Decompiling x86 Deep Neural Network Executables

Zhibo Liu, Yuanyuan Yuan, Shuai Wang∗
The Hong Kong University of Science and Technology
{zliudc,yyuanaq,shuaiw}@cse.ust.hk
Xiaofei Xie
Singapore Management University
xfxie@smu.edu.sg
Lei Ma
University of Alberta
ma.lei@acm.org

Abstract

Due to their widespread use on heterogeneous hardware devices, deep learning (DL) models are compiled into executables by DL compilers to fully leverage low-level hardware primitives. This approach allows DL computations to be undertaken at low cost across a variety of computing platforms, including CPUs, GPUs, and various hardware accelerators.

We present BTD (Bin to DNN), a decompiler for deep neural network (DNN) executables. BTD takes DNN executables and outputs full model specifications, including types of DNN operators, network topology, dimensions, and parameters that are (nearly) identical to those of the input models. BTD delivers a practical framework to process DNN executables compiled by different DL compilers and with full optimizations enabled on x86 platforms. It employs learning-based techniques to infer DNN operators, dynamic analysis to reveal network architectures, and symbolic execution to facilitate inferring dimensions and parameters of DNN operators.

Our evaluation reveals that BTD enables accurate recovery of full specifications of complex DNNs with millions of parameters (e.g., ResNet). The recovered DNN specifications can be re-compiled into a new DNN executable exhibiting identical behavior to the input executable. We show that BTD can boost two representative attacks, adversarial example generation and knowledge stealing, against DNN executables. We also demonstrate cross-architecture legacy code reuse using BTD, and envision BTD being used for other critical downstream tasks like DNN security hardening and patching.

1 Introduction

Recent years have witnessed increasing demand for applications of deep learning (DL) in real-world scenarios. This demand has led to extensive deployment of DL models in a wide spectrum of computing platforms, ranging from cloud servers to embedded devices. Deployment of models in such a spread of platforms is challenging, given the diversity of hardware characteristics involved (e.g., storage management and compute primitives) including GPUs, CPUs, and FPGAs.

A promising trend is to use DL compilers to manage and optimize these complex deployments on multiple platforms [20, 58, 77]. A DL compiler takes a high-level model specification (e.g., in ONNX format [4]) and generates corresponding low-level optimized binary code for a variety of hardware backends. For instance, TVM [20], a popular DL compiler, generates DNN executable with performance comparable to manually optimized libraries: it can compile models for heterogeneous hardware backends. To date, DL compilers are already used by many edge devices and low-power chips vendors [44, 66, 67, 75]. Cloud service providers like Amazon and Google are also starting to use DL compiler in their AI services for performance improvements [12, 92]. In particular, Amazon and Facebook are seen to spend considerable effort to compile DL models on Intel x86 CPUs through the usage of DL compilers [45, 55, 61].

Compilation of high-level models into binary code typically involves multiple optimization cycles [20, 58, 77]. DL compilers can optimize code utilizing domain-specific hardware features and abstractions. Hence, generated executables manifest distinct representations of the high-level models from which they were derived. However, we observe that different low-level representations of the same DNN operator in executables generally retain invariant high-level semantics, as DNN operators like ReLU and Sigmoid, are mathematically defined in a rigorous manner. This reveals the opportunity of reliably recovering high-level models by extracting semantics from each DNN operator’s low-level representation.

Extracting DNN models from executables can boost many security applications, including adversarial example generation, training data inference, legacy DNN model reuse, migration, and patching. In contrast, existing model-extraction attacks, whether based on side channels [28, 42, 43, 95, 96, 103] or local retraining [68, 70, 71, 85], assume specific attack environments or can leak only parts of DNN models with low accuracy or high overhead.
We propose BTD, a decompiler for DNN executables. Given a (stripped) executable compiled from a DNN model, we propose a three-step approach for full recovery of DNN operators, network topology, dimensions, and parameters. BTD conducts representation learning over disassembler-emitted assembly code to classify assembly functions as DNN operators, such as convolution layers (Conv). Dynamic analysis is then used to chain DNN operators together, thus recovering their topological connectivity. To further recover dimensions and parameters of certain DNN operators (e.g., Conv), we launch trace-based symbolic execution to generate symbolic constraints, primarily over floating-point-related computations. The human-readable symbolic constraints denote semantics of corresponding DNN operators that are invariant across different compilation settings. Experienced DL experts can infer higher-level information about operators (e.g., dimensions, the memory layout of parameters) by reading the constraints. Nevertheless, to deliver an automated pipeline, we then define patterns over symbolic constraints to automatically recover dimensions and memory layouts of parameters. We incorporate taint analysis to largely reduce the cost of symbolic execution which is more heavyweight.

BTD is comprehensive as it handles all DNN operators used in forming computer vision (CV) models in ONNX Zoo [69]. BTD processes x86 executables, though its core technique is mostly platform-independent. Decompiling executables on other architectures requires vendor support for reverse engineering toolchains first. We also find that DNN “executables” on some other architectures are not in stand-alone executable formats. See the last paragraph of Sec. 2 for the significance of decompiling x86 DNN executables, and see Sec. 8 for discussions on cross-platform support.

BTD was evaluated by decompiling 64-bit x86 executables emitted by eight versions of three production DL compilers, TVM [20], Glow [77], NNFusion [58], which are developed by Amazon, Facebook, and Microsoft, respectively. These compilers enable full optimizations during our evaluation. BTD is scalable to recover DNN models from 65 DNN executables, including nearly 3 million instructions, in 60 hours with negligible errors. BTD, in particular, can recover over 100 million parameters from VGG, a large DNN model, with an error rate of less than 0.1% (for TVM-emitted executable) or none (for Glow-emitted executable). Moreover, to demonstrate BTD’s correctness, we rebuild decompiled model specifications with PyTorch. The results show that almost all decompiled DNN models can be recompiled into new executables that behave identically to the reference executables. We further demonstrate that BTD, by decompiling executables into DNN models, can boost two attacks, adversarial example generation and knowledge stealing. We also migrate decompiled x86 DNN executables to GPUs, and discuss limits and potential future works. In summary, we contribute the following:

- This paper, for the first time\(^1\), advocates for reverse engineering DNN executables. BTD accepts as input (stripped) executables generated by production DL compilers and outputs complete model specifications. BTD can be used to aid in the comprehension, migration, hardening, and exploitation of DNN executables.
- BTD features a three-step approach to recovering high-level DNN models. It incorporates various design principles and techniques to deliver an effective pipeline.
- We evaluate BTD against executables compiled from large-scale DNN models using production DL compilers. BTD achieves high accuracy in recovering (nearly) full specifications of complex DNN models. We also demonstrate how common attacks are boosted by BTD.

We provide BTD and data (e.g., recompiled executables) at [9]. We will maintain BTD to benefit follow-up research. We also provide an extended version of this paper at [9] which includes full details of this work.

2 Preliminary

Fig. 1(a) depicts DNN model compilation. DNN compilation can be divided into two phases [53], with each phase manipulates one or several intermediate representations (IR). Computation Graph. DL compiler inputs are typically high-level model descriptions exported from DL frameworks like PyTorch [72]. DNN models are typically represented as computation graphs in DL frameworks. Fig. 1(b) shows a simple graph of a multilayer convolutional neural network (CNN). These graphs are usually high-level, with limited connections.

---
\(^1\) This paper was submitted to USENIX Security 2022 (Fall Round) on October 12, 2021. We received the Major Revision decision and re-submitted the revised version to USENIX Security 2023 (Summer Round) on June 07, 2022. When preparing the camera-ready version, we notice a parallel work DnD [94], which considers decompiling DNN executables across architectures (BTD only considers x86 executables). Nevertheless, DnD does not deeply explore the impact of compiler optimizations compared to our work.
to hardware. DL frameworks export computation graphs often in ONNX format [4] as DL compiler inputs.

