Hardening Deep Neural Network Binaries against Reverse Engineering Attacks

Anonymous Author(s)

Abstract

Deep Neural Networks (DNNs) are proprietary assets due to the expertise, confidential data, and high development costs involved in model training. Well-trained DNN models are compiled into DNN binaries to be efficiently executed on various platforms, such as edge devices and cloud infrastructures. Recent research on DNN binary decompilation shows the potential of stealing DNN models via binary reverse engineering techniques. While obfuscation is a well-studied technique to hamper binary reverse engineering, general obfuscation schemes are not designed for this new type of binary and have limitations in concealing information within DNN binaries due to the unique characteristics of DNN binaries.

In this paper, we show that existing reverse engineering attacks on DNN binaries can recover 98.5% of DNN operators from DNN binaries that have been obfuscated using general obfuscators. We then propose new obfuscation schemes tailored for DNN binaries, namely, (1) Flexible Operator Fusion; (2) Fake Operator Insertion; and (3) Operator Computation Reordering. We implement our dedicated obfuscation schemes as an end-to-end obfuscation toolchain called *NeuroShield*. Experiments show that *NeuroShield* is resilient to existing model reverse engineering attacks while introducing a reasonable overhead. Specifically, *NeuroShield* reduces the operator recovery rate to 3.03% for CV models and 47.18% for NLP models. Moreover, it has comparable binary size overhead and significantly lower execution time overhead (7.8% - 36.1%) compared to OLLVM, one of the commonly used general obfuscators.

CCS Concepts

• **Security and privacy → Software reverse engineering**.

Keywords

Binary Analysis, Deep Neural Network, Reverse Engineering

ACM Reference Format:
Anonymous Author(s). 2018. Hardening Deep Neural Network Binaries against Reverse Engineering Attacks. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 16 pages. https://doi.org/XXXXXXX.XXXXXXX

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Deep Neural Networks (DNNs) are considered proprietary assets due to the specialized expertise, confidential user data, and substantial development costs required to train them. Firstly, the model architecture needs to be well designed, which requires either human expertise or automatic neural architecture searching in a large searching space [17, 18]. Secondly, companies typically use private user data to train DNN models, which are often confidential. These data are vulnerable to leakage, especially in scenarios where the DNN model semantics are known by attackers (i.e., white-box attacks [58]). A recent article from NVIDIA reports a total training time of 34 days, and a cost of $4.6M to train the GPT-3 model [43]. These advanced models represent a unique form of intellectual property, which companies try to keep secret for the purpose of keeping user data confidential and staying competitive in the market [63].

To deploy a DNN model, DNN compilers (e.g., TVM [3]) compile DNN models into stand-alone binaries that can run on a dedicated backend device. For example, compiled DNN models have been widely deployed as web services on the cloud [12, 44, 47], and directly on edge devices (e.g., Google Chrome [49], Android apps [15]). In both cases, users can gain access to these DNN binaries, e.g., via compromising the cloud infrastructures or extracting binary code from the distribution packages. Encryption and model packing are two common ways to protect models [15, 63], yet are vulnerable to dynamic analysis [13, 26, 48, 63, 68, 77].

Recent research has shown the potential of reverse engineering DNN model structures from DNN binaries through static analysis [74] and dynamic analysis [35]. They exploit the following characteristics of DNN binaries: (1) DNN operators are typically compiled into distinct functions, providing explicit operator function boundary information; This allows attackers to extract the function corresponding to each individual operator and perform intra-procedure analysis; (2) The lack of input-dependent control flow in DNN binaries enables attackers to perform dynamic analysis on the instruction traces of random input, as any input can achieve full coverage of the binary code; (3) DNN operators are floating-point operations over tensors, which are typically implemented as nested loops with computation over buffers inside the loop bodies; This provides opportunities for both static and dynamic attackers to quickly reconstruct the semantics of a DNN operator, by recognizing the loop patterns.

In this paper, we first demonstrate the ineffectiveness of protecting DNN binaries from reverse engineering attacks by applying general binary obfuscation schemes. We implement an extended version of the attack described in [35] which successfully recovers the semantics of 98.5% operator functions from six DNN models (Mnist, Resnet, Mobilenet, FastText, ESM and AlBert) under the obfuscation of four general obfuscators (Fusor [75], OLLVM [25], Tigress [46] and Loki [56]).

To defend against existing reverse engineering attacks on DNN binaries, we propose three dedicated obfuscation schemes: (1) **Flexible Operator Fusion** to create fused operators with more complex semantics; (2) **Fake Operator Insertion** to introduce fake control flow to each fused operator by leveraging the *Bounded Neuron Activation* characteristic [37] of DNN models; (3) **Operator Computation Reordering** to distribute computation evenly across outputs for each operator. We implement these obfuscation schemes in a compilation toolchain called *NeuroShield*. Experiments show that *NeuroShield* is robust against existing reverse engineering attacks while maintaining reasonable overhead. Specifically, *NeuroShield* reduces the operator recovery rate to 3.03% for three CV models (Mnist, Resnet, and Mobilenet) and 47.18% for three NLP models (FastText, ESM, and AlBert). In terms of binary size, the overhead of *NeuroShield* is comparable to OLLVM, and is much smaller than Tigress and Loki. Moreover, *NeuroShield* has a much smaller execution time overhead (7.8% - 36.1%) compared to OLLVM, Tigress and Loki.

