 117] has seen its commercial adoption with impacts across the field. It also increases the demand for Deep Learning (DL) based services and the need to deploy deep learning model on edge devices like mobile phones [84, 137]. For example, to help users who are blind or have low vision, some DNN models need to be deployed on the phone so that users can use them to identify nearby objects more conveniently. Also, giant AI providers provide the so-called service privatization to sell their high-quality DNN models to other companies and organizations with a license fee.

However, over the past decades, the explosion of DNN frameworks [1, 21, 26, 93, 108] and the explosion of hardware backend (e.g., CPUs, GPUs, and FPGAs) increase the difficulty of deploying DL based services to target platforms, which requires significant manual effort. The deep learning compiler kills two birds with one stone and draws the attention of many stakeholders. Several DL compilers have been proposed by both industrial and academic actors recently, such as TVM [22], Tensor Comprehension [125], Glow [116], nGraph [32], and XLA [76]. The DL compilers take the models from different DL frameworks as input and compile them into a lightweight binary with faster inference efficiency for the target hardware platform. TVM, in particular, has developed a large community across the industry and academia [6]. However, the DNN program generated from DL compilers makes it possible for an attacker to leak the internal work of DNN models by reversing the stand-alone programs [20, 89, 138]. The deployable DNN program generated by DL compilers has two deployment modes, Ahead-of-time (AOT) and Just-in-time (JIT) [81]. Unlike the existing work targeting the AOT scheme, which generates self-contained executables, in this paper, we narrow down the threat model where we only have access to the DNN runtime library generated by the JIT scheme. We propose a framework named LibSteal to leak the network architecture information of the target DNN model using only the runtime library.

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

We implement the prototype of LibSteal based on Uroboros [131] and adopt the idea proposed by Asm2Vec [36] to accomplish the similarity comparison between layer functions. To demonstrate the practicality and the effectiveness of our attack framework, we evaluate it against binaries of four widely-used DNN models, MNIST [77], VGG [119], ResNet [50], and MobileNet [54]. We choose TVM [22] as the target DL compiler, LLVM [75] as the target host, and CPU as the target hardware device to deploy. All victim models are initially designed in Keras framework [26] with Tensorflow [1] as the backbone. The experimental result shows that our framework can effectively extract the neural network architecture information. The reconstructed models have similar or even equivalent network architecture to the original. We then re-trained the extracted models and they all achieved accuracy comparable to that of the original models.

In summary, we make the following contributions:

- We narrow down the threat model from the DNN executable to the DNN runtime library. With limited input, we are able to leak essential information about the DNN model architecture.
- We design and implement the framework LibSteal to achieve our goal, which consists of three parts and combines various techniques to deliver a decent pipeline.
- We have evaluated our framework on four widely-used model binaries using the TVM as the DL compiler. The results indicate that our framework can handle MNIST, VGG, ResNet, and MobileNet DNN models. With the stolen information, the reconstructed models have similar or even equivalent network architecture to the original and can achieve inference accuracy comparable to that of the original.

The rest is organized as follows. We formulate the problem about reversing-based model extraction attack in Section 3.2 and claim our threat model in Section 3.3. Then, we present the design of our LibSteal framework in Section 3.4. The experimental results are shown in Section 3.5.

3.2 Problem Statement

3.2.1 General Challenges

A reversing-based model extraction attack aims to reconstruct the architecture and functional behavior of a DNN model by reverse engineering the DNN program generated

by DL compilers, such as runtime libraries or standalone executables, without access to the original DNN model description, training dataset, or model parameters. Unlike traditional model extraction attacks that query APIs or analyze runtime behavior, reversing-based model extraction attack focuses on the analysis of DNN programs to infer layer types, depth, dimensions, and connectivity, which remain three crucial general challenges (GCs):

- **GC1: Information Scaricity.** When DL compilers compile A DNN model into the DNN Programs, it loses explicit layer metadata, such as layer dimensions and attributes. Furthermore, deploying the DNN program as a commercial off-the-shelf (COTS) product strips helpful information that may lead us to the layer metadata. In this context, extracting essential information from the DNN program and reconstructing the DNN model is challenging.

- **GC2: Code-Semantic Gap.** DL compilers pack high-level DNN model layer into function. In order to infer the type of each layer, we need to map low-level assembly to high-level DNN layers. However, constrained by **GC1**, filling this gap seems unattainable.

- **GC3: Limited Resources.** In addition to **GC1**, we can only access to the DNN runtime library. Without the JSON files containing the connection information, it is almost impossible to reconstruct the DNN model even if we successfully tackle **GC1** and **GC2** and infer layer-related information.

3.2.2 Our Solutions

With constrain from the about three general challenges, it is almost impossible to extract architecture information from only DNN runtime library and reconstruct DNN model. We overcome hurdle by several observations from careful investigation of the DNN programs. More specifically,

- To **address GC1**, the data section of DNN runtime library contains information related to layer metadata.

- To **address GC2**, the special computation pattern bind to each layer type remains consistently even when being compiled to low-level assembly code.

- To **address GC3**, with extracted layer information, it is possible to heuristically search out the connection information and reconstruct the DNN model

3.2.2.1 Reverse Engineering Framework

For **GC1**, we reply on a reverse engineering framework to decompile and analyze the DNN runtime library. Binary reverse engineering focuses on analyzing and understanding binary programs without access to their source code. This technique is often used to uncover the functionality and structure of software, especially when the source code is unavailable or inaccessible. Over the past decade, both academia and industry have introduced several sophisticated reverse engineering frameworks designed to automate and enhance the analysis of binary programs. Industry-standard tools such as IDA Pro [42], Binary Ninja [58], and Ghidra [115] are widely adopted for their user-friendly interfaces, extensive plugin ecosystems, and support for multiple instruction set architectures. On the other hand, academic research has produced frameworks like BAP (Binary Analysis Platform) [17], angr [128], and BitBlaze [120], which emphasize static and dynamic analysis techniques, symbolic execution, and formal methods for program reasoning. Among these, Uroboros, introduced by Wang et al. [131], offers a particularly notable advancement through its reassembleable disassembly approach. In this work, we adopt Uroboros as the primary framework for reverse engineering the DNN runtime library. Its ability to generate accurate and reassembleable representations of disassembled code provides a robust foundation for understanding and reconstructing the control and data flows within DNN programs.

3.2.2.2 Binary Similarity Detection

To address **GC2**, we actually need to finish a task about the binary similarity detection. Since the computation pattern of the same layer type with the same attributes remains similar across different DNN programs, we can generate a repo containing all candidate layer types for the victim layer function. Then, we leverage the binary similarity detection technology to compare the victim layer function with all candidate layer types and determine its layer type. To achieve this goal, we must choose a binary similarity detection methodology wisely. Early frameworks such as BinDiff [44] and BinSlayer [14] rely on graph-based structural comparisons of control flow graphs (CFGs) to measure similarity. While effective in some scenarios, these methods often suffer from limited resilience to code transformations, such as compiler optimizations. More recently, learning-based approaches have been proposed to improve robustness and efficiency. DeepBinDiff [40], for example, incorporates deep learning models to generate embeddings that capture functional similarities between binary code segments, even

when the code has undergone heavy transformations. However, DeepBinDiff primarily targets the function-level comparison without fine-grained adaptability to specific code patterns like those found in DNN layer functions. In this work, we adopt Asm2Vec [36], a state-of-the-art static binary similarity detection framework designed to enhance robustness against both code obfuscation and compiler optimization. Asm2Vec generates embeddings for assembly instructions by leveraging control flow context and sequential semantics, producing vector representations that reflect the functional behavior of binary code. Compared to graph-based or purely syntactic approaches, Asm2Vec offers superior resilience to variations introduced by different compilation settings, making it particularly well-suited for analyzing layer functions in DNN runtime libraries. Its efficiency and accuracy in identifying semantically similar functions allow us to reliably map the victim layer function to its corresponding candidate in our repository. By leveraging robust similarity detection of Asm2Vec, we can confidently determine the layer type of a given function within a DNN program despite variations caused by different compilers, optimization levels, or minor implementation differences.

As for **GC3**, we innovatively adopt heuristic search to figure out the connection information from DNN program and reconstruct DNN models. Based on the information extracted by solving **GC1** and **GC2**, we can build a directed graph to search out possible architecture. In order to get as complete and accurate architecture as possible, we adopt several assumptions to do the prune. The technical detail will be discussed in Section 3.4.4.

3.3 Threat Model

In this section, we break down the victim's capability and the attacker's goal and capability.

3.3.1 Victim's Capability

The DNN programs are compiled by TVM [22] with optimization level `-O0` and are not obfuscated because, to the best of our knowledge, DL compilers themselves do not apply any software defensive mechanism to the generated binary. Also, the target host platform is LLVM, which means the ultimate DNN programs are ELF binaries on the x86/x64 system.

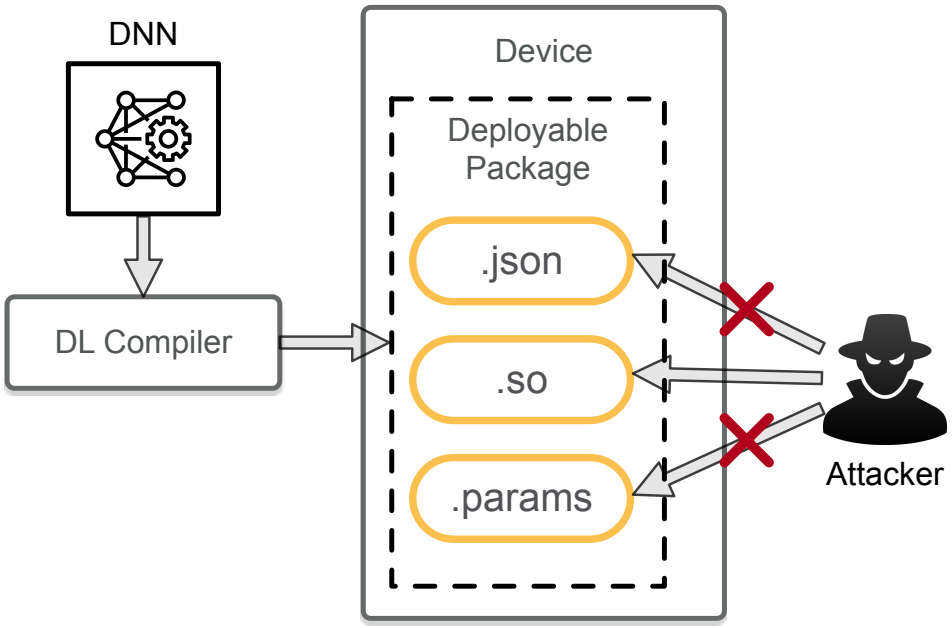


Figure 3.1: Threat Model. The deployable package generated by a DL compiler includes three components: a JSON specification, a runtime library (.so) containing layer functions, and parameter weights (.params). The attacker is restricted to accessing only the runtime library (.so), which holds compiled layer implementations. This limitation aligns with practical deployment scenarios, where configuration files (JSON/params) are more easily secured, and Just-in-Time (JIT) compilation, widely adopted in DL compilers, separates model logic from runtime execution.

3.3.2 Attacker's Capability

This work assumes the attacker is motivated to leak the DNN model architecture information for malicious usage and can only access the shared library. The deployable DNN program generated by DL compilers has two deployment modes, Ahead-of-time (AOT) and Just-in-time (JIT) [81]. AOT scheme uses general-purpose compiler backends and generates self-contained executables. They can run directly on the target devices and serve the same function as the general binary program. On the contrary, JIT will produce two artifacts: a DNN configuration file describing the model and a runtime library that contains all the layer implementations. As shown in Figure 3.1, the deployable package generated by TVM consists of three parts: a JSON-formatted specification file, a runtime library (.so), and parameter weights (.params). Among them, the JSON and the parameter weights files are DNN configuration files, which are important for inferring information about the neural network architecture. The JSON file contains the connection topology between the layers of the DNN models, and the shared library

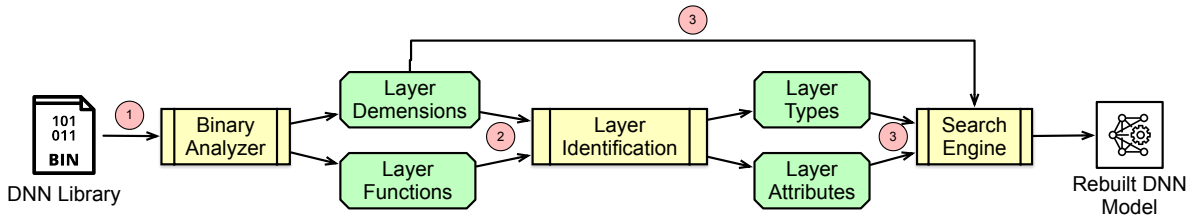


Figure 3.2: LibSteal Workflow Overview. ① The binary analyzer takes DNN runtime library as input and disassembles it to get the sliced layer function. With further analysis, we also extract the layer dimensions at this step; ② At the layer identification step, we leverage the immutable computation pattern to compare the similarity of unknown victim layer functions with customized candidate layer functions and identify their layer types and attributes. ③ Based on layer types, attributes, and dimensions, the search engine finds the connectivity between layers and rebuilds the network architecture of the model.

includes all the unique layer functions. A survey [81] shows that the JIT scheme is more popular and adopted by a majority of the DL compilers. Moreover, as the data files, the JSON and parameter weights files are much easier to protect than the runtime library files. Therefore, limiting the attacker's capability to only access runtime library files is reasonable and has a practical impact.

3.4 Attack Design

3.4.1 Overview

As shown in Figure 3.2, our attack framework consists of three parts, i.e., the binary analyzer, the layer identification module, and the search engine. Correspondingly, the attacking process has three steps. First, we feed the DNN library to the binary analyzer. The analyzer disassembles the binary and slices it into different layer functions. We leverage the information from the data section to extract the layer dimensions for each layer function. Moreover, we apply nested loop analysis to identify each layer function's existing nested loop. The details will be discussed in Section 3.4.2.

Then we pass the extracted layer dimensions and sliced layer functions to the next step to identify the layer type and attributes of the layer functions. We use the layer dimensions to generate all possible layer functions with the exact I/O dimension and store them in the candidate layer repository. Since the computation pattern of each kind of layer remains the same across different models, the layers with the exact layer

dimensions, type, and attributes will lead to very similar layer functions. Therefore, in order to obtain layer types and attributes, we use collected layer candidates to train the representative learning model so that we can check the similarity between victim layer functions and candidate layers functions. We present the detail in Section 3.4.3.

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

3.4.2 Binary Analysis

The Binary Analyzer uses Uroboros [131] as the binary reverse engineering framework to disassemble and analyze the victim DNN library. Uroboros can recover the relocation information and solve the symbolization problem. With these pieces of information, we slice the program into separated layer functions and recover the precise control flow graph of each function.

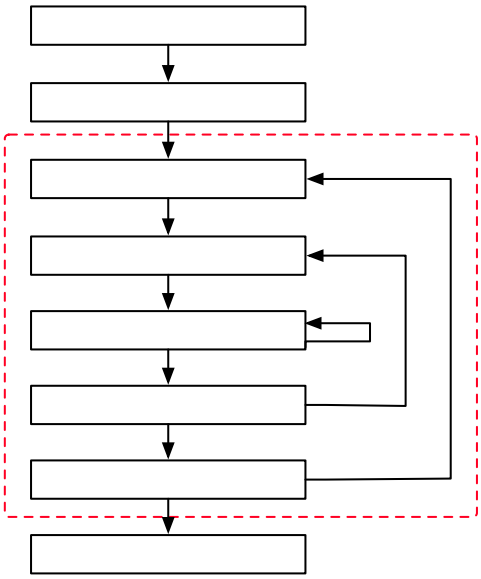
3.4.2.1 Layer Dimensions

This section shows how we infer the layer dimensions. Figure 3.3 shows a part of a layer function from VGG16, which is abstracted from the actual result, where we only replace the recovered symbol with a more readable one. In order to extract the I/O dimensions of each layer function, the first thing we need to do is find its related code in the function. We find that every layer function will check the constraint of the data dimensions before the computation so that the memory will not mess up during the runtime. Therefore, we locate the Basic Block invoking the error report function. In line 9, the code loads the address of `__TVMAPISetLastError`, which sets the last error message before return, to the register `rax`. The function is indirectly called in line 11 with the error message set to `STR` in line 10. According to the message carried by `STR`, this is the exception caused by the input and actual data dimensions mismatch. When we trace back to line 2 in Figure 3.3, we find the comparison between a memory-loaded number and a constant number in line 1 and figure out that one of the data dimension numbers is equal to 1. Following the same routine, we get a set of numbers. The numbers can be separated into groups based on their memory address. For example, four numbers in Figure 3.3 are in one

