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 Preliminary

Fig. 1(a) depicts DNN model compilation. DNN compilation can be divided into two phases [13], with each phase manipulates one or several intermediate representations (IR).

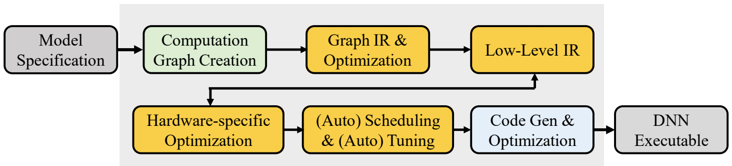
Computation Graph. DL compiler inputs are typically high-level model descriptions exported from DL frameworks like PyTorch [18]. DNN models are typically represented as computation graphs in DL frameworks. Fig. 1(b) shows a simple graph of a multilayer convolutional neural network (CNN). These graphs are usually high-level, with limited connections to hardware. DL frameworks export computation graphs often in ONNX format [1] 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 [5, 21].

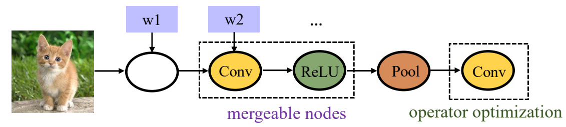
Backend: Low-Level IRs and Optimizations. Hardware-specific low-level IRs are generated from graph IRs. Instead of translating graph IRs directly into standard IRs like LLVM IR [12], low-level IRs are employed as an intermediary step for customized optimizations using prior knowledge of DL models and hardware characteristics. Graph IR operators can be converted into low-level linear algebra operators [21]. Such representations alleviate the hurdles of directly supporting many high-level operators on each hardware target.

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 [20] introduces a scheduling language with manual and automated schedule optimization primitives. Recent works explore launching auto scheduling and tuning to enhance optimization [3, 5, 6, 17, 25, 28, 29]. 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 to explore hardware-specific optimizations. For instance, Glow [21] can perform fine-grained loop-oriented optimizations in LLVM IR. DL compilers like TVM and Glow compile optimized IR code into standalone executables.



(a) DNN compilation pipeline.



(b) Sample DNN computation graph. DNN compiler frontend looks for holistic opt. chances like mergeable nodes, whereas backend explores efficient machine code for each operator.

Figure 1: The high-level workflow of DL compilation.

2 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. 3 details BTD's processes to recover each component.

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.

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 [7], and to date, decompiled C/C++ code mainly aids (human-based) analysis and comprehension, *not* recompilation. 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.

3 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.

BTD delivers practical decompilation based on the *invariant semantics* of DNN operators. 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. Besides, as invariant semantics reflect high-level information, e.g., operator types and dimensions, we can infer model abstractions accurately.

Fig. 2(a) depicts the BTD workflow. We first train a neural model to map assembly functions to DNN operators. Recent works perform representation learning by treating x86 opcodes as natural language tokens [8, 9, 14, 19, 26]. 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.

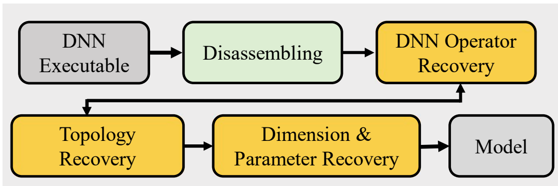
Given recovered DNN operators, we reconstruct the network topology using dynamic analysis. In this step, DNN operators are chained into a computation graph. Specifically, a DNN operator has a fixed number of inputs and outputs [2]. According to our observation, DL compilers compile DNN operators into assembly functions and pass inputs and outputs as memory pointers through function arguments. We use Intel Pin [15], 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 then use trace-based symbolic execution to extract operator semantics from assembly code and then recover dimensions and parameters with semantics-based patterns (Sec. **??**). 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.

Dimensions and parameters configure DNN operators. Fig. 2(b) shows representative cases. We define patterns over the extracted symbolic constraints, which enable recovering dimensions and parameter layouts. We then use Intel Pin to dump parameters to disk at runtime. With recovered dimensions and dumped parameters in data bytes, we can recover well-formed parameters.

4 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



(a) Workflow.

