Decompiling x86 Deep Neural Network Executables Supplementary Document

The the best of our knowledge, BTD is the first decompiler that decompiles x86 DNN executables compiled by DL compilers. From a holistic view, reverse engineering tools are unavoidably sensitive to the underlying platforms and compilation tool chains. While this should *not* undermine the research merit, practical reverse engineering tools, including BTD, often come with a number of heuristics and assumptions [1, 7, 6, 2, 8]. Hence, in addition to our submitted main paper, this Appendix paints a more complete picture of BTD to discuss other tasks need to be addressed, which involves new heuristics.

1. Recover Parameters & Dimensions of Other DNN Operators

Table 1: Configurations required to infer DNN operator parameters from corresponding assembly function arguments compiled by TVM.

Operator	Configuration
Conv	in, weights, out
Conv (fused)	in, weights, biases, out
Dense	in, weights, out
Dense (fused)	in, weights, biases, out
BiasAdd	in, biases, out
AveragePooling	in, out
MaxPooling	in, out
Sqrt	in, out
ReLU	in, out
Divide	in_1, in_2, out
Multiply	in_1, in_2, out
Negative	in, out
Flatten	in, out
Reshape	in, out
Transpose	in, out
Transform	in, out

Table 2: Configurations required to infer DNN operator parameters from corresponding assembly function arguments compiled by Glow.

Operator	Configuration
Conv	out, in, weights, biases
ConvDKKC8	out, in, weights, biases
Matmul	out, in, weights
MaxPooling	in, out
AveragePooling	in, out
Relu	in/out
BatchAdd	out, in, biases
Transpose	in, out, dims
AddReLU	in_1/out, in_2

As has clarified in the main paper, the calling convention of DNN executables have eased the task of localizing inputs and parameters for each DNN operator: we use Intel Pin [3] to hook callsites of assembly functions belonging to the DNN operators, and log the starting addresses of parameters and inputs during runtime.

With respect to different DNN operators, we pre-define configurations to describe their assembly function prototypes, particularly the meaning of each function parameter. Table 1 and Table 2 report the full configurations in BTD. Table 1 describes the assembly function prototypes in TVM-emitted executables. For instance, the configuration of Conv indicates that there are three arguments in its compiled assembly function, where the 1st argument is a pointer toward Conv inputs, the 2nd argument points to Conv parameters (i.e., weights), and the 3rd argument points to Conv outputs. In optimized code (TVM -O3), Conv operators are often fused with following operators which contribute "Bias". As a result, the assembly function of Conv accepts two pointers toward the memory regions of weights and biases, respectively. Similarly, Table 2 reports the assembly function prototypes in Glow-emitted executables. Note that some optimized Conv operators are renamed as ConvDKKC8 by Glow. AddReLU has two inputs, where one input (in_1) shares the memory region with the output.

2. Parameters & Dimensions Recovery

BTD can recover dimensions and parameters of commonly-used DNN operators. We have demonstrated the comprehensiveness of BTD, by showing that three tested DNN models contain nearly all DNN operators that exist in all image classification models provided by ONNX Zoo [4].

General Procedure. Due to the limited space, we only detail the recovery of parameters and dimensions for the Conv operator, which is also the most complex task BTD needs to solve. As introduced in our main paper, we log execution trace using Intel Pin, simplify the trace with taint analysis, and then conduct symbolic execution to summarize symbolic constraints over inputs, parameters, and outputs. We then classify memory addresses collected over constraints into input addresses and parameter addresses. This way, we are able to gradually scope the memory layouts of inputs and parameters, thus inferring dimensions with *heuristics*. Parameters are then dumped from memory to disk by collecting all accesses to the input memory region, and with the recovered dimensions, we are able to convert parameters (in data bytes dumped from memory) into well-formed parameters that can be directly processed by PyTorch.