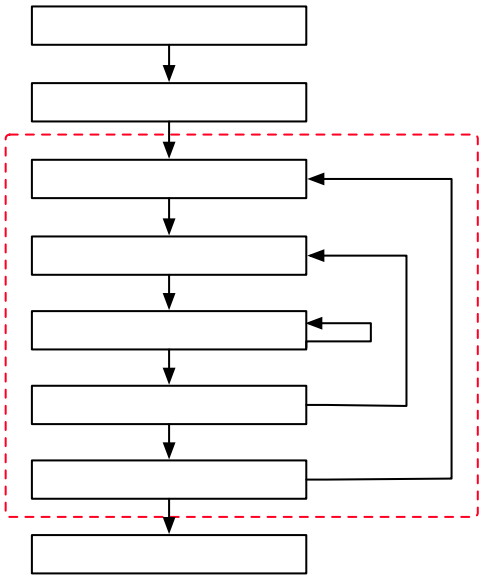
```
   section .text
    ...
   LAYER_FUNC:
    ...
 1  cmp dword [rax], 0x1
 2  jne LABEL
 3  cmp dword [rax + 0x8], 0x40
 4  jne (label)
 5  cmp dword [rax + 0x10], 0x20
 6  jne (label)
 7  cmp dword [rax + 0x18], 0x20
 8  jne (label)
    ...
   LABEL:
 9  mov rax, qword [LABEL_GOT]
10  lea rdi, [STR]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

Figure 3.3: Example of a layer function from VGG16 DNN model. The complete message carried by `STR` is `"Assert fail: (1==int32(arg.placeholder.shape[0])), Argument arg.placeholder.shape[0] has an unsatisfied constraint: (1==int32(arg.placeholder.shape[0]))"`

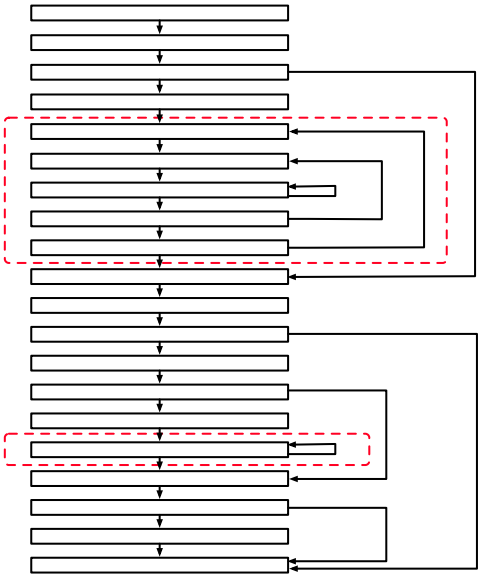
group. According to our observation, each group represents one data dimension of the layer function. Therefore, the data dimension extracted from Figure 3.3 is $1, 64, 32, 32$. Typically, one layer function only has one input dimension and one output dimension. However, some layers require multiple inputs, like `Add` layers whose input can be a list of tensors with the same shape. Fortunately, all constraints of dimension data are checked in order.



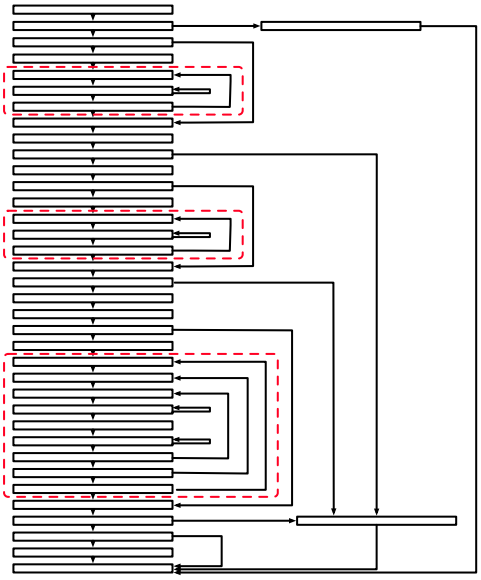
(a) Victim layer



(b) MaxPooling2D layer



(c) AveragePooling2D layer



(d) Conv2D layer

Figure 3.4: The comparison between the CFGs of layer functions with the same I/O dimensions. All four layer functions have the same input dimension (1,64,32,32) and the same output dimension (1,64,16,16). Figure 3.4a is the layer function of figure 3.3; figure 3.4b is the MaxPooling2D layer function with `pool_size=2`; figure 3.4c is the AveragePooling2D layer function with `pool_size=2`; figure 3.4d is the Conv2D layer function with `filters=64, kernel_size=2, strides=2`, and the padding option is `"same"`.

Algorithm 1 Nested Loop Analysis Algorithm

Input: $G = (V, E)$ {Control flow graph}

$stack \leftarrow \emptyset$
$stack.push(G.entryBlock)$
$visitedBlocks \leftarrow \{G.entryBlocks\}$
while not $stack.empty()$ **do**
  $u \leftarrow stack.top()$
  if $v \in u.successors$ **and** $v \notin visitedBlocks$ **then**
    $stack.push(v)$
    $visitedBlocks.add(v)$
  else
    $stack.pop()$
    $timestamp(u)$
  end if
end while
$V' \leftarrow V$
$E' \leftarrow reversed(E)$ {the directions of all edges are opposite}
$G' \leftarrow (V', E')$
$nestedLoops \leftarrow \emptyset$
while not $V'.empty()$ **do**
  for $u \in V'$ **do**
    $v \leftarrow u : v \ ?\ u.timestamp > v.timestamp$
  end for
  $blockSet \leftarrow traverse(G', v)$
  $V' \leftarrow V' - blocksSet$
  if $blockSet.size() > 1$ **or** $v.isSelfLoop()$ **then**
    $nestedLoops.add(blockSet)$
  end if
end while

3.4.2.2 Nested Loop Analysis

The main goal of nested loop analysis is to find the most significant computation features of each layer function. After the DNN model is compiled into the binary, the layer functions contain computation code and trivial instructions like push/pop and data load/store. The existence of these instructions will affect the effect of the representation learning in the next step. Moreover, as shown in Figure 3.4, different types of layer functions have different numbers of nested loops, which is reasonable because the computation of pooling layers is relatively more straightforward than the Conv layer. This feature can be used to validate the result of the layer identification process to increase accuracy.

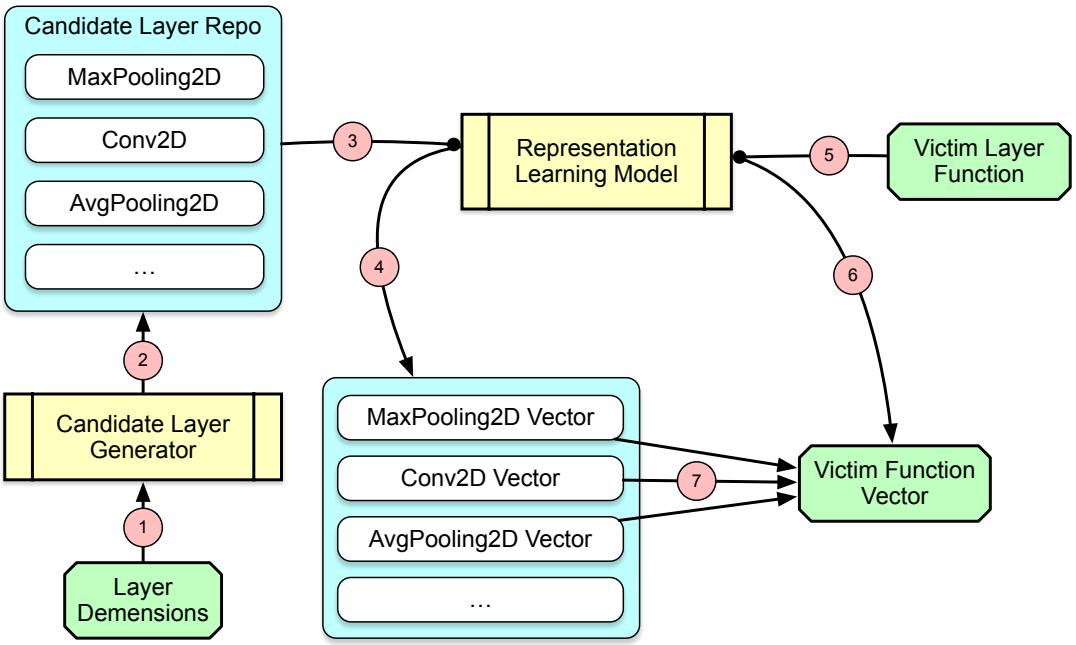


Figure 3.5: The basic workflow of layer identification. The candidate layer generator uses the layer dimension information (①) to generate the potential layer functions (②). The generated candidate layer functions are then used to train the representation model (③) and produce vectors for each candidate (④). Also, we use the trained representation model to generate the vector of the victim layer function (⑤, ⑥). Finally, we compute the similarity between candidate function vectors and victim function vector (⑦) to choose the most similar to infer the layer type and attributes of the victim layer function.

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

3.4.3 Layer Identification

Figure 3.5 shows the basic workflow of the whole process of layer identification. In this section, we provide more details and insights.

3.4.3.1 Layer Generator

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

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

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

where P is the value of padding, D is the value of dilation, K is kernel size, and S is the stride value. Therefore, based on the layer dimensions, we generated a decent number of the candidate layer functions. As shown in Figure 3.4, (a) is the computation part of the layer function we discuss in Figure 3.3. After applying the method we present in Section 3.4.2, we get its I/O dimensions as $(1, 64, 32, 32)$ and $(1, 64, 16, 16)$. There are several possible layers with this kind of I/O dimensions. We choose three representative layers to illustrate our approach: MaxPooling2D with `pool_size=2` (b)), AveragePooling2D with `pool_size=2` (c), and Conv2D with `filters=64, kernel_size=2, strides=2, padding=same` (d). As we can see, the CFG of the victim layer function shown in (a) is the same as the layer function shown in (b) (MaxPooling2D).

3.4.3.2 Layer Function Representation Learning

Our representation learning model is built based on the idea proposed by Asm2Vec [36]. After finishing the training of the representation learning model, we first use the nested loop analysis approach in Section 3.4.2.2 to lift the computation pattern of candidate

layer functions. We then use the trained model to produce the vector of each candidate layer function. The vector of the victim function is also generated from the nested loop of the functions. Finally, we calculate the similarity score and make the inference decision for the victim function.

As for the example shown in Figure 3.4, the similarity between the victim layer and MaxPooling2D is 0.9356, while its similarity with AveragePooling2D is 0.522 and with Conv2D is 0.272. Therefore, we determine that the layer function from VGG16 is MaxPooling2D with `pool_size=2`. This result is reasonable because the computation patterns MaxPooling2D and AveragePooling2D are similar. They follow the same routine to slide through the data. However, the final operations are different.

3.4.4 Model Reconstruction

After the binary analysis and layer identification, we have stolen valuable information, including layer dimensions, types, and attributes. In this section, using all the above information, we rely on the search engine to infer the network architecture.

At first, we build a directed graph, $G_{nn} = (V_{nn}, E_{nn})$, where each layer represents a vertex v, and an edge e, pointing from v_1 to v_2 means the output dimension of v_1 matches the input dimension of v_2. We search all possible combinations from the input layer, whose output dimension is the data resolution of the model, to the output layer, whose input dimension is the number of model classifications. We also did some heuristic pruning based on the layer types, such as activation layer and batch normalization. Typically, these layers do not affect the data resolution, so their layer dimensions stay the same. Furthermore, the layer functions of these supportive layers are the same when they have the exact I/O dimensions because they do not have variable attributes, which means different supportive layers may share the same layer functions. If we directly add them to our search space, much effort will be wasted. However, these layers are critical in the training and inference, so we cannot ignore them. Therefore, we divide the layer into supportive layers (e.g., ReLu, BN) and functional layers (e.g., Conv2D, Pooling2D). Each functional layer can only be used once during the search and should all be used in the final rebuilt model. As for the supportive layers, we do not limit their usage. Once a functional layer finds out it can link supportive layers, we add them directly to the path. Additionally, when we meet the layer with multiple inputs, we trace back the graph to find another input and link it to the former path. Once we find the path that is satisfying, we can reconstruct the DNN model.

Table 3.1: Victim Models Information.

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

Table 3.2: Statistics of the comparison between the original model and extracted model.

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

3.5 Evaluation

In this section, we will present the evaluation result of our framework. We implement the prototype of our attack framework based on Uroboros [131] and adopt the idea of asm2vec [36] to achieve layer-type identification. we evaluate LibSteal by answering the following research questions (RQs).

- ***RQ1: (Architectural Completeness)*** *How complete is the reconstructed DNN model compared to the original victim DNN model?*
- ***RQ2: (Accuracy of Extracted Models)*** *Does the DNN model reconstructed by* LibSteal *still apply the inference functionality properly?*

3.5.1 Experiment Setup

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

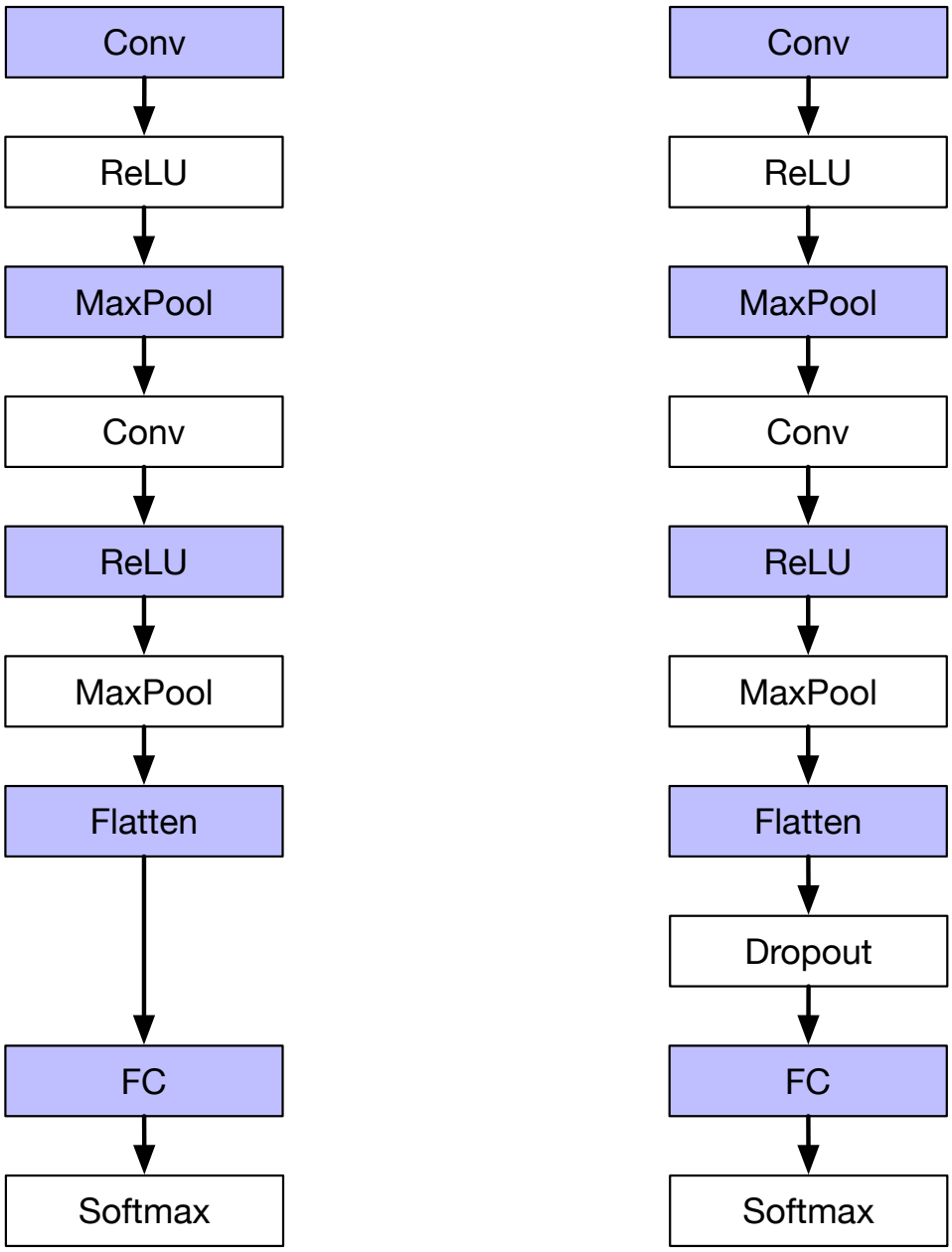


Figure 3.6: Architecture Comparison For MNIST. This figure partially shows the network architecture difference between the original and extracted models.

Victim Model: We evaluate our attack on four widely-used DNN models: MNIST, VGG16, ResNet20, and MobileNet. The detailed information of all victim models is shown in Table 3.1. All the pre-trained models are designed in the Keras framework [67, 68] with Tensorflow as the backbone and compiled by TVM [22] to generate the binaries. We use LLVM [75] as our host platform.

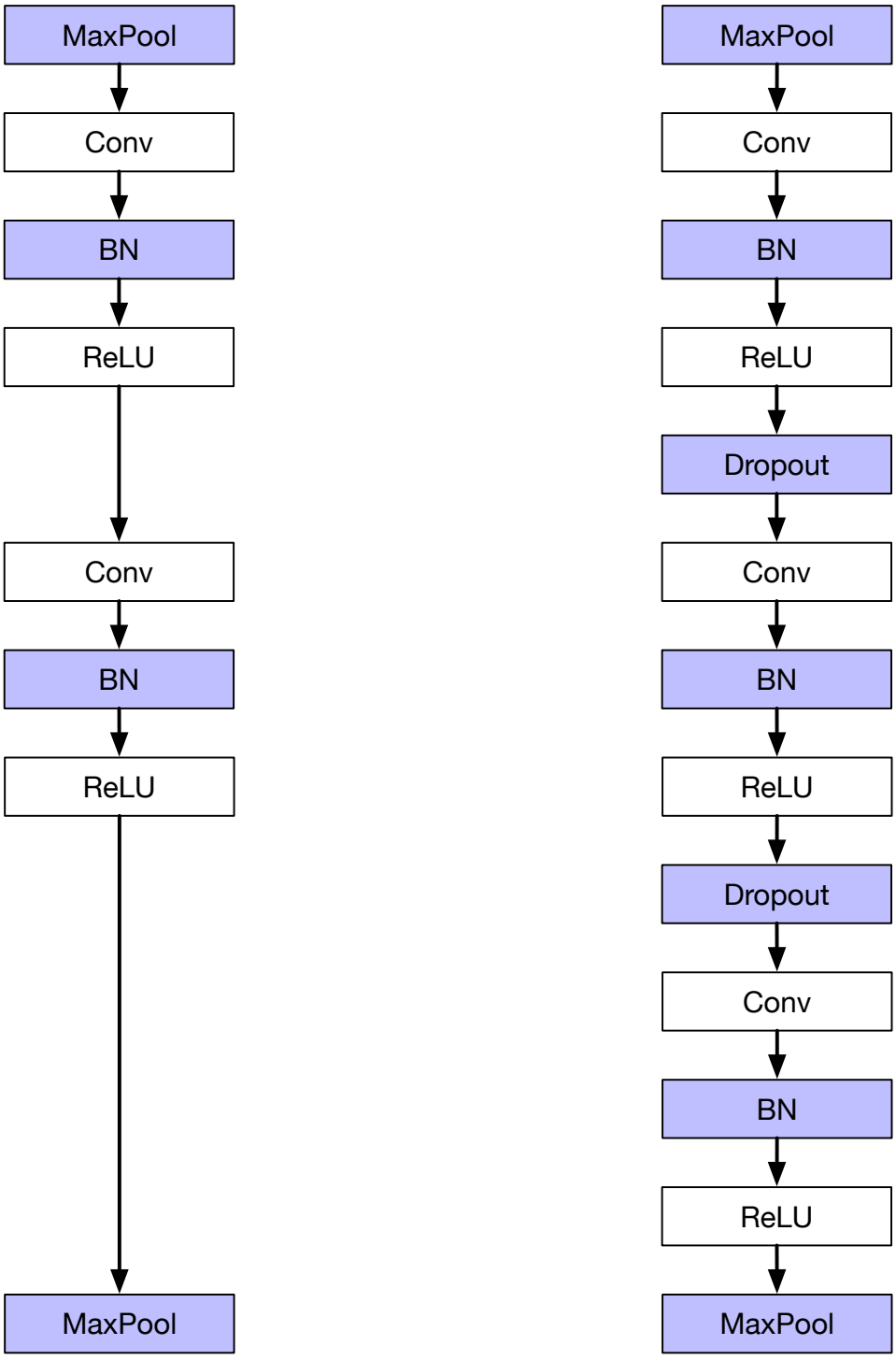


Figure 3.7: Architecture Comparison For VGG16. This figure partially shows the network architecture difference between the original and extracted models.

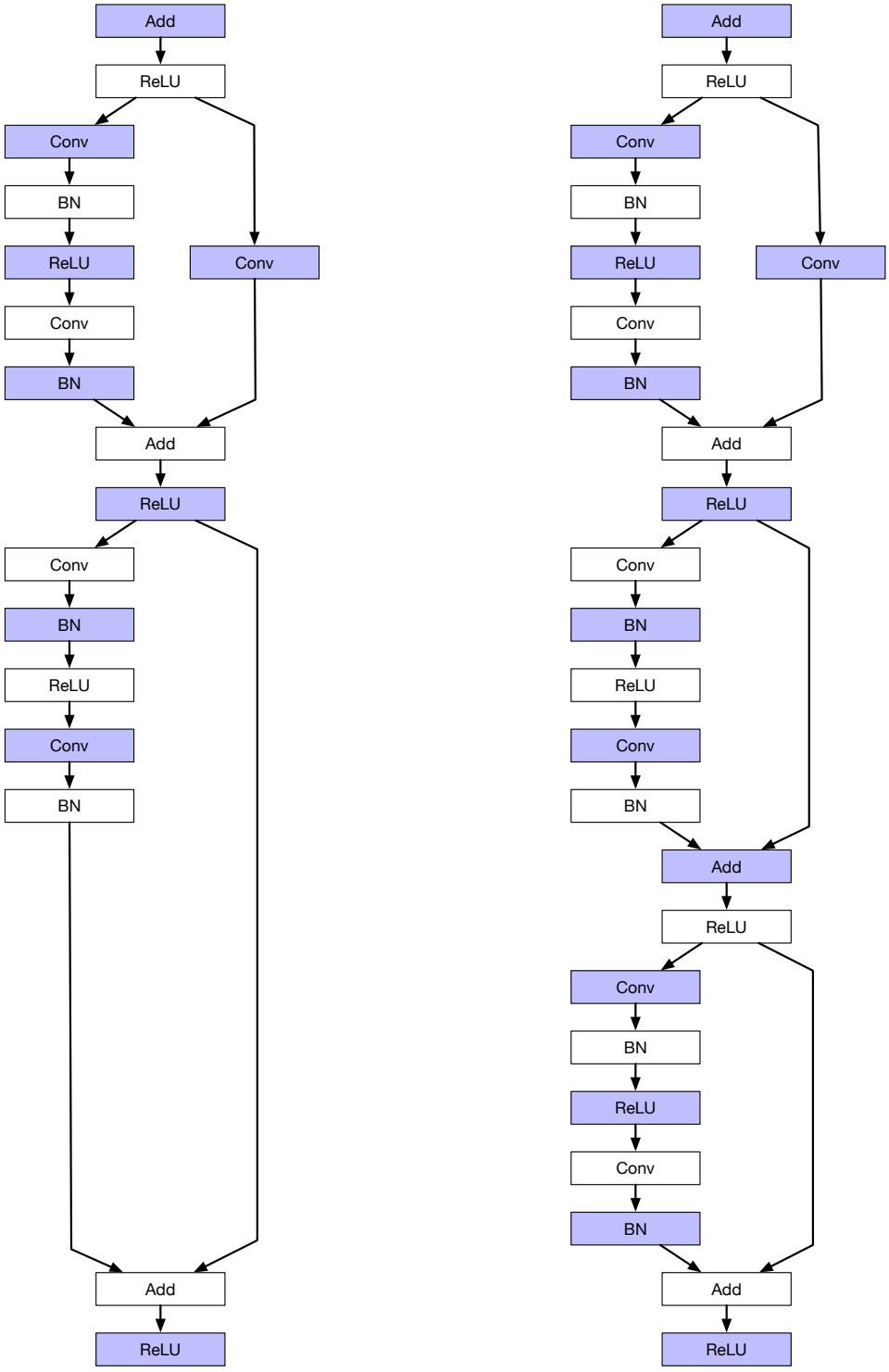


Figure 3.8: Architecture Comparison For ResNet20. This figure partially shows the network architecture difference between the original and extracted models.

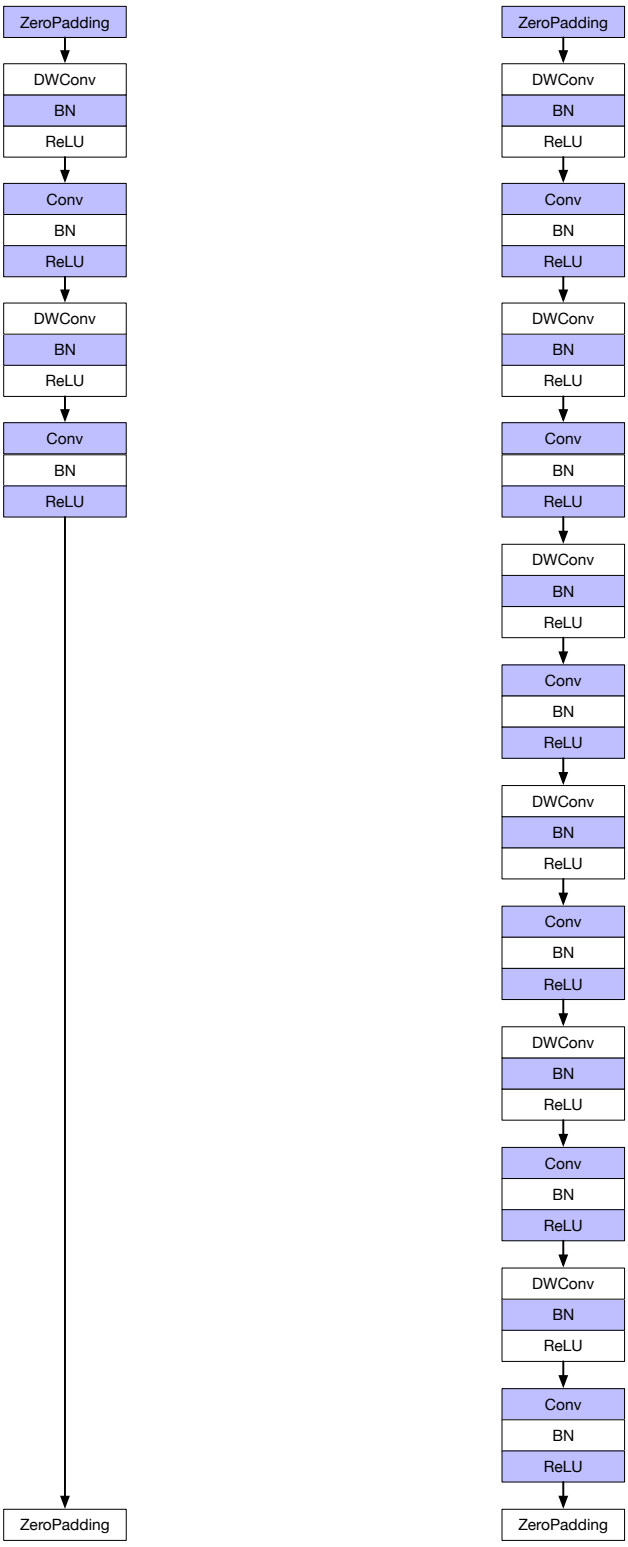


Figure 3.9: Architecture Comparison For MobileNet. This figure partially shows the network architecture difference between the original and extracted models.

3.5.2 (RQ1) Architectural Completeness

This section compares the architectures of the original models and the extracted models. Figure 3.6, Figure 3.7, Figure 3.8, and Figure 3.9 visualize part of the architecture of original models and extracted models for MNIST, VGG16, ResNet20, and MobileNet. Each block represents a DNN layer. As we can see, the basic architecture remains the same in extracted models. However, several layers are missing. First, the dropout layer is used to avoid over-fitting during the training phase by setting randomly selected input units to 0, so it will be invalid during the inference. Therefore, the DL compiler does not compile it into the layer function, and we cannot extract its related information. Nevertheless, this does not invalidate the purpose of the attack since the lack of dropout layers does not hurt the performance of the extracted models.

Regarding other missing layers, as we mentioned in Section 3.4.3, the same type of layer with the same input dimension, output dimension, and layer attributes will produce the same layer function. This rule of thumb allows us to identify the layer type and its related attributes but also leads to the situation that the layer function can be reused in the DNN executable. As shown in Figure 3.9, the recovery of MobileNet is the least satisfying. The reason for missing a larger chunk in this model is that the original DNN model is the composition of the same pattern repeated four times. Since our attack targets the DNN library, inferring the number of repetitions from the code is difficult, as the same code can be executed arbitrarily during inference. On the other hand, Figure 3.6 shows that as long as all the layers of the DNN model have their unique features, we are able to fully recover the whole network architecture.

3.5.3 (RQ2) Accuracy of Extracted Models

We compared other statistics information between original DNN models with the extracted ones as shown in Table 3.2. In order to evaluate the functionality of the extracted model, we re-trained the extracted models and compared their accuracy with the original models. The training and testing dataset (MNIST and CIFAR-10) is obtained from *keras.datasets*. MNIST [77] is the dataset of hand-written digits with 60,000 training and 10,000 test gray-scale images. CIFAR-10 [72] contains 60,000 training and 10,000 test colour images. As for the training settings, we use the same hyperparameter as the original one for a more precise comparison. For the MNIST model, the extracted model achieves 99.04% accuracy compared to the original 99.17% accuracy. The extracted VGG16 model and ResNet20 model achieve 90.59% accuracy compared to the original 93.16% and 83.92%

accuracy compared to the original 91.65% accuracy, respectively. Moreover, extracted MobileNet model achieves 75% accuracy compared to the original 83.16%. With the missing layers, it is not surprising that the accuracy of these three extracted DNN models is worse than the original ones. However, we can see that the accuracy drop of ResNet20 and MobileNet is more significant than that of VGG16, although we recover more layers for ResNet20 and MobileNet. We guess that the importance of layers varies inside the neural network architecture. Although VGG16 missed more layers, the key skeleton still remains. As for the layer types, so far, the only layer we are not able to recover is the Dropout layer, which will disappear during the inference process.

Chapter 4
Protecting Deep Neural Network Program from Reversing Attacks

In this chapter, we propose FLATD [147], an advanced defense mechanism that protects Deep Neural Network Programs from reversing-based model extraction attacks while preserving efficiency.

4.1 Introduction

Due to the widespread success [47,50,66,78] of Deep Learning (DL) across various domains, the demand for DL-based services has surged in recent years. This has led service providers to deploy the Deep Neural Network (DNN) models across a wide range of hardware devices, from cloud servers to embedded devices [46], to meet diverse requirements. However, deployment across multiple platforms presents challenges due to the differences in on-chip memory architecture and compute primitives across CPU, GPU, and TPU-like accelerators [61]. Additionally, the rapid growth of DL frameworks [1, 21, 26, 93] further complicates the situation.

The DL compilers [22, 76, 91, 116] ease this process by automatically compiling the models into standalone DNN programs with decent optimization using multiple intermediate representations (IR) during compilation. Generally, a DL compiler can support various frameworks as input and generate programs for different hardware devices. Some also [22, 116] allow third-party toolchains such as LLVM [75] and CUDA [136] for further code generation. The powerful automated optimization provided by DL compilers suddenly attracted the attention of both academia and industry. Gaint AI providers such as Google, Amazon, and Facebook are all considering embedding the DL compilers into their AI infrastructure to enhance the performance of their AI services. [5, 60, 88, 97, 134].

