black hat USA 2023

AUGUST 9-10, 2023

BRIEFINGS

BTD: Unleashing the Power of Decompilation for x86 Deep Neural Network Executables

Zhibo Liu, Yuanyuan Yuan, Xiaofei Xie, Tianxiang Li, Wenqiang Li, Shuai Wang

Outline

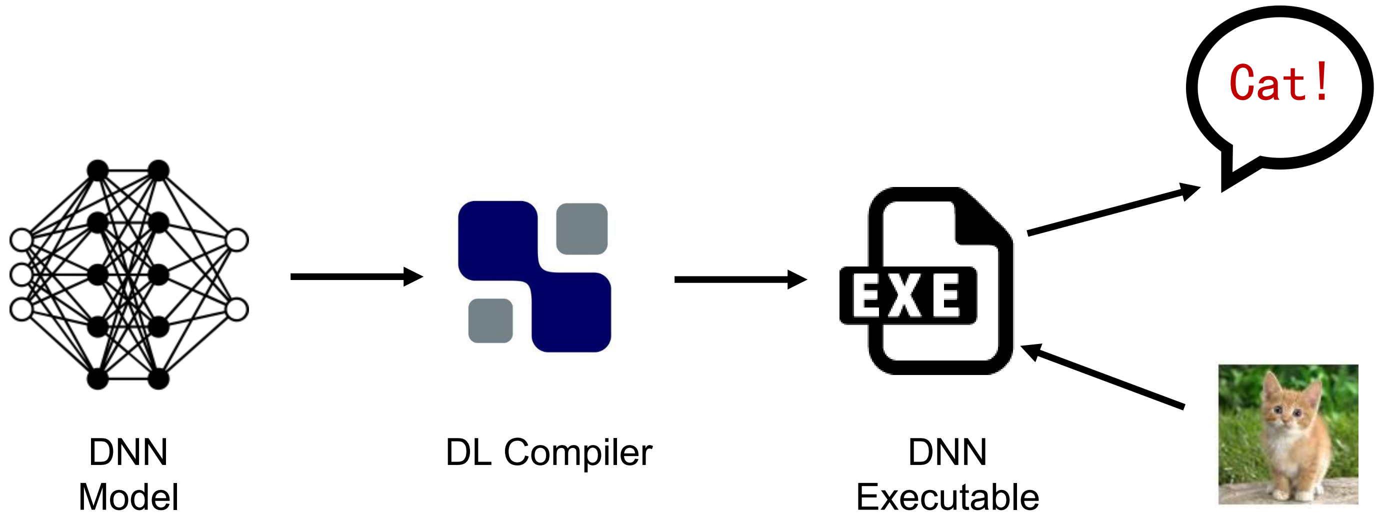
- Background
- Motivation
- Related Work
- Decompiling DNN executables
- Evaluation

Outline

- Background ◁
- Motivation
- Related Work
- Decompiling DNN executables
- Evaluation

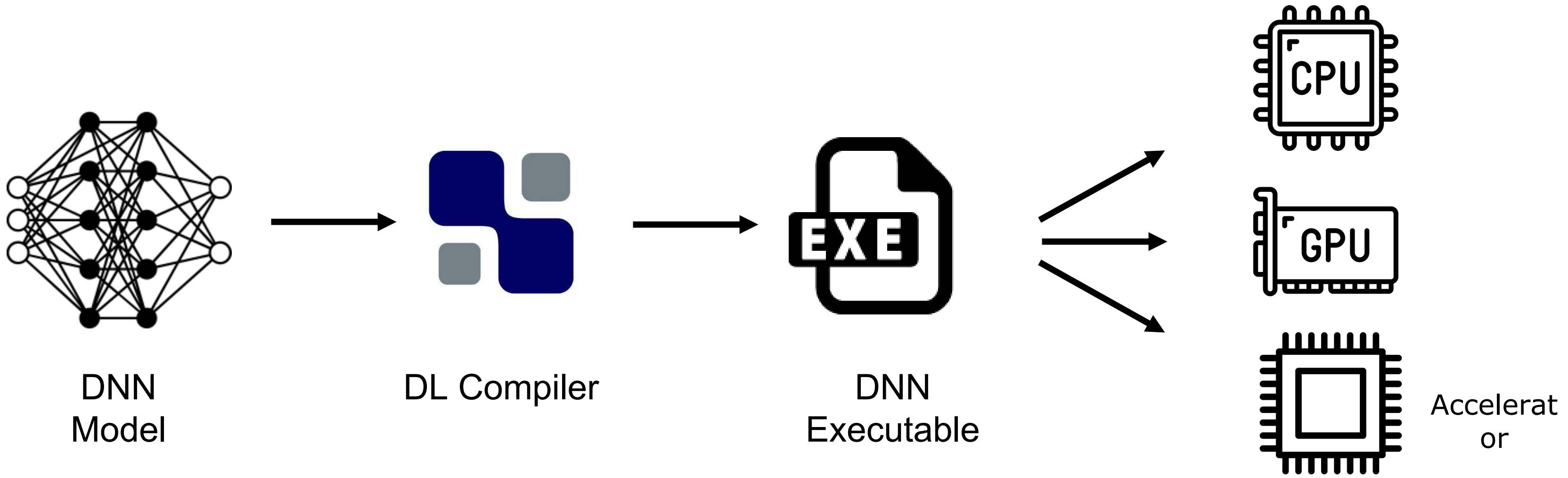
DNN Executable

- What is DNN executable?
  - Output of deep learning compilers.
  - Performing the DNN model inference at runtime.
  - In standalone binary format.



DNN Executable

- Why we need DNN compilation/executable?
    - To fully leverage low-level hardware primitives for fast model inference.
    - To deploy DNN models on heterogeneous hardware devices.