In addition to solutions of Conv, the rest of this section lists solutions of all other DNN operators that appear in our dataset. We emphasize that the **General Procedure** is indeed applicable to **all** DNN operators considered in this work. Given that said, heuristics used to infer dimensions, as expected, need to be defined specifically for different operators. Also, Conv denotes the most complex DNN operator that requires the recovery of both parameters and dimensions. Many other operators only contain dimensions or parameters, as will be clarified below.

2.1. Fully-Connected (FC) Operator

FC operators are commonly-used to serve the last few operators of many popular DNN models. In general, neurons in a FC operator are fully connected to all activations in its previous operator. The activations of neurons in a FC operator can be typically computed with a *matrix multiplication* followed by a bias. Let the input I of a FC operator be a matrix of size $(1,M)$, and the output O be a matrix of size $(N,1)$. Hence, parameters P of a FC operator can be represented as a (M,N) matrix, and Bias B is a matrix of $(N,1)$. The output can be calculated as $O = I \times P + Bias$.

Inferring Dimensions M and N. For dimensions, we need to first recover the input memory size M and the output memory size N. Given matrix multiplication $O = I \times P$ is performed in a FC operator, it is easy to see that each element o in the output matrix of size $(N,1)$ is calculated using M elements in the input and M elements from the parameters P. Therefore, once we have performed symbolic execution to summarize the semantics constraint over inputs, parameters and one output o, it is easy to infer M, by directly counting the number of input elements that are found in the symbolic constraint. To infer N, we re-run Pin to log all memory writes, scope the size of the output region M_o, and divide M_o by the size of an output element o. The size of o is 4 bytes, denoting a 32-bit floating point number, on 64-bit x86 platforms.

Inferring B. We discuss the following three situations to handle bias B used by FC operators. 1) The FC operator does not employ B. 2) The FC operator does employ B; however, for most of the time, given that the addition operator in FC is handled by a trailing Add operator, it is the Add operator that contributes the bias (see how Add is analyzed in Sec. 2.3). 3) The FC operator employs B, and due to DL compiler optimizations, the trailing Add operator that contributes B are fused with FC. Hence, the starting address of B can be directly acquired from the arguments of the fused assembly function. See the corresponding entry in Table 1.

2.2. Pooling Operator

A pooling operator downsamples the output of its prior operator. It is commonly used in dimension reduction to reduce the number of operations required for the following operators, but still retaining sufficient information from its previous operators. Despite that there are several different pooling schemes, e.g., max pooling (MaxPool), average pooling (AveragePool), and min pooling (MinPool), we infer the dimensions of pooling operators in a unified manner.

Heuristics on Dimension Inference. In general, for a pooling operator, we need to recover its stride S and the kernel size K. Similar with computations launched in a Conv operator, the pooling operator will iteratively extract a sub-matrix, namely *kernel*, for pooling. We clarify that kernel size K is inferred in exactly the same way as how kernel size is inferred for Conv. To recover stride S, we first launch symbolic execution and generate the symbolic constraints for two consecutive output elements, namely A and B. The offsets of input memory addresses shall indicate stride S. Specifically, we calculate S through the following formula:

$$S = \frac{addr_1 - addr_2}{size}$$

where $addr_1$ denotes the smallest input address found in constraint A and $addr_2$ denotes the smallest input address found in constraint B. *size* denotes the size of one input element, which is 4 bytes in our research context.

2.3. Arithmetic Operators

In general, arithmetic operators in DNN models can be divided into two categories: *parameterized* operators and *parameter-free* operators. For parameter-free operators, such as Add, Sub, Div, Sqrt, they perform element-wise arithmetic operations over one (for Sqrt) or two inputs. Since the operations are element-wise, there is no dimension to be recovered.