While the DL compilers significantly impact the AI industry, they also introduce a new attack interface from the binary analysis side. Due to the lack of defense applied at the binary level, the DNN programs are vulnerable to reversing-based model extraction attacks. The targets of traditional model extraction attacks [41, 55, 56, 103–105, 135, 139, 143, 152] can mainly be classified into three categories: side channel information, sniffing bus traffic, and prediction pairs from black box models. These information sources are limited and sometimes depend on strict assumptions. Unlike these sources, DNN programs always contain complete information that can be used to run in an isolated environment. To date, there are four state-of-the-art model extraction attack frameworks [20, 89, 138, 146] that can fully or partially reconstruct models by reversing the DNN programs. However, to our knowledge, effective defense mechanisms still need to be developed to countermeasure these reversing attacks. Moreover, training DNN models at an industry scale often involves processing TB-sized datasets [33, 141] with high training costs. For example, using a v2 Tensor Processing Unit (TPU) in a cloud environment costs approximately $4.50 per hour, and completing an entire training cycle may exceed $400,000 [35, 110], which emphasizes the importance of protecting DNN models.

Unlike the typical binary program, the DNN program is generated directly from the model without any source code, which excludes source-code-level defense frameworks like Tigress [29]. Moreover, the DNN Program is more sensitive to performance and scale than a typical binary program, making the time-consuming framework unavailable. Fortunately, most state-of-the-art DL Compilers [22, 116] support third-party code-gen tools (e.g., LLVM [75]) for users to apply the customized transformation, which leaves us the window to shield DNN programs.

We carefully investigated the basic logic and workflow of attacking frameworks to gain more insight into the reversing-based model extraction attack. These frameworks include the same components to rebuild the model: operator-type recovery, topology recovery, and metadata recovery (including dimensions, parameters, and attributes). Although the methodologies vary from framework to framework, they share the idea of using the computation pattern to recover the operator type. Specifically, each kind of operator in the DNN model has a formula for transforming the input data to the next operator. For instance, the ReLU activation function uses the formula 4.1, and the Tanh activation function uses the formula 4.2. They exhibit entirely different syntax and semantics meanings when represented in the program. This feature helps attackers infer the operator type by using binary similarity comparison. On the other hand, it also

guides the protection of DNN programs because we found that the Control Flow Graph (CFG) plays a vital role in all attack frameworks.

$$f(x) = x^+ = \max(0, x) = \frac{x + |x|}{2} \tag{4.1}$$

$$f(x) = \frac{e^x + e^{-x}}{e^x - e^{-x}} \tag{4.2}$$

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

We implemented FlatD on the top of O-LLVM [62] and embedded it into the code generation part of TVM [22]. We used O-LLVM as the baseline and evaluated FlatD on eight real-world pre-trained models and one self-trained model from four frameworks. Our experiment results show that compared to the traditional Control Flow Flattening, FlatD can more effectively counterwork state-of-the-art reversing-based model extraction attacks while preserving the functionality of the original DNN programs. Moreover, the DNN program transformed by FlatD performs similarly to the one using traditional Control Flow Flattening in most cases and always has a lower scale.

In summary, we make the following contributions:

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

- We design and implement FlatD, the advanced defense framework targeting compiled models toward reversing-based model extraction attacks. FlatD conceals the original Control Flow Graph of DNN programs based on Control Flow Flattening and ensures minimal information gained by attackers through statistical analysis.

- We successfully apply FlatD on DNN programs compiled from large-scale models using TVM to evaluate these DNN programs regarding functionality, performance, and resilience. We use O-LLVM as the baseline to compare the results. Our

experiment demonstrates that DNN programs transformed by FLATD can prevent leaking information from reversing-based model extraction attacks more effectively than traditional Control Flow Flattening with similar performance and lower scale.

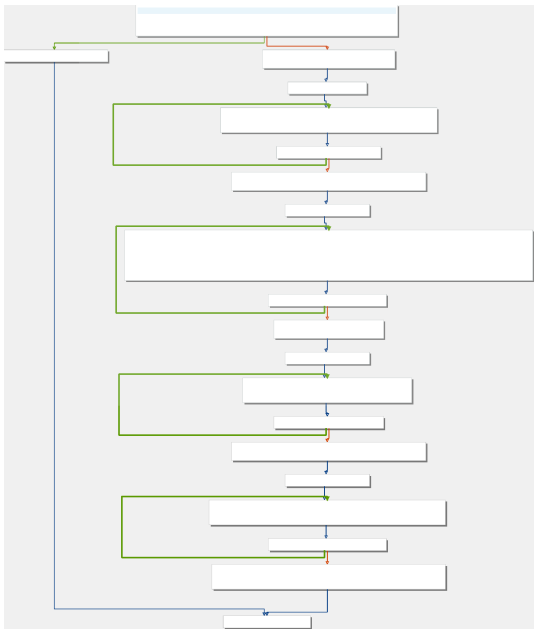
The rest of the paper is organized as follows. We first show the insight of reversing-based model extraction attacks is presented in Section 4.2. Next, we present the workflow and design logic of FLATD in Section 4.3. The experimental results are presented in Section 4.4.

4.2 Inspiration from Attacks

Table 4.1: Reversing-based Model Extraction Attacks. (★stands for fully support and fully recover ☆stands for partial recover.)

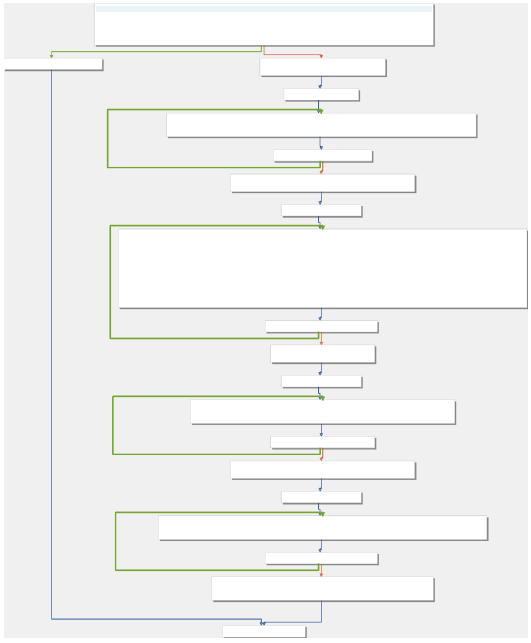
Tool Name		NNReverse	DnD	BTD	LibSteal
Target Compilers	TVM	★	★	★	★
	Glow		★	★	
	NNFusion			★	
Platform Support	x86-64	★	★	★	★
	ARM	★	★		
	AArch64	★	★		
Results	Architecture	★	★	★	☆
	Parameters		★	★	

In this section, we compare the methodology and design logic between four state-of-the-art reversing-based model attacks to determine their similarities and differences. Table 4.1 lists attack frameworks that extract the essential information from the DNN model to rebuild the DNN model by reversing the victim DNN program. Specifically, NNReverse [20] is a learning-based method that can fully recover the architecture from DNN programs compiled from TVM across platforms. DnD [138] implemented a cross-architecture DNN decompiler based on symbolic execution, which can fully recover the architecture and parameters of DNN programs compiled from TVM and Glow. Instead, BTD [89] focuses on the decompiling DNN Programs on x86 platforms. By utilizing the neural identifier model and dynamic analysis, BTD can reconstruct models from DNN programs compiled from three DL compilers. Unlike the above attack frameworks, all targeting the standalone DNN programs, LibSteal [146] partially recovers the DNN architecture using only a shared library. Although the target and methodology vary from frameworks, the logic and workflow align the same. In order to rebuild the original DNN



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

(a) VGG16 softmax function



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

(b) ResNet50 softmax function

Figure 4.1: The comparison of CFG and Opcode sequence of softmax function between VGG16 (Figure 4.1a) and ResNet50 (Figure 4.1b) DNN Programs, which compiled by TVM with optimization -O0. VGG16 and ResNet50 are pretraining on ImageNet and loaded from Keras Application Zoo. Therefore, they have the same output dimension (1,1000). However, the input dimension of the VGG16 softmax function is (1,4096), while the input dimension of the ResNet50 softmax function is (1, 2048).

model, the following characteristics must be recovered: operator types (Section 4.2.1), topology (Section 4.2.2), and metadata, including dimensions, attributes, and parameters (Section 4.2.3). We detail the review of each part below and summarize our findings in Section 4.2.4.

4.2.1 Operator-type Recovery

The critical component of each attack framework is to recover operator types of DNN programs. The idea of operator-type recovery is also the most straightforward due to the unique computation pattern of each DNN operator. We take the softmax function as an example. As shown in Figure 4.1, both VGG16 and ResNet50 are loaded from Keras Application Zoo and compiled by TVM with the configuration -O0. Figure 4.1a is the CFG and opcode sequence of the VGG16 softmax function, and Figure 4.1b is the CFG and opcode sequence of the ResNet50 softmax function. Both the CFGs and opcode sequences are obtained from Binary Ninja [58]. We can see that no matter which feature is compared, the VGG16 softmax function is almost identical to the ResNet50 softmax function, although they do not even have the exact input dimensions. With this insight, all reversing attacks coincidentally choose to infer operator type based on binary similarity. NNReverse combines the syntax representation (opcodes and operands) and topology representation (CFG) to train an embedding model and find the most similar function in the dataset based on the semantic representation. DnD uses symbolic execution to lift each operator function to an Abstract Syntax Tree (AST) and match it with a template AST to infer the operator type. BTD chose to train a model with a sequence of Atomic Opcodes from each DNN function and predict the operator type. LibSteal trained a representation model with loop structures extracted from each operator function and compared the victim operator function with functions in the dataset.

4.2.2 Topology Recovery

At this step, attack frameworks start to use different methodologies to achieve the goal. NNReverse directly leverages the graph file generated along with the shared library file to recover the topology of DNN architecture. DnD reconstructs the DNN topology structure by utilizing the sequence of DNN operator executions within the inference function and the data dependencies among the DNN operators. BTD uses the Intel PinTool [90] to hook every callsite as the operator function's inputs and outputs are transmitted via memory pointers in the function arguments. Subsequently, BTD seamlessly links the

operator function using the identical memory address. Since LibSteal can only access the shared library file, it heuristically searches the possible topology combination to link all the inferred operators.

4.2.3 Data Recovery

Recovering the data of the DNN models, including parameters, attributes, and dimensions, is tricky. LibSteal does not support the recovery of parameters due to limited resources. However, it extracts the dimensions and attributes information about each operator by analyzing the data flow of the functions that validate the input data before the computation. NNReverse still uses parameter files directly generated by the TVM. As for BTD, the whole workflow includes multiple steps. First, it records execution traces and applies taint analysis to condense them. Following this, symbolic execution is utilized to summarize the input-output constraints of each operator function, which is used to infer dimensions and parameters later. DnD recovers the attributes and parameters of each operator by leveraging the lifted AST and reconstructed topology structure.

4.2.4 Inspiration

Upon conducting a comprehensive analysis of the workflows employed in four cutting-edge reverse-engineering-based extraction attacks, it becomes evident that the type of operator emerges not only as the most foundational aspect to be considered during the execution of an attack but also as the critical characteristic warranting robust protection. This assertion is confirmed in Section 4.2.1 and visually illustrated in Figure 4.1, where we detail how the unique computational patterns of each operator cause their corresponding functions to be susceptible to inference attacks.

A key observation from our analysis is that these distinct computational patterns give rise to specific structural features within the CFG, especially loop structures. It is widely acknowledged that operations within DNN models mainly involve matrix computations, which result in a nested loop structure within operator functions. Such structures are not merely incidental but serve as significant signatures that attackers can exploit to their advantage. Therefore, defensive technologies capable of obscuring the original CFG structure, as discussed in Section 2.2.3, could play a transformative role in mitigating the effectiveness of these extraction attacks. Nonetheless, integrating these defensive mechanisms into DNN programs is challenging because we need to ensure the efficiency of the DNN programs while preserving security and privacy. The balance between enhancing

Algorithm 2 Flattening Algorithm

Input: P {DNN Program}

1: S_F = getAllFunctions(P)
2: **for** $F \in S_F$ **do**
3: **if** F is necessary to be flattened **then**
4: S_{BB} = getAllBasicBlocks(F)
5: breakCFG(F)
6: BB_{old} = getOldEntry(S_{BB})
7: BB_{new} = createNewEntry(BB_{old})
8: T = createDispatcher(BB_{new}) {Return the switch table that guide the control flow in new CFG}
9: attachBBToDispatcher(T, S_{BB})
10: $Salt$ = initializeSalt()
11: F_{hash} = initializeHashFunc()
12: updateSwitchVar(D, $Salt$, F_{hash})
13: createBBAddrTable(S_{BB})
14: encodeSwitchTable(T)
15: F_{decode} = initializeDecodeFunc()
16: updateDispatcher(T, F_{decode})
17: inlineDispatcher(T)
18: **end if**
19: **end for**

Output: Program with flattened operator functions.

security measures and ensuring computational efficiency requires a detailed approach to design, which not only conceals CFG structures but also carefully considers the potential impacts on the performance and functionality of DNN programs. Therefore, we propose FlatD to secure DNN programs against sophisticated extraction attacks.

4.3 Design

4.3.1 Overview

Algorithm 2 uses pseudo code to represent the basic workflow of our defense framework, FlaD. For the given DNN Program P, we extract and iterate all the functions (Line 1-2). Before applying the flattening, we check the specific function's necessity to increase the performance (Line 3). The rules are discussed in Section 4.3.5. Then, we break the

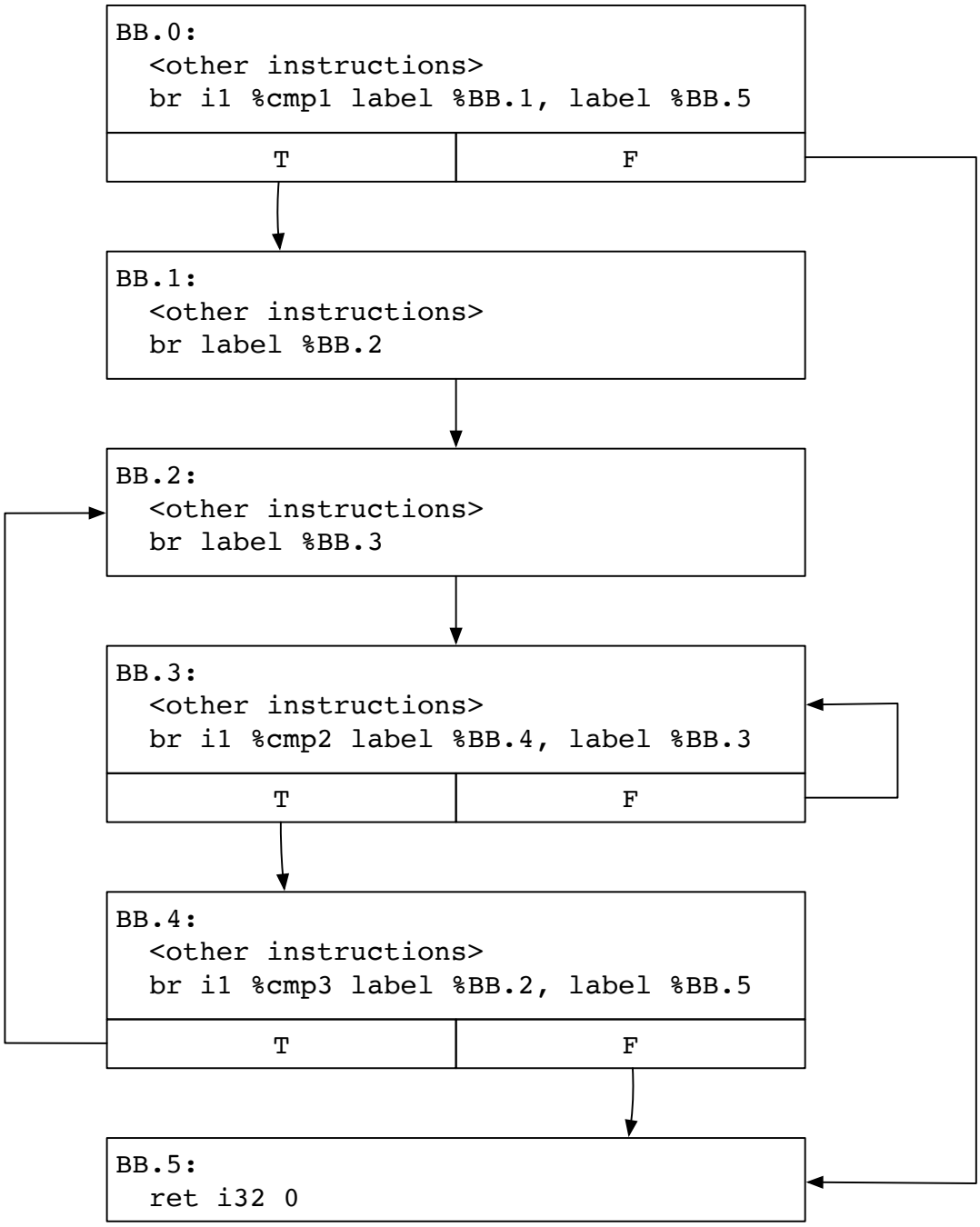


Figure 4.2: Original Control Flow Graph of the ReLU operator function from MNIST compiled by TVM -O0. The figure only shows the control flow-related instructions in LLVM IR format to simplify the graph.

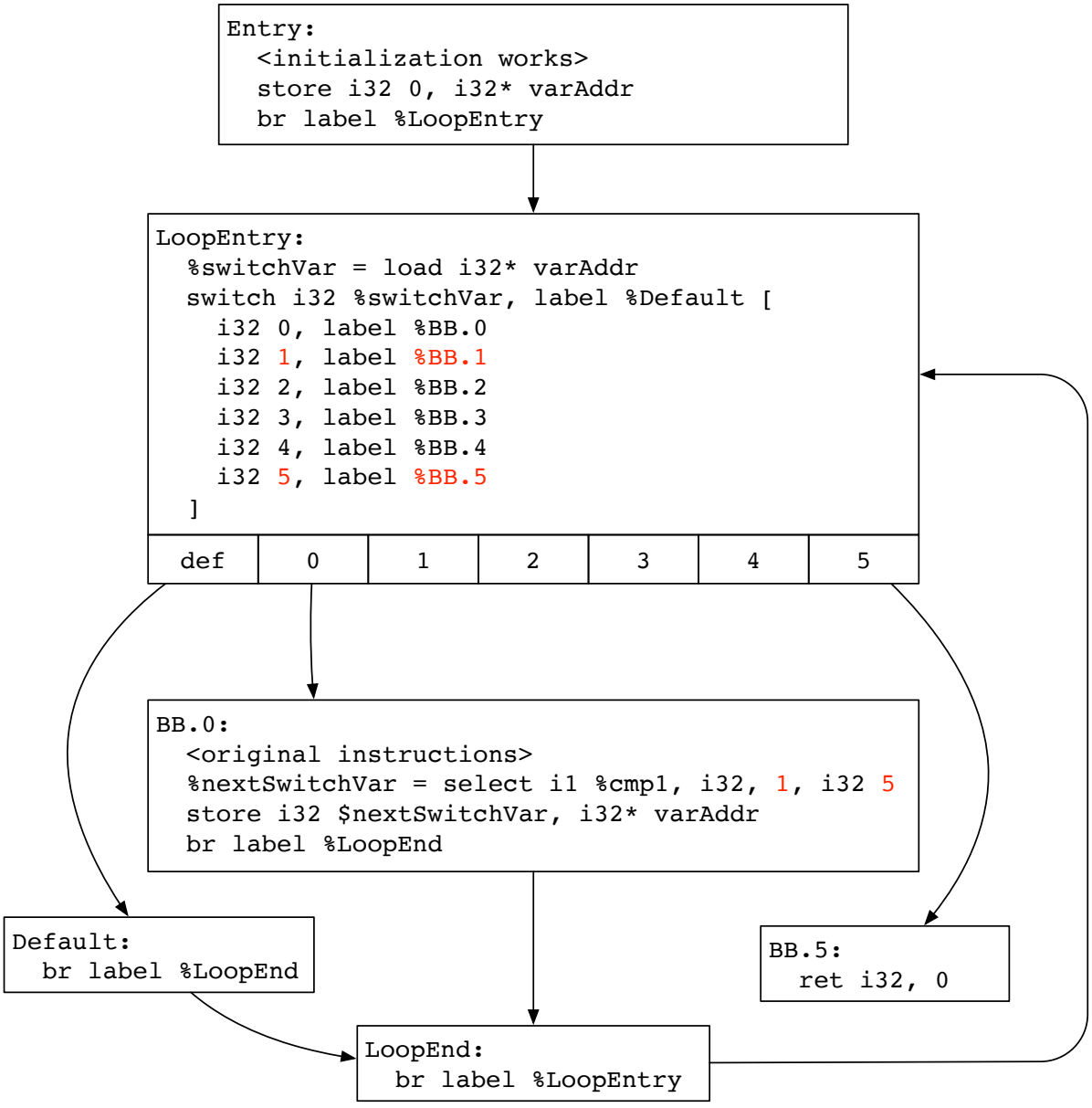


Figure 4.3: CFG after applying traditional Control Flow Flattening to Figure 4.2. The figure only shows part of the resulting CFG because the modifications of all basic blocks are similar except for basic blocks with label `BB.5` and `Default`, where `BB.5` is the exit block of this function, and `Default` is added by switch instruction to avoid assertion. Although we only include flows to `BB.0`, `BB.5` and `Default`, flows to other basic blocks still exist.

original CFG and rebuild a new one using the dispatcher and switch table (Line 4-9), which follows the implementation steps of traditional CFF [74]. However, as shown in Section 4.3.2, traditional CFF is still vulnerable to static analysis.

To increase the robustness of the resulting program, we continue transforming using the following strategies. First, we hide the visible label by introducing the hashing method and a randomly chosen secret, `Salt` (Line 10-12). We provide more details in Section 4.3.3. Second, to further hide the loop structure, we create a table containing the address of each Basic Block and encode the switch table, which are both stored in the global variable list (Line 13-14). Then, we create a decode function (Line 15-16) to accomplish the indirect control flow. The function decodes the address of the corresponding basic block from the encoded switch table and switch variable. Finally, we inline the dispatcher to each basic block to hide the loop structure completely. Section 4.3.4 takes an example of the final operator function to illustrate the whole process.