Frontend: Graph IRs and Optimizations. DL compilers typically first convert DNN computation graphs into graph IRs. Hardware-independent graph IRs define graph structure. Network topology and layer dimensions encoded in graph IRs can aid graph- and node-level optimizations including operator fusion, static memory planning, and layout transformation [20, 77]. For instance, operator fusions and constant folding are used to identify mergeable nodes in graph IRs after precomputing statically-determinable components. Graph IRs specify high-level inputs and outputs of each operator, but do not restrict how each operator is implemented.

Backend: Low-Level IRs and Optimizations. Hardware-specific low-level IRs are generated from graph IRs. Instead of translating graph IRs directly into standard IRs like LLVM IR [51], low-level IRs are employed as an intermediary step for customized optimizations using prior knowledge of DL models and hardware characteristics. Graph IR operators can be converted into low-level linear algebra operators [77]. For example, a fully connected (FC) operator can be represented as matrix multiplication followed by addition. Such representations alleviate the hurdles of directly supporting many high-level operators on each hardware target. Instead, translation to a new hardware target only needs the support of low-level linear algebra operators. Low-level IRs are usually memory related. Hence, optimizations at this step can include hardware intrinsic mapping, memory allocation, loop-related optimizations, and parallelization [15, 20, 76, 98].

Backend: Scheduling and Tuning. Policies mapping an operator to low-level code are called schedules. A compiler backend often searches a vast combinatorial scheduling space for optimal parameter settings like loop unrolling factors. Halide [76] introduces a scheduling language with manual and automated schedule optimization primitives. Recent works explore launching auto scheduling and tuning to enhance optimization [10, 20, 21, 62, 88, 100, 101]. These methods alleviate manual efforts to decide schedules and optimal parameters.

Backend: Code Gen. Low-level IRs are compiled to generate code for different hardware targets like CPUs and GPUs. When generating machine code, a DNN operator (or several fused operators) is typically compiled into an individual assembly function. Low-level IRs can be converted into mature tool-chains IRs like LLVM or CUDA IR [65] to explore hardware-specific optimizations. For instance, Glow [77] can perform fine-grained loop-oriented optimizations in LLVM IR. DL compilers like TVM and Glow compile optimized IR code into standalone executables. Kernel libraries can be used by DL compilers NNFusion [58] and XLA [86] to statically link with DNN executables. Decompiling executables statically linked with kernel libraries are much easier: such executables contain many wrappers toward kernel libraries. These wrappers (e.g., a trampoline to the Conv implementation in kernel libraries) can be used to infer DNN models. This work mainly focuses on decompiling “self-contained” executables emitted by TVM and Glow, given their importance and difficulty. For completeness, we demonstrate decompiling NNFusion-emitted executables in Sec. 4.4.

Real-World Significance of DL Compilers. DL compilers offer systematic optimization to improve DNN model adoption. Though many DNN models to date are deployed using DL frameworks like Tensorflow, DL compilers cannot be disregarded as a growing trend. Edge devices and low-power processors suppliers are incorporating DL compilers into their applications to reap the benefits of DNN models [44, 66, 67, 75]. Cloud service providers like Amazon and Google include DL compilers into their DL services to boost performance [12, 92]. Amazon uses DL compilers to compile DNN models on Intel x86 CPUs [45, 55]. Facebook deploys Glow-compiled DNN models on Intel CPUs [61]. Overall, DL compilers are increasingly vital to boost DL on Intel CPUs, embedded devices, and other heterogeneous hardware backends. We design BTD, a decompiler for Intel x86 DNN executables. We show how BTD can accelerate common DNN attacks (please see the extended paper [9] for details) and migrate DNN executables to GPUs (Sec. 8). Sec. 8 explains why BTD does not decompile executables on GPUs/accelerators. GPU/accelerator platforms lack disassemblers/dynamic instrumentation infrastructures, and the DL compiler support for GPU platforms is immature (e.g., cannot generate standalone executables).

3 Decompiling DNN Executables

Definition. BTD decompiles DL executables to recover DNN high-level specifications. The full specifications include: ① DNN operators (e.g., ReLU, Pooling, and Conv) and their topological connectivity, ② dimensions of each DNN operator, such as #channels in Conv, and ③ parameters of each DNN operator, such as weights and biases, which are important configurations learned during model training. Sec. 4 details BTD’s processes to recover each component.

Query-Based Model Extraction. Given a (remote) DNN model with obscure specifications, adversaries can continuously feed inputs x to the model and collect its prediction outputs y. This way, adversaries can gradually assemble a training dataset (x, y) to train a local model [71, 87].

This approach may have the following challenges: 1) for a DNN executable without prior knowledge of its functionality, it is unclear how to prepare inputs x aligned with its normal inputs; 2) even if the functionality is known, it may still be challenging to prepare a non-trivial collection of x for models trained on private data (e.g., medical images); 3) local retraining may require rich hardware and is costly; and 4) existing query-based model extraction generally requires prior knowledge of model architectures and dimensions [71]. In contrast, BTD only requires a valid input. For instance, a meaningless image is sufficient to decompile executables of CV models. Also, according to the notation in Definition, local retraining
assumes $1 + 2$ as prior knowledge, whereas BTD fully recovers $1 + 2 + 3$ from DNN executables.

**Model Extraction via Side Channels.** Architectural-level hints (e.g., side channels) leaked during model inference can be used for model extraction [28, 42, 43, 95, 96, 103]. These works primarily recover high-level model architecture, which are $1$ or $1 + 2$ according to our notation in **Definition.** In contrast, BTD statically recovers $1$ and then dynamically recovers $2 + 3$ from DNN executables (but coverage is not an issue; see Sec. 4.2 for clarification). Sec. 9 further compares BTD with prior model extraction works.

**Comparison with C/C++ Decompilation.** BTD is different from C/C++ decompilers. C/C++ decompilation takes executable and recovers C/C++ code that is visually similar to the original source code. Contrarily, we explore decompiling DNN executables to recover original DNN models. The main differences and common challenges are summarized below.

**Statements vs. Higher-Level Semantics:** Software decompilation, holistically speaking, line-by-line translates machine instructions into C/C++ statements. In contrast, BTD recovers higher-level model specifications from machine instructions. This difference clarifies that a C decompiler is not sufficient for decompilation of DNN executables.

**Common Uncertainty:** There is no fixed mapping between C/C++ statements and assembly instructions. Compilers may generate distinct low-level code for the same source statements. Therefore, C/C++ decompilers extensively use heuristics/patterns when mapping assembly code back to source code. Likewise, DL compilers may adopt different optimizations for compiling the same DNN operators. The compiled code may exhibit distinct syntactic forms. Nevertheless, the semantics of DNN operators are retained, and we extract the invariant semantics from the low-level instructions to infer the high-level model specifications. See Sec. 4.3 for details.

**End Goal:** C/C++ compilation prunes high-level program features, such as local variables, types, symbol tables, and high-level control structures. Software decompilation is fundamentally undecidable [23], and to date, compiled C/C++ code mainly aids (human-based) analysis and comprehension, not recompilation. Generating “recompilable” C code is very challenging [30, 89, 90, 93]. In this regard, DNN compilation has comparable difficulty, as compilation and optimization discard information from DNN models (e.g., by fusing neighbor operators). BTD decompiles DNN executables into high-level DNN specifications, resulting in a functional executable after recompilation. Besides helping (human-based) comprehension, BTD boosts model reuse, migration, security hardening, and adversarial attacks. See discussion in Sec. 8 and case studies in extended paper [9].

**Opacity in DNN Executables.** Fig. 2 compares VGG16 [81] executables compiled using three DL compilers. For simplicity, we only plot the control flow graphs (CFGs) of VGG16’s first Conv operator. These CFGs were extracted using IDA-Pro [38]. Although this Conv is only one of 41 nodes in VGG16, Glow compiles it into a dense CFG (Fig. 2(a)). Sec. 2 has introduced graph-level optimizations that selectively merge neighbor nodes. Comparing CFG generated by TVM -O0 (Fig. 2(b)) and by TVM -O3 (Fig. 2(c)), we find that optimizations (e.g., operator fusion) in TVM can make CFG more succinct. We also present CFGs emitted by NNFusion in Fig. 2(d): NNFusion-emitted executables are coupled with the Mlas [60] kernel library. This CFG depicts a simple trampoline to the Conv implementation in MlasGemm.

