Decompling x86 Deep Neural Network Executables

Zhibo Liu, Yuanyuan Yuan, Shuai Wang
The Hong Kong University of Science and Technology

Xiaofei Xie
Singapore Management University

Lei Ma
University of Alberta
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

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

Support from industry → DL compilers → Academic output

- Apache
- Meta
- Microsoft

- tvm → OSDI’18
- Glow → arXiv
- NNFusion → OSDI’20
Problem

• Currently, DL compiler community mainly focuses on performance

• Our questions:
  • What is the difference between DNN exe and traditional exe?
  • Can we do reverse engineering on DNN executable?
Problem

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

Is it incomprehensible?

Or is it vulnerable?

Which assumption is true?
Challenges

- The traditional software reverse engineering techniques are unable to tackle DNN executables.

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

Decompiled with IDA
Challenges

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

Weights of a Conv

\[
\begin{align*}
&\quad [O_c, I_c, K, K] \\
&[64, 3, 3, 3] \\
&\quad [O_c/8, I_c, K, K, 8] \\
&[8, 3, 3, 3, 8]
\end{align*}
\]
Our Work

• The traditional software reverse engineering techniques are unable to tackle DNN executables.

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

- Binary access

With pre-trained parameters inside

Can read the DNN executable image directly

Hardware Devices

Downstream Tasks

- Watch
- Speaker
- Cleaner
Observation

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:

\[
\text{out}(N_i, C_{out_j}) = \text{bias}(C_{out_j}) + \sum_{k=0}^{C_{in}-1} \text{weight}(C_{out_j}, k) \ast \text{input}(N_i, k)
\]

Semantics of different implementation should be consistent!
Observation

• Differences between DNN executables and general software
  ➢ overwhelming arithmetic operations
    → hard to understand
  ➢ only one valid execution path!
    → no path explosion problem
    → get high-level semantics with symbolic execution!

• Give us an opportunity to summarize the semantics from low-level binary code
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: DNN Operator Recovery

• We train an LSTM model to map assembly functions to DNN operators.
  • Treat x86 opcodes as language tokens.
  • Segment x86 opcodes using Byte Pair Encoding (BPE).
  • Multiclass classification task
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.

```
Conv → ReLU → Pool → Conv → ... → Conv → ReLU → Pool → Conv → ...
```
Step 3: Dimension & Parameter Recovery

• Idea: we launch trace-based **symbolic execution** (SE) to infer dimensions and localize parameters for DNN operators.

---

**Diagram:**

- **DNN exe** → **assembly trace** → **SE** → **symbolic constraints** → **heuristics** → **dimensions**

---

**Examples:**

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

- **Input:** 3x3x1
- **Filter W0:** (2x2x1)
- **Output:** (2x2x1)

- **Symbols:**
  - `w0`: weights
  - `0x29b8`
  - `0x29c4`
  - `0x4470`
  - `0x4478`

- **Formulas:**
  - `output = load(0x29b8, 4) * load(0x4470, 4) + load(0x29bc, 4) * load(0x4474, 4) + load(0x29c4, 4) * load(0x4478, 4) + load(0x29c8, 4) * load(0x447c, 4)`

- **Concrete Value:**
  - `0111`
Step 3: Dimension & Parameter Recovery

- Symbolic constraints extracted from vastly different binaries are mostly consistent.
  - Our (symbolic constraint-based) heuristics are general and cross-compilers

```
(a) Symbolic Constraint of Glow
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)

(b) Symbolic Constraint of TVM –O0
output =
( 0 +
  load(0x284dcd8,4) * load(0x7a9180,16) +
  load(0x284dcd0,4) * load(0x7a9200,16) +
  load(0x284dcd4,4) * load(0x7a9280,16) +
  load(0x284dcd8,4) * load(0x7a9300,16) +
  ...)
mem address: input locations
mem address: weight locations

(c) Symbolic Constraint of TVM –O3
```
Step 3: Dimension & Parameter Recovery

• We infer operator dimensions (e.g., kernel size, #input channels, #output channels, stride) from extracted symbolic constraints with a set of heuristics.

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

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

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

<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 (07a82bd9fe97dfd)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2021 (97835cec670bd2f)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2022 (793fec7fb0269db)</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>
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
Results

• Step 1: DNN operator inference

Table 3: Average accuracy of DNN operator inference.

<table>
<thead>
<tr>
<th>Model</th>
<th>Glow 2020</th>
<th>Glow 2021</th>
<th>Glow 2022</th>
<th>TVM -O0 v0.7</th>
<th>TVM -O0 v0.8</th>
<th>TVM -O0 v0.9.dev</th>
<th>TVM -O3 v0.7</th>
<th>TVM -O3 v0.8</th>
<th>TVM -O3 v0.9.dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet18</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>99.79%</td>
<td>99.84%</td>
<td>100%</td>
<td>98.15%</td>
<td>99.06%</td>
<td>99.69%</td>
</tr>
<tr>
<td>VGG16</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>99.95%</td>
<td>99.79%</td>
<td>99.57%</td>
<td>99.75%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Inception</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>99.98%</td>
<td>99.88%</td>
<td>99.98%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>ShuffleNet</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>99.96%</td>
<td>99.82%</td>
<td>100%</td>
<td>99.62%</td>
<td>99.71%</td>
<td>99.31%</td>
</tr>
<tr>
<td>MobileNet</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>99.35%</td>
<td>99.46%</td>
<td>99.40%</td>
<td>99.80%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>EfficientNet</td>
<td>100%</td>
<td>100%</td>
<td>100%</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>
Results

• Step 3:
  • Parameter layout/dimension inference.

• BTD fails on two cases
  • Because of DL compiler optimizations
  • (details in our paper)
Results

• BTD is able to extract functional models in most cases.

<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>
<tr>
<td>VGG16</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>FastText</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Inception</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>ShuffleNet</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>MobileNet</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>EfficientNet</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>

• Thus, we can enable white-box attacks (e.g., adversarial example) on a black-box, obscure DNN executable.
Implement

• BTD is released at: https://github.com/monkbai/DNN-decompiler
  • With a demo docker image

• With badges Available, Functional, Reproduced
Takeaways

• It is hard to reverse DNN executables with existing techniques due to complex control/data flow.

• There is only one execution path, giving us an opportunity to summarize the semantics with symbolic execution.

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

Q&A

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