4.3.2 Traditional Control Flow Flattening

Figure 4.2 shows the original CFG of the ReLu operator function from a simple MNIST convnet model [28] compiled from TVM [22] with optimization configuration -O0, and Figure 4.3 demonstrates the CFG after flattening. We omit unnecessary instructions for both figures and only retain the control flow related instructions to make the figures clear and tidy. Moreover, the code in each basic block is represented in LLVM assembly language format (LLVM IR) because our defense mechanism is implemented on top of LLVM. For Figure 4.3, we do not include all the basic blocks originally shown in Figure 4.2 because the modification applied to all basic blocks is similar except for basic block `BB.5` and `Default`, where `BB.5` is the exit block of this function, and `Default` is added by switch instruction to avoid assertion by default. Therefore, we choose one basic block (`BB.0`) to show the basic logic and explain the vulnerability of traditional CFF. As we can see in Figure 4.3, after flattening, all basic blocks in the original implementation share the same dominator, `LoopEntry`, and post-dominator, `LoopEnd` (except for Basic Block `BB.5`). Instead of condition jump controlled by `%cmp1` and `br`, the dispatch variable `%switchVar` manipulates the control flow. Although the CFG is transformed, attackers can reconstruct it by analyzing the operand value of the selection instruction in `BB.0` with the operand value of the switch instruction in `LoopEntry` to get the successors of each basic block. For example, as shown in Figure 4.3, we can infer that either `BB.1` or `BB.5` can be the next execution target after the execution of `BB.0` (marked as red).

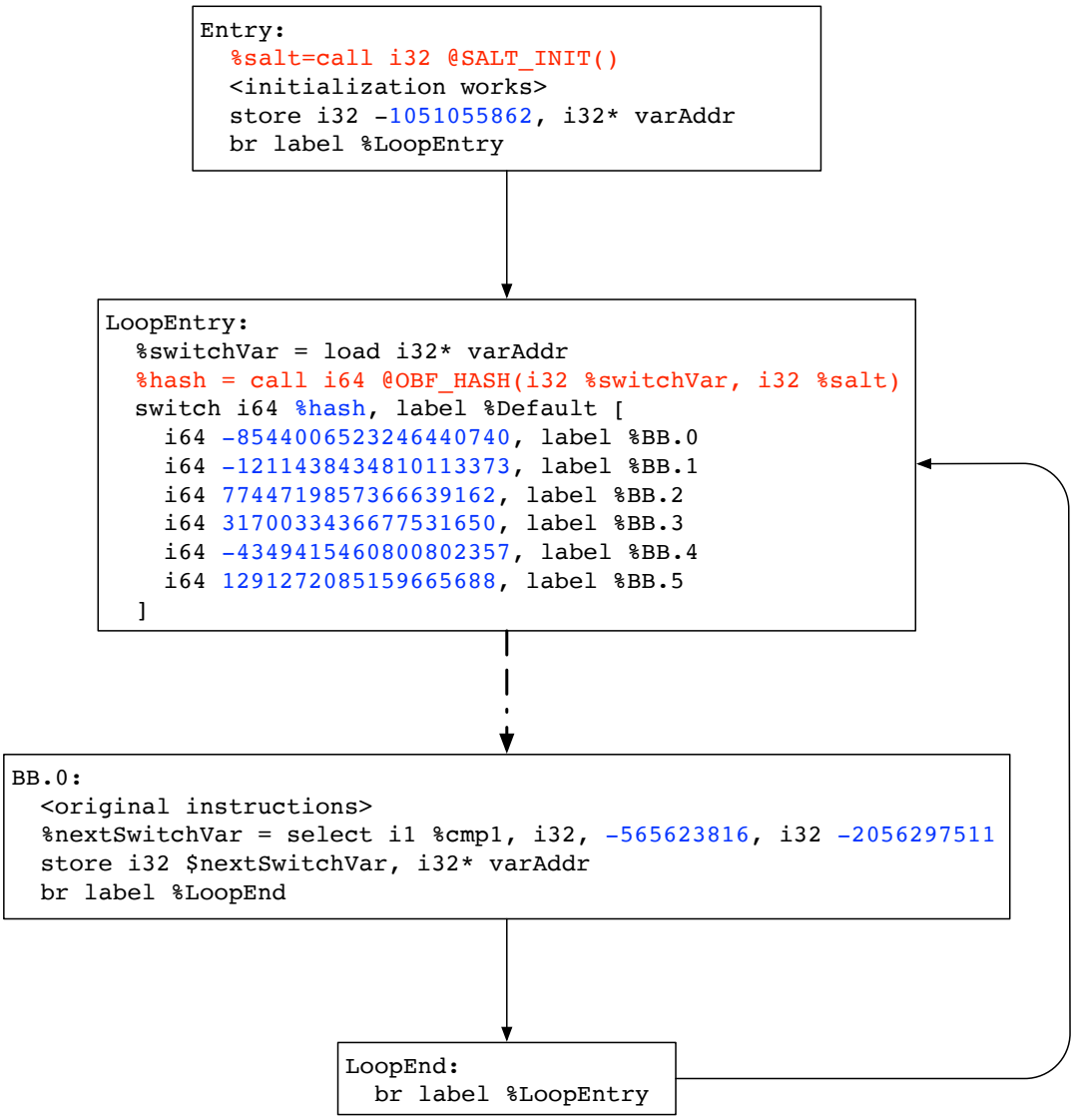


Figure 4.4: CFG after hiding the visible label of Figure 4.3. To achieve the goal, we introduce a 32-bit secret number, `%salt` and initialize it (red) at the Basic Block `Entry`. Then we compute `%hash` using a one-way cryptographic hashing function (red) based on the old `%switchVar` value and `%salt`. Finally, we use the value of `%hash` to determine the control flow. In this case, the value assigned to `%switchVar` does not show in the switch table anymore (blue).

4.3.3 Hide Visible Label

The first strategy to increase the resilience of traditional CFF is to hide the statically visible dispatch labels by introducing secret information and employing one-way cryptographic hashing. Figure 4.4 shows the CFG after hiding the visible label (marked as blue) of Figure 4.3. The value of `%salt` is initialized at the Block `%Entry` (marked

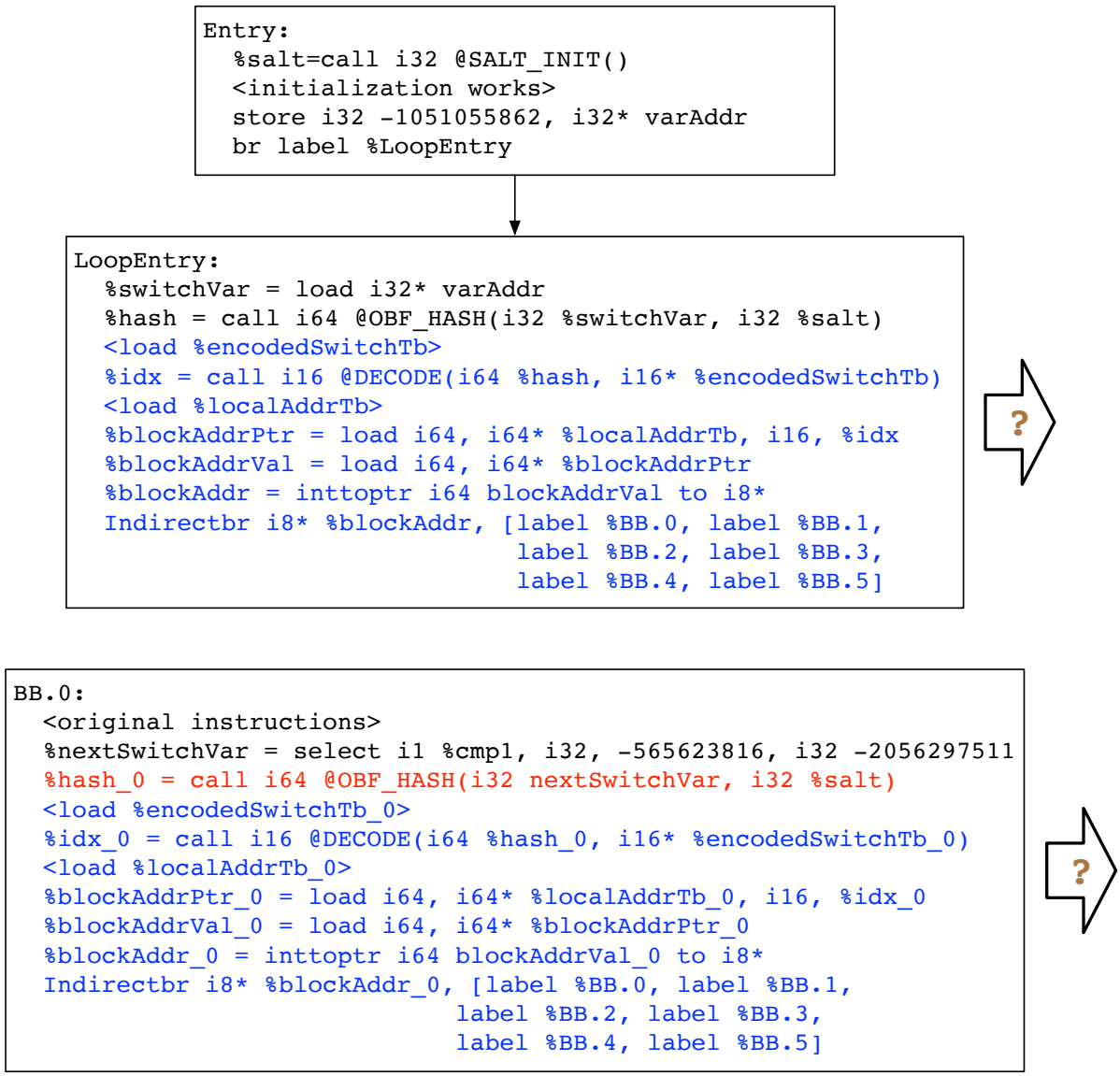


Figure 4.5: CFG after hiding the loop structure of Figure 4.4. Instead of directly using hashed dispatcher label `%hash` to determine the subsequent control flow, we use it to decode the switch table and get `%idx` to retrieve the address of the target basic block in the basic block address table so that we can implicitly go to the following basic block. The potential candidate can be all the original basic blocks. Moreover, the main body of the dispatcher is inlined into each basic block, and the loop structure is completely removed.

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

4.3.3.1 Hash function

We want to ensure the hash function has preimage resistance, which means finding any input that maps to a given output hash is computationally infeasible. In other words, given a hash value h, it should be tough to find any original input x such that $hash(x) = h$. This property is crucial for security, as it prevents attackers from reverse-engineering the hash to discover the original data. The most common method used in practical life is SHA-256/512 [49]. Preimage resistance keeps the original data secure and practically impossible to deduce even if the hash value is exposed. This feature perfectly fits our requirements since the hash values are used in the switch table to determine the following control flow, and we do not want the attackers to match it with the original data in each basic block. Moreover, the overhead of the hash function is relatively low and does not affect the performance of DNN programs.

4.3.3.2 Salt computation

Although preimage resistance prevents attackers from inferring the original data from the hash value, they can directly compute the hash value from the original data because regardless of which hash function we choose, its body is included in the DNN program file. Therefore, it is necessary for us to introduce a secret value, `%salt`, and keep it unknown from attackers. The `%salt` is computed at run time and should not be easily statically revealed to make attackers unable to recover the original CFG statically. To achieve the goal, we leverage the concept of opaque predicate [30, 38], which can either be a constant function that does not look constant, like the value of $x * (x + 1)\%2$, which always returns 0 regardless of the value of x, or the fact only known to developers, like the color of a particular pixel of some app resource. An adequate opaque predicate should be resilient to static analysis. Therefore, for the computation of `%salt`, the compiler first randomly generates for each function. At run time, each bit of the `%salt` is computed by a "query," which can be an opaque predicate, making `%salt` computation statically obscure and

dynamically confidential. Since the computation of `%salt` is a one-time job for each function, it also does not affect the time overhead of the DNN programs.

4.3.4 Hide Loop Structure

After hiding the statically visible dispatcher label, attackers have already struggled to recover the original CFG or gain important information from CFG. However, as the loop structure is retained in each function, the control flow still explicitly goes to each basic block and comes back to the dispatcher (`%LoopEntry` in Figure 4.4), which leaves a window for attackers to leak essential information of DNN programs. For example, attackers at least know all the original basic blocks contained inside a function. To conceal the CFG further, we aim to hide the loop structure by turning all the explicit flow into the implicit flow (i.e., indirect jump). The first thing we need to do is to remove the switch table. Thus, we encoded and embedded the original switch table in the global variable list. Besides, for each function, we create a table to store the addresses of basic blocks. Then, we create a decode function that uses the hashed dispatcher label `%hash` to get the address of the following basic block. In this way, the flows from the dispatcher (`%LoopEntry`) to each basic block are removed. For the last step, we remove the flows from basic blocks to the dispatcher (`%LoopEntry`) by inlining the dispatching part (mark as blue) into every flattened basic block. Figure 4.5 shows the part of the final result transformed from the original CFG from Figure 4.2. Apparently, attackers do not see a loop-like structure (or even part of it) in such a CFG because all basic blocks are floated in the DNN programs.

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

4.3.5 Optimization

Undoubtedly, after transforming the original CFG from Figure 4.2 to Figure 4.5, the time overhead increases due to the increasing of instructions, indirect jump, and function calls. In nested loops, even a single added instruction in the inner loop can execute thousands of times, causing significant performance overhead. In order to weaken the overhead introduced by the transformation, we propose two optimization methods: reduce the flattened functions and inline added function calls.

Table 4.2: A classification of the number of Basic Blocks in each operator function. **BB** refers to Basic Block.

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

4.3.5.1 Reduce the Transformed Functions

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

- We only consider transforming the function with the computation. For example, the DNN programs that TVM generates contain functions that check the input and output data layout constraint before starting the computation. These functions typically have more basic blocks than the actual computation function. Applying transformation to such a function increases not only the time overhead but also the scale of the DNN program.

- We want to ignore the functions with few blocks because the transformation would be trivial in this case. However, if we refer to the number of such blocks as N, determining the value of N is tricky since we do not want to exclude the vital operator functions. Thus, we collect information about the number of basic blocks in each operator function as shown in Table 4.2. We find that only activation functions like `ReLU`, element-wise arithmetic operators like `Add`, and operator function `BatchFlatten` have less than five basic blocks. Since these operators are unimportant, we ignore the functions with less than five basic blocks. Note that the DNN program with a high optimization level rarely contains such a function because operators can be fused into one operator function.

- Transforming a function with too many blocks (e.g., 100) is unnecessary, which makes the code generation step suffer too much, though people typically do not care about the compilation overhead.

4.3.5.2 Inline Function Call

As described in Section 4.3.3 and Section 4.3.4, our algorithm integrates two specialized function calls within the dispatch segment of the program. Recognizing that additional function calls may incur a notable runtime overhead is essential. This overhead primarily stems from the potential for extensive jumps between function calls, alongside the requisite establishment of stack frames, each of which demands considerable processing time and can impede overall system performance.

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

4.4 Evaluation

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

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

To explore the above RQs and provide a comprehensive evaluation, we evaluate FlatD with eight real-world pre-trained models and one self-trained model from four different frameworks and use the well-known obfuscator, O-LLVM [62] as the baseline. We only use the control flow flattening (-fla) obfuscation of O-LLVM to transform the

Table 4.3: Statistics of DNN models. ResNet18 is loaded from three different frameworks: PyTorch (**P**), ONNX (**O**), and MXNet (**M**). All other models are loaded from Keras Application Zoo.

Model		# of Parameters	# of Operators
MNIST [28]		34,826	12
VGG16 [119]		138,357,544	23
VGG19		143,667,240	26
Xception [27]		22,910,480	134
ResNet50 [50]		25,636,712	177
ResNet101		44,707,176	347
ResNet152		60,419,944	517
MobileNet [54]		4,253,864	91
ResNet18	**P**	11,689,512	51
	O	11,699,112	69
	M	11,699,112	171

program. All models are optimized and compiled by TVM [22] to generate DNN programs. FlatD and O-LLVM are both applied during the compilation and optimization. During the evaluation, all the models only used the data from ImageNet [33] to do the inference task.

4.4.1 Experimental Setup

We implement FlatD on the top of O-LLVM [62, 75] (version 8.0), primarily written in C++ with about 5K LOC. The current implementation obfuscates and evaluates DNN Programs in the ELF format on x86 platforms.

4.4.1.1 Deep Learning Compiler

For our evaluation, we adopt TVM [1], a state-of-the-art deep learning compiler, to compile DNN models into executable programs. TVM is chosen due to its flexibility, which allows seamless integration of our defense mechanism, FlatD, into the workflow during the code generation phase. For most of our evaluation, we use TVM v0.13.0 with the highest optimization level (O3) during the compilation. However, for the resilience evaluation, since the iteration of the TVM version is relatively fast, in order to align the attack environment, we chose TVM v0.9 to get the victim DNN programs with both the lowest optimization level (O0) and the highest optimization level (O3).

Table 4.4: Comparison of the inference results of the obfuscated DNN programs from O-LLVM and FlatD to the original DNN programs. Here **P** refers to PyTorch, **O** refers to ONNX, **M** refers to MXNet

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

4.4.1.2 Datasets

Table 4.3 shows all DNN models used for evaluation. All these models, except MNIST-convnet, are pre-trained models loaded from different frameworks. Among them, MNIST-convnet is a self-construct and self-trained model following the guide from [28], and ResNet18 is used to evaluate the effect of FlatD across the frameworks, so we loaded it from PyTorch [108], ONNX [10], and MXNet [21] respectively. The rest of the models are all loaded from Keras application zoo [26, 67].

4.4.1.3 Runtime Environment

We perform our evaluation in an Ubuntu 22.04 system on a machine that has Intel(R) Xeon(R) Silver 4114 CPU (2.20 GHz) with 40 cores and 219GB RAM. Note that our tool can only applied to ELF format on x86 platforms, so our experiments are all running on the CPUs rather than the GPU. Although the inference overhead is vital for DNN programs, the compilation overhead is not generally of concern. The total time cost for applying FlatD to DNN programs at code generation usually ranges from a few seconds to a few minutes, depending on the operator numbers of each model.

4.4.2 (RQ1) Correctness

To evaluate the impact of FlatD on preserving the inference accuracy of DNN programs, we compare the inference outcomes of the original DNN programs with the counterparts transformed from FlatD and O-LLVM. The primary metric for this comparison is to check

if the prediction results of the two models are identical. We assumed the original DNN program results were the ground truth and computed the identical percentage for the programs generated from FlatD and O-LLVM. To ensure a diverse and representative sample of test inputs, we sourced the test dataset from the ImageNet obtained from TorchVision [94]. We randomly select $10,000$ test inputs from this dataset as our evaluation set. Moreover, to illustrate the adaptability of FlatD, we evaluated ten versions. Within this set, three versions of ResNet18 were sourced from three distinct frameworks: PyTorch, ONNX, and MXNet. The remaining seven models were obtained from the Keras Application Zoo.

The summarized results are presented in Table 4.4. The findings from this table indicate that the inference results of the transformed DNN programs generated by FlatD align perfectly with those of the original programs across all the sampled inputs, which is under the expectation. However, we surprisingly found that after being obfuscated by O-LLVM, the MobileNet Program lost functionality. This outcome underscores the effectiveness of FlatD in preserving the original functionality and prediction accuracy of the DNN models.

4.4.3 (RQ2) Resilience

In this section, we evaluate the resilience of our defense framework. We first describe our evaluation setup (Section 4.4.3.1). Then, we show how FlatD influences the operator-type inference to the reversing-based extraction attacks (Section 4.4.3.2) by comparing the result between FlatD and O-LLVM.