DL Compiler

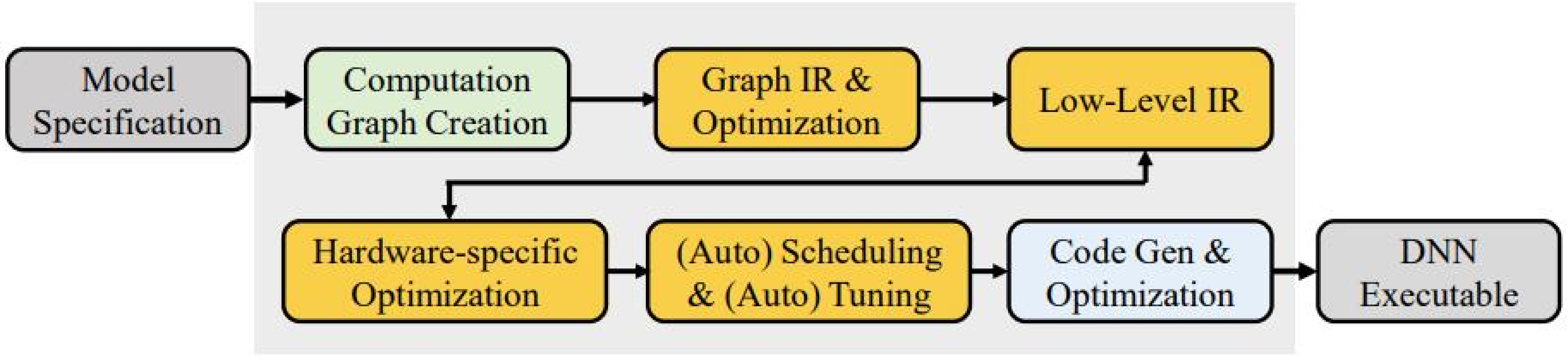
- Compile high-level models into binary code.

- Can optimize code utilizing domain-specific hardware features (e.g., Intel SIMD) and abstractions.

- Further squeeze (low-power) hardware performance potential.

DL Compiler

- Compilation process typically involves multiple optimization cycles.



DNN compilation pipeline.

DL Compiler

- Many resources from academia and industry have been devoted to this field.

Support from industry	DL compilers	Academic output
APACHE SOFTWARE FOUNDATION →	tvm →	OSDI'18
Meta →	GLOW →	arXiv
Microsoft →	NNFusion →	OSDI'20

Real-World Applications

- Low-power processors suppliers (e.g., NXP, Qualcomm) are incorporating DL compilers into their applications

- Cloud service providers (e.g., Amazon and Google) include DL compilers into their DL services to boost performance

aws

Amazon SageMaker Neo uses Apache TVM and partner-provided compilers and acceleration libraries to deliver the best available performance for a given model and hardware target. AWS contributes the compiler code to the Apache TVM project and the runtime code to the Neo-AI open-source project, under the Apache Software License, to enable processor vendors and device makers to innovate rapidly on a common compact runtime.

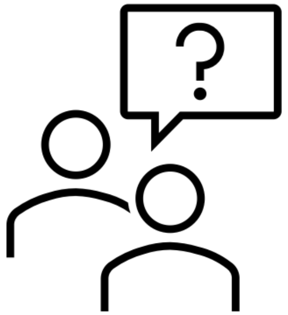
https://aws.amazon.com/sagemaker/neo/

Outline

- Background
- Motivation ◁
- Related Work
- Decompiling DNN executables
- Evaluation

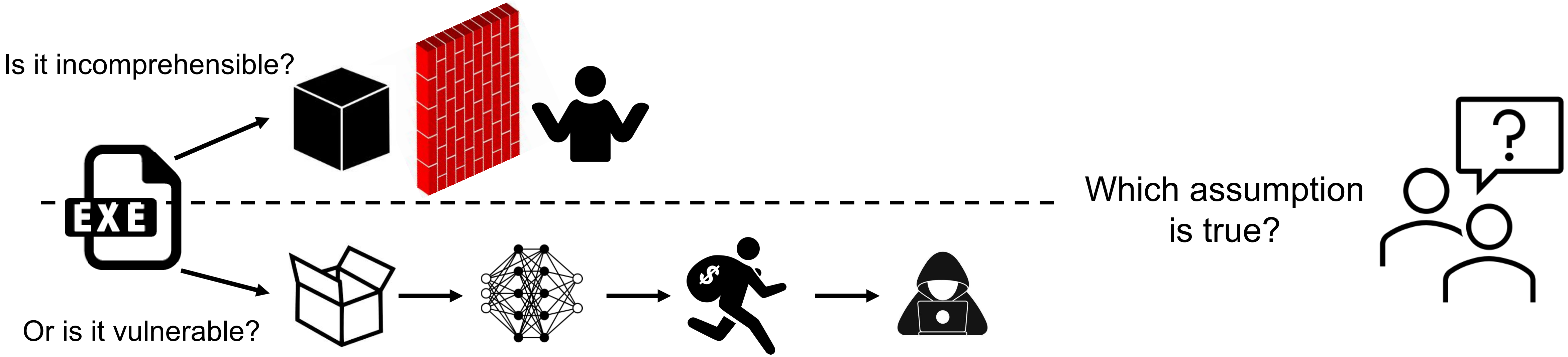
Problem

- Currently, DL compiler community mainly focuses on performance
- Our questions:
  - What is the difference between DNN executables and traditional software?
  - How should we safely use DL compilers?
  - What are the potential security risks of using DL compilers?