As in Fig. 2, different compilers and optimizations can result in complex and distinct machine code realizations. However, BTD is designed as a general approach for decompilation of executables compiled by these diverse settings.

**Design Focus.** Reverse engineering is generally sensitive to the underlying platforms and compilation toolchains. As the first piece of work in this field, BTD is designed to process common DNN models compiled by standard DL compilers. Under such conservative and practical settings, BTD delivers highly encouraging and accurate decompilation. Similarly, obfuscation can impede C/C++ decompilation [56]. Modern C/C++ decompilers are typically benchmarked on common software under standard compilation and optimization [14, 19, 90, 93], instead of extreme cases. We leave it as a future work to study decompiling obfuscated DL executables.

### 4 Design

Decompiling DNN executables is challenging due to the mismatch between instruction-level semantics and high-level model specifications. DNN executables lack high-level information regarding operators, topologies, and dimensions. Therefore, decompiling DNN executables presents numerous reverse engineering hurdles, as it is difficult to deduce high-
level model specifications from low-level instructions. We advocate DL decompilers to satisfy the following criteria:

**R1** *(Generalizability)*: Avoid brittle assumptions. Generalize across compilers, optimizations, and versions.

**R2** *(Correctness)*: Use effective, resilient methods and produce correct outputs.

**R3** *(Performance)*: Be efficient when necessary.

**R4** *(Automation)*: Avoid manual analysis and automate the decompilation process.

BTD delivers practical decompilation based on the invariant semantics of DNN operators that aims to meet all four criteria. Our intuition is simple: *DL compilers generate distinct low-level code but retain operator high-level semantics, because DNN operators are generally defined in a clean and rigorous manner.* Therefore, recovering operator semantics should facilitate decompilation generic across compilers and optimizations *(R1)*. Besides, as invariant semantics reflect high-level information, e.g., operator types and dimensions, we can infer model abstractions accurately *(R2)*.

Fig. 3(a) depicts the BTD workflow. Sec. 4.1 describes learning-based techniques for recognizing assembly functions as DNN operators like Conv. Given recovered DNN operators, we reconstruct the network topology using dynamic analysis *(Sec. 4.2)*. We then use trace-based symbolic execution to extract operator semantics from assembly code and then recover dimensions and parameters with semantics-based patterns *(Sec. 4.3.2)*. Some operators are too costly for symbolic execution to analyze. We use taint analysis to keep only tainted sub-traces for more expensive symbolic execution to analyze *(R3)*, as noted in Sec. 4.3.1. BTD is an end-to-end, fully automated DNN decoder *(R4)*. BTD produces model specifications that behave identically to original models, whose focus and addressed challenges are distinct from C/C++ decompilation. BTD does not guarantee 100% correct outputs. In Sec. 5, we discuss procedures users can follow to fix errors.

Dimensions and parameters configure DNN operators. We show representative cases in Fig. 3(b). Type I operators, including activation functions like ReLU and element-wise arithmetic operators, do not ship with parameters; recovering their dimensions is trivial, as clarified in the caption of Fig. 3. Type II and III operators require dimensions or parameters, such as Pooling’s stride $S$ and kernel size $K$. In addition to simple arithmetic operators, BiasAdd involves bias $B$, as extra parameters. Type IV operators require both parameters and dimensions. These operators form most DNN models. Sec. 7.1 empirically demonstrates “comprehensiveness” of our study.

BTD recovers dimensions/parameters of all DNN operators used by CV models in ONNX Zoo *(see Sec. 7.1)*. Due to limited space, Sec. 4.3 only discusses decompiling the most challenging operator, Conv. The core techniques explained in Sec. 4.3 are utilized to decompile other DNN operators. However, other operators may use different (but simpler) patterns. Our extended paper [9] lists other operator patterns. We further discuss the extensibility of BTD in Sec. 7.3.

**Disassembling and Function Recovery.** BTD targets 64-bit x86 executables. Cross-platform support is discussed in Sec. 8. BTD supports stripped executables without symbol or debug information. We assume that DNN executables can be first flawlessly disassembled with assembly functions recovered. According to our observation, obstacles that can undermine disassembly and function recovery in x86 executables, e.g., instruction overlapping and embedded data [30], are not found in even highly-optimized DNN executables. We use a commercial decompiler, IDA-Pro [38] *(ver. 7.5)*, to maximize confidence in the credibility of our results.

**Compilation Provenance.** Given a DNN executable $e$, compilation provenance include: 1) which DL compiler is used, and 2) whether $e$ is compiled with full optimization -O3 or no optimization -O0. Since some DNN operators *(e.g., type IV in Fig. 3(b))* in $e$ are highly optimized when compiled, the compilation provenance can be inferred automatically by analyzing patterns over sequences of x86 instructions derived from $e$. We extend our learning-based method from Sec. 4.1 to predict compilation provenance from assembly code. Our evaluation of over all CV models in ONNX Zoo finds no errors. Overall, we assume that compilation provenance is known to BTD. Therefore, some patterns can be designed separately for Glow- and TVM-emitted executables; see details in the extended paper [9]. To show $e$’s decompilation is flawless, we must recompile decompiled DNN models with the same provenance *(see Sec. 7.1.4)*. Using different compilation provenances may induce (small) numerical accuracy discrepancies and is undesirable.

This section focuses on decompilation of self-contained DNN executables compiled by TVM and Glow. Decompilation of NNFusion-emitted executables is easier because of its
distinct code generation paradigm. We discuss decompiling NNFusion-emitted executables in Sec. 4.4.

4.1 DNN Operator Recovery

As introduced in Sec. 2, one or a few fused DNN operators are compiled into an assembly function. We train a neural model to map assembly functions to DNN operators. Recent works perform representation learning by treating x86 opcodes as natural language tokens [26, 27, 54, 73, 97]. These works help comprehend x86 assembly code and assist downstream tasks like matching similar code. Instead of defining explicit patterns over x86 opcodes to infer DNN operators (which could be tedious and need manual efforts), we use representation learning and treat x86 opcodes as language tokens.

Atomic OPs. Launching representation learning directly over x86 opcodes syntax can result in poor learning quality. Due to x86 instructions’ flexibility, opcodes with (nearly) identical semantics may have distinct syntactic forms, e.g., vmulps and mulps denoting multiply over floating numbers of different sizes. Rare words machine translation [80] are recently advanced from the observation that natural language words can be divided into atomic units. Translators can use atomic units to translate rare words. Accordingly, we define atomic OPs over x86 opcodes: an atomic OP represents an atomic and indivisible unit of an x86 opcode. Each opcode is thus split into atomic OPs. While a DNN operator could be compiled into various x86 opcode sequences, induced atomic OP sequences can better reflect “semantics” in a noisy-resilient way.

Dividing Opcodes into Atomic OPs. As a common approach, we segment opcodes using Byte Pair Encoding (BPE) [33]. BPE iteratively replaces the most frequent consecutive bytes in a sequence with a single, unused byte. We split each opcode into a sequence of characters and counted consecutive characters to find the most frequent ones. BPE iterates until the opcodes of all atomic OPs have been merged. For instance, opcodes vmulps and mulps are first split into “v m u l p s” and “m u l p s”, and an atomic OP mulps is eventually extracted.

Learning over Atomic OPs. We train a neural identifier model with a sequence of atomic OPs from an assembly function as inputs. This model outputs a 1D vector with N dimensions (N is the total number of unique DNN operators), where multiple “1” in the vector implies that this assembly function represents several fused DNN operators. All “0” in the vector implies this function may be a DL compiler-inserted utility function (e.g., for memory management). The order of fused operators is represented in symbolic constraints extracted in Sec. 4.3. Thus, predicted operator labels and network topology will be refined after symbolic execution. Our model’s frontend learns a neural embedding for each atomic OP and then embeds a function’s entire atomic OP sequence. The order of atomic OPs is found to be vital in prediction. Therefore, we preserve the order of collected atomic OPs within the assembly function. We encode orders with LSTM [40] (see Sec. 5) and enhance learning with neural attention [16].