Our contributions can be summarized as follows:

- We conduct a study to show the ineffectiveness of general obfuscation schemes against existing DNN binary reverse engineering attacks, by extending the state-of-the-art attack on DNN binaries, and using it to effectively recover six DNN models' semantics from DNN binaries under general obfuscations.
- We propose three dedicated obfuscation schemes to enhance DNN binaries against existing reverse engineering attacks.
- We implement our dedicated obfuscation schemes as an end-to-end DNN compilation toolchain called *NeuroShield*.

We will open source *NeuroShield* upon the paper's acceptance.

2 Background

2.1 DNN and DNN Compilers

Deep Neural Network (DNN) is a machine learning method widely used in computer vision, voice recognition, and natural language processing. Structurally, DNNs are composed of multiple layers of neural operators, including, but not limited to, convolutional layers that help in feature extraction by processing data through various filters, and pooling layers like max pooling that reduce dimensionality and computational complexity while retaining important features. Each operator performs mathematical operations on its input (which is typically the outputs from the upstream layers), utilizing trainable parameters to compute its output. This output is then fed into the downstream operators. The training of DNNs is primarily conducted through back-propagation, an efficient algorithm for adjusting the internal parameters of the network. Back-propagation works by calculating the gradient of the loss function (a measure of prediction error) with respect to each parameter, and iteratively updating these parameters in a direction that minimizes the loss. After training, DNNs perform inference, utilizing their learned parameters to process new, unseen data and make predictions, demonstrating their ability to generalize from the data they were trained on to solve complex, real-world problems.

Trained DNN models are commonly represented as model files in formats like ONNX [50], TFLite [64], and PyTorch [53]. These model files are compiled by DNN compilers (e.g., TVM [3], Glow [55] and

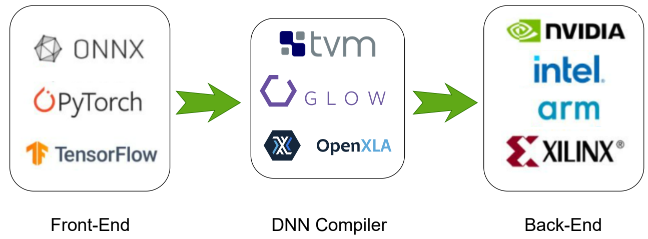


Figure 1: Typical front-end and back-end for DNN compilers

OpenXLA [51]) to binaries that can be executed on target platforms like GPUs, FPGAs and edge devices. Figure 1 shows some typical front-end model file formats and back-end platform targets for DNN compilers.

2.2 General Obfuscation Schemes

Opaque Predicates [11] involve inserting conditional statements into target code to create additional control flows. These conditions, known as opaque predicates, are always true/false, thus the execution paths associated with the false/true branch condition will never be executed. Dummy semantics can be added to these branches, which are called *Bogus Control Flow*. Opaque predicates are crafted to look complex and indecipherable to confuse humans attempting to understand the code or slow down the symbolic execution engine in solving complex path constraints. Effective opaque predicates must be constructed over input (or input tainted) values, which are considered as symbols in symbolic execution engines. However, most of the opaque predicate constructions [75] are based on integer operations, which can not be applied in DNN binaries as DNN models (if not quantized) are typically based on floating-point operations. Existing opaque predicate constructions over floating point operations rely on equality checking [73, 75] — the true branch of floating equality comparison is unlikely to be executed due to the precision loss characteristic of floating point operations. However, these opaque predicates follow the same equality-check pattern, making them trivial to detect.

Control Flow Flattening (CFF) and Virtualization (VM) [8, 9] transforms the code's original structured control flow into a flattened, switch-case-like structure, where the execution sequence is controlled by a dispatcher. Each piece of the original code is turned into a case within a large switch statement, and the order of execution is determined by the value of a state variable that is altered dynamically. According to the granularity of the flattened code piece, it can be classified as *Control Flow Flattening* [8] (basic block-wise) and *Virtualization* [9] (instruction-wise). This method is effective against static analysis as it makes the control flow appear linear and sequential, obscuring the actual logical control flow structures (e.g., loop structures). However, it is demonstrated to be vulnerable to dynamic semantic attacks [77] (e.g., dynamic back-slicing and symbolic execution).

Mixed Boolean Arithmetic (MBA) [82] transforms simple arithmetic and logical operations to equivalent expressions that use a mix of both arithmetic operations and bit-wise boolean operations. For example, a simple operation like addition might be redefined using a complex combination of AND, OR, XOR, and other arithmetic operations. Equation 1 shows some examples of equivalent MBA expressions for a single integer addition. The key advantage of MBA obfuscation is that it transforms straightforward operations into expressions that are mathematically correct but practically difficult to simplify and understand. The limitation is that *MBA* can only be applied to transform integer operations [82], while the dominant operations in DNN binaries are floating-point arithmetics.

$$x + y = \begin{cases} x - \neg y - 1 \\ (x \oplus y) + 2 \cdot (x \wedge y) \\ (x \vee y) + (x \wedge y) \\ 2 \cdot (x \vee y) - (x \oplus y) \end{cases} \tag{1}$$

2.3 Dynamic Reverse Engineering Attack

Dynamic Symbolic Execution [5, 57] is a systematic way to analyze the semantics of a program. By treating program inputs as symbolic values rather than concrete data, this technique allows attackers to capture the program output semantics as symbolic expressions of program input. A well-known drawback of symbolic execution is that it tries to statically explore every possible execution path within a program. This approach can result in a path explosion problem in complex systems, where the number of paths becomes excessively large. Dynamic symbolic execution addresses the issue of path explosion by combining the principles of concrete execution with symbolic analysis, which starts with a specific execution path with concrete input, symbolizes program inputs, and solves constraints along the concrete execution path to find input values that result in other execution paths.