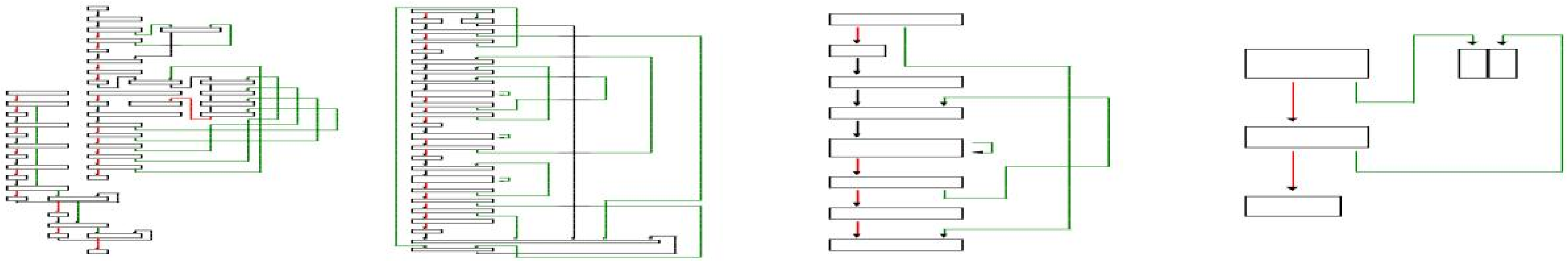
Problem

- Specifically, should we view a DNN executable as a black-box or white-box?



Challenges

- The traditional software reverse engineering techniques can hardly tackle DNN executables.



(a) Glow (b) TVM -O0 (c) TVM -O3 (d) NNFusion

Figure 2: Compare CFGs of a Conv operator in VGG16 compiled by different DL compilers. TVM refers to enabling no optimization as "-O0" while enabling full optimizations as "-O3". Glow and NNFusion by default apply full optimizations.

Challenges

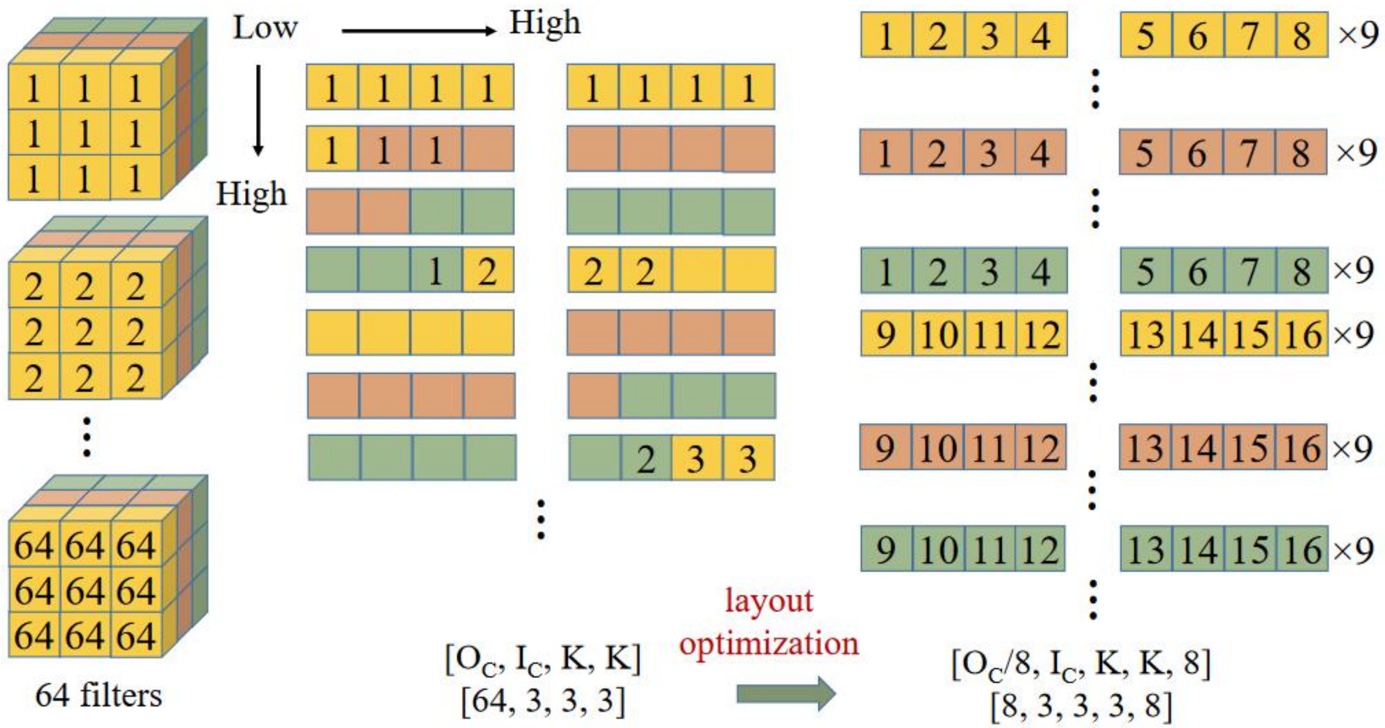
- Complex data flow during DNN inference.

```
v52 = (__m128)*(unsigned int *)(v7 + 4 * v29 + 1024);
v53 = _mm_shuffle_ps(v52, v52, 0);
v159 = _mm_add_ps(_mm_mul_ps(*(__m128 *)(v8 + 4 * v42), v53), v159);
v160 = _mm_add_ps(_mm_mul_ps(*(__m128 *)(v8 + 4 * v45), v53), v160);
v161 = _mm_add_ps(_mm_mul_ps(*(__m128 *)(v8 + 4 * v46), v53), v161);
v162 = _mm_add_ps(_mm_mul_ps(*(__m128 *)(v8 + 4 * v47), v53), v162);
v163 = _mm_add_ps(_mm_mul_ps(*(__m128 *)(v8 + 4 * v48), v53), v163);
v164 = _mm_add_ps(_mm_mul_ps(*(__m128 *)(v8 + 4 * v49), v53), v164);
v165 = _mm_add_ps(_mm_mul_ps(*(__m128 *)(v8 + 4 * v50), v53), v165);
v166 = _mm_add_ps(_mm_mul_ps(*(__m128 *)(v8 + 4 * v51), v53), v166);
v54 = (__m128)*(unsigned int *)(v7 + 4 * v29 + 1536);
v55 = _mm_shuffle_ps(v54, v54, 0);
v167 = _mm_add_ps(_mm_mul_ps(*(__m128 *)(v8 + 4 * v42), v55), v167);
v168 = _mm_add_ps(_mm_mul_ps(*(__m128 *)(v8 + 4 * v45), v55), v168);
v169 = _mm_add_ps(_mm_mul_ps(*(__m128 *)(v8 + 4 * v46), v55), v169);
v170 = _mm_add_ps(_mm_mul_ps(*(__m128 *)(v8 + 4 * v47), v55), v170);
v171 = _mm_add_ps(_mm_mul_ps(*(__m128 *)(v8 + 4 * v48), v55), v171);
v172 = _mm_add_ps(_mm_mul_ps(*(__m128 *)(v8 + 4 * v49), v55), v172);
v173 = _mm_add_ps(_mm_mul_ps(*(__m128 *)(v8 + 4 * v50), v55), v173);
v174 = _mm_add_ps(_mm_mul_ps(*(__m128 *)(v8 + 4 * v51), v55), v174);
v56 = (__m128)*(unsigned int *)(v7 + 4 * v29 + 2048);
v57 = _mm_shuffle_ps(v56, v56, 0);
```