For arithmetic operators with parameters, the memory starting addresses of parameters can be obtained from the assembly function inputs (see Table 1 and Table 2). Furthermore, we need to recover the size of parameters in order to dump parameters from memory. For example, DL compilers will use BiasAdd (or BatchedAdd) to implement the "add bias" operation in a Conv operator. To extract biases, we first need to know its size. However, the size of parameters cannot be inferred from the symbolic constraints, because the operation is element-wise, thus only one input element and one parameter will be involved in each symbolic constraint to compute an output element. Recall when analyzing Conv and FC, we use Pin to record all memory accesses toward parameters to gradually scope the memory region size of parameters. Similarly, here we conduct the same process to obtain the memory region size of biases.

2.4. Embedding Operator

Despite the popularity of DNN models in CV applications, natural language processing (NLP) models are also extensively adopted in real-life scenarios. Despite all common DNN operators that can be handled by BTD, Embedding is unique to NLP models. Embedding is frequently used to preprocess input text data by encoding text into embedding vectors. Embedding is essentially implemented as a hash table that maps semantically similar inputs to similar vectors in the latent embedding space. To recover an embedding operator, we need to

infer the embedding dimension D (the size of the embedding vector), and the number of embeddings N (the size of the vocabulary or embedding table). For Glow-compiled executable, given that `memcpy` functions are used to implement Embedding, we can directly hook these `memcpy` functions and log its copied memory size, which equals to D. For TVM-compiled executable, to infer D, we use Pin to log all memory addresses that are accessed for memory read. The size of the longest and continuous memory chunk equals to D. To obtain N, we record the entire memory region of the embedding table used by Embedding, the size of which is denoted by M. Then, we compute N as $\frac{M}{D}$.

2.5. Misc.

Activation Functions. Typical DNN models include a large volume of non-linear activation functions like Sigmoid, tanh, Softmax, and ReLU. To ease the presentation, we divide these activation functions into two categories: *element-wise* activations and *tensor-wise* activations. As aforementioned, element-wise activations like ReLU, Mish, and Tanh, will apply activation function one element each time. For cases of this category, there are no dimensions to be recovered. On the other hand, tensor-wise activations will apply activation functions to an n-dimensional input tensor, e.g., Softmax, Softmin, and LogSoftmax. For these cases, recovering the dimension N of the input tensor is straightforward, given that for each symbolic constraint over inputs and one output element, the number of involved input elements equal to N.

Local Response Normalization (LRN). LRN usually follows a Conv operator, which applies normalization across channels. More specifically, the dimensions of output is the same as the dimension of input, and each output element is calculated using input elements from n neighboring channels. While there is no parameter involved, we need to recover the number of neighboring channels n, as a configuration of LRN. As expected, given a LRN operator whose number of neighboring channels is n, there will be n input elements involved in the symbolic constraint to compute one output element. Thus, we can infer n by counting the number of input addresses found in the symbolic constraint.

Transpose, ExpandDims & BatchFlatten. These operators do not perform arithmetic operations. Instead, they change the dimensions, alter, or reshape the layout of inputs. Hence, while no parameters are explicitly involved in the computations, we need to recover the attributes involved in their operations. To this end, we define heuristics based on the input dimensions and output dimensions. For example, Transpose permutates the input layout. Given the input dimensions is $[128,64,3,3]$ and the output dimensions is $[128,3,3,64]$, we can thus infer the attributes of this operator as $(0,2,3,1)$, i.e., it exchanges the 2nd and 4th axes of the input. Similarly, the attributes of ExpandDims are inferred, by comparing the inputs and outputs to decide which axis position in the input is extended with a new axis. Overall, it is worth noting that our current approach is *not* flawless. It should be easy to see that data flow analysis is required, in order to more accurately reveal the computation launched in these operators and how inputs are mutated and propagated to the outputs. We leave it as one future work to use taint analysis at this step to recover attributes of these operators.