Type	Dimension	Parameter	Operators
I	NA	NA	ReLU; Sigmod; … Add; Sub; Negative; Sqrt; … ExpandDims; BatchFlatten; …
II	✓	NA	Pooling;
III	NA	✓	BiasAdd; Multiply; Divide; BN;
IV	✓	✓	Conv; FC; Embedding

(b) Four types of operators.

Figure 2: Decompilation workflow. Here "NA" in the "Dimension" column denotes an easy case where output dimension of an operator `O` equals to its input dimension and no other dimensions associated with `O`. We find that in non-trivial DNN, it is sufficient to decide `O`'s dimensions after propagating dimensions from other operators on the DNN computation graph.

ELF format on x86 platforms. We use LSTM for DNN operators inference in an "out-of-the-box" manner. BTD is evaluated by the USENIX Security'23 AE committee and awarded with *Available*, *Functional*, and *Reproduced* badges. We open source BTD at https://github.com/monkbai/DNN-decompiler.

Table 1: Compilers evaluated in our study.

Tool Name	Publication	Developer	Version (git commit)
TVM [5]	OSDI '18	Amazon	v0.7.0 v0.8.0 v0.9.dev
Glow [21]	arXiv	Facebook	2020 (07a82bd9fe97dfd) 2021 (97835cec670bd2f) 2022 (793fec7fb0269db)
NNFusion [16]	OSDI '20	Microsoft	v0.2 v0.3

5 Evaluation

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. Table 1 lists compilers we used. Table 2 lists all evaluated DNN models.

Overall, BTD can decompile DNN executables with negligible errors over all settings. Four dimension failures in ResNet18 (TVM -O0) are due to a Conv optimization, 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 (0.7%) parameters of an optimized Conv operator in ResNet18 (TVM -O3) due to its specially-optimized memory layout. In addition to that, all other models can be decompiled and recompiled into new models manifesting identical behavior.

6 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.

References

[1] Onnx. https://onnx.ai/, 2021.

[2] ONNX Operators . https://github.com/onnx/onnx/blob/main/docs/Operators.md, 2022.

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

[4] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *TACL*, 5:135–146, 2017.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX OSDI*, pages 578–594, 2018.

[6] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *NeurIPS*, 31:3389–3400, 2018.

[7] Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Softw. Pract. Exper.*, 25(7):811–829, July 1995.

[8] S. H. Ding, B. M. Fung, and P. Charland. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *IEEE S&P*, 2019.

[9] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. DeepBinDiff: Learning program-wide code representations for binary diffing. In *NDSS*, 2020.

Table 2: Statistics of DNN models and their compiled executables evaluated in our study.

Model	#Parameters	#Operators	TVM -O0		TVM -O3		Glow -O3	
			Avg. #Inst.	Avg. #Func.	Avg. #Inst.	Avg. #Func.	Avg. #Inst.	Avg. #Func.
Resnet18 [10]	11,703,912	69	49,762	281	61,002	204	11,108	39
VGG16 [22]	138,357,544	41	40,205	215	41,750	185	5,729	33
FastText [4]	2,500,101	3	9,867	142	7,477	131	405	14
Inception [23]	6,998,552	105	121,481	615	74,992	356	30,452	112
Shufflenet [27]	2,294,784	152	56,147	407	34,637	228	33,537	59
Mobilenet [11]	3,487,816	89	69,903	363	46,214	228	37,331	52
Efficientnet [24]	12,966,032	216	89,772	546	49,285	244	13,749	67

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[11] 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.

[12] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[13] 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.

[14] Xuezixiang Li, Qu Yu, and Heng Yin. PalmTree: Learning an assembly language model for instruction embedding. In *ACM CCS*, 2021.

[15] 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. In *PLDI*, 2005.

[16] 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 OSDI*, pages 881–897, 2020.

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

[18] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

[19] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. TREX: Learning execution semantics from micro-traces for binary similarity. arXiv, 2021.

[20] 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*, 2013.

[21] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint*, 2018.

[22] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[23] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *CVPR*, pages 2818–2826, 2016.

[24] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*, pages 6105–6114. PMLR, 2019.

[25] 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.

[26] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. Order matters: Semantic-aware neural networks for binary code similarity detection. AAAI, 2020.

[27] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *CVPR*, 2018.

[28] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX OSDI*, pages 863–879, 2020.

[29] 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 *ASPLOS*, 2020.