Decompile with IDA

Challenges

- Hardware-aware optimizations during compilation.
  - memory layout optimization → better memory locality & compatible with SIMD



Outline

- Background
- Motivation
- Related Work ◁
- Decompiling DNN executables
- Evaluation

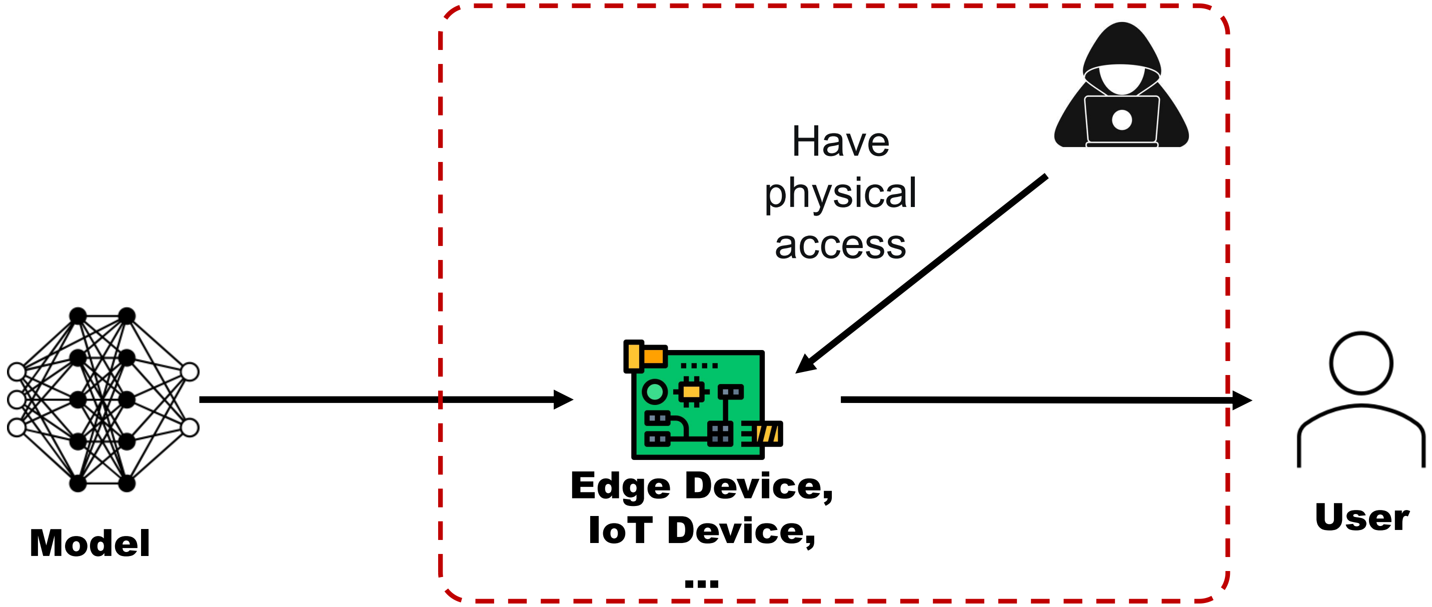
Related

- Attacking DNN models is not new
- Previous works mainly focus on DL frameworks (e.g., PyTorch and TensorFlow):
  - Cache side channel
  - Power side channel
  - Electromagnetic emanations (EM) side channel
  - Bus snooping