Dynamic Backward Slicing is a taint analysis procedure that starts from the program output and works backward to figure out the sections of code that contribute to the computation of the output. Non-tainted code sections are considered unrelated to the output and are excluded from further analysis. This method is extremely effective in analyzing some obfuscation schemes such as Control Flow Flattening and Virtualization — for flattened or virtualized programs, most of the code in the dynamic execution traces is related to the switch-case-like code dispatcher, which does not contribute to the computation of true program output. This semantically insignificant code can be removed by backward slicing [77].

3 Motivation

In this section, we discuss some characteristics of DNN binaries that render general obfuscation schemes ineffective, and present a proof-of-concept attack to reconstruct the semantics of DNN binaries obfuscated by general obfuscators.

3.1 DNN Binary Characteristics

[C1] Explicit function calls to DNN operators. In compiled DNN binaries, a function typically contains the semantics of a single complex operator (e.g., `Conv2D)`, or a fusion of an operator with its downstream element-wise operator (e.g., `Conv2D+Relu`). The arguments of each function are pointers to input/output buffers that the operator reads input data from/writes output data to. Input buffers hold model parameters or computation results (output) of upstream operators. After mathematical computation over inputs and parameters, the resulting operator outputs (or intermediate results) are written to output buffers. Listing 1 shows a typical (decompiled) operator function prototype of a `Conv2D` operator generated by Glow [55]. Function arguments `a1, a2, a3` are pointers to buffers for input, filter weight, and output, respectively. This characteristic provides the opportunities to perform (1) dedicated semantical analysis (e.g., intra-procedure analysis on each function) on a single operator and (2) memory access analysis near the argument buffer address to figure out the input/output buffers and their sizes. Unless changing the behavior of the DNN compiler, the information leaked by explicit function calls cannot be prevented because for most of the general obfuscators [25, 46, 56, 75], obfuscation transformations are applied independently to each function.

```
void conv2d(uint64 a1, uint64 a2, uint64 a3);
```

Listing 1: A typical DNN operator function prototype

[C2] Lack of input-dependent control flow. Input-dependent control flows only exist in a small set of DNN operators, such as `MaxPool` and `Relu`. These control flows involve selecting a larger value among multiple values (e.g., `a>b?a:b`). However, they are compiled to control-flow-free instructions (e.g., `MAXPS` in x86). Therefore, for most DNN operators, analyzing the dynamic behavior of a single random input is enough to recover operator semantics, as any input can achieve full code coverage. This characteristic makes symbolic execution (which suffers from the path explosion issue) ideal for DNN operator semantic extraction [35]. Opaque Predicate [11] is a widely used obfuscation scheme to add bogus input-dependent control flows. However, state-of-the-art opaque predicate constructions [75] are either based on integer operations, or easy to be detected, as discussed in Section 2.2.

[C3] Floating-point operations. Output of DNN operators is essentially a combination of floating-point operations over input and parameters. Existing obfuscation scheme on arithmetic operations, i.e., Mixed Boolean Arithmetic (MBA) [82], can only be applied to integers. Consequently, in DNN binaries only the integer instructions used for concrete address calculations (for buffer accessing) can be obfuscated. However, this offers little protection against dynamic analysis, since the actual memory read/write addresses are concrete and can be resolved at runtime.

[C4] Tensor Computation and its Nested Loop Implementation. DNN operators perform computation over tensors (i.e., multidimensional arrays). Each element in the output tensor is computed over different parts of the input tensors, but with a similar combination of arithmetic. Therefore, DNN operators are generally implemented as nested loops with computation over tensor elements (indexed by loop iterator variable) inside loop bodies. Figure 2 shows a `Conv2D` operator with 3×3 input, 2×2 weight (filter) and 2×2 output (for simplicity, batch and channel sizes are set to 1). A typical implementation of this operator in TVM [3] (represented in TIR [20]) is shown in Listing 2. The outer loop (Line 8) iterates over all possible indexes of the output buffer (with loop variables `oh` and `ow`) and each iteration generates the complete result of a single output element. This can be leveraged by the attacker to speed up reverse engineering the semantics of the operator. More specifically, they can (1) perform analysis of the execution of a single

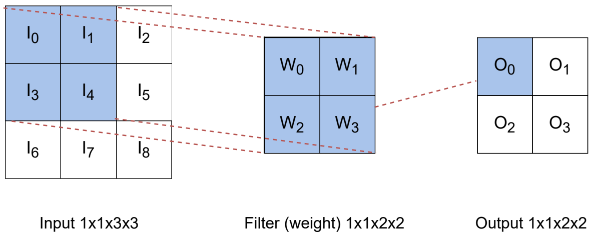


Figure 2: Example Conv2D operator. The numbers indicate the tensor element index within its flattened buffer.

```
from tvm.script import tir as T
@T.prim_func
def tvmgen_default_nn_conv2d(
    input_: T.Buffer((9,), "float32"),
    output_: T.Buffer((4,), "float32"),
    weight_: T.Buffer((4,), "float32")
):
    for oh, ow in T.grid(2, 2):
        output_[oh * 2 + ow] = T.float32(0)
        for kh, kw in T.grid(2, 2):
            output_[oh*2+ow] += input_[oh*3+kh*3+ow+kw]
     * weight_[kh*2+kw]
```

Listing 2: A typical implementation of Conv2D in TVM TIR

outer loop iteration, which contains the computation of a complete output element; (2) recover the semantics of a single output, e.g., $out[0] = in[0] * w[0] + in[1] * w[1] + in[3] * w[2] + in[4] * w[3]$, which is enough to recover the complete semantics of a whole operator using some heuristics [35].