4.4.3.1 Evaluation Setup

To align with the attack environment of prior reversing-based model extraction attacks [89, 146], we choose the TVM with released version v0.9.0 and use MNIST [34] and VGG16 [119] as two of our test models. We acquire MNIST by following the guide from [28] and VGG16 from Keras Application Zoo [67]. To test the effect of our defense mechanism on a more diverse set of DNN programs, we compile the two models above with two different optimizations (O0 and O3).

4.4.3.2 Operator Type Inference

As mentioned in Section 4.2, all the reversing-based model extraction attacks fully or partially include four parts: Operator-type recovery, Topology recovery, Parameter recovery,

Table 4.5: The accuracy change in DNN operator inference before and after applying FlatD and O-LLVM. "N/A" means the attack framework does not support the DNN programs with the settings.

Victim Model	Attack Framework	TVM -O0			TVM -O3		
		Orig	O-LLVM	FlatD	Orig	O-LLVM	FlatD
MNIST	BTD	100%	91.67%	50.00%	100%	92.31%	38.46%
	LibSteal	100%	58.34%	25.00%	N/A		
VGG16	BTD	100%	96.88 %	40.63%	100%	91.23%	64.91%
	LibSteal	100%	40.63%	9.38%	N/A		

and Dimensions (Attributes) Recovery. Since FlatD mainly focuses on manipulating the CFG of Operator functions, which is only related to the Operator Type recovery, we only evaluate our defense mechanism on how it can affect the inference of Operator Type of each reversing-based model extraction attack. Moreover, Operator-type recovery is the most essential and fundamental step in reconstructing the final models because, in some attacks [20, 138, 146], the recovery of other parts highly depends on the recovery of Operator-type.

We report the difference in the accuracy of DNN operator inference between the original version and the transformed version generated from O-LLVM and FlatD in Table 4.5. We apply the same metrics to compute the accuracy of each attack used, where the prediction of operator type is regarded as correct only when the predicted result describes precisely the same operation as the ground truth. Since LibSteal [146] cannot deal with the situation when multiply operators are fused into one operator function, we only evaluate the DNN programs compiled with configuration -O3 on BTD [89]. As we can see, compared to the O-LLVM, FlatD can effectively reduce accuracy in DNN operator inference for each attack framework. Notably, while BTD can still achieve over 90% accuracy decompiling the program transformed by O-LLVM, FlatD decreases the accuracy to around 60% and even lower. Specifically, for MNIST with TVM -O0, the accuracy of BTD reduces to 50.00%; for VGG16 with TVM -O0, the accuracy of BTD reduces to 40.63%; for the optimization level -O3, BTD only gets 38.46% accuracy when targeting the transformed MNIST program compared to 92.31% targeting the MNIST program obfuscated by O-LLVM. BTD can achieve 61.54% accuracy when targeting the VGG16 program transformed by FlatD. When facing the LibSteal Attack, although O-LLVM has already significantly reduced the accuracy, FlatD can still outperform it (58.34% compared to 25.00% for MNIST TVM -O0 and 40.63% compared to 9.38% for VGG TVM -O0).

Table 4.6: Comparison between the performance of the transformed DNN programs generated by FlatD and O-LLVM and the original DNN programs. We use the time overhead of the original program as the baseline (100%). This table reports the time overhead of each DNN program running the inference to one specific picture from ImageNet and compares it to the original version to indicate the increasing time overhead. Here, **P** refers to PyTorch, **O** refers to ONNX, and **M** refers to MXNet.

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

LibSteal and BTD rely heavily on the complete CFG information to infer the operator type. However, FlatD completely conceals the CFG by breaking the visible control flow between basic blocks. Even IDA Pro cannot extract the complete CFG without manual effort. On the other hand, although O-LLVM changes the control flow structure, the basic blocks are still visibly connected in the same chunk, which is still risky for the operator type to be inferred. We did not evaluate all the attacks mentioned in Section 4.2 due to the failure of setting up the attacks, which is discussed in Chapter 6.

4.4.4 (RQ3) Performance and Scale

Runtime performance and scale are critical to a DNN program, especially when deploying the model on devices with limited resources, like edge devices or low-power processors. Therefore, in this section, we compare the scale change between the original DNN programs and transformed versions (Section 4.4.4.2), as well as their performance of inference tasks (Section 4.4.4.1). To demonstrate the compatibility of FlatD, we evaluate ten versions of DNN models. Among them, we loaded three versions of ResNet18 from three different frameworks (PyTorch, ONNX, MXNet), and all seven other models are loaded from the Keras application Zoo. Compared to O-LLVM, the programs generated by FlatD have a lower scale while maintaining similar performance.

Table 4.7: Comparison between the scale of the transformed DNN programs generated by FlatD and O-LLVM and the original DNN programs. We use the original DNN program size as the baseline (100%). This table shows the increased percentage between transformed DNN programs and original DNN programs. Here, **P** refers to PyTorch, **O** refers to ONNX, and **M** refers to MXNet.

Diff(%)		FlatD	O-LLVM
VGG16		17.43	27.90
VGG19		17.48	27.57
Xecption		18.76	34.71
ResNet50		12.91	28.59
ResNet101		12.91	28.44
ResNet152		12.90	28.88
MobileNet		17.04	34.24
ResNet18	**P**	22.45	36.73
	O	21.93	35.53
	M	21.94	35.72

4.4.4.1 Inference Time Overhead

Since the time overhead is sensitive to the runtime environment and can fluctuate wildly due to unexpected reasons, we run all the DNN programs in an isolated environment to mitigate the influence of the runtime environment and reduce such fluctuation. Specifically, we randomly chose one picture from ImageNet and used it as the input for all the inference tasks. For each DNN program, we run the inference 100 times and record the mean as the evaluation result. As shown in Table 4.6, the results demonstrate that the increasing time overhead introduced by FlatD is similar to O-LLVM for most DNN models except the Xecption model.

4.4.4.2 Program Scale

Table 4.7 shows the scale change between the transformed DNN programs generated by FlatD and O-LLVM and the original DNN programs. Since the transformation process does not affect the parameter part of the DNN program, we only compare the scale change of the shared library files, which only contain the operator functions. To note, $\textbf{Diff} = (\frac{S_t}{S_o} - 1) * 100(\%)$ where S_t refers to the scale of a transformed DNN program and S_o refers to the scale of its original version. As we can see, the final percentages increased by FlatD to the DNN programs are much less than O-LLVM. While the size increased by FlatD can range less than 20%, the program generated from O-LLVM may increase over 30%.

Chapter 5
Securing Integrity of Deep Neural Network Program

In this chapter, we realize the potential risk of fault injection attacks to the Deep Neural Network Program and propose a prototype to secure the integrity of DNN Programs.

5.1 Introduction

Deep Neural Networks (DNNs) have revolutionized numerous fields by delivering state-of-the-art performance in tasks such as image recognition [50, 119], natural language processing [16], and complex decision-making systems [118]. These advantages speed up their widespread adoption across various platforms, ranging from large-scale cloud infrastructures to resource-constrained edge devices and embedded systems. The growing demand for low-latency, privacy-preserving inference has accelerated the deployment of DNN models directly on end-user devices.

To address the computational demands of deploying DNN models on such platforms, modern Deep Learning (DL) compilers like TVM [22], Glow [116], and XLA [76] have become essential components in the deployment workflow. These compilers translate high-level DNN models from different DL frameworks into optimized low-level executable programs tailored for efficient execution on specific hardware architectures. Although the appearance of DL compilers eases the deployment of the DNN model, it also introduces new risks from the perspective of the DNN program. For example, compared to the model extraction attacks based on side-channel information or prediction queries, the information leak caused by a reversing-based model extraction attack is more severe because the DNN program always carries the complete information to run as a standalone program or on a lightweight runtime stack, which even includes the essential model characteristics

like its architecture, trained parameters, and hyperparameters. Consequently, attackers are increasingly encouraged to reverse-engineer DNN programs to extract sensitive model information, exploit implementation vulnerabilities, or tamper with inference behavior.

In addition to model extraction attacks, it is unknown whether other threats to DNN models can become more severe due to the emergency of the DNN program. Nevertheless, a state-of-the-art fault injection attack toward DNN models reveals the potential risk to the integrity of DNN programs. A fault injection attack intentionally introduces faults into DNN models, leading to misclassifications, performance degradation, or unauthorized data leakage. The attack can be performed on different interfaces. While existing research predominantly focuses on model parameters or training data, FrameFlip [82], which exploits hardware vulnerabilities, particularly the Rowhammer bug [70], applies the attack by directly bit-flipping the code base of the shared computational library, such as the Basic Linear Algebra Subprograms (BLAS), which are fundamental components of widely used DL frameworks like PyTorch [108] and TensorFlow [1]. Even minor corruption can cause reductions in inference accuracy. FrameFlip represents a proof-of-concept attack highlighting the real-world feasibility of using fault injection to undermine AI infrastructure. Yet, its focus on shared libraries underestimates the potential impact of similar attacks when targeting standalone compiled DNN programs, which include the entire computational pipeline within a single executable. When targeting these compiled DNN programs, the attack surface of the adversary increases substantially. Unlike shared libraries that are typically protected and isolated by operating system mechanisms, standalone DNN programs often lack robust runtime defenses, making them even more vulnerable to fault injection attacks.

Despite the growing severity of fault injection threats, existing defenses for protecting software code integrity remain insufficient, particularly against hardware-based fault injection attacks like Rowhammer. Common solutions such as code signing, Control Flow Integrity [2], and Runtime Application Self-Protection (RASP) [106] primarily focus on enforcing software-level security guarantees. While these approaches are effective against traditional software tampering and control flow hijacking, they typically assume trusted hardware environments and lack mechanisms to detect or prevent bit flips caused by hardware faults. Consequently, they offer limited protection against advanced fault injection techniques that exploit physical vulnerabilities in memory subsystems.

To fill the gap, this chapter investigates three general challenges facing fault injection attacks and proposes a crash-based integrity protection framework to protect the integrity of the DNN program. The framework is designed to detect and respond to unauthorized

code modifications in real time. By integrating dynamic hashing techniques and indirect control flow enforcement into the compilation pipeline, the framework ensures that any tampering attempt results in an immediate system halt. This fail-stop behavior prevents attackers from leveraging tampered code for inference or further system compromise. The protection framework operates in two distinct phases:

- **Compilation Phase.** During compilation, the DNN program is instrumented with verification points, where hashes of critical code segments are calculated and embedded. These verification points are strategically placed to cover sensitive computation paths, ensuring comprehensive protection.

- **Post-Compilation Phase.** After compilation, runtime monitors verify these embedded hashes against recalculated values during execution. Any differences between the stored and computed hashes indicates a potential integrity violation, triggering the crash mechanism to halt execution immediately.

To evaluate the effectiveness of this framework, we conducted experiments on six widely used real-world DNN models(e.g., VGG-16 [119]). Using ResNet-50 [50] as a case study, the framework was able to identify and respond to FrameFlip-like attacks, ensuring that no degraded outputs were produced as a result of code manipulation. Our evaluation demonstrates that our framework successfully detects single-bit fault injections and prevents inference under compromised conditions.

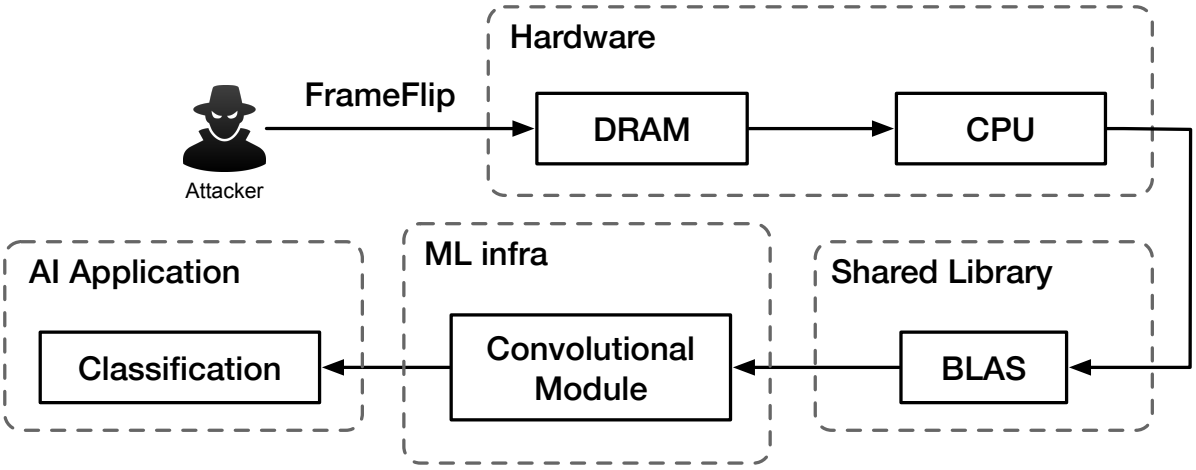
In summary, we make the following contributions.

- We identify the emerging threat of fault injection attacks targeting DNN programs and analyze their potential for disrupting inference reliability and system security.

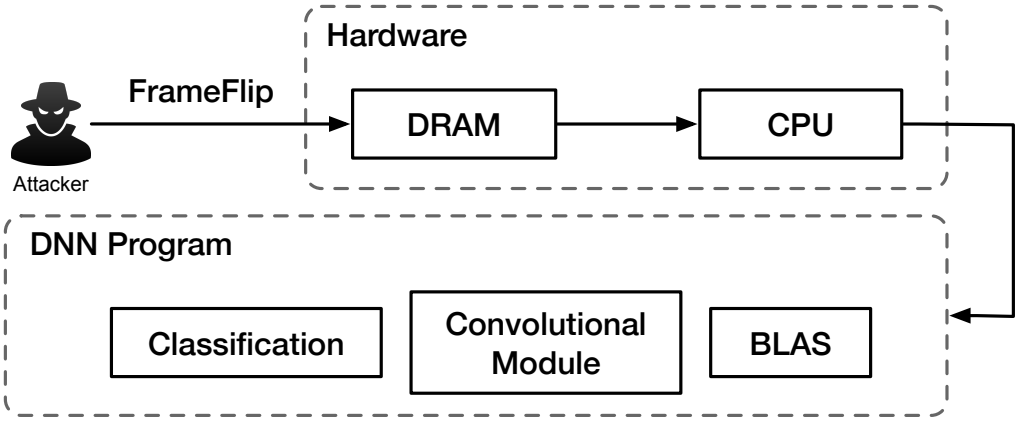
- We propose a crash-based integrity protection framework that enforces runtime code integrity verification, providing robust defense against unauthorized code modifications, including those induced by hardware-level fault injection attacks like Rowhammer.

- We implement and evaluate our framework across multiple DNN models, demonstrating its effectiveness in preserving inference correctness and mitigating runtime tampering, while maintaining acceptable runtime performance.

The rest of the paper is organized as follows. We first demonstrate our motivation for this framework in Section 5.2. Next, we discuss the general challenges and our solution in Section 5.3. Then we present the basic work flow of our framework in Section 5.4. Section 5.5 presents the evaluation and case study.



(a) Original FrameFlip Attack Overflow



(b) Attack towards DNN Program

Figure 5.1: Comparison between the attack towards DNN models and DNN Programs. As shown in figure 5.1a, the original frameflip uses DARM Rowhammer to affect the bit in the shared library, Basic Linear Algebra Subprograms (BLAS), and further affects computation results of the Convolutional module and the final result of the AI classification Application. When the attacker uses the same strategy targeting DNN programs (figure 5.1b), all DNN model-related computation, including shared library (BLAS), Convolutional module, and final classification, are integrated into a standalone DNN program. Therefore, the attacker can target any vulnerable bit inside the DNN program to affect the final result.

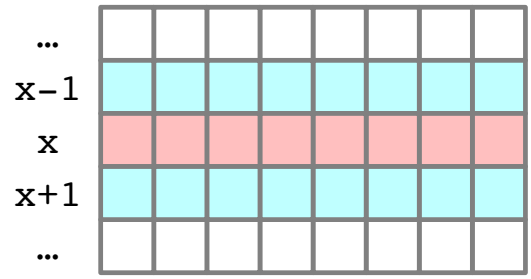
5.2 Motivation

5.2.1 Potential Risk

With the widespread adoption of Deep Neural Network (DNN) models, they have faced numerous security and privacy risks [113, 121, 132]. These risks stem from various attack