  ……

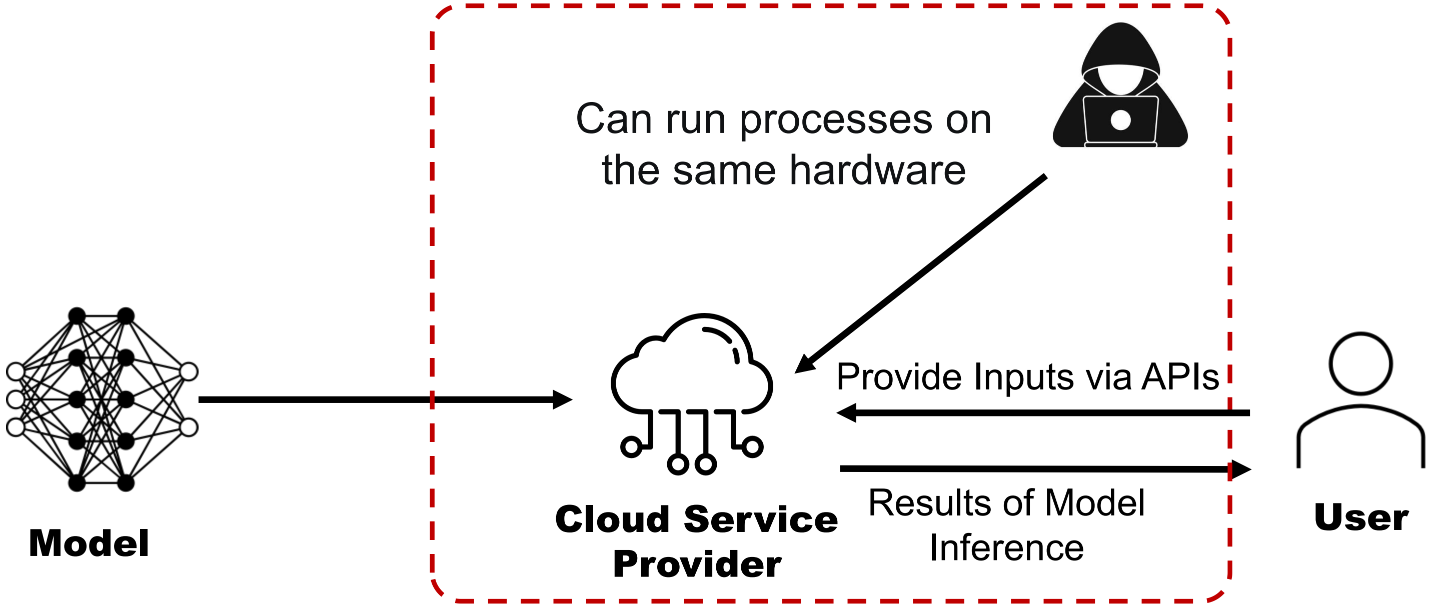
Threat Model

- Physical access



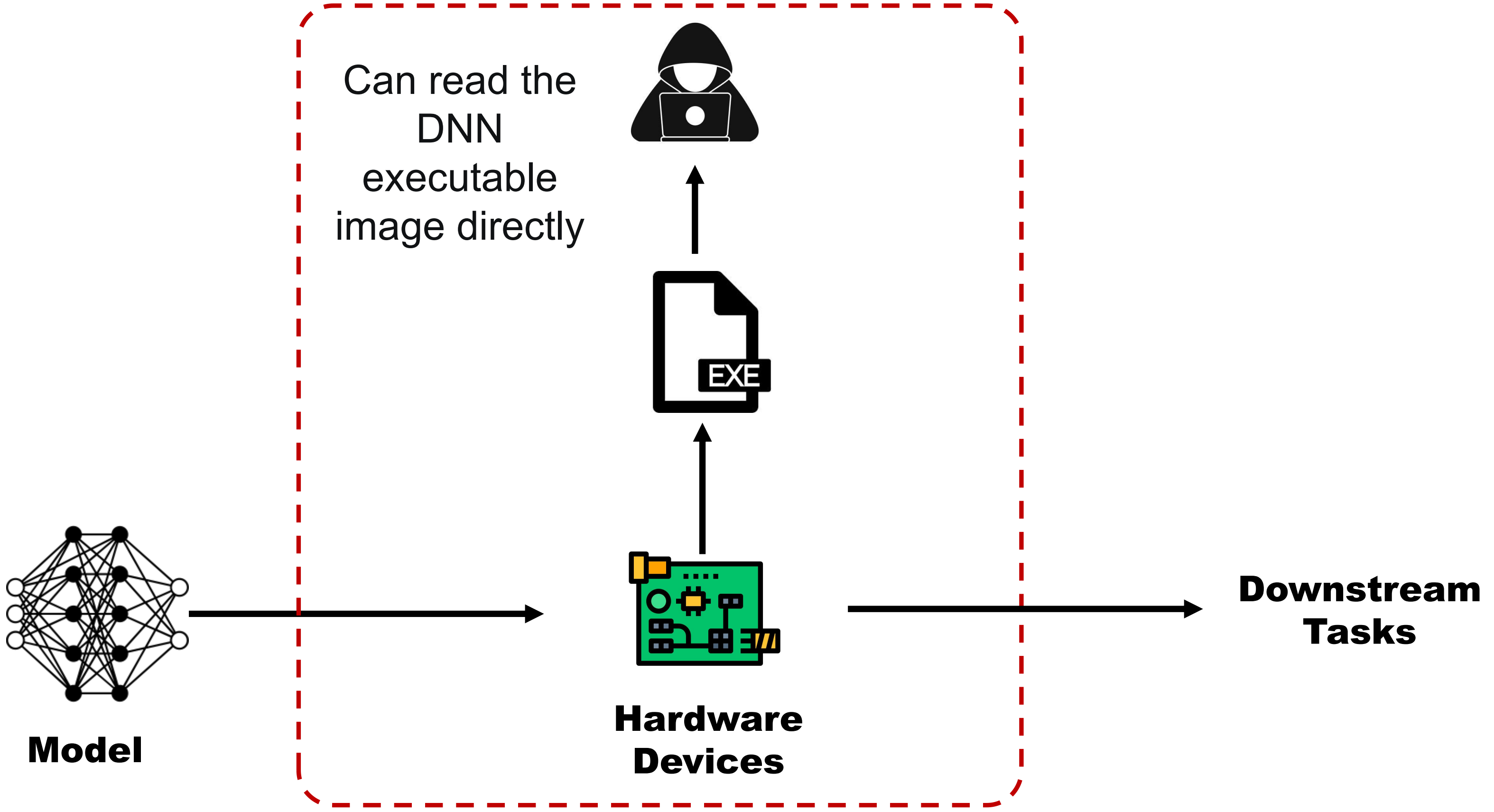
Threat Model

- Remote access



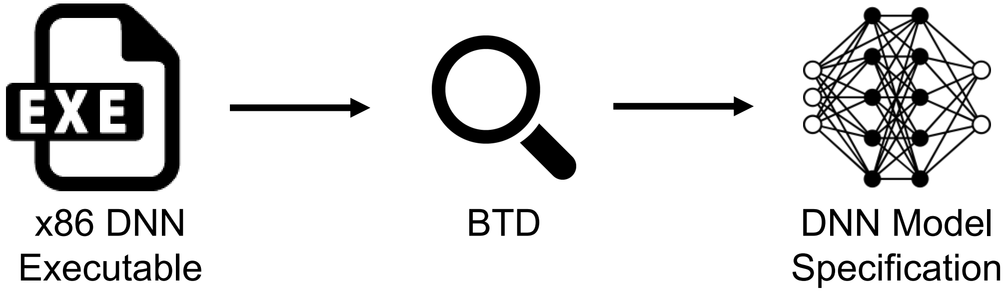
Threat Model

- Our assumption: binary access



Our Work

- We propose BTD (Bin-To-DNN), the first DNN executable decompiler.



Outline

- Background
- Motivation
- Related Work
- Decompiling DNN executables ◁
- Evaluation

Observation

- Differences between DNN executables and general software:
  - Complex data flow (millions of floating-point multiplications in DNN exe)
    → difficult to summarize