3.2 Proof-of-Concept Attack

Threat model and assumption. We assume the attackers have the following abilities: (1) They have access to the victim binary and can extract the function boundaries in the binaries (e.g., using common disassemblers like IDA and Ghidra); (2) Attackers can dynamically execute the binary and perform analysis dynamically (e.g., log the information of each executed instruction). The threat model is aligned with the previous work [35, 74].

Scope. This paper focuses on recovering the semantics of each operator function in DNN binaries, since model-level information (e.g., connection between operators) can be directly inferred from the model's operator dispatch function [35, 74]. Operator semantic is defined as its type and attributes (e.g., input/output/filter shapes). A successful recovery entails reconstructing both the operator type and its attributes.

To demonstrate the limitation of general obfuscation schemes on DNN binaries, we design a general dynamic symbolic execution attack to reverse engineer the semantics of the obfuscated DNN binaries under general obfuscation schemes mentioned in Section 2.2. We prototype our basic attack based on the DNN characteristics in Section 3.1. Our basic attack applies the following steps to each operator function (with function boundaries extracted by the IDA Pro disassembler [23]).

❶ **Identifying read/write buffer**. For each operator function with #args arguments, we create #args buffers with random contents and pass the buffer pointers to it. We then dynamically run the function with the random input. For each memory operation we encounter, we log (1) whether the instruction is a memory read or memory write instruction; (2) the dynamic memory address and size for the memory operation. We then group memory read and write together to get continuous read/write buffers, and their sizes.

❷ **Identifying output buffer**. For each function, ❶ outputs a single write buffer which is the output buffer of the operator (or fused operator) related to that function, due to the following characteristics of DNN compilers: (1) Output elements (tensor) of an operator are grouped in a single continuous buffer; (2) As mentioned in Secition 3.1, a single function in the DNN binary contains the implementation of at most one complex operator (e.g., `Conv2D`), with potential downstream element-wise operator fused to it. As a result, only one output buffer is needed because, for the fused operator, the downstream element-wise operator reuses the output buffer of the complex operator to perform its computation. For this output buffer, we re-run the operator function to log the number of times the first element in the buffer is written to, denoted as $\#w\_fisrt\_ele$.

❸ **Instruction Trace Logging.** We re-run the operator function again. For each instruction encountered, we log the instruction machine code, the memory type of the instruction (read/write/non-mem), and the dynamic memory address it operates on. We stop the logging when the times of memory writes to the first output buffer element reach $\#w\_fisrt\_ele$, which we obtained from ❷.

❹ **Recovering the first output semantic**. We set the first element in the output buffer as tainted, and performed back-slicing on the trace we logged in ❸. After that, we get a sliced trace of instructions that contribute to the computation of the first output element. Then we set all elements in the input buffers we identified in ❶ as symbolic, and perform symbolic execution on the sliced trace, after which we obtain the semantics of the first output element as a symbolic expression of input elements.

❺ **Operator semantic reconstruction**. With the symbolic expression and input/output buffers we identified in previous steps, use the heuristics in [35] to reconstruct the semantics of the whole operator.

Our proof-of-concept attack is mostly aligned with the BTD attack [35], with the following extensions: (1) BTD stops trace logging after completing executing the first iteration of the outermost loop, while we stop logging after the last memory write to the first output element is finished; (2) We extend BTD's symbolic engine to support more instructions that appear in the obfuscated binaries but not in the original binaries.

Experimental results show that our proof-of-concept attack can successfully recover 98.5% of compiled operator functions among six DNN models obfuscated by four general obfuscators, which proves the effectiveness of our extended BTD attack. We discuss the details in Section 6.

4 Dedicated Obfuscation Schemes for DNN Binaries

Given the limitation of general obfuscation schemes discussed above, we propose three dedicated obfuscation primitives for DNN

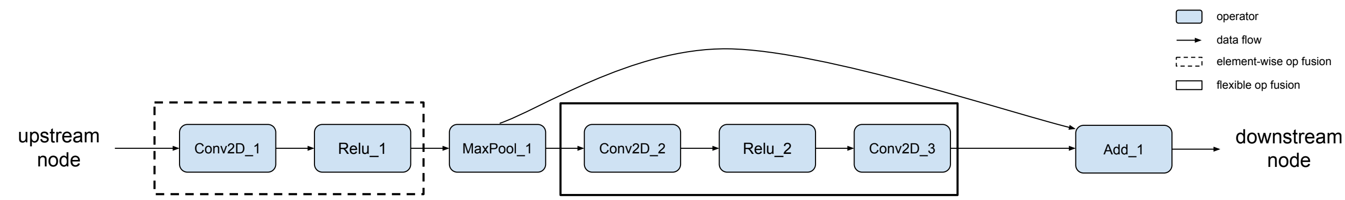


Figure 3: Example DNN computational graph with a common shortcut structure

binaries, namely, (1) Flexible Operator Fusion, (2) Fake Operator Insertion, and (3) Operator Computation Reordering.