From Operators to Compilation Provenance. As noted in Sec. 4, our decompilation pipeline requires compilation provenance. We extend the model presented in this section to recover compilation provenance. The extended model predicts compilation provenance using embeddings of all functions in an executable as its input (function embeddings have been generated above). We clarify this task is generally simple; humans can easily distinguish assembly functions in executables from different compilation provenances.

4.2 DNN Network Topology Recovery

Recovering DNN network topology is straightforward, regardless of underlying operator semantics. DNN operators are chained into a computation graph. Generally, a DNN operator has a fixed number of inputs and outputs [6]. According to our observation, DL compilers compile DNN operators into assembly functions and pass inputs and outputs as memory pointers through function arguments. We use Intel Pin [57], a dynamic instrumentation tool, to hook every callsite. During runtime, we record the memory addresses of inputs/outputs passed to callsites and connect two operators if the successor’s inputs match the predecessor’s outputs.

We do not rely on any compiler-specific assumptions like function signatures. This step is independent of later steps and only uses shallow information readily available in binaries. In case the inputs and outputs are passed differently (e.g., not using pointers) in further compiler implementation changes, we envision updating the instrumented code accordingly without much engineering effort required. Also, we clarify that this dynamic analysis is not limited by “coverage”. We do not require “semantically meaningful” inputs (not like a query-based model extraction [71, 87]), just a format-valid input to record how each operator in the executable accesses memory. One format-valid, trivial (meaningless) input can achieve 100% coverage. Besides, whether a model accepts images or text as its valid inputs is easy to determine.

4.3 Dimension and Parameter Recovery

As in Fig. 3, certain complex DNN operators are configured with dimensions and parameters. This section details solutions to recover parameters/dimensions. To present a comprehensive working example, we use Conv, the most complex operator in our dataset, to introduce our solutions. Nonetheless, our solutions are general enough to cover all operators in CV models of ONNX Zoo (see Sec. 7.1) and can be extended to support more operators with trivial effort (see Sec. 7.3).

Fig. 4(a) shows the input, the kernel, and the output of a simple Conv computation. Suppose no optimization is applied, we present the memory layout of Conv in Fig. 4(b). Inputs and parameters are typically stored separately in memory, whereas
We launch SE over tainted x86 instructions. While existing symbolic execution tools do not support pervasive SSE instructions in DNN executables, we reimplement a trace-based SE engine that models all SSE floating-point computations encountered in tainted traces. We ignore irrelevant semantics like CPU flags. As with taint analysis, symbolic pointers are computed using concrete values. For instance, movss xmm1, dword ptr [rcx] will load floating numbers from memory pointed by rcx. Given that dword denotes 4 bytes, if rcx is 0x29b8, we create (0x29b8, 4) as xmm1’s symbolic value while the upper 16-4 bytes are reset to zero. After performing SE on tainted trace, we get a (simplified) symbolic constraint as in Fig. 4(c), which shows how inputs and parameters in memory (see Fig. 4(b)) are used to computing an output.

Identifying Memory Layouts. To determine if each address in the symbolic constraint points to inputs or parameters, we form a once-for-all configuration that records the meaning of each argument (inputs or parameters) of the corresponding assembly function for different operators (see extended paper [9]). We can collect and identify inputs and parameters’ memory addresses by querying the configuration. For the constraint in Fig. 4(c), we will identify memory addresses and classify them into weights (marked in red) and inputs (marked in yellow). Furthermore, by logging and identifying all memory addresses accessed during an operator’s computation, we can cluster all addresses of the same parameter to scope that parameter’s memory region (i.e., the starting address and size).

4.3.3 Dimension Recovery

For reverse engineering, heuristic are hardly avoidable [89,90]. We now present patterns defined over the extracted constraints, which enable recovering dimensions and parameter layouts. Without compromising generality, we mainly introduce patterns we use to recover Conv operator dimensions and layouts. Other operators in our dataset can be covered smoothly with simpler patterns, as we stated in Sec. 4.3.

Kernel Size $K$, Input Channel $I_C$, Zero Padding $P$. Consider Fig. 4(c), given the relative offsets of four marked input memory addresses are $[0, 4, 12, 16]$, we can infer the kernel shape as $2 \times 2$ (the continuous sequence has length 2), indicating that $K = 2$. Further, we calculate #input channels $I_C = \frac{4}{2 \times 2} = 1$, as to compute one output element, we recognize four inputs (which belong to one input channel) in the symbolic constraint. Also, considering the memory layout in figure b, it should be easy to infer that “12” denotes the first element from the next row (element at row 3, column 2). Hence, we can compute the shape of the input matrix as $\frac{12}{3} = 4$ where 4 denotes the size of one floating number on 64-bit x86 platforms. The input shape is therefore $3 \times 3$. As the network topology has been recovered in Sec. 4.2, we compare the output of the prior operator with the input of this Conv to

Figure 4: Launching trace-based symbolic execution (SE) to infer dimensions and localize parameters for Conv operators.
With recovered dimensions and dumped parameters in data
we do not need to log every instruction and its associated
memory locations when executing Conv. Since we do not need to log every instruction and its associated
context (which involves lots of string conversions and I/Os),
Pin runs much faster than being used to log execution traces
in Sec. 4.3.1. We then re-launch the analysis detailed in Identifying Memory Layouts to identify all addresses belonging
to weights and then determine the size of the memory region
that stores weights, which implies the size of weights. Let the memory region size be \( M_w \), \( O_C \) can be computed as \( \frac{M_w}{k_c \times k \times k} \).

**Stride** \( S \). Let the input (output) height be \( IH_i \) (\( OH_i \)), we compute stride \( S \) using the following dimension constraint:

\[
OH_i = \left\lfloor \frac{(IH_i + 2P - K)}{S} \right\rfloor + 1
\]

The memory region size \( M_i \) of Conv inputs can be decided in the same way as deciding \( M_w \). Hence, the input height

can be computed as \( IH_i = \sqrt{\frac{M_i}{O_i}} \). Similarly, we can compute \( OH_i \) as \( \sqrt{\frac{M_o}{O_i}} \), where \( M_o \) is the memory region size of Conv

outputs. Stride \( S \) is thus computed using the constraint above.

In sum, BTD extracts a series of facts based on the symbolic constraint and runtime information of DNN executables, including 1) the memory regions size of inputs, outputs, and parameters, 2) the relative offsets of the input memory addresses, 3) the relative offsets of the parameter memory addresses, and 4) the number of specific arithmetic operations inside a symbolic constraint. Lacking any of these cannot fully recover a (complex) DNN operator. Nevertheless, since these facts are consistently presented in DNN executables generated by different DL compilers, our dimension recovery techniques are generic across compilers and optimizations. See [9] for handling other operators (in the same procedure) and Sec. 7.2 for the generalization evaluation.

### 4.3.4 Recover Parameters

Recovering parameters requires to identify their starting addresses and memory layouts. As clarified in **Output Channels** \( O_C \), we identify memory region \( M_w \) that stores parameters. We then use Pin to dump parameters to disk at runtime. With recovered dimensions and dumped parameters in data bytes, we can recover well-formed parameters. Operators may have multiple parameters, and more than one pointers to distinct parameters may appear in the assembly function arguments (see extended paper [9] for function interfaces of operators). Each pointer’s parameter is recovered separately.

**Handling Compiler Optimizations**. Fig. 4(b) depicts a Conv’s memory layout. However, compilers may optimize Conv to reduce runtime cost. Both TVM and Glow may perform layout alteration optimizations to take advantage of SSE parallelism by reading 4 (or 8) floating numbers from contiguous memory into one register. These floating numbers can be computed with one SSE instruction (optimized memory layout is depicted in Sec. 7.1.6). These optimizations modify Conv’s standard memory layout, impeding parameter recovery. Similar to dimension inference, we use patterns to identify optimized layouts. We detail patterns in extended paper [9]. In short, BTD is nearly flawless; see discussion in Sec. 7.1.6.