Batch Normalization (BN). BN normalizes (e.g. scaling and shifting) operator inputs to enhance model learning quality and robustness. BN operators typically ship with parameters that were updated during training. The input and output dimensions are the same. It is worth noting that BN operators do *not* need to be handled specifically if the DL compiler optimizations are enabled: according to our observation, BN operators are typically fused with its prior operators (e.g., Conv) in the compiled executable, which changes the values of parameters in prior operators, but does not impede our solutions. In case that optimizations are disabled, BN operators are formed by simple arithmetic and utility operators. For example, when compiling with TVM -O0, the BN operator will be compiled as a sequence of basic arithmetic operations: [Add, Sqrt, Divide, Multiply, ExpandDims, Multiply, Negative, Multiply, Add, ExpandDims, Add.] Solutions for ExpandDims has just been given above, and all of these arithmetic operators can be smoothly decompiled following the solution detailed in Sec. 2.3.

InsertTensor & ExtractTensor. InsertTensor and ExtractTensor are used by Glow to process tensors. For example, Glow may use InsertTensor to implement padding operations by inserting the input tensor into another wider tensor filled with zeros. For these operators, we need to recover the attributes such as offsets, axises and counts employed to manipulate tensors. Similar to our solutions for Transpose, we can further design heuristics by locating inputs in outputs. That is, we can infer the attributes of InsertTensor by comparing the input and output tensors to decide the position where the input is inserted into the output. For ExtractTensor, we record the position from where the output is taken from the input. Again, we point out that data flow analysis is required to more accurately reveal the attributes of these operators. We plan to perform taint analysis to track how inputs are manipulated and propagated to outputs.

3. Recover Compilation Provenance

As mentioned in our paper, given a DNN executable e, our analysis requires to recover the compilation provenance: 1) which compiler is used, and 2) if TVM is used, whether the executable is compiled with full optimization -O3 or no optimization -O0. While most of our analysis is general to process executable compiled different provenance, still, there are some more cases that require different techniques, e.g., decompiling Embedding operators (Sec. 2.4) or handling memory layout alteration optimization (Sec. 4). In addition, obtaining this compilation provenance is a must, because to show decompiling e is flawless, we will have to re-compile recovered DNN models with the *same provenance* to compare with e.

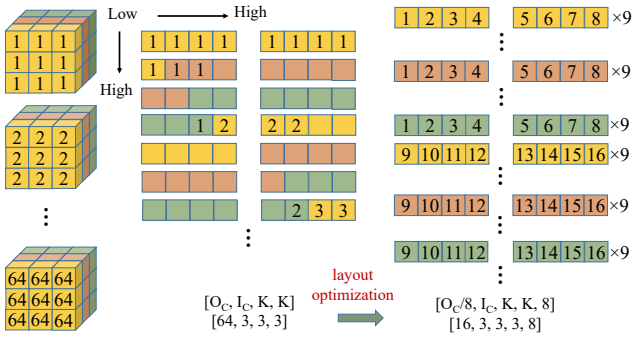

Figure 1: Layout optimization launched by Glow. TVM performs similar optimizations that can also be handled by BTD. To be consistent with notations used in our main paper, we use $[O_C, I_C, K, K]$ to denote the memory layout of weights. It is also referred to as $[N, H, W, C]$ in many previous literatures.

We clarify that compilation provenance can be automatically acquired. In particular, NNFusion-compiled executables are linked with kernel libraries, and many wrapper functions are placed in the executable files to invoke functions in the kernel libraries. It is thus easy to determine if an executable is compiled by NNFusion.

To determine if a piece of "self-contained" executable is compiled by TVM or Glow, we can search for TVMPlatformMemoryAllocate [5], a memory allocation routine code that is placed in assembly functions compiled by TVM. Further, to decide if a TVM-emitted executable is compiled using full optimization -O3 or no optimization -O0, we look for assembly functions with more than 3 parameters. This indicates operator fusion optimization, as assembly functions of fused operators will have at least two arguments pointing to more than one parameters of DNN operators, e.g., Conv (fused) in Table 1. We assess the correctness of these heuristics over all 15 image classification models provided by ONNX Zoo [4] compiled by TVM -O0, TVM -O3, and Glow. We report that compilation provenance can be correctly recovered for all cases with no error.