Algorithm 1 Flexible Fusion

```
Require: G, grps, maxOp
1: for node in G.topo_order() do
2:     if node.outputs.size() ≠ 1 then
3:         continue
4:     end if
5:     sink ← node.outputs[0]
6:     if grp[node].root = grp[sink].root then
7:         continue
8:     end if
9:     if grps[node].numOp+grps[sink].numOp > maxOp then
10:        continue
11:    end if
12:    grps.union(node, sink)
13: end for
```

4.1 Flexible Operator Fusion

DNN models are generally represented as *Computational Graphs*, with graph nodes denoting computation operators and edges denoting data flows. Computation operators can be divided into two categories: (1) *Complex* operators (e.g., `Conv2D`, `Dense`, `MaxPool`), which compute each output element from multiple input elements; (2) *Math* operators (e.g., `Add`, `Relu`), which perform element-wise mathematical operations over the input. Figure 3 shows the computational graph of a shortcut structure commonly found in DNN models.

Most DNN compilers use *Graph-Level IR* to represent the computation graph structure of a DNN model (e.g., Relay IR in TVM [3], HLIR in Glow [55]). DNN compilers perform operator fusion on Graph-Level IR as a general optimization technique. However, most compilers (e.g., TVM and Glow) restrict fusion to narrow cases – complex operators can only be fused to element-wise math operators concatenated to them. For example, concatenating operators `Conv2D_1` and `Relu_1` in Figure 3 are fused together. As a result, a compiled fused operator function contains the semantics of only one complex operator (called *"anchor"* operator in TVM). The simplicity of the operator function is beneficial to subsequent compilation optimization. However, strict fusion rules result in a limited number of fused operator types and ease the effort for attackers to reverse engineering the DNN binary.

We propose to relax the strict fusion rules and aggressively fuse operators, as long as they meet the following criteria:

- **[R1]** Upstream operators cannot be fused to downstream operators if it has multiple downstream operators. For example, `MaxPool_1` in Figure 3 cannot be fused to `Conv2D_2` because it has a shortcut to another downstream operator `Add_1`.
- **[R2]** The number of complex operators fused together does not exceed a predefined limit (denoted as max_fuse_depth);

Algorithm 1 describes our Flexible Fusion procedure. It takes a computation graph G and current fusion groups grp (a union-find data structure with computation graph node as key) as input. We process nodes in topological order (Line 1). For each node, Lines 2–3 enforce criterion R1, Lines 6–7 check whether it has already been fused with its downstream node. Lines 9–10 check for criterion R2. If all these checks succeed, we merge the fusion groups of the two nodes (Line 12).

Under relaxing operator fusion rules, more complex operators can be created by fusing a larger number of operators (potentially with multiple complex operators, e.g., `Conv2D_2 + Relu_2 + Conv2D_3` with a $fuse\_depth$ of 2 (#complex operators) in Figure 3).

Discussion. Flexible Fusion provides the following two benefits: (1) It increases the complexity of data dependency in an operator function, compared to the default fusion strategy which results in one single output buffer in a function; (2) The complex function semantics provide more opportunity for further obfuscation, which we discuss in Section 4.2. However, too complicated semantics make it difficult for subsequent compilation optimization and result in inference overhead. We discuss the value selection for max_fuse_depth in Section 6.

4.2 Fake Operator Insertion

As discussed in Section 3.1, the lack of input-dependent control flow in DNN binaries makes them vulnerable to dynamic analysis, as any concrete input to the binary can achieve full code coverage. To tackle this issue, we add input-dependent control flow to DNN binaries by leveraging the *Bounded Neuron Activation* characteristic of DNN models [37]: each neuron (element in operators' output) has a valid activation range. To further obfuscate a fused operator, we construct fake control flows by inserting random operators conditioned on neuron values that fall outside of their valid value ranges, namely, Fake Operator Insertion.

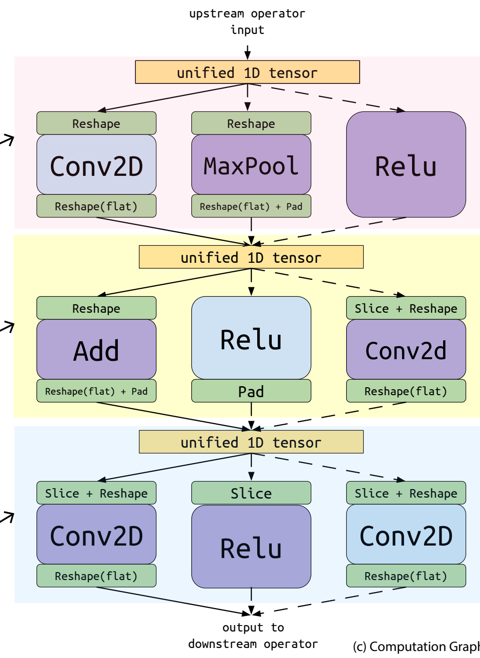
Overview. We explain the procedure of Fake Operator Insertion using the fused `Conv2D + Relu + Conv2D` example in Figure 3. Figure 4a shows its related Relay IR representation. Before we start inserting fake operators, the value range of outputs from each real operator is profiled using the model training dataset.

```
1  def @ff_op(%input[1,3,32,32]) {
2    %0 = conv2d(%input, param0[64,3,3,3], pad=true);
3    %1 = relu(%0);
4    %2 = conv2d(%1, param1[64,64,3,3], pad=true)
5  }
```