```
L1:
   mov r10, (x+1)
   mov r10, (x-1)
   clflush (x+1)
   clflush (x-1)
   jmp L1
```

(a) Code snippet



(b) Target Physical Address

Figure 5.2: Demonstration of Double-sided Rowhammer attack exploiting DRAM vulnerabilities. As shown in 5.2a, the code snippet repeatedly accesses and flushes adjacent memory rows $x-1, x+1$ via `mov` and `clflush` instructions in a loop (L1), bypassing the cache to stress DRAM cells directly. Physical address layout (5.2b) highlighting the victim row x flanked by aggressor rows $x-1, x+1$. Rapid access to these adjacent rows induces bit flips in the victim row, enabling unauthorized memory manipulation.

vectors that threaten model integrity, confidentiality, and robustness. On the other hand, the emergence of the Deep Learning Compiler, which compiles the model into DNN programs, introduces a new attack vector. For example, the critical attack vector, model extraction attacks [138, 146], has gained considerable attention. These attacks aim to steal essential DNN model information, including network architecture, parameters, and hyperparameters, by reversing DNN Programs, allowing adversaries to replicate or exploit valuable models developed by leading AI providers.

Beyond model extraction attacks, fault injection attacks [52, 82, 112] present another serious risk for the DNN program that demands attention. Fault injection attacks involve deliberately introducing faults into a system to induce incorrect behavior, compromise security, or expose latent vulnerabilities. By manipulating execution flow, adversaries can force misclassifications, degrade performance, or bypass security mechanisms to leak sensitive information or manipulate model behavior. Among various fault injection strategies, the recently proposed FrameFlip attack by Li et al. [82] stands out due to its novel and highly effective approach. As shown in Figure 5.1a, unlike traditional fault injection methods, FrameFlip does not target individual neural network weights or activations but instead exploits shared libraries, such as the Basic Linear Algebra Subprograms (BLAS), which are foundational components of widely used machine learning frameworks like PyTorch [108] and TensorFlow [1]. By flipping a single bit in these shared libraries, FrameFlip affects the computation of the Convolutional module in the machine learning infrastructure and further disrupts the DNN inference control flow in AI applications, causing substantial performance degradation. The attack leverages DRAM

Rowhammer [70] to perform end-to-end fault injections, making it effective across diverse model architectures (LeNet [79], VGG-16 [119], ResNet-50 [50]) and datasets (CIFAR-10, ImageNet [33]). The findings demonstrate that even a single bit flip can degrade inference accuracy, reducing performance to levels comparable to random guessing. Inspired by the FrameFlip attack, we recognize the critical need to protect the integrity of DNN programs. Figure 5.1 compares the original FrameFlip attack and the potential attack workflow pipeline if it targets the DNN programs. As we can see in Figure 5.1b, an attack on DNN programs expands the threat landscape to the entire DNN model computation pipeline because they are all integrated into a standalone executable, including involved shared library (like BLAS), Machine Learning infrastructure (Convolutional Module), and final classification. Furthermore, targeting the vulnerable bits in the DNN program containing the whole computation pipeline is far more convenient than just exploring the shared library codebase. In order to emphasize the severity of the FrameFlip attack, we first dive into the critical technology it uses, Rowhammer.

5.2.1.1 Rowhammer

Dynamic Random-Access Memory (DRAM) is the backbone of modern computing, providing high-density, high-speed, volatile storage. On the architectural level, DRAM is organized into several channels. Each channel sits between the memory controller and the physical DRAM modules, which are known as Dual Inline Memory Modules (DIMMs). Data is organized hierarchically in these modules into ranks, banks, rows (i.e., word-lines), and columns (i.e., bit-lines). Each memory cell stores data as an electric charge in capacitors, which are controlled and accessed by transistors through activating targeted word-lines.

The Rowhammer bug is a fatal hardware vulnerability in many DRAM devices [70], typically affecting those with high-density production technologies. It was first recognized through experimental studies showing that if specific DRAM rows are rapidly and repetitively turned on again, then capable electromagnetic interference or capacitive coupling is induced across adjacent memory rows. Double-sided hammering is a typical exploit to trigger the Rowhammer bug, which interferes with data accuracy in adjacent rows, causing random bit flips. Such an offset compromises the data integrity of neighboring rows by casually turning all 0s into 1s instead. Figure 5.2 demonstrates the code snippet and the target physical address of double-sided Rowhammer. As shown in Figure 5.2b, the aggressor rows with row index $x + 1$ and $x - 1$ lie on either side of a target victim row x. If an attacker rapidly accesses these two aggressor rows via `mov` and `clfush`

instructions, the electromagnetic interference will eventually corrupt data stored inside the victim row x. When this happens, a memory cell in row x storing a 1 signal can suddenly convert into a 0 signal or vice versa.

Although hardware manufacturers have introduced mitigation strategies, such as Target Row Refresh (TRR) [98], in subsequent generations of DRAM (e.g., DDR4), research has shown that these defenses are not absolute. For example, TRRespass [43] proposes an attack that bypasses TRR mechanisms by utilizing advanced Rowhammer solutions that aim simultaneously at multiple rows, called many-sided Rowhammer. Thus, despite TRR and other defensive measures, the Rowhammer bug is still a daunting and significant issue for modern commodity DRAM.

5.2.2 Existing Code Integrity Mechanism

Ensuring the integrity of software code during its lifecycle, from development to deployment and execution, is a foundational basement of software security. Code integrity protection mechanisms aim to guarantee that software remains untampered, authentic, and behaves as intended, even in adversarial environments. Over the years, several techniques and frameworks have been developed to preserve code integrity, each addressing different stages and components of software execution. However, none can prevent DNN programs from the risk of bit-flip attacks using Rowhammer. Note that in this section, we only discuss the software-level defense mechanism. Although hardware-level mitigation policies perform better against the Rowhammer, they have limitations when applied to DNN programs. For instance, Error-Correcting Code (ECC) memory can detect and correct single-bit errors, which mitigates some Rowhammer effects. However, ECC is not universally available, especially in edge devices where DNN programs are often deployed. Therefore, software-based defenses are necessary.

5.2.2.1 Static Protection

Code Signing and Digital Signatures. Code signing is a cryptographic technique designed to ensure the authenticity and integrity of software prior to its execution. Developers generate digital signatures by hashing the code and encrypting this hash with their private key. At runtime or load time, the signature is verified using the corresponding public key to confirm that the code has not been altered. However, this method predominantly verifies integrity only before execution, leaving the running code potentially vulnerable.

Integrity Measurement Architecture (IMA) and Trusted Platform Module (TPM). IMA is a Linux kernel subsystem that calculates hashes of critical executable files before execution and stores them in tamper-resistant logs for attestation purposes. TPM, combined with Secure Boot and Measured Boot mechanisms, provides hardware-based verification of software components during the boot process, ensuring that the system starts in a known good state. These mechanisms focus on initial system trust but do not continuously monitor runtime integrity.

5.2.2.2 Dynamic Protection

Control Flow Integrity (CFI). CFI is a critical code integrity protection mechanism designed to prevent control flow hijacking attacks. First introduced by Abadi et al. [2], CFI has become an essential defense against Return-Oriented Programming (ROP), Jump-Oriented Programming (JOP), and other sophisticated code-reuse attacks that leverage vulnerabilities such as buffer overflows and memory corruption. Zhang and Sekar [148] and Mohan et al. [96] further improve the efficiency by reducing performance impact. Control Flow Integrity ensures that a program's execution adheres to its intended control flow graph (CFG), preventing adversaries from redirecting execution to unintended locations. It enforces two primary properties: Forward-Edge ensures that function calls and indirect jumps follow valid transitions within the CFG, and Backward-Edge protects return addresses to prevent stack-based control flow attacks. Static CFI, like LLVM-CFI [100] and Microsoft Control Flow Guard [123], enforces control flow correctness at compile-time by instrumenting code with valid control flow constraints. Dynamic CFI enforces integrity during execution by monitoring control transfers and blocking unauthorized branches. For example, Shadow Stack Techniques [18] maintains a separate secure stack to verify return addresses at runtime.

Runtime Application Self-Protection (RASP). RASP technologies embed security checks within an application's runtime environment to detect and mitigate attacks as they happen. RASP monitors application inputs, API calls, and behavior to block common application-layer threats, such as SQL injection, cross-site scripting (XSS), and command injection. However, RASP primarily operates at the software abstraction level, assuming that the hardware and memory layers are trustworthy.

5.2.2.3 Limitation

Unfortunately, Rowhammer is capable of bypassing these code integrity mechanisms. The primary limitation of existing code integrity mechanisms lies in their foundational assumption: the reliability of the underlying hardware. Traditional software integrity protections are designed to prevent software-based tampering and unauthorized modifications. Still, they are not equipped to detect or deter bit-level manipulations originating from Rowhammer. Specifically, they have the following disadvantages against Rowhammer:

- **Threat Model Mismatch.** Code signing and IMA verify software integrity before execution, but they do not monitor runtime changes. Once the code is loaded into memory, it becomes susceptible to Rowhammer-induced modifications.
- **Post-Execution Blind Spots.** TPM-backed Secure Boot and Measured Boot confirm system integrity during startup. Still, they do not protect against dynamic memory corruption during runtime, leaving systems exposed to Rowhammer after the initial attestation phase.
- **Lack of Hardware Awareness.** Control Flow Integrity (CFI) and RASP focus on enforcing legitimate control flow and runtime behavior. However, they typically do not include low-level memory integrity verification. Bit flips induced by Rowhammer that subtly alter data or instructions without violating expected control flow or triggering behavioral anomalies go unnoticed.

Therefore, DNN programs still face the threat of bit-flip attacks like FrameFlip. It is critical to devise a solution to mitigate this threat.

5.3 Problem Statement

5.3.1 General Challenges

Designing a compiler-integrated defense framework to safeguard the code integrity of DNN programs against fault injection attacks, especially bit-flip attacks using Rowhammer, like FrameFlip, is challenging. There are three crucial general challenges (GCs):

- **GC1: Detection bit-flip Tampering at Runtime.** Traditional integrity checks fail to address runtime modifications because they apply the check at load or boot time or after the control flow graph is not hijacked. A robust framework

must detect any bit-flips in critical code regions (e.g., Convolutional computation module) in real time.

- **GC2: Preventing Forgery of Integrity Verification.** Traditional integrity checks (e.g., static hash functions) are vulnerable to bypass or reverse engineering, allowing attackers to recompute valid hashes after tampering. A defense must ensure verification mechanisms cannot be easily forged even if code sections are modified.

- **GC3: Halting Exploitation of Compromised Execution.** Many defenses allow execution to continue after detecting tampering, enabling attackers to exploit transient or partial control-flow hijacking. A robust framework must prevent attackers from leveraging even partially corrupted states.

5.3.2 Our Solution

To mitigate this threat, we propose a prototype to secure the integrity of the DNN program against unauthorized modifications. Our approach introduces a crash-based security mechanism, which ensures that any attempt to manipulate the codebase or execution flow triggers an immediate application failure. By enforcing strict integrity verification, we negate fault injection attempts before they can cause damage, thus preserving the reliability and security of deployed DNN programs. Our work introduces a more robust defense model that addresses some of the shortcomings of traditional code integrity solutions by solving the general challenges. Specifically, when facing hardware-based attacks like Rowhammer, our framework is more robust than traditional code integrity solutions.

- **Runtime Code Integrity Verification.** To address **GC1**, our framework enforces continuous runtime verification of control flow integrity. By using customized hash functions and indirect control flow mechanisms, it ensures that any unauthorized modification to the critical codebase of DNN programs or control flows is detected during execution.

- **Customized Hash Function for Code Integrity.** To address **GC2**, the system employs a customized hash function to generate verification values based on the program's code base. This customization adds an additional layer of security by making it difficult for an attacker to generate valid hash values after flipping

bits in the code. Without knowledge of the hash function, an attacker leveraging Rowhammer cannot recompute correct hashes, and any modification will trigger the framework's detection mechanisms.

- **Immediate Crash-Based Reaction.** To address **GC3**, upon detecting unauthorized changes, the mechanism initiates a crash-based reaction, halting execution to prevent further exploitation. This immediate response limits the attacker's ability to leverage Rowhammer-induced modifications to perform privilege escalation or arbitrary code execution.

For the solution of **GC2**, even if the attackers can get the customized hash function by reverse engineering and recompute correct hashes for the target modification, the attack is still extremely hard or even impossible to be successful. We assume the attacker targets the vulnerable bit in row X, and the original hash value stores at row Y. The attacker needs to flip bits in both rows X and Y simultaneously to bypass the defense mechanism while applying the desired attack effect. Otherwise, the mismatch of hash values will trigger the crash reaction and halt the execution to prevent further exploitation. Therefore, our framework offers greater robustness against Rowhammer because it verifies runtime integrity rather than relying on pre-execution checks, provides a customized hash function to complicate forgery attempts, and reacts to tampering proactively by terminating execution rather than attempting to continue operating under compromised conditions.

5.4 Design

5.4.1 Overview

Figure 5.3 presents the workflow of our integrity protection methodology. The framework is composed of two primary phases: the Code Generation (CodeGen) stage and the Post-Compilation stage. During the CodeGen phase, which is implemented using the LLVM compiler infrastructure [75], a relocatable Executable and Linkable Format (ELF) binary is produced. This intermediate binary includes strategically inserted placeholders, which are designed to accommodate later critical values to ensure code integrity during execution. Furthermore, this phase integrates the computed hash values with indirect branching instructions to protect the control flow integrity of the final DNN program.

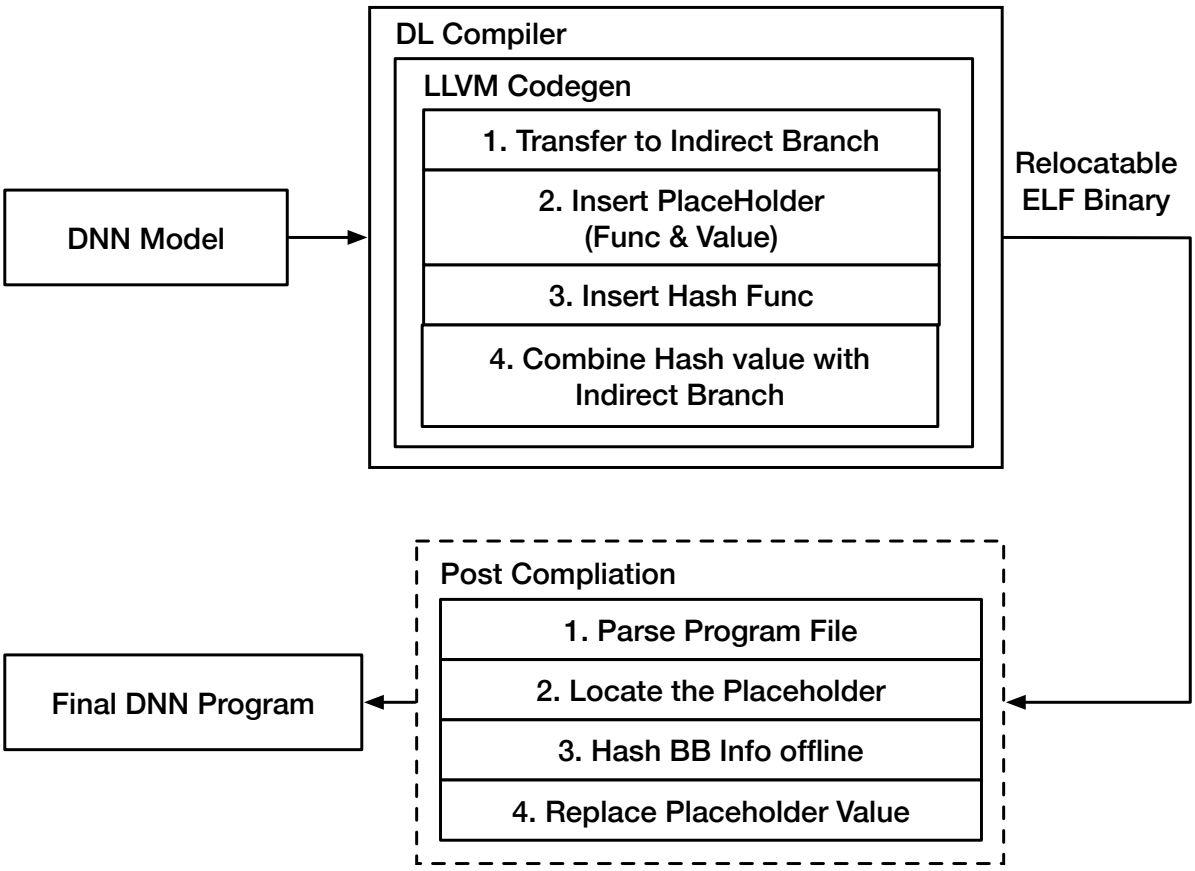


Figure 5.3: Workflow Pipeline Overview. The process begins with a DNN model, which is compiled into a final program using a DL compiler. During LLVM-based code generation, integrity checks are injected: (1) the Original Control Flow Graph is transferred to indirect branch instructions, (2) placeholder values and functions are inserted, (3) cryptographic hash functions are integrated. (4) The hash results are combined with indirect branch logic. At the post-compilation phase, the relocatable ELF binary is processed to locate placeholders, compute the hash value of Basic Block metadata offline, and replace placeholder values.

At the Post-Compilation stage, we leverage a Python-based parsing and analysis tool to process the relocatable ELF binary. The tool systematically identifies the placeholders embedded during the earlier CodeGen stage. After locating these placeholders, it computes the appropriate hash values based on pre-analyzed basic block information, and substitutes the placeholders with these finalized values.