4. Recovering Parameters From Optimized Memory Layouts

Our main paper has clarified that parameters of Conv can be dumped from memory with its starting address obtained from assembly function inputs together with the inferred dimensions. However, to take full advantage of SSE parallelism on x86 platforms, DL compilers may perform layout alteration optimization to change the standard memory layout of parameters. We have illustrated the standard memory layouts and its optimized form in the main paper (Fig. 8). To ease the presentation, we re-present this layout comparison in Fig. 1. Overall, we find that Glow may change the memory layout from $[O_C, I_C, W, C]$ into $[O_C/A, I_C, K, K, A]$ (as shown in Fig. 1). Similarly, TVM may change the layout into a 6-dimensional tensor $[O_C/B, I_C/A, K, K, A, B]$.

Hence, inferring A and B becomes a pre-requisite to comprehend the optimized memory layouts. Given a summarized symbolic constraint over inputs, parameters, and one output element, we extract the memory offsets (w.r.t. the smallest memory address on the constraint) of all memory addresses belonging to weights, which will derive the following expression:

$$
\begin{aligned}
&[\\
&Y_0[x_{0,0},\ x_{0,1},\ \ldots,\ x_{0,(W-1)},\\
&\quad x_{1,0},\ x_{1,1},\ \ldots,\ x_{1,(W-1)},\\
&\quad ,\ \ldots,\\
&\quad x_{(A-1),0},\ x_{(A-1),1},\ \ldots,\ x_{(A-1),(W-1)}],\\
&Y_1[\ldots,\ \ldots,\ \ldots],\\
&\quad ,\ldots,\\
&Y_{C\times H/A}[x_{0,0},\ x_{0,1},\ \ldots,\ x_{0,(W-1)},\\
&\quad x_{1,0},\ x_{1,1},\ \ldots,\ x_{1,(W-1)},\\
&\quad ,\ \ldots,\\
&\quad x_{(A-1),0},\ x_{(A-1),1},\ \ldots,\ x_{(A-1),(W-1)}]\\
&]
\end{aligned}
$$

where each $x_{i,j}$ is a memory offset of one weight element (i.e., the memory address of one small box in the optimized memory layout in Fig. 1). Here Y is a notation to ease the representation, otherwise x will be defined using 3 dimensions.

Furthermore, the optimized memory layout in Fig. 1 can indeed derive the following constraint

$$Y_k[x_{i,j}] = (i-1)\times B + j\times A\times B + k\times W\times A\times B$$

Hence, A and B can be inferred, by matching the above constraint with two elements in the extracted memory offsets, e.g., $Y_0[x_{0,1}]$ and $Y_0[x_{0,3}]$.

For example, let the offset list of weight memory addresses in a symbolic constraint be

$$[0, 1024, 2048, 32, 1056, 2080, 64, 1088, 2112, \ldots]$$

, and the filter shape $[O_C, I_C, K, K]$ recovered in advance be $[256, 128, 3, 3]$. From the 2nd offset in the list, we can infer that $A \times B = 1024$. Similarly, from the 4th offset, we can infer that $B = 32$. This way, A and B are recovered, and the layout configuration becomes $[O_C/32, I_C/32, K, K, 32, 32] = [256/32, 128/32, 3, 3, 32, 32]$. Knowing the layout can enable smoothly loading all parameters out from memory and reforming the original filters in Conv.

References

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

[2] Bauman Erick, Lin Zhiqiang, and Hamlen Kevin W. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 2018.

[3] 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 *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, 2005.

[4] ONNX. ONNX Zoo: A collection of pre-trained, state-of-the-art models in the ONNX format. https://github.com/onnx/models, 2021.

[5] TVM. Tvmplatformmemoryallocate. https://tvm.apache.org/docs/api/doxygen/platform_8h.html#a133959eaf3ec68c568bdb71fcb94ddcb, 2021.

[6] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *NDSS*, 2017.

[7] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *USENIX Security*, 2015.

[8] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 133–147, 2020.