(a) Relay IR after Flexible Fusion

```
1  def @fi_op(%input[3072]) {
2    // %u0 = unified 1d computation output of conv2d,
3    //   maxpool, and relu over %input[3072]
4    %n0 = gather(%u0, idx=0);
5    %cond0 = less(%n0, 1.5f);
6    %u1 = if (%cond0) {
7      %l0 = reshape(%u0, shape=(1,64,32,32));
8      %fop0 = add(%l0, fparam0[64]);
9      %l1 = reshape(%fop0, shape=(65536));
10     %l2 = pad(%l1, pad_size=64);
11   } else {
12     %cond1 = less(%n0, n0_r0=2.5f);
13     if (%cond1) {
14       %rop = relu(%u0);
15       %l3=pad(%rop, pad_size=64);
16     } else {
17       %l4 = slice(%u0, range=(0,32768));
18       %l5 = reshape(%l4, shape=(8,16,16,16));
19       %fop1 = conv2d(%l5, fparam1[82,16,7,7], pad=false);
20       %l6 = reshape(%fop1, shape=(65600));
21     }
22   };
23   // downstream conv2d, relu conv2d
24   //   computation over %u1
25 }
```

(b) Relay IR after Fake Op Insertion



(c) Computation Graph

Figure 4: (a) Relay IR after Flexible Operator Fusion. (b) Relay IR after Fake Operator Insertion. (c) Computational graph representation of an obfuscated operator with $insert\_depth = 3$ and $insert\_width = 2$. Blue boxes = real operators, purple boxes = fake operators, yellow boxes = unified output, green boxes = legalization operators, solid arrow = real data flow, dashed arrow = fake data flow.

We perform insertion for nodes in the topological order of the computation graph — when starting fake operator insertion for the second operator $\%1 = relu(\cdot)$ (Figrue 4a Line 3), the output from upstream $\%0 = conv2d(\cdot)$ (Figrue 4a Line 2) is flattened to 1D unified form $\%u0$ with shape=(65536,) (Figrue 4b Lines 2-3). Assume the first (index $idx = 0$) neuron of $\%u0$ has a value range of $[1.5, 2.5]$ and we decide to use it to construct conditions for fake control flows.

In Figure 4b, we first extract the neuron value $\%n0$ using a `Gather` operator (Line 4), then construct a condition $\%cond0$ by doing less comparison between $\%n0$ and the lower bound 1.5 of $\%n0$'s valid value range (Line 5). We then use a relay `If` expression on $\%cond0$ to construct two branches (Line 6). The true branches ($\%n0 < 1.5$) lies outside the valid range $[1.5, 2.5]$, thus we insert the first fake operator $\%fop0 = add(\cdot)$ in this branch (Line 8). As $\%fop0$ takes a input of shape $(1, 64, 32, 32)$, the unified 1D output $\%n0$ is first reshaped before fed to $\%fop0$ (Line 7).

Similarly, in the false branch of $\%cond0$, we construct a nested condition $\%cond1$ based on the upper bound 2.5 of $\%u0$'s neuron range (Line 12). The corresponding true branch lies in the valid range of $\%n0$, therefore, we put the real operator $rop = relu(\cdot)$ here (Line 14). In the false branch, we insert another fake operator $\%fop1 = conv2d(\cdot)$ (Line 19). The input shape for $\%fop1$ is $(8, 16, 16, 16)$ (with size 32768) which is smaller than the size 65536 of unified input $\%u0$. Thus we use a `Slice` operator to extract the first 32768 elements and then reshape it to shape $(8, 16, 16, 16)$ (Lines 17-18), before feeding it to $\%fop1$.

To unify the outputs from $\%fop0$ and $\%fop1$, we flatten them to 1D shapes (Line 9 and 20). Note that the real `Relu` operator $\%rop$ can operate on any shapes, thus no reshaping is required. As $\%fop1$ has output size 65600 (shape=(8, 82, 10, 10)) which is larger than the output size 65536 of $\%rop$ and $\%fop0$, we use `Pad` to pad additional 64 elements for $\%rop$ and $\%fop0$ respectively (Line 10 and 15). At present, we have finished the fake operator insertion for the second operator $\%1 = relu(\cdot)$(Figure 4a Line 3), with unified output $\%u1$ (Figure 4b Line 6). $\%u1$ is then used for the fake control flow construction for downstream operator $\%2 = conv2d(\cdot)$ (Figure 4a Line 4).

The computation graph of the resulting obfuscated function is shown in Figure 4c. The operators used for shape transformations (`Slice`, `Pad` and `Reshape`) are called *Legalization operators* and marked in green. Original real operators are marked in blue. Fake operators are marked in purple.

Insertion Depth/Width. We define *Insertion Depth* as the number of (maximum) original operators in a fused function that we used to insert fake operators. In the above example, $insert\_depth = 3$ because all three operators in the fused function receive insertion. We define *Insertion Width* as the number of fake operators added per original operator. We have $insert\_depth = 2$ for the above example. This setting results in 3x3x3=27 input-dependent control flows, marked as arrows in Figure 4c. We discuss the value selection for $insert\_depth$ and $insert\_depth$ in Section 6.