### 4.4 Executables Emitted by NNFusion

The procedure described in Sec. 4 decompiles self-contained DNN executables — outputs of the dominant compilers TVM and Glow. As introduced in Sec. 2, some DL compilers, including NNFusion [58] and XLA [86], generate executables statically linked with kernel libraries. It is easier to decompile NNFusion- and XLA-emitted executables since they contain wrapper functions to invoke operator implementations in kernel libraries. We can easily determine DNN operator types by matching those wrapper functions. During runtime, we use Pin to recover network topology and intercept data sent via wrappers. While this work primarily focuses on decompiling more challenging code generation patterns adopted by industrial-strength DNN compilers, TVM [20] and Glow [77], we only empirically demonstrate the feasibility of decompiling NNFusion executables in Sec. 7.1.5.

### 5 Implementation

BTD is primarily written in Python with about 11K LOC. Our Pin plugins contain about 3.1K C++ code. The current implementation decompiles 64-bit executables in the ELF format on x86 platforms. See discussion on cross-platform support in Sec. 8. We use LSTM for DNN operators inference in an “out-of-the-box” manner to deal with distinct optimized low-level code of the same type of operator resulting from different dimensions. The model is a one-layer LSTM [40] whose hidden dimension is 128. The LSTM is implemented using PyTorch [72], with CUDA 10.0 [64] and cuDNN [22].

### 6 Usage & Error Fixing

BTD offers an end-to-end, automated decompilation. All tasks of Fig. 3(a) require no human intervention. However, decompilation is inherently challenging, and BTD may make mistakes. This section first explains how a user use BTD in practice, and then discuss error fixing.

**Usage.** Given a DNN executable, a user first disassembles it (e.g., using IDA-Pro) and recovers all assembly functions. The user also need to provide a format-valid input of this executable for use. Next, as an end-to-end procedure, BTD predicts compilation provenance and each disassembly function’s operator type. BTD then launches the network topology recovery before conducting symbolic execution and recovering dimensions and parameters for each operator, as ex-
plained in Sec. 4. Note that at this step, BTD uses a set of error detection rules (see below) to detect and fix potential errors. Decomposition process is then re-invoked if errors are fixed. If the error cannot be resolved, human intervention is required. The user needs to read and understand the symbolic constraints to fix the error. Human comprehension at this step is the only uncertain but necessary step of the decomposition if complex errors occur. Finally, the user can rebuild the model using the recovered model specification on DL frameworks like PyTorch.

**Error Fixing.** To augment BTD’s decomposition pipeline, we provide a set of rules based on the basic knowledge of ML models, whose violation uncovers decomposition errors. Some rules have error-fixing actions, but not all errors can be fixed in an automated manner. In that case, the user can inspect and fix those errors manually. Also, the user may extend the rules based on their observation and experience of ML models. Currently, we have six error detection rules, of which Rules 1–4 have follow-up automated fixing actions, while Rules 5–6 require human intervention for error fixing:

1. Dimensions of Conv operators must be integers. Otherwise, BTD reports an error and instead uses input/output of predecessor/successor operators to form the dimensions; see a relevant case study in the extended paper [9].
2. Inputs of Add operators must be other operators’ outputs. If not, BTD infers this operator type as BiasAdd.
3. The Split operator’s output memory region size should be smaller than its input memory region size. If not, BTD infers this operator type as Concatenate.
4. The symbolic constraint of an operator with ReLU label (e.g., “Conv+ReLU”) must contain a “max” operation. If not, BTD fixes its inferred operator type by removing the “ReLU” label (e.g., “Conv+ReLU” → “Conv”).
5. If operator inference model’s confidence score is below 80%, and no error is detected by Rules 2-4, BTD throws an error and requires human intervention.
6. An operator’s input shape must match its predecessor’s output shape. Otherwise, BTD throws an error and requires human intervention.

Conceptually, Rule 1 and 6 validate dimension inference, whereas Rules 2–5 validate operator inference results. Rules 2–5 are designed based on observations of our manual exploration, not failed cases when inferring test models.

### Table 1: Compilers evaluated in our study.

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Publication</th>
<th>Developer</th>
<th>Version (git commit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TVM [20]</td>
<td>OSDI ’18</td>
<td>Amazon</td>
<td>v0.7.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>v0.8.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>v0.9.dev</td>
</tr>
<tr>
<td>Glow [77]</td>
<td>arXiv</td>
<td>Facebook</td>
<td>2020 (07a82bd9ef977dd)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2021 (97853ecc670bd2f)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2022 (791f4c7b6260db)</td>
</tr>
<tr>
<td>NNFusion [58]</td>
<td>OSDI ’20</td>
<td>Microsoft</td>
<td>v0.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>v0.3</td>
</tr>
</tbody>
</table>

7 Evaluation

In this section, we evaluate BTD by exploring the following four research questions (RQs) below:

**RQ1 (Comprehensiveness and Correctness):** Is BTD comprehensive and correct to process all operators used in common DL models compiled with different compilers and optimization options?

**RQ2 (Robustness):** Is BTD robust to survive frequent DL compiler implementation changes?

**RQ3 (Extensibility):** Can BTD be easily extended to support new operators and models? What efforts are needed?

**RQ4 (Error Fixing):** How does BTD handle decompilation errors?

We evaluated BTD with seven real-world CV models and an NLP model compiled with eight versions of compilers to provide a comprehensive evaluation. BTD can produce correct model specifications on 59 of 65 DNN executables, and experienced users can quickly fix 3 of 6 remaining errors. Nevertheless, we recognize that some errors cannot be easily fixed by normal users. In the evaluation, we only use ground truths to verify the correctness of decompilation results. BTD is designed to cope with real-world settings and does not rely on any ground truth. Setup and results are below.

**Compilers.** Table 1 lists compilers we used. We select eight versions of three state-of-the-art DL compilers. Glow does not have a release yet, so we use three versions that are at least six months apart. Glow and NNFusion only generate fully-optimized executables, but TVM can be configured to use different optimization levels. Therefore, we use TVM with and full optimizations to build two sets of executables, while using Glow and NNFusion with default settings. All models are compiled into 64-bit x86 executables.

**Test Data.** Table 2 lists all evaluated DNN models. All these models, except FastText, are extensively used in CV tasks. NNFusion can only compile its own shipped VGG11. Our large-scale dataset includes a total of 675 operators and more than 178 million parameters. Note that these operators have covered all types of DNN operators used in the CV models in ONNX Zoo. FastText is a common NLP model that contains Embedding, FC, and Pooling. Embedding is a frequently-used DNN operator in NLP models that encodes text into embedding vectors. Since Embedding is not included in the training dataset, we manually label functions in FastText.

These ONNX files are compiled with TVM and Glow and then disassembled with IDA-Pro. Table 2 presents assembly code statistics. In general, one (or several fused) operator corresponds to one TVM/Glow compiled assembly function. Besides, TVM and Glow will add utility functions (e.g., for memory allocation) that are not covered in the table.
Table 2: Statistics of DNN models and their compiled executables evaluated in our study.

<table>
<thead>
<tr>
<th>Model</th>
<th>#Parameters</th>
<th>#Operators</th>
<th>TVM -O0</th>
<th>TVM -O3</th>
<th>Glow -O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet18 [50]</td>
<td>11,703,912</td>
<td>69</td>
<td>49,762</td>
<td>281</td>
<td>61,002</td>
</tr>
<tr>
<td>VGG16 [81]</td>
<td>138,357,544</td>
<td>41</td>
<td>40,205</td>
<td>215</td>
<td>41,750</td>
</tr>
<tr>
<td>FastText [18]</td>
<td>2,500,101</td>
<td>3</td>
<td>9,867</td>
<td>142</td>
<td>7,477</td>
</tr>
<tr>
<td>Inception [83]</td>
<td>6,998,552</td>
<td>105</td>
<td>121,481</td>
<td>615</td>
<td>74,992</td>
</tr>
<tr>
<td>ShuffleNet [99]</td>
<td>2,294,784</td>
<td>152</td>
<td>56,147</td>
<td>407</td>
<td>34,637</td>
</tr>
<tr>
<td>MobileNet [41]</td>
<td>3,487,816</td>
<td>89</td>
<td>69,903</td>
<td>363</td>
<td>46,214</td>
</tr>
<tr>
<td>EfficientNet [84]</td>
<td>12,966,032</td>
<td>216</td>
<td>89,772</td>
<td>546</td>
<td>49,285</td>
</tr>
</tbody>
</table>

Table 3: Average accuracy of DNN operator inference.