  - But only one execution path!
    → no path explosion

Give us an opportunity to summarize the semantics from low-level binary code (i.e., floating-point arithmetic)

Observation

- Moreover

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.

E.g., mathematical definition of Conv:

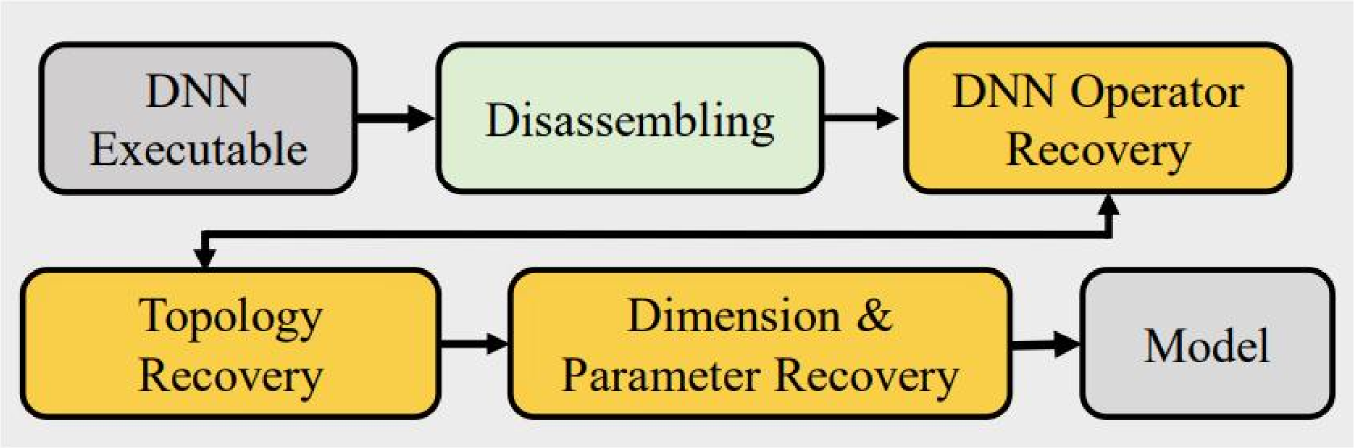
$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

Idea

- Summarize the invariant operator semantics with trace-based symbolic execution

Workflow

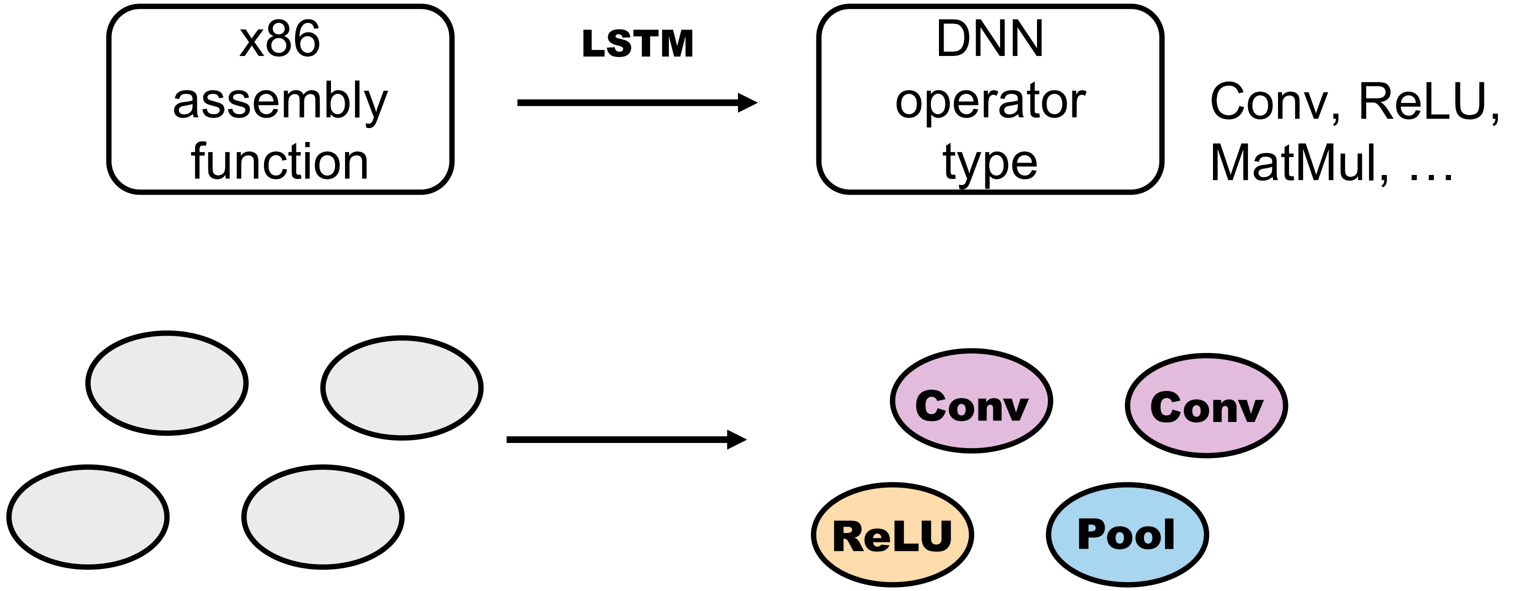
- BTD consists of 3 steps: operator recovery, topology recovery, dimension & parameter recovery.