Neuron selection and neuron range handling. Let a neuron n has a profiled range $[l, h]$ over the model's training dataset. For

```
from tvm.script import tir as T
@T.prim_func
def default_nn_conv2d(
  input_: T.Buffer((784,), "float32"),
  output_: T.Buffer((576,), "float32"),
  weight_: T.Buffer((25,), "float32")
):
  for oh, ow in T.grid(24, 24):
    output_[oh*24+ow] = T.float32(0)
    for kh, kw in T.grid(5, 5):
      output_[oh*24+ow] += input_[oh*28+kh*28+ow+kw] *
     weight_[kh*5+kw]
```

Listing 3: Default implementation for a Conv2D operator generated by TVM

```
from tvm.script import tir as T
@T.prim_func
def reordered_nn_conv2d(
  input_: T.Buffer((784,), "float32"),
  output_: T.Buffer((576,), "float32"),
  weight_: T.Buffer((25,), "float32")
):
  for oh, ow in T.grid(24, 24):
    output_[oh*24+ow] = T.float32(0)
  for kh, kw in T.grid(5, 5):
    for oh, ow in T.grid(24, 24):
      output_[oh*2+ow] += input_[oh*28+kh*28+ow+kw] *
     weight_[kh*5+kw]
```

Listing 4: An implementation of the same Conv2D operator with different loop order

each operator's output tensor, we select the neuron n that has the median range length $h - l$ for fake operator insertion. To preserve the semantics of compiled DNN model, we increase the range $[l, h]$ to $[\hat{l} = l - \delta, \hat{h} = h + \delta]$. We discuss the value selection for δ in Section 6.1. To randomize the value range location of the real operator and fake operators, given a *insert_width*, we chose a random location i from $(1, ..., insert\_width+1)$ for the real operator. We then partition the left range $[-\infty, \hat{l}]$ and right range $[\hat{h}, \infty]$ to $i - 1$ and $insert\_width + 1 - i$ sub-ranges respectively, for fake operator generations.

Fake Operator Generation. Given the size s_i of upstream unified output, and the size s_o of real operator output, we construct fake operators following two goals: (1) make the fake operator input size as close as possible to s_i but not exceeding it; (2) make fake operator output sizes as close as possible to s_o. We first randomly select an operator type from the common computation operators. Then we concretize the selected operator type with input shape, output shape and attributes. Apart from `Conv2D`, the concretization of other operators are straightforward — they either have a fixed output size given input size (e.g., element-wise math operators `Add`, `Relu`), or the attributes that map the input size to output size can be directly computed (e.g., `Dense`, `Maxpool`). We discuss the details of `Conv2D` concretization in Appendix A.

Overhead. Fake Operator Insertion introduces multiple sources of overhead. Firstly, for neurons selected to construct fake control flows, we need to store their index and neuron ranges. As we select at most one neuron for each operator, and the number of operators is much less than the number of parameters in a DNN model, the overhead introduced by fake control flow construction is negligible. Moreover, some types of fake operators (e.g., `Conv2D`) can contain additional parameters. We eliminate this overhead by reusing the parameters from real operators. More specifically, TVM stores all model operators in a unified continuous buffer. We thus set the pointers for fake parameters to some random offsets in the buffer, making sure the chosen offset plus the fake parameter size doesn't exceed the buffer's end. This requires the fake parameter to be smaller than the whole DNN model parameter size, which is easy to satisfy as the parameter size of the whole model is typically large. We simply discard and re-generate a fake operator if it has a substantially large parameter size.

Secondly, fake operators can generate outputs larger than the real operator, which incurs extra workspace memory overhead. We limit this by discarding and regenerating any fake operator whose output size exceeds the real operator's by more than 50%.

Thirdly, Fake Operator Insertion introduces additional shape transformation operators in the model's Relay IR representation (e.g., `Reshape`). Note that they are just used to perform shape legalization and make a valid IR representation. Most of them can be optimized away in later compilation stages, as long as the input and output layout of these `Reshape` operators remain the same.

Discussion. Existing opaque predicate construction on floating point operations is based on equality checks, which generate unsatisfiable branch with a predictable pattern. As a result, the bogus control flow branch can be easily detected by symbolic execution attacks. In comparison, Fake Operator Insertion uses floating-point less-than comparisons to produce two satisfiable branches, making them harder for symbolic engines to resolve.

4.3 Operator Computation Reordering

To lower a fused operator in the computation graph (Relay IR), TVM chooses an operator implementation generation scheme (e.g., `Strategy` [60] in TVM), which takes the operator attributes (e.g., shape of input, output and weight) as input and generates an implementation for the operator. TVM uses TIR [20] to represent the implementation of operators in the form of nested loop computation. Consider the example of a `Conv2D` with input shape $(1, 1, 28, 28)$, output shape $(1, 1, 24, 24)$ and weight shape $(1, 1, 3, 3)$. TVM generates an implementation shown in Listing 3. The implementation places the output buffer iterators (oh, ow) in the outer loop, and the weight buffer iterators (kh, kw) in the inner loop. However, this implementation offers attackers greater opportunities to reverse-engineer the operator's semantics — each iteration of the outer loop computes a full output element, which can be used to infer the complete functionality of the operator (as discussed in Section 3.1).