```
define weak i8* @PLACEHOLDER_FUNC() noinline optnone readonly {
  %retaddr = call i8* @llvm.returnaddress(i32 0)
  ret i8* %retaddr
}
```

Figure 5.4: LLVM IR placeholder function designed to capture the return address of its caller. Marked as `weak`, `noinline`, `optnone`, and `readonly`, this function is ensured not to be optimized during compiler optimizations and other code generation processes. It uses `@llvm.returnaddress` to retrieve the runtime return address and returns it.

```
%begin_addr = call i8* @PLACEHOLDER_FUNC()
 <Original Basic Block>
%end_addr = call i8* @PLACEHOLDER_FUNC()
%check_sum = call i32 @HASH_FUNC(i8* %begin_addr, i8* %end_addr)
%masked_value = xor i32 %next_bb_address, %check_sum
%next_bb_address = xor i32 % masked_value, <PLACEHOLDER_VAL>
```

Figure 5.5: Result After CodeGen. Placeholder functions (`@PLACEHOLDER_FUNC`) capture the start and end addresses of a basic block. A hash function (`@HASH_FUNC`) computes a checksum over the block's address range. The checksum is combined with the next basic block's address via the `XOR` operation, which ensures runtime tamper detection by validating code integrity during execution. The placeholder value (`<PLACEHOLDER_VAL>`) is replaced post-compilation to finalize the integrity checks.

5.4.2 CodeGen

The initial operation within the Code Generation (CodeGen) phase is converting all direct branch instructions present in the computational segments of the DNN program into indirect branches. This transformation enhances flexibility by allowing dynamic modifications to target addresses during runtime. To assist in identifying the boundaries of basic blocks during the Post-Compilation phase, the framework introduces placeholder functions at both the entry and exit points of each basic block of critical computation component, as demonstrated in Figure 5.4. These placeholder functions fulfill several critical roles and adhere to specific design requirements:

- **Boundary Delimitation.** They explicitly mark the beginning and end of each basic block, enabling precise detection of block boundaries when performing integrity verification after compilation.

- **Runtime Address Retrieval.** The placeholder function calls the LLVM intrinsic `llvm.returnaddress`, allowing it to capture the runtime return address. This