- BTD is able to recover full model specification (including operators, topologies, dimensions, and parameters) from DNN executable.

Step 1: Operator Recovery

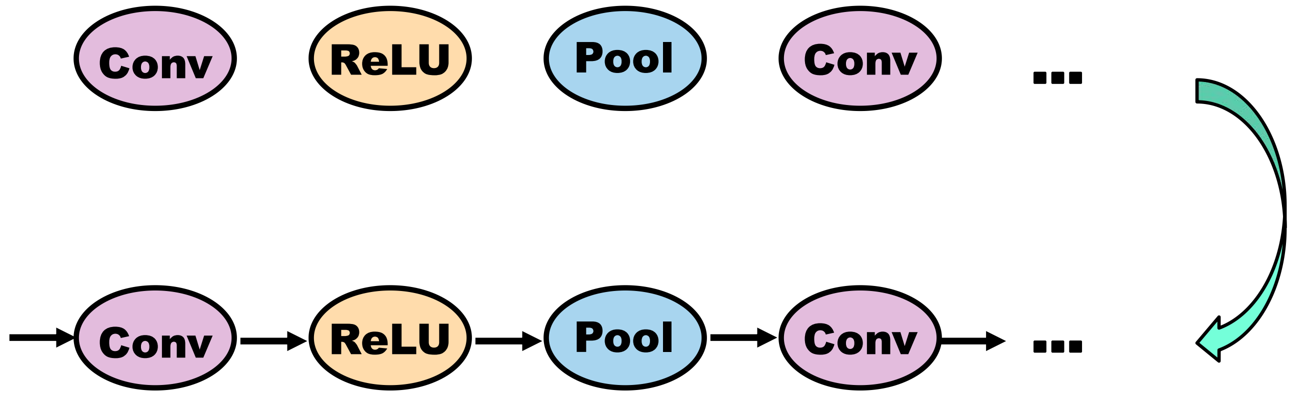
- We train a LSTM model to map assembly functions to DNN operators.
  - Treat x86 opcodes as language tokens.
  - Segment x86 opcodes using Byte Pair Encoding (BPE).



Step 2: Topology Recovery

- DL compilers compile DNN operators into assembly functions and pass inputs and outputs as memory pointers through function arguments.

- We hook every call site to record the memory address, and chain operators into computation graph.



Step 3: Dimension & Parameter

- We launch trace-based symbolic execution (SE) to infer dimensions and localize parameters for DNN operators
- We filter trace with taint analysis to only keep parts related to operator output.

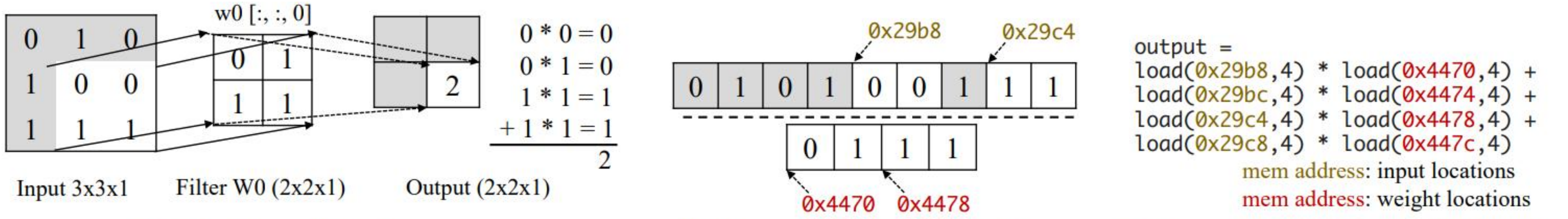
assembly trace $\xrightarrow{\text{SE}}$ symbolic constraints

Human readable operator semantics

Step 3: Dimension & Parameter

- The gap (offset) between inputs implies the dimension information.

(0x29c4-0x29b8) / sizeof(float) = 3
each row has 3 float values.



(a) One Convolution Operation (b) Memory Layout and Addresses (c) Corresponding Symbolic Formula

Step 3: Dimension & Parameter

- Symbolic constraints extracted from vastly different binaries are mostly consistent.

```
output =
max(
 (load(0x22a5a84,4) * load(0x7e1f54,4) +
  load(0x22a5a7c,4) * load(0x7e1f4c,4) +
  load(0x22a5a80,4) * load(0x7e1f50,4) +
  load(0x22a5a78,4) * load(0x7e1f48,4) +
  ...),
0)
```

(a) Symbolic Constraint of Glow

```
output =
( 0 +
  load(0x29cfe98,4) * load(0x293cd60,16) +
  load(0x29cfe9c,4) * load(0x293cde0,16) +
  load(0x29cfea0,4) * load(0x293ce60,16) +
  load(0x29cfea4,4) * load(0x293cee0,16) +
...)
```

(b) Symbolic Constraint of TVM –O0

```
output =
( 0 +
  load(0x284dcc8,4) * load(0x7a9180,16) +
  load(0x284dccc,4) * load(0x7a9200,16) +
  load(0x284dcd0,4) * load(0x7a9280,16) +
  load(0x284dcd4,4) * load(0x7a9300,16) +
...)
```

(c) Symbolic Constraint of TVM –O3

mem address: input locations
mem address: weight locations

Step 3: Dimension & Parameter

- We infer operator dimensions (e.g., kernel size, #input channels, #output channels, stride) from extracted symbolic constraints.