<table>
<thead>
<tr>
<th>Model</th>
<th>2020</th>
<th>2021</th>
<th>2022</th>
<th>v0.7</th>
<th>v0.8</th>
<th>v0.9.dev</th>
<th>v0.7</th>
<th>v0.8</th>
<th>v0.9.dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet18</td>
<td>99.79%</td>
<td>99.84%</td>
<td>100%</td>
<td>99.81%</td>
<td>99.91%</td>
<td>99.95%</td>
<td>99.79%</td>
<td>99.84%</td>
<td>100%</td>
</tr>
<tr>
<td>VGG16</td>
<td>99.95%</td>
<td>99.79%</td>
<td>99.57%</td>
<td>99.80%</td>
<td>99.91%</td>
<td>99.79%</td>
<td>99.95%</td>
<td>99.79%</td>
<td>100%</td>
</tr>
<tr>
<td>Inception</td>
<td>99.98%</td>
<td>99.88%</td>
<td>99.98%</td>
<td>99.96%</td>
<td>99.82%</td>
<td>99.96%</td>
<td>99.80%</td>
<td>99.84%</td>
<td>100%</td>
</tr>
<tr>
<td>ShuffleNet</td>
<td>99.96%</td>
<td>99.82%</td>
<td>100%</td>
<td>99.62%</td>
<td>99.71%</td>
<td>99.96%</td>
<td>99.95%</td>
<td>99.79%</td>
<td>99.31%</td>
</tr>
<tr>
<td>MobileNet</td>
<td>99.35%</td>
<td>99.46%</td>
<td>99.40%</td>
<td>99.33%</td>
<td>99.46%</td>
<td>99.40%</td>
<td>99.80%</td>
<td>99.80%</td>
<td>100%</td>
</tr>
<tr>
<td>EfficientNet</td>
<td>99.65%</td>
<td>99.68%</td>
<td>99.59%</td>
<td>99.65%</td>
<td>99.68%</td>
<td>99.59%</td>
<td>99.81%</td>
<td>99.91%</td>
<td>100%</td>
</tr>
</tbody>
</table>

The training dataset contains landmark CV models and a common NLP model listed in Table 2. These CV models contain all kinds of operators used by CV models in ONNX Zoo. We first present an in-depth evaluation of decompiling CV models, assessing the correctness and comprehensiveness of BTD’s technical pipeline over all included DNN operators. We then discuss the comprehensiveness over NLP and audio processing models.

7.1.1 Predicting DNN Operator Type

In our test dataset, Glow-compiled executables have 14 types of DNN operators and TVM-compiled executables have 30. As introduced in Sec. 4.1, our operator identifier outputs a 1D vector of 14 or 30 elements for each assembly function, where a “1” in $k$th element indicates that this function should be labeled to $k$th operator. We allow multiple “1”, because operators can be fused into one assembly function. As a result, DNN operator inference is performed as two-class classification tasks over 14 or 30 labels. DL compilers provide the ground truth (function labels). We report the overall accuracy in Table 3, where prediction of an function is correct, when the predicted label describes exactly the same operation as the ground truth label. We interpret the prediction as highly accurate. Particularly, we achieve 100% accuracy for all executables compiled by Glow. We check all errors in TVM and discuss the root causes as follows:

Data Bias. Conv is commonly used with a following ReLU for feature extraction. Given “Conv+ReLU” patterns are frequent in training data, a ConvAdd operator emitted by TVM -O3 is mislabeled as ConvAddReLU. In contrast, Dense is often used in the last few layers of DNN without ReLU. Therefore, when ReLU is fused with DenseAdd under TVM -O3, our model mislabels DenseAddReLU as DenseAdd. Unbalanced real-world training data causes these mislabels. Errors may be eliminated by post-checking if symbolic constraints indicating ReLU (i.e., containing “max”) exists.

Operators with Similar Assembly Code. In TVM generated code, BiasAdd can be predicted as Add, and vice versa. As expected, assembly code of these two operators are similar.
Table 4: Parameter/dimension inference. Each column reports dimension inference accuracy/parameter inference accuracy. The complete data is available in the extended paper [9].

<table>
<thead>
<tr>
<th>Model</th>
<th>Glow (2020, 2021, 2022)</th>
<th>TVM -O0 (v0.7, v0.8, v0.9.dev)</th>
<th>TVM -O3 (v0.7, v0.8, v0.9.dev)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet18</td>
<td>100%/100%</td>
<td>92.15%/99.37%</td>
<td>100%/99.37%</td>
</tr>
</tbody>
</table>

7.1.2 DNN Network Topology Recovery

Recovering network topology (Sec. 4.2) is straightforward and rapid. To validate correctness, we compare the recovered network topology with the reference DNN’s computation graph for executables compiled by TVM -O0. For all evaluated DNN models, the recovered network structure is fully consistent with the reference. As for executables compiled by TVM -O3 and Glow, optimizations can change the high-level graph view of models. Thus, it becomes difficult to compare the recovered topology with reference models. Nevertheless, we note that all these test cases are shown as flawless in Table 4.

7.1.3 Parameter and Dimension Recovery

Table 4 reports parameter/dimension recovery accuracy. We only list results for ResNet (as its recovery at this step has defects). Besides ResNet, the accuracies for all other models are 100%, and results are consistent across different compiler versions; see complete data in [9]. Except for TVM -O0, it is difficult to compare the recovered dimensions/parameters with the reference due to compiler optimizations. Hence, #failures in Table 4 equals #dimensions or #parameters that need to be fixed before the recovered models can be compiled into executables showing identical behavior with the references. Some operator inference failures do not involve dimensions/parameters, and are thus not reflected in Table 4.

Overall, BTD can determine dimensions of different DNN operators with negligible errors over all settings. Four failures in ResNet18 (TVM -O0) are due to a Conv optimization (see Sec. 7.1.6), while all dimensions of ResNet18 (TVM -O3) are correctly recovered. Besides, despite huge volume of parameters in each model, the results are promising. BTD failed to recover about 73K parameters of an optimized Conv operator in ResNet18 (TVM -O3) due to its specially-optimized memory layout; see root causes in Sec. 7.1.6.

7.1.4 Recompilation

Recompilation is an active field in reverse engineering, though recompiling decompiled C/C++ code is challenging [30, 89, 90, 93]. This section demonstrates the feasibility of recompiling decompiled DNN models. Recompilation requires a fully fledged decompilation, with the end results again being a functional executable exhibiting identical behavior with the reference. This demonstrates the feasibility of DNN model reuse, migration, and patching. To do so, we re-implement DNN models in PyTorch using recovered DNN models, then export models as ONNX files and compiled into DNN executables using the same compilation provenance.

It is not desirable to directly compare the recovered high-level model specifications with the reference model’s specifications: compilation and optimization inevitably change DNN model representation (e.g., fusing operators). Thereby inconsistency of two high-level specifications does not necessarily indicate a difference in model outputs. Instead, We compare recompiled and reference executables directly. Specifically, we compare the predicted labels and confidence scores yielded by recompiled and reference executables over every input from validation dataset. Two executables are deemed identical if labels and confidence scores are exactly identical or with only negligible floating-point precision loss. For image classification models, we randomly select 100 images from 100 different categories in ImageNet [25] to form a validation dataset. For FastText, we randomly crafted 50 inputs.

Table 5: Recompilation. “NA” means that some errors are not fixed, thus the rebuilt models manifest inconsistent behavior. See full results in the extended paper [9].