```
Relocation section '.rela.text.<section_name>' at offset 0x231a0 contains 30 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000024d  002400000004 R_X86_64_PLT32    0000000000008960 PLACEHOLDER_FUNC - 4


Symbol table '.symtab' contains 54 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
    36: 0000000000008960     5 FUNC    WEAK   DEFAULT    2 PLACEHOLDER_FUNC
```

Figure 5.6: Partial results from readelf showing relocation and symbol table entries for the `PLACEHOLDER_FUNC` in an ELF binary. The relocation entry (type `R_X86_64_PLT32`) references the weak symbol `PLACEHOLDER_FUNC` at offset `0x24d`. The symbol table confirms `PLACEHOLDER_FUNC` as a weakly bound function with fixed address `0x8960`. These entries enable post-compilation patching, where placeholder offsets and values are replaced with runtime-derived hashes for control-flow integrity verification.

address information is essential for the computation of hash values used in verifying the integrity of control flow during execution.

- **Compiler Directive Compliance.** To ensure the placeholder function remains intact and is not subject to alteration during compiler optimizations, specific attributes are applied. These include `noinline`, `optnone`, and `readonly`, which collectively prevent inlining, disable optimization, and declare that the function has no side effects, respectively.

Following these modifications, the CodeGen phase produces an intermediate representation of the program, where each basic block is bracketed by calls to the placeholder function. As illustrated in Figure 5.5, the runtime system retrieves both the starting and ending addresses of a basic block using these functions. These addresses are then passed to a hash function to compute a checksum that validates the block's integrity. At this stage of compilation, the actual binary layout of the program is not finalized. Therefore, a temporary placeholder value is inserted in place of the actual checksum. This placeholder will later be replaced with the valid hash corresponding to the finalized basic block during the Post-Compilation phase. Finally, two sequential XOR operations are integrated to enforce control flow integrity. The first XOR combines the computed checksum with the address of the next basic block, while the second XOR reverses this masking process using the placeholder (which will later be replaced by the correct hash). If an attacker tampers with the basic block, the checksum verification will fail, resulting in an unexpected indirect branch address. This deviation causes the program to terminate abnormally, thereby preventing unauthorized code modifications and preserving the integrity of the execution flow.

5.4.3 Post Compliation

The Post Compilation phase ensures that the final DNN program contains the correct security-enforcing mechanisms by replacing placeholders with computed hash values. This step requires precise identification of placeholder positions, both function and value, within the relocatable ELF binary.

To confirm the placeholder function's position, we first use readelf to analyze the symbol table and relocation section and identify placeholder call sites. As shown in Figure 5.6, the relocation section of the ELF binary contains information about where PLACEHOLDER_FUNC is referenced. Specifically, at offset 0x24d, the instruction calls PLACEHOLDER_FUNC with relocation type R_X86_64_PLT32, and the corresponding function symbol is located at address 0x8960, which is consistent with the symbol table entry. Upon these findings, we utilize pyelftools, a Python library designed to parse relocatable ELF binary and locate each placeholder function call site.

Once the placeholder function locations are identified, the next step is to search for the placeholder value. Since we use a predefined, unique placeholder value (0xdeadbeef in our case) within the binary, locating offsets that store these values is straightforward. Then, we combine two position lists to get an offset list that includes basic block start address, basic block end address, and instruction address containing the placeholder value sequentially. Since it is impossible for the DNN program to contain function symbols or values matching our predefined placeholders, the offset list follows a strict order. Finally, we compute the hash values of each basic block offline using the previously extracted boundaries and replace the placeholder value.

5.5 Evaluation

This section evaluates our prototype using the following research questions (RQs). We mainly focus on the functionality of our prototype from two sides.

- ***RQ1: (Functionality of DNN Programs)*** *After applying our prototype to the DNN programs, can they still apply the inference functionality properly?*
- ***RQ2: (Functionality of Prototype)*** *Does our prototype effectively secure the DNN programs integrity?*

To explore the RQ1, we evaluate our prototype with six real-world pre-trained models and use the data from ImageNet [33]. All models are optimized and compiled by TVM [22]

Table 5.1: Statistics of DNN models. All models, except MNIST, are loaded from the Keras Application Zoo.

Model	# of Parameters	# of Operators
MNIST	34,826	12
VGG16	138,357,544	23
VGG19	143,667,24	26
Xception	22,910,480	134
ResNet50	25,636,712	177
MobileNet	4,253,864	91

to generate DNN programs. As for the RQ2, we target the DNN program compiled from ResNet-50 as the case study to discuss the effectiveness of our methodology. In the evaluation, we use TVM [22] as the DL compiler to generate DNN programs from DNN models with the highest optimization level (O3) during the compilation.

5.5.1 (RQ1) Functionality of DNN Program

5.5.1.1 Datasets

Table 5.1 shows all DNN models used for functionality evaluation. All models, except MNIST-convnet, are pre-trained models loaded from the Keras application zoo [26,67,68]. MNIST-convnet is a self-construct and self-trained model following the guide from [28]. All the experiments are run on a server with an Ubuntu 22.04 system, Intel(R) Xeon(R) Silver 4114 CPU (2.20 GHz) with 40 cores and 219GB RAM. Note that our prototype can only be applied to the ELF format on x86 platforms, so our experiments run on the CPU rather than the GPU.

5.5.1.2 Results

To assess how our prototype impacts the inference accuracy of DNN programs, we compare their prediction outcomes before and after applying our approach. The key evaluation metric involves determining whether both versions produce identical predictions. Given a test set of N inputs, we capture the outputs from the original and transformed programs and quantify their consistency. Specifically, we compute the percentage of test inputs for which both models yield the exact predictions, providing a direct measure of how the transformation process influences the predictive behavior of the DNN programs. Our assessment is conducted on six different DNN models. We draw samples from the ImageNet dataset sourced through TorchVision [94]. A total of 100 test inputs are

Table 5.2: Comparison between inference results of the DNN programs with and without applying our methodology. This table shows that the predicted labels do not change after transformation

MNIST	VGG16	VGG19	Xecption	ResNet50	MobileNet
100%	100%	100%	100%	100%	100%

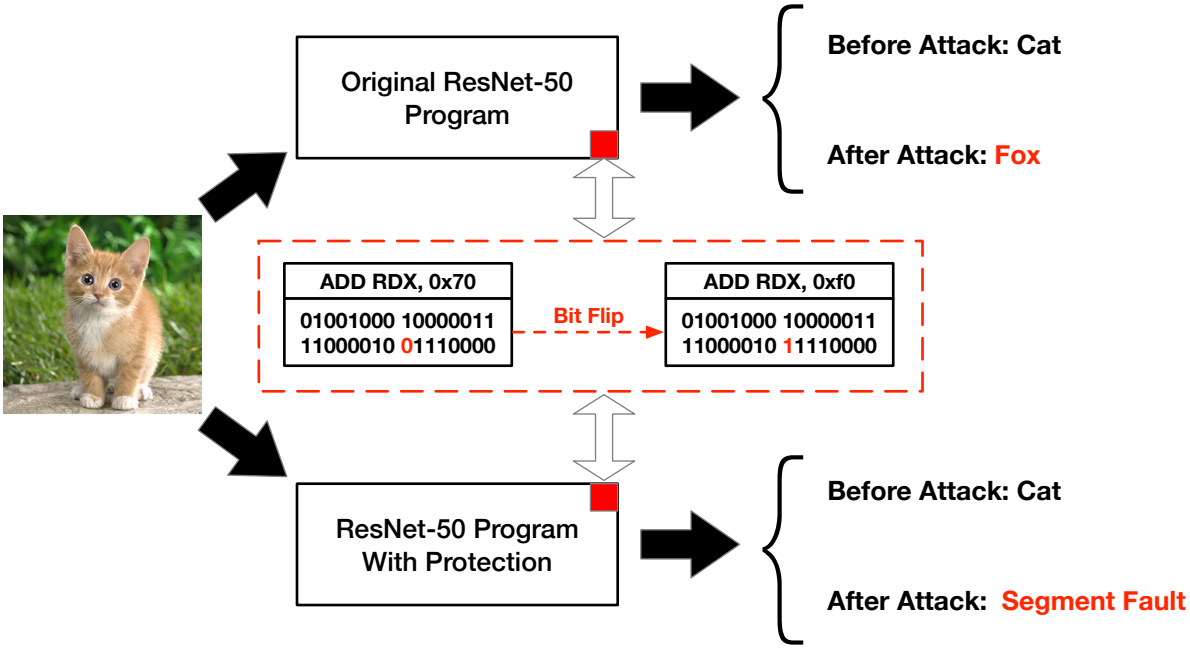


Figure 5.7: Case Study Result Overview on FrameFlip-like bit flip attack against ResNet-50. We manually identify a vulnerable bit ((highlighted in red)) in the original program's assembly instruction (`ADD RDX, 0x70`) and alter it to `ADD RDX, 0xF0` via Rawhammer, causing misclassification from "Cat" to "Fox". On the other hand, the protected DNN program triggers a segmentation fault upon the same attack, demonstrating runtime integrity enforcement. The protection framework detects tampering and halts execution to prevent erroneous outputs.

randomly selected, allowing our evaluation to cover a broad range of scenarios and input variations.

Table 5.2 summarizes the results. The findings show that the transformed DNN programs consistently produce identical inference results compared to their original versions across all sampled inputs. This outcome highlights the effectiveness of our prototype in preserving both the integrity and predictive accuracy of the DNN programs.

5.5.2 (RQ2) Functionality of Prototype

5.5.2.1 Assumption

Victime Capability. We use ResNet-50 [50] as a case study to demonstrate the impact of potential bit-flip attacks on a DNN program and evaluate how our prototype enhances integrity protection by triggering an alarm upon detection of unauthorized modifications. We use a pre-trained ResNet-50 from the Keras Application Zoo [67, 68] and compile it into a deployable DNN program using TVM. To align with the prior bit-flip attacks [52, 82, 112, 144], we assume that the decompiled ResNet-50 program is deployed and run on a resource-sharing platform.

Attack Capability. The attacker aims to undermine the functional reliability of the DNN inference program by introducing a single-bit fault into the DNN program runtime codebase. The injected fault is designed to disrupt the standard control flow of the DNN program, thereby resulting in a measurable decline in the predictive accuracy and overall performance of the DNN program. Meanwhile, the attacker strives to preserve a high level of stealth, ensuring that the fault remains undetected for an extended duration. We assume that the attacker process resides on the same platform as the target DNN service, allowing both to share computational resources. In particular, the adversary is an unprivileged user and yet still accesses the same physical memory space as the victim process. This assumption is widely recognized and has been adopted in previous research [48, 52, 82, 144]. Also, we assume that the attacker can obtain the DRAM address mapping scheme.

5.5.2.2 Case Study

To simulate a bit-flip attack similar to FrameFlip [82], we manually identify a vulnerable bit in the ResNet-50 Program and flip it using RawHammer. The impact of this bit-flip attack is visualized in Figure 5.7. Specifically, when the original (unprotected) ResNet-50 program is subjected to the attack, a test image of a "cat" is misclassified as a "fox", highlighting the vulnerability of the DNN program to even minimal modifications. Furthermore, we randomly selected 100 test images from ImageNet and measured the accuracy after the attack. Without protection, the accuracy drops from 100% to 70%, demonstrating a substantial degradation in performance due to the attack. Conversely, when the protected ResNet-50 program is tested under the same attack conditions, it maintains its original classification accuracy before the attack. As expected, the model correctly identifies the image as a "cat", consistent with our evaluation result in Section

5.5.1. However, when a bit-flip attack is attempted on the protected model, the system detects the integrity violation and immediately terminates execution via a segmentation fault, preventing any further inference.

This case study demonstrates the robustness of our methodology in securing DNN program integrity against fault injection attacks. By enforcing a fail-safe mechanism, our approach detects unauthorized modifications and halts execution to prevent compromised models from producing incorrect outputs.

Chapter 6
Discussion

In this chapter, we discuss several limitations of our presented works and provide potential solutions for them.

6.1 LibSteal

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

The emergence of the DL compilers has attracted attackers' attention to the binary attack interface [20, 89, 138]. However, a few software defense strategies can still be used to mitigate this attack interface. The most common one is obfuscation, which transforms the program into another format with the same functionality and semantics. However, only some obfuscation techniques are suitable for this scenario because the DNN executable still requires efficiency. For example, although code virtualization performs outstandingly against reverse engineering, it will significantly increase the execution time overhead, which is unacceptable for the DL inference binary.

Another approach is white-box cryptography (WBC) [15]. Existing attacks can all be considered white-box attacks since they assume complete control over the DNN executable. WBC is an essential technology against the white-box binary attack. WBC

allows transforming the given binary into a robust representation so that it can hinder the reverse engineering and analysis of the binary. However, this technology also increases the execution time overhead.

6.2 FlatD

Protect other characteristics. The primary purpose of FlatD is to protect the CFG of the operator function so that attackers cannot infer the operator types accordingly. However, we do not provide protection to other characteristics of the DNN model, such as graph topology, operator attributes, and parameters, which should be protected from different views. For example, the data flow of DNN programs is also an essential feature attackers use to extract DNN model information, like graph topology. Attackers [89, 138] utilize the data dependency between operator functions to determine the graph topology structure because the input data of the successor operator function and the output data of the predecessor operator function share the same memory address.

DNN program integrity and dynamic analysis. In section 4.3.3, we introduce the secret information, *salt*, and one-way cryptographic hashing to secure the dispatcher label. Actually, we also consider the probability of such a strategy to secure the code integrity (i.e., tamper-proof). For example, we can leverage the value of *salt*, which is randomly chosen for each function. Theoretically, we can make the *salt* related to the code of each basic block, like the hash value of the code chunk. The value of the *salt* can be calculated after executing each basic block. Thus, If the code is compromised, the value of *salt* will not match the original one, and the execution sequence of the basic block will not follow the original control flow. Most likely, the code crashes. A tamper-proof program can prevent malicious instruments from collecting valuable information.

Supporting Other DL Compilers and Platforms. In Chapter 4, though we only discussed and evaluated the availability of FlatD on TVM and x86-64 platforms, our defense framework can also provide support to other DL compilers and platforms thanks to the compatibility of LLVM [75]. According to a complete survey of DL compilers [81], the low-level IR used by the majority of DL compilers can ultimately be converted into LLVM IR. To better support more compilers and platforms, we need to consider the optimization work after the code transformation.

Other attacks. Although we failed to evaluate NNreverse [20], and DnD [138], the defense effect of FlatD towards these two attacks should be more significant than the result from BTD because they both highly rely on accurate CFG. The advantage of

NNReverse compared to other binary mapping tools [36] is that it combines syntax and topology structure representation. However, after the transformation, the loop structure of CFG will be hidden, and all the CFG structures will look similar. In other words, the proposed advantage is cut off. DnD implemented its framework on the top of anger [128] and utilized symbolic execution to recover all the essential information for reconstructing the model. Nevertheless, after FLATD turns all branch instructions into indirect jumps, each operator function is transformed into a state-machine-like format, which is fatal to symbolic execution-based tools. Overall, FLATD can also effectively hinder these attacks [95].

6.3 DNN Program Integrity

Data Flow Integrity. Apart from leveraging control flow to enforce the integrity of the DNN program, ensuring data flow integrity represents another crucial defense strategy against fault injection attacks because, all in all, the ultimate target of such attacks is often the computational pipeline of the DNN model (i.e., the data flow of DNN program), which includes critical weights [52] and vulnerable bits in the branch condition of shared library [82]. Moreover, it is not necessary to enforce strict integrity checks across the entire data flow of the DNN program because a DNN model inherently possesses some tolerance to minor variations in weight and can often continue functioning correctly even if less critical components of the computational pipeline, such as activation functions, are compromised or malfunctioned. Given this resilience, focusing on monitoring and protecting the most critical data portion is more practical and efficient. By selectively supervising these key data portions, it is possible to preserve the performance and accuracy of the DNN program while minimizing the additional time overhead.

Hardware-assisted Defense. While traditional code integrity mechanisms provide critical protections against software-based tampering, they are generally insufficient to defend against hardware-level attacks like Rowhammer. Our third work enhances robustness by incorporating runtime integrity verification and immediate reaction mechanisms. However, it is important to acknowledge that our framework is still at the software level and cannot replace the need for hardware-level defense. As we discussed in Section 5.3.2, if Rowhammer flips bits in memory holding the hash values used for integrity verification and vulnerable bits in the computation pipeline at the same time, which is extremely hard to achieve, the framework can be bypassed. Therefore, although the target hardware platform of DNN programs does not universally have the condition to

shield with hardware-assisted defense, it is reasonable to raise a discussion. A proper robustness defense framework against hardware-level attacks like Rowhammer requires co-design with hardware-level protections such as Trusted Execution Environments (TEEs) and memory isolation strategies. When combined with hardware-based protections, our framework can strengthen defenses against Rowhammer and similar fault injection attacks. Hardware protections prevent Rowhammer, and software protections ensure tampering is noticed if hardware fails.

Chapter 7
Conclusion

In this thesis, we present three of our works about the reverse and anti-reverse engineering of the Deep Neural Network program. In the first work, we notice the new security and privacy risk brought about by the rise of the Deep Learning Compiler and its production, the DNN Program. This work observes this scenario and identifies a new attack surface. We are the first several teams to demonstrate that it is possible to leak the essential information of DNN models by binary reverse engineering. Based on this new attack surface, we propose a novel model extraction attack, namely LibSteal, the advanced model extraction attack framework using binary reverse engineering to steal the architecture of DNN models. We develop solutions based on empirical ways to get the types, attributes, dimensions, and connectivity of the DNN model layers. We implemented a prototype of LibSteal and evaluated it on four DNN programs compiled from TVM. The evaluation results indicate that our framework can reconstruct a similar or even equivalent model architecture compared to the original one. We aim to raise the community's security concern about the possible threat from the perspective of binary reverse engineering.

The great value of DNN programs does not only attract our attention. Several reversing-based model extraction attacks have been proposed. Therefore, followed by the first work, we focus on improving the robustness of DNN programs. In the second work, we investigate current state-of-the-art reversing-based model extraction attacks and implement FlatD, an advanced defense framework for protecting DNN programs based on control flow flattening. FlatD makes it challenging for attackers to recover the control flow graph (CFG) statically and gain necessary information from DNN programs. Compared to the traditional control flow flattening, our evaluation shows that FlatD is an effective, adequate, and practical defense framework that prevents DNN programs from leaking essential information while ensuring their performance and program scale.

Our third work investigates the state-of-the-art fault injection attack against DNN models and finds that it risks affecting the integrity of the DNN Program. By systematically integrating static integrity enforcement mechanisms, our framework secures the integrity of deployed DNN programs, ensuring they remain resistant to potential fault injection attacks. We implement a prototype and evaluate its functionality. The result shows that our methodology can prevent the fault injection attack while preserving the accuracy of the DNN program.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[3] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4):1–12, 2019.

[4] Shun-ichi Amari. Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5(4-5):185–196, 1993.

[5] Amazon. Amazon SageMaker Neo uses Apache TVM for performance improvement on hardware target. https://aws.amazon.com/sagemaker/neo/, 2021.

[6] Apache TVM. Apache TVM conference archive, 2021.

[7] Omid Aramoon, Pin-Yu Chen, and Gang Qu. Aid: Attesting the integrity of deep neural networks. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 19–24. IEEE, 2021.

[8] Amir Averbuch, Michael Kiperberg, and Nezer Jacob Zaidenberg. Truly-protect: An efficient vm-based software protection. *IEEE Systems Journal*, 7(3):455–466, 2013.

[9] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable

code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE, 2019.

[10] Junjie Bai, Fang Lu, and Ke Zhang. ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx, 2019.

[11] Arini Balakrishnan and Chloe Schulze. Code obfuscation literature survey. *CS701 Construction of compilers*, 19, 2005.

[12] Qinkun Bao, Zihao Wang, James R Larus, and Dinghao Wu. Abacus: a tool for precise side-channel analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 238–239. IEEE, 2021.

[13] Qinkun Bao, Zihao Wang, Xiaoting Li, James R Larus, and Dinghao Wu. Abacus: Precise side-channel analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 797–809. IEEE, 2021.

[14] Martial Bourquin, Andy King, and Edward Robbins. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, pages 1–10, 2013.

[15] Wyseur Brecht. White-box cryptography: hiding keys in software. Technical report, NAGRA Kuelski Group, 2012.

[16] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[17] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.

[18] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 985–999. IEEE, 2019.

[19] Jan Cappaert. Code obfuscation techniques for software protection. *Katholieke Universiteit Leuven*, pages 1–112, 2012.

[20] Simin Chen, Hamed Khanpour, Cong Liu, and Wei Yang. Learning to Reverse DNNs from AI Programs Automatically. *arXiv preprint arXiv:2205.10364*, 2022.

[21] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[22] 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 *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[23] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems*, 31, 2018.

[24] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. One engine to fuzz'em all: Generic language processor testing with semantic validation. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 642–658. IEEE, 2021.

[25] Yongheng Chen, Rui Zhong, Yupeng Yang, Hong Hu, Dinghao Wu, and Wenke Lee. μfuzz: Redesign of parallel fuzzing using microservice architecture. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security'23)*, 2023.

[26] François Chollet. Keras. https://github.com/fchollet/keras, 2015.

[27] François Chollet. Xception: Deep Learning with Depthwise Separable Convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[28] François Chollet. Simple MNIST convnet. https://keras.io/examples/vision/mnist_convnet/.

[29] Christian Collberg. Tigress. https://tigress.wtf/, 2014.

[30] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[31] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167, 2008.

[32] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers,

Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning. *arXiv preprint arXiv:1801.08058*, 2018.

[33] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[34] Li Deng. The MNIST database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[35] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186, 2019.

[36] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489. IEEE, 2019.

[37] Stephen Dolan. mov is Turing-complete, 2013. University of Cambridge, Technical Report.

[38] Daniel Dolz and Gerardo Parra. Using exception handling to build opaque predicates in intermediate code obfuscation techniques. *Journal of Computer Science & Technology*, 8, 2008.

[39] Christopher Domas. Turning 'mov' into a soul-crushing RE nightmare. In *Proceeding of the 2015 Annual Reverse Engineering and Security Conference, ser. REcon*, volume 15, 2015.

[40] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing. In *Network and Distributed Systems Security (NDSS) Symposium 2020*, 2020.

[41] Vasisht Duddu, Debasis Samanta, D Vijay Rao, and Valentina E Balas. Stealing Neural Networks via Timing Side Channels. *arXiv preprint arXiv:1812.11720*, 2018.

[42] Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, San Francisco, CA, USA, 2nd edition, 2011.

[43] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor Van Der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the many sides of target row refresh. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 747–762. IEEE, 2020.

[44] Zynamics GmbH. Bindiff. https://www.zynamics.com/bindiff.html, 2010.

[45] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

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

[47] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech Recognition with Deep Recurrent Neural Networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. Ieee, 2013.

[48] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 245–261. IEEE, 2018.

[49] Shay Gueron, Simon Johnson, and Jesse Walker. SHA-512/256. In *2011 Eighth International Conference on Information Technology: New Generations*, pages 354–358. IEEE, 2011.

[50] 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*, pages 770–778, 2016.

[51] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, and Brian Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[52] Sanghyun Hong, Pietro Frigo, Yiğitcan Kaya, Cristiano Giuffrida, and Tudor Dumitraș. Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks under Hardware Fault Attacks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 497–514, 2019.

[53] Fateme S Hosseini, Qi Liu, Fanruo Meng, Chengmo Yang, and Wujie Wen. Safeguarding the Intelligence of Neural Networks with Built-in Light-weight Integrity MArks (LIMA). In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 1–12. IEEE, 2021.

[54] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv preprint arXiv:1704.04861*, 2017.

[55] Xing Hu, Ling Liang, Lei Deng, Shuangchen Li, Xinfeng Xie, Yu Ji, Yufei Ding, Chang Liu, Timothy Sherwood, and Yuan Xie. Neural Network Model Extraction Attacks in Edge Devices by Hearing Architectural Hints. *arXiv preprint arXiv:1903.03916*, 2019.

[56] Weizhe Hua, Zhiru Zhang, and G Edward Suh. Reverse Engineering Convolutional Neural Networks Through Side-Channel Information Leaks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[57] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J Rossbach, and Emmett Witchel. Telekine: Secure Computing with Cloud GPUs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 817–833, 2020.

[58] Vector 35 Inc. Binary Ninja. https://binary.ninja/.

[59] Texas Instruments. The AM335x microprocessors support TVM. software-dl.ti.com/processor-sdk-linux/esd/docs/linux/FoundationalComponents/MachineLearning/tvm.html, 2021.

[60] Animesh Jain, Shoubhik Bhattacharya, Masahiro Masuda, Vin Sharma, and Yida Wang. Efficient execution of quantized deep learning models: A compiler approach. *arXiv preprint arXiv:2006.10226*, 2020.

[61] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

[62] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM–software protection for the masses. In *2015 IEEE/ACM 1st International Workshop on Software Protection*, pages 3–9. IEEE, 2015.

[63] Mika Juuti, Sebastian Szyller, Samuel Marchal, and N Asokan. PRADA: protecting against DNN model stealing attacks. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 512–527. IEEE, 2019.

[64] Ari Karchmer. Theoretical Limits of Provable Security Against Model Extraction by Efficient Observational Defenses. In *2023 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, pages 605–621. IEEE, 2023.

[65] Sanjay Kariyappa, Atul Prakash, and Moinuddin K Qureshi. Maze: Data-free model stealing attack using zeroth-order gradient estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13814–13823, 2021.

[66] Shinpei Kato, Eijiro Takeuchi, Yoshio Ishiguro, Yoshiki Ninomiya, Kazuya Takeda, and Tsuyoshi Hamada. An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68, 2015.

[67] Keras. Keras applications. https://keras.io/api/applications/, April 2022.

[68] Keras. Keras examples. https://keras.io/examples/, April 2022.

[69] Nikhil Ketkar. Introduction to Keras. In *Deep learning with Python*, pages 97–111. Springer, 2017.

[70] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[71] Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[72] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, University of Toronto, Toronto, Ontario, 2009.

[73] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Advances in neural information processing systems*, 25, 2012.

[74] Tımea László and Ákos Kiss. Obfuscating C++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30(1):3–19, 2009.

[75] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[76] Chris Leary and Todd Wang. XLA: TensorFlow, compiled. *TensorFlow Dev Summit*, 2017.

[77] Yann LeCun. The MNIST database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[78] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep Learning. *Nature*, 521(7553):436–444, 2015.

[79] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[80] Dawei Li, Di Liu, Ying Guo, Yangkun Ren, Jieyu Su, and Jianwei Liu. Defending against model extraction attacks with physical unclonable function. *Information Sciences*, 628:196–207, 2023.

[81] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, and Depei Qian. The Deep Learning Compiler: A Comprehensive Survey. *arXiv preprint arXiv:2002.03794*, 2020.

[82] Shaofeng Li, Xinyu Wang, Minhui Xue, Haojin Zhu, Zhi Zhang, Yansong Gao, Wen Wu, and Xuemin Sherman Shen. Yes, One-Bit-Flip Matters! Universal DNN Model Inference Depletion with Runtime Code Fault Injection. In *Proceedings of the 33th USENIX Security Symposium*, 2024.

[83] Xiaoting Li, Lingwei Chen, Jinquan Zhang, James Larus, and Dinghao Wu. Watermarking-based defense against adversarial attacks on deep neural networks. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2021.

[84] Yi Liang, Zhipeng Cai, Jiguo Yu, Qilong Han, and Yingshu Li. Deep Learning-Based Inference of Private Information Using Embedded Sensors in Smart Devices. *IEEE Network*, 32(4):8–14, 2018.

[85] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM, 2003.

[86] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. *ACM SIGPLAN Notices*, 50(4):87–101, 2015.

[87] Chang Liu, Michael Hicks, and Elaine Shi. Memory trace oblivious program execution. In *2013 IEEE 26th Computer Security Foundations Symposium*, pages 51–65. IEEE, 2013.

[88] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing CNN model inference on CPUs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1025–1040, 2019.

[89] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, Xiaofei Xie, and Lei Ma. Decompiling x86 Deep Neural Network Executables. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7357–7374, 2023.

[90] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[91] 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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897. USENIX Association, November 2020.

[92] Anirban Majumdar, Stephen J Drape, and Clark D Thomborson. Slicing obfuscations: design, correctness, and evaluation. In *Proceedings of the 2007 ACM workshop on Digital Rights Management*, pages 70–81. ACM, 2007.

[93] A Markham and Y Jia. Caffe2: Portable High-Performance Deep Learning Framework from Facebook. *NVIDIA Corporation*, 2017.

[94] Meat. Torchvision datasets. http://pytorch.org/vision/main/datasets.html.

[95] MLC team. MLC-LLM. https://github.com/mlc-ai/mlc-llm, 2023-2025.

[96] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W Hamlen, and Michael Franz. Opaque Control-Flow Integrity. In *NDSS*, volume 26, pages 27–30, 2015.

[97] Timothy Prickett Morgan. Inside Facebook's Future Rack And Microserver Iron. https://www.nextplatform.com/2020/05/14/inside-facebooks-future-rack-andmicroserver-iron/.

[98] Janani Mukundan, Hillery Hunter, Kyu-hyoun Kim, Jeffrey Stuecheli, and José F Martínez. Understanding and mitigating refresh overheads in high-density DDR4 DRAM systems. *ACM SIGARCH Computer Architecture News*, 41(3):48–59, 2013.

[99] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 35(4):1–11, 2016.

[100] Paul Muntean, Matthias Neumayer, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. Analyzing Control Flow Integrity with LLVM-CFI. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 584–597, 2019.

[101] NXP. NXP uses Glow to optimize models for low-power NXP MCUs. www.nxp.com/company/blog/glow-compiler-optimizesneural-networks-for-low-power-nxp-mcus:BL-OPTIMIZES-NEURAL-NETWORKS, 2020.

[102] OctoML. OctoML leverages TVM to optimize and deploy models. https://octoml.ai/features/maximize-performance/, 2021.

[103] Seong Joon Oh, Bernt Schiele, and Mario Fritz. Towards Reverse-Engineering Black-Box Neural Networks. In *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, pages 121–144. Springer, 2019.

[104] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. Knockoff nets: Stealing functionality of black-box models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4954–4963, 2019.

[105] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. Prediction poisoning: Towards defenses against dnn model stealing attacks. *arXiv preprint arXiv:1906.10908*, 2019.

[106] OWASP. Runtime Application Self-Protection (RASP). https://owasp.org/www-project-runtime-application-self-protection/.

[107] 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 2017 ACM on Asia conference on computer and communications security*, pages 506–519, 2017.

[108] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS-W*, 2017.

[109] Qualcomm. Qualcomm contributes Hexagon DSP improvements to the Apache TVM community. https://developer.qualcomm.com/blog/tvm-open-source-compiler-now-includesinitial-support-qualcomm-hexagon-dsp, 2020.

[110] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[111] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[112] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. TBT: Targeted Neural Network Attack with Bit Trojan. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 13198–13207, 2020.

[113] K Ren, QR Meng, SK Yan, and Z Qin. Survey of artificial intelligence data security and privacy protection. *Chin J Netw Inf Secur*, 7(1):1–10, 2021.

[114] Nicholas Roberts, Vinay Uday Prabhu, and Matthew McAteer. Model weight theft with just noise inputs: The curious case of the petulant attacker. *arXiv preprint arXiv:1912.08987*, 2019.

[115] Roman Rohleder. Hands-on ghidra-a tutorial about the software reverse engineering framework. In *Proceedings of the 3rd ACM Workshop on Software Protection*, pages 77–78, 2019.

[116] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, Jack Montgomery, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, Misha Smelyanskiy, and Man Wang. Glow: Graph Lowering Compiler Techniques for Neural Networks. *arXiv preprint arXiv:1805.00907*, 2018.

[117] Jürgen Schmidhuber. Deep Learning in Neural Networks: An Overview. *Neural networks*, 61:85–117, 2015.

[118] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[119] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[120] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *International conference on information systems security*, pages 1–25. Springer, 2008.

[121] Lei Song. Machine learning security and privacy: a survey. *Chinese Journal of Network and Information Security*, 4(8):1–11, 2018.

[122] Emil Stefanov, Marten Van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. *Journal of the ACM (JACM)*, 65(4):1–26, 2018.

[123] Jack Tang and Trend Micro Threat Solution Team. Exploring Control Flow Guard in Windows 10. *Trend Micro*, 2015.

[124] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing Machine Learning Models via Prediction APIs. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 601–618, 2016.

[125] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[126] Binghui Wang and Neil Zhenqiang Gong. Stealing Hyperparameters in Machine Learning. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 36–52. IEEE, 2018.

[127] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical report, Technical Report CS-2000-12, University of Virginia, 12 2000, 2000.

[128] Fish Wang and Yan Shoshitaishvili. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9. IEEE, 2017.

[129] Pei Wang, Shuai Wang, Jiang Ming, Yufei Jiang, and Dinghao Wu. Translingual obfuscation. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 128–144. IEEE, 2016.

[130] Pei Wang, Jinquan Zhang, Shuai Wang, and Dinghao Wu. Quantitative Assessment on the Limitations of Code Randomization for Legacy Binaries. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 1–16. IEEE, 2020.

[131] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable Disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 627–642, 2015.

[132] Tianyang Wang, Ziqian Bi, Yichao Zhang, Ming Liu, Weiche Hsieh, Pohsun Feng, Lawrence KQ Yan, Yizhu Wen, Benji Peng, Junyu Liu, Keyu Chen, Sen Zhang, Ming Li, Chuanqi Jiang, Xinyuan Song, Junjie Yang, Bowen Jing, Jintao Ren, Junhao Song, Hong-Ming Tseng, Silin Chen, Yunze Wang, Chia Xin Liang, Jiawei Xu, Xuanhe Pan, Jinlang Wang, and Qian Niu. Deep Learning Model Security: Threats and Defenses. *arXiv preprint arXiv:2412.08969*, 2024.

[133] Zihao Wang, Pei Wang, Qinkun Bao, and Dinghao Wu. Source Code Implied Language Structure Abstraction through Backward Taint Analysis. In *Proceedings of the 18th International Conference on Software Technologies (ICSOFT)*. SCITEPRESS-Science and Technology Publications, 2023.

[134] Sally Ward-Foxton. Google and NVIDIA Tie in MLPerf; Graphcore and Habana Debut. https://www.eetimes.com/google-and-nvidia-tie-inmlperf-graphcore-and-habana-debut, 2021.

[135] 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 34th Annual Computer Security Applications Conference*, pages 393–406, 2018.

[136] Nicholas Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013.

[137] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. Machine Learning at Facebook: Understanding Inference at the Edge. In *2019 IEEE international symposium on high performance computer architecture (HPCA)*, pages 331–344. IEEE, 2019.

[138] 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, 2022.

[139] Yun Xiang, Zhuangzhi Chen, Zuohui Chen, Zebin Fang, Haiyang Hao, Jinyin Chen, Yi Liu, Zhefu Wu, Qi Xuan, and Xiaoniu Yang. Open DNN Box by Power Side-Channel Attack. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 67(11):2717–2721, 2020.

[140] Feng Xiao, Jinquan Zhang, Jianwei Huang, Guofei Gu, Dinghao Wu, and Peng Liu. Unexpected Data Dependency Creation and Chaining: A New Attack to SDN. In *2020 IEEE symposium on security and privacy (SP)*, pages 1512–1526. IEEE, 2020.

[141] Jianxiong Xiao, James Hays, Krista A Ehinger, Aude Oliva, and Antonio Torralba. Sun database: Large-scale scene recognition from abbey to zoo. In *2010 IEEE computer society conference on computer vision and pattern recognition*, pages 3485–3492. IEEE, 2010.

[142] Hui Xu, Yangfan Zhou, Yu Kang, and Michael R Lyu. On Secure and Usable Program Obfuscation: A Survey. *arXiv preprint arXiv:1710.01139*, 2017.

[143] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2003–2020, 2020.

[144] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. DeepHammer: Depleting the Intelligence of Deep Neural Networks through Targeted Chain of Bit Flips. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1463–1480, 2020.

[145] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. Alphaz: A system for design space exploration in the polyhedral model. In *Languages and Compilers for Parallel Computing: 25th International Workshop, LCPC 2012, Tokyo, Japan, September 11-13, 2012, Revised Selected Papers 25*, pages 17–31. Springer, 2013.

[146] Jinquan Zhang, Pei Wang, and Dinghao Wu. LibSteal: Model Extraction Attack towards Deep Learning Compilers by Reversing DNN Binary Library. In *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, 2023.

[147] Jinquan Zhang, Zihao Wang, Pei Wang, Rui Zhong, and Dinghao Wu. FlatD: Protecting Deep Neural Network Program from Reversing Attacks. In *Proceedings of the 47th International Conference on Software Engineering: Software Engineering in Practice*, 2025.

[148] Mingwei Zhang and R Sekar. Control flow and code integrity for COTS binaries: An effective defense against real-world ROP attacks. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 91–100, 2015.

[149] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating High-Performance tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

[150] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–873, 2020.

[151] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. Squirrel: Testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 955–970, 2020.

[152] Yuankun Zhu, Yueqiang Cheng, Husheng Zhou, and Yantao Lu. Hermes attack: Steal DNN models with lossless inference accuracy. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

Vita

Jinquan Zhang

Jinquan Zhang obtained his Ph.D. from the College of Information Sciences and Technology at The Pennsylvania State University in 2025 under the supervision of Dinghao Wu. He specializes in software security, program analysis, and reverse engineering. He also has experience in AI security.