- Then instrument the DNN executable to dump parameters (e.g., weights, biases) during execution.

- With all extracted information (i.e., types, topology, dimensions, and parameters), we can rebuild a new model showing identical behavior with the original model.

Implementation

- BTD is open available at: https://github.com/monkbai/DNN-decompiler

- BTD passed the artifact evaluation of USENIX Security With Available, Functional, Reproduced badges

Outline

- Background
- Motivation
- Related Work
- Decompiling DNN executables
- Evaluation ◁

Evaluation

- 8 version of 3 state-of-the-art, production level DL compilers

Table 1: Compilers evaluated in our study.

Tool Name	Publication	Developer	Version (git commit)
TVM [20]	OSDI '18	Amazon	v0.7.0 v0.8.0 v0.9.dev
Glow [77]	arXiv	Facebook	2020 (07a82bd9fe97dfd) 2021 (97835cec670bd2f) 2022 (793fec7fb0269db)
NNFusion [58]	OSDI '20	Microsoft	v0.2 v0.3

Evaluation

- 7 models cover all operators used in the CV models from ONNX Zoo https://github.com/onnx/models

- real-world image classification models trained on ImageNet

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 [36]	11,703,912	69	49,762	281	61,002	204	11,108	39
VGG16 [81]	138,357,544	41	40,205	215	41,750	185	5,729	33
FastText [18]	2,500,101	3	9,867	142	7,477	131	405	14
Inception [83]	6,998,552	105	121,481	615	74,992	356	30,452	112
Shufflenet [99]	2,294,784	152	56,147	407	34,637	228	33,537	59
Mobilenet [41]	3,487,816	89	69,903	363	46,214	228	37,331	52
Efficientnet [84]	12,966,032	216	89,772	546	49,285	244	13,749	67

Results

- Step 1: DNN operator inference

Table 3: Average accuracy of DNN operator inference.

Model	Glow			TVM -O0			TVM -O3		
	2020	2021	2022	v0.7	v0.8	v0.9.dev	v0.7	v0.8	v0.9.dev
ResNet18	100%	100%	100%	99.79%	99.84%	100%	98.15%	99.06%	99.69%
VGG16	100%	100%	100%	99.95%	99.79%	99.57%	99.75%	100%	100%
Inception	100%	100%	100%	99.98%	99.88%	99.98%	100%	100%	100%
ShuffleNet	100%	100%	100%	99.96%	99.82%	100%	99.62%	99.71%	99.31%
MobileNet	100%	100%	100%	99.35%	99.46%	99.40%	99.80%	100%	100%
EfficientNet	100%	100%	100%	99.65%	99.68%	99.59%	99.81%	99.91%	100%

- Errors can be eliminated by post-checking symbolic constraints, e.g.,
  - predicted types → Conv+ReLU
  - but no max operation in constraints
  - remove ReLU label and get the correct Conv type

Results

- Step 3: Parameter layout/dimension inference

Table 10: Parameter/dimension inference. Lines 2–8 report each executable's total #dimensions, correctly-inferred dimensions, and accuracy rate for dimension inference. Lines 9–15 report total #parameters and accuracy rate for parameter inference. Different versions of the same compiler produce the same results, therefore we merge their columns.

Model	Glow (2020, 2021, 2022)	TVM -O0 (v0.7, v0.8, v0.9.dev)	TVM -O3 (v0.7, v0.8, v0.9.dev)
ResNet18	65/65/100%	51/47/**92.15%**	78/78/100%
VGG16	54/54/100%	59/59/100%	52/52/100%
FastText	7/7/100%	7/7/100%	7/7/100%
Inception	235/235/100%	223/223/100%	222/222/100%
ShuffleNet	82/82/100%	71/71/100%	71/71/100%
MobileNet	124/124/100%	144/144/100%	125/125/100%
EfficientNet	133/133/100%	133/133/100%	132/132/100%
ResNet18	11,684,712/100%	11,703,912/**99.37%**	11,684,712/**99.37%**
VGG16	138,357,544/100%	138,357,544/100%	138,357,544/100%
FastText	2,500,101/100%	2,500,101/100%	2,500,101/100%
Inception	6,998,552/100%	6,998,552/100%	6,998,552/100%
ShuffleNet	2,270,514/100%	2,294,784/100%	2,270,514/100%
MobileNet	3,487,816/100%	3,487,816/100%	3,487,816/100%
EfficientNet	12,950,384/100%	12,966,032/100%	12,950,384/100%

- BTD fails on two cases because of DL compiler optimizations
  (details in our paper)

Results

- Overall, BTD is able to extract functional models in most cases.

Table 11: Recompilation. "NA" means that some errors in DNN models are not fixed, and thus the rebuilt models manifest inconsistent behavior.

Model	Glow (2020, 2021, 2022)	TVM -O0 (v0.7, v0.8, v0.9.dev)	TVM -O3 (v0.7, v0.8, v0.9.dev)
ResNet18	100%	100% (with fixing)	NA → 100%
VGG16	100%	100%	100%
FastText	100%	100%	100%
Inception	100%	100%	100%
ShuffleNet	100%	100%	100%
MobileNet	100%	100%	100%
EfficientNet	100%	100%	100%

- Thus, we can enable white-box attacks (e.g., Adversarial Example, Knowledge Stealing) on a black-box, obscure DNN executable!

Example

- We can use DeepInversion (CVPR'20) to attack a ResNet18 executable decompiled with BTD.

Synthesized Images

- The results are the same as attacking the original model

Thanks!

Q&A

- BTD: https://github.com/monkbai/DNN-decompiler