<table>
<thead>
<tr>
<th>Model</th>
<th>Glow (2020, 2021, 2022)</th>
<th>TVM -O0 (v0.7, v0.8, v0.9.dev)</th>
<th>TVM -O3 (v0.7, v0.8, v0.9.dev)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet18</td>
<td>100%</td>
<td>100% (with fixing)</td>
<td>NA → 100%</td>
</tr>
</tbody>
</table>

recovery.
We also measure the size and speed of recompiled and reference executables. We report that no noticeable changes can be observed comparing recompiled and original executables.

### 7.1.5 Decompiling NNFusion Outputs

As clarified in Sec. 4.4, decompiling executables emitted by NNFusion and XLA are much easier, as these executables are linked with kernel libraries. For completeness, we run an automated process to decompile executables emitted by NNFusion v0.2 and v0.3. NNFusion cannot compile VGG provided by ONNX Zoo. We thus compile the VGG11 model shipped by NNFusion. To verify the correctness, we reimplement VGG in PyTorch using recovered VGG descriptions and parameters. We follow the same step in Sec. 7.1.4 to validate the recovered model. Note that PyTorch and DNN executables may show negligible deviation between results, which, we believe is from numerical accuracy instead of errors. We set a threshold to allow $10^{-4}$ difference in the outputs of VGG in PyTorch and in executable. All validation inputs are passed. Therefore, we conclude that decompilation is correct.

![Diagram](a) Data Flow of Standard Conv (b) Data Flow in Optimized Executable

Figure 5: Reshapes inserted before Conv by TVM.

### 7.1.6 Root Cause Analysis

Besides errors due to mislabeled operators, two failures occurred when inferring parameters and dimensions. We now discuss the root causes.

**Case One.** All dimension/parameter recovery failures of ResNet (TVM -O3) are from one Conv operator. The failure is due to a Reshape operator inserted by TVM before Conv. To speed up program execution, TVM may add extra code to reshape the input at runtime before Conv calculation, as shown in Fig. 5. However, the reshaped input addresses extracted over symbolic constraints no longer solely reflect elements in the Conv kernels, thus restraining our patterns.

**Case Two.** Another parameter recovery failure also roots in Conv (TVM -O3). Fig. 6 presents the standard memory layout of a Conv which is denoted as $[O, I, K, K]$. While this layout can be flawlessly recovered by BTD, as mentioned in Sec. 4.3.4, DL compilers may alter memory layout of parameters to take advantage of SSE instructions. As shown in Fig. 6, the parameter layout can be optimized as $[O/A, I, K, K_A]$. BTD can correctly infer memory layouts for 33 of 34 Conv operators optimized this way (see our patterns in the extended paper [9]). Nevertheless, we still encounter one rare case where weights are not loaded in the order assumed in our patterns, which is likely due to TVM’s auto scheduling. This breaks our patterns to recover parameters. In contrast, while Glow also extensively optimize memory layout, BTD can flawlessly recover all Conv operators compiled by Glow.

![Diagram](chart)

Figure 6: Layout optimization.

### 7.1.7 Other Models

**NLP Models.** We also tried to incorporate NLP models into our evaluation. However, existing DL compilers still lack complete support for basic NLP operators, such as RNN and LSTM. Table 6 reports the results of preliminary investigation. We select two common NLP models, Char-RNN [7] and LSTM [8], from PyTorch tutorial. Only the current version of Glow can successfully compile both models. Thus, we evaluated BTD with Char-RNN compiled with all versions of Glow and LSTMD compiled with Glow 2022, and BTD could smoothly output the correct model specifications. With manual inspection, we find that typical NLP operators, such as RNN, GRU, and LSTM, are decomposed into sub-operators during compilation, including FC operators and Element-wise arithmetic operators. We note that these decomposed operators are already included in the ONNX Zoo CV models [69].

**Audio Processing Models.** We expect that BTD can also decompile audio processing models without extension. To clarify, in the era of deep learning, audios are often converted into 2D representations and then processed using CV models [37], or directly processed as sequences using NLP models [74].
output = max((load(0x22a5a84,4) * load(0x7e1f54,4) + load(0x22a5a7c,4) * load(0x7e1f4c,4) + load(0x22a5a80,4) * ... + load(0x284dcd0,4) * load(0x7a9280,16) + load(0x284dcd4,4) * load(0x7a9300,16) +…)(c) Symbolic Constraint of TVM –O3

(a) Symbolic Constraint of Glow

(b) Symbolic Constraint of TVM –O0

(c) Symbolic Constraint of TVM –O3

Figure 7: Mostly consistent symbolic constraints extracted from vastly different binaries.

Table 7: GitHub repo commits investigation results. We report the #Commits in the past two years, #Commits related to CPU code generation, the average LOC per CPU commit, and #CPU commits with substantial changes (over 100 LOC).

<table>
<thead>
<tr>
<th>Compiler</th>
<th>#Commits</th>
<th>CPU Commits Avg. LOC</th>
<th>Substantial</th>
</tr>
</thead>
<tbody>
<tr>
<td>TVM</td>
<td>4,292</td>
<td>121</td>
<td>17</td>
</tr>
<tr>
<td>Glow</td>
<td>1,435</td>
<td>22</td>
<td>47</td>
</tr>
<tr>
<td>NNFusion</td>
<td>200</td>
<td>13</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 8: Categorize ONNX operators. #with Dims. denotes operators with dimensions to be recovered, #with Opt. denotes operators with compile-time optimizations, Workload denotes the amount of work required to cover new operators, and #Covered denotes operators that currently supported.

<table>
<thead>
<tr>
<th>Category</th>
<th>#Ops</th>
<th>#with Dims</th>
<th>#with Opt</th>
<th>Workload</th>
<th>#Covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element-wise Op</td>
<td>56</td>
<td>2</td>
<td>0</td>
<td>Low</td>
<td>2/2</td>
</tr>
<tr>
<td>Tensor Op</td>
<td>32</td>
<td>8</td>
<td>0</td>
<td>Low</td>
<td>6/8</td>
</tr>
<tr>
<td>Matrix Op</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Pooling</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>Low</td>
<td>5/9</td>
</tr>
<tr>
<td>Heavyweight</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>High</td>
<td>7/11</td>
</tr>
<tr>
<td>Normalization</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>Low</td>
<td>2/6</td>
</tr>
<tr>
<td>Transpose</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Random</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Others</td>
<td>33</td>
<td>0</td>
<td>0</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Total</td>
<td>163</td>
<td>36</td>
<td>11</td>
<td>NA</td>
<td>22/36*</td>
</tr>
</tbody>
</table>

* 22 of the 36 operators with dimensions that require specific patterns to recover are already covered in BTD. 4 of the remaining 8 operators can be covered with low effort, and the rest 4 (ConvInteger, MatMulInteger, QLinearConv, and QLinearMatMul) are rare in common models.

Second, we investigated commits related to CPU instruction generation in DL compilers’ GitHub repos over the past two years (April 2020 to April 2022). As in Table 7, while these compilers are frequently updated, most commits aim to increase support for alternative hardware and model formats, where CPU-related code has changed little. We manually reviewed all “substantial” commits, i.e., commits with more than 100 LOC changes, and confirmed that they do not change optimization strategies or binary code generation that may affect BTD. Besides, DL compilers heavily use parallel instruction extensions (e.g., SSE) to speed up model inference on CPUs. These extensions have been stable and unchanged over the long term. To answer RQ2, we again underline that BTD’s essential assumption is that symbolic constraints extracted from each DNN operator’s assembly function should be invariant across compilers and optimizations. Other features, such as function signatures, operator fusion, and optimization strategies, are independent of BTD’s core techniques and are also unlikely to be largely changed in the near future.

Answer to RQ2: BTD is robust enough against changes in current and prior versions of DL compilers. We anticipate that compiler changes are unlikely to affect the robustness of BTD in the near future.

7.2 RQ2: Robustness

BTD involves patterns during decompilation. **RQ2 arises: Is this method robust to survive frequent DL compiler implementation changes?** To answer this question, we evaluated BTD with prior versions of DL compilers released in the past two years (see Table 1). In short, after testing BTD using seven CV models and one NLP model, we report that BTD produces exactly identical results for different versions of compilers.

We interpret this highly encouraging result from two aspects. First, although BTD leverages patterns to recover dimensions and layouts of parameters, these patterns are based on semantics constraints, instead of syntax. Since DNN operators like Conv and ReLU are defined cleanly and rigorously, these semantics-level information are consistent across compiler implementation changes. Fig. 7 illustrates Conv constraints derived from Glow, TVM -O0, and TVM -O3. Although DL executables are drastically different, semantic constraints preserve mostly the same pattern. Notice that Fig. 7(a), an extra max exists due to Glow optimizations. When designing patterns, we deliberately pick components that co-exist across different constraints to recover dimensions and layouts.

Despite complex optimizations imposed by compilers, we find that our focused components are consistent and robust.

7.3 RQ3: Extensibility

As stated in Sec. 7.1, BTD can cover all operators used in the CV models from ONNX Zoo. This section measures BTD’s extensibility through the lens of all DNN operators supported by ONNX Zoo (RQ3). Note that not all operators are for CV models, and not all operators have been used in DNN models; some of them are rarely used in common models. Overall, while most techniques (i.e., operator inference and symbolic execution) used in BTD are independent of operator types, patterns described in Sec. 4.3.3 are designed for each complex operator to recover their parameters/dimensions. Supporting
a new operator may need new or existing patterns. Symbolic constraints are generally human readable, and we typically need several hours to design and validate a new pattern for operators without complex optimization, like BiasAdd and Pooling. Developing new patterns for complex operators like Conv may take days due to complex optimization strategies.

We classified all ONNX operators [6] to scope BTD’s applicability and the engineering effort required to extend BTD. Consider Table 8, where “Low” in Workload column represents hours of effort, and “High” represents several days of work. While ONNX has 163 DNN operators [6], most of them do not have dimensions to be recovered. Besides, our patterns can be reused with minor modifications to support currently uncovered tensor operators, pooling operators, and normalization operators. For heavyweight calculation (Conv, MatMul/FC, GRU, RNN, LSTM, and their variants), we have already covered 7 out of 11 operators. GRU, RNN, and LSTM can be covered because they are decomposed into sub-operators, including FC and element-wise operators, as explained in Sec. 7.1.7. Standard models rarely use these remaining operators, including ConvInteger, MatMulInteger, QLinearConv, and QLinearMatMul. BTD cannot handle these four operators for the time being, but we expect it will not be challenging to design patterns for these operators. Essentially, they are variants of Conv and MatMul operators. We leave support for these operators to our future work.

**Answer to RQ3:** Users experienced in DL models can spend reasonable effort to add support for new operators and models by modifying existing patterns in BTD.

### 7.4 RQ4: Error Fixing

This section clarifies how BTD performs error fixing (RQ4).

On one hand, with rules presented in Sec. 6, BTD can detect and automatically fix the errors exposed when decompiling the ResNet18 executable (compiled with TVM -O0). The recovered model specification, after fixing, are completely correct, as noted in Table 5. We detail the error fixing procedure as a case study over the ResNet18 executable in [9].

On the other hand, when errors can not be fixed automatically (ResNet compiled with TVM -O3), users are required to read the symbolic constraints and fix errors manually. Generally, fixing an error requires users to be familiar with both standard operators used in DL models and x86 assembly language. Nevertheless, even partially recovered models may boost attacks like query-based model extraction [71, 87].

**Answer to RQ4:** To cope with decompilation defects, BTD provides error detection & automated fixing mechanism, including a collection of rules derived from domain-specific knowledge and observations.

### 8 Discussion

**Downstream Applications & Countermeasures.** Previous model extraction attacks rely on repetitive queries or side channels to leak parts of DNNs. BTD, as a decompiler, reveals a new and practical attack surface to recover full DNNs when DNN executables are accessible. Our extended paper [9] will show that BTD can boost DNN attacks. In addition, legacy DNN executables can be inspected, hardened, and migrated to new platforms. To show the feasibility, we migrated decompiled x86 DNN executables onto GPUs. This step only requires to use different compiler options over our recovered DNN models.

DNNs may provide business advantages. Potential security concerns raised by BTD may be mitigated using obfuscation [50]; particularly, code obfuscation could likely impede DNN operator inference whereas data obfuscation may likely undermine our patterns over memory layouts.

**Cross-Platform.** As reviewed in Sec. 2, DL compiler can generate executables on various platforms. The core techniques of BTD are platform independent. We analyze the cross-platform extension of BTD from the following aspects. First, decompiling DNN executables on devices like hardware accelerators requires appropriate disassemblers. This demands vendor support and considerable engineering work. While certain GPU makers like Nvidia provides disassemblers [3], the architecture and ISA of such devices are only partially or not revealed, preventing migrating BTD to these devices. Intel recently released a dynamic instrumentation tool, GTPin [5], but it is immature and limited to Intel processor graphics. Without vendor support, it is extremely difficult, if not impossible, to implement disassemblers and dynamic instrumentors on our own for various devices.

Second, DL compilers produce distinct executables on GPUs and CPUs. For example, TVM creates a standalone DNN executable on CPU, but a runtime library, including detailed model information, and an OpenCL/CUDA executable on GPU. Glow has an immature support for OpenCL using JIT. In short, we see x86 CPU decompilation as more difficult because inputs are typically standalone executables.

### 9 Related Work

**Software Reverse Engineering.** Software decompilation has achieved primary success. Algorithms are proposed to improve decompiled C/C++ code, including refining type recovery [52,79], variable recovery [13,17], and control structure recovery [29,31]. ML accelerates decompilation [24,32,35,48]. Some decoders are commercially available [1,2,38]. In addition, recent works have designed decoders for Ethereum smart contracts [34,102]. We focus on decompiling DNN executables by addressing domain challenges to convert DNN executables to high-level models. We envision that BTD will
meet demands to comprehend, exploit, and harden real-world DNN executables.

**Model Extraction.** We have reviewed DL compilation techniques in Sec. 2. BTD enables a novel perspective to extract DNN models. As introduced in Sec. 3, current model extraction works mostly take “black-box” forms [68, 70, 71, 85], where adversaries can assemble a training dataset \((x, y)\) by continuously feeding inputs \(x\) to a target model and collecting its prediction outputs \(y\). The resulting training datasets can be used to train a local model. Side channels leaked during inference are also used for model extraction, including timing, cache, and power side channels [28,42,43,95,96,103]. BTD is orthogonal to these side channel-based methods. As noted in Sec. 2, BTD can recover full model information whereas they conduct partial recovery. We also notice reverse engineering efforts targeting image processing software [11,46,59]. These works use static analysis and heuristics to map (assembly) image processing code (e.g., blurring) to high-level operators. They analyze image processing software (e.g., Photoshop), not DNN models.

**10 Conclusion**

We presented BTD, a decompiler for x86 DNN executables. BTD recovers full DNN models from executables, including operator types, network topology, dimensions, and parameters. Our evaluation reports promising results by successfully decompiling and further recompiling executables compiled from popular DNN models using different DL compilers.

**Acknowledgments**

We thank the anonymous reviewers for their valuable comments. HKUST authors are supported in part by a RGC ECS grant under the contract 26206520. Lei Ma’s research is supported in part by the Canada First Research Excellence Fund as part of the University of Alberta’s Future Energy Systems research initiative, Canada CIFAR AI Chairs Program, Amii RAP program, the Natural Sciences and Engineering Research Council of Canada (NSERC No.RGPIN-2021-02549, No.RGPAS-2021-00034, No.DGECR-2021-00019), as well as JSPS KAKENHI Grant No.JP20H04168, No.JP21H04877, JST-Mirai Program Grant No.JPMJMI20B8.

**References**


[53] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. TPDS, 2020.


[78] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE SP*, 2010.


[96] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn dnn architectures. In *USENIX Sec’20*.