To tackle this issue, we apply Operator Computation Reordering for each complex operator (`Conv2D`, `Dense`, `MatMul`) implementation. Specifically, if the iterators over the output buffer are not located in the innermost loop, we move them inward by one level, stopping once an invalid TIR exception is triggered. Listing 4 shows the reordered implementation of the `Conv2D` example, where the order of output buffer iterators (oh, ow) and weight buffer iterators (kh, kw) are switched.

Discussion. Operator Computation Reordering can cause overhead for execution time. In the `Conv2D` example, the output size 576 is

Table 1: Model statistics for 6 tested models. #op = number of operators. #cmpl. op = number of compiled operator functions that contain fused operators.

model	type	#op	#cmpl. op	bin sz. (byte)	param sz. (byte)	mem sz. (byte)	exe. time (s)
Mnist	cv-cnn	12	5	142944	23984	54272	0.000778
Resnet	cv-cnn	70	22	173024	44676648	1048576	0.734
Mobilenet	cv-cnn	155	55	251520	13951488	11239424	0.471
FastText	nlp-embeding	11	5	137544	84473024	114000	0.000129
ESM	nlp-transformer	545	163	239048	30082056	6063616	1.457
AlBert	nlp-transformer	888	296	284760	46550052	5189632	15.848

much greater than the weight size 25. Compared to the reordered implementation, the default implementation has better cache locality, as it avoids repeatedly accessing the large output buffer by placing its iterators in the outer loop. To prevent substantial execution time overhead, we discard the reordered implementation if it results in more than a 10% increase in execution time.

5 Implementation

We prototype our three obfuscation primitives as a DNN compilation toolchain called *NeuroShield*, based on the TVM compilation infrastructure. Given a DNN model file (e.g., in ONNX [50] format) and its training dataset as input, the pipeline of *NeuroShield* can be described as follows. ❶: The model file is converted to Relay IR using TVM's front-end. ❷: The model training dataset is fed to the Relay representation of the model, which is directly (without compilation) executed by the Relay IR executor to profile the value range for each neuron. For each operator's output tensor, the neuron with the median profiled length is selected to be used for fake control flow construction in later stages. The selected neuron (represented by its index within the tensor) and its value range are attached to the operator as attributes. ❸: To explore more operator fusion opportunities, TVM's built-in `FuseOps` pass is first applied. Subsequently, the Flexible Operator Fusion pass is executed to enable additional fusion of complex operators beyond the default TVM fusion rules. ❹: Annotated neuron index and value ranges are used by Fake Operator Insertion to construct input-dependent control flows. ❺: The Relay IR representation of the obfuscated model is then lowered to TIR with the TVM built-in `LowerTE` pass. ❻: Operator Computation Reordering is applied to the TIR representation of complex operator implementations. ❼: The TIR module is further lowered to LLVM-IR by the TVM codegen module, and compiled to binary by `clang`.

Table 2: Obfuscation schemes enabled in general obfuscators for evaluation

Obfuscators	Obfuscation schemes
OLLVM	Control Flow Flattening, Opaque Predicates, Mixed Boolean Arithmetic
Fusor	Opaque Predicates
Tigress	Virtualization, Opaque Predicates
Loki	Virtualization, Mixed Boolean Arithmetic

We implement the three obfuscation schemes as configurable switches within the compilation toolchain. They can be applied independently, except that Fake Operator Insertion depends on Flexible Operator Fusion. Our implementation consists of over 5000 LOC (including addition and deletion) modifications to the TVM code base. The changes include: (1) We implement Flexible Operator Fusion as a Relay pass, based on the TVM built-in `FuseOps` Pass to enable more complicated operator fusion; (2) We instrument the built-in `FuseMutator` pass for Fake Operator Insertion; (3) As TVM currently does not support lowering for Relay `If` expression to TIR [1], we modify the `AOTExecutor` module to support the lowering of our obfuscated operator, which consists of control flow encoded by Relay `If` expressions; (4) We instrument the `LowerTE` pass to inspect the loop order in the generated TIR implementations of operators, and apply corresponding Operator Computation Reordering.

6 Experiments

DNN model set. We perform an evaluation on three image classification models — Mnist, Resnet, and Mobilenet, and three natural language processing models — FastText, ESM [54], and AlBert [29]. We compile these models with TVM as the baseline for the experiments. Table 1 shows the statistics of the six models.

General obfuscators. To evaluate our three obfuscation primitives against general obfuscation schemes, we choose the following four general obfuscators: (1) Obfuscator-LLVM (OLLVM) [25], a widely used obfuscation framework to implement obfuscation schemes at the LLVM-IR level, which has been used to build dedicated obfuscators [45, 75]; (2) Fusor [75], which implements a set of strong opaque predicates and has support for floating-point-based opaque predicates; (3) Tigress [46], the state-of-the-art general obfuscator in academia which implements a broad range of obfuscation transformations over C code; (4) Loki [56], the state-of-the-art MBA obfuscator, which enhances the Virtualization obfuscation scheme by applying MBA extensively. Table 2 shows the enabled obfuscation schemes in these obfuscators for our evaluation. We believe they represent the state-of-the-art general obfuscation schemes that are available to the public. Commercial obfuscators such as VMProtect [69] and Themida [66] work only on Windows executables (PE format) and are not freely available.

To generate obfuscated binaries, we use TVM to compile the target DNN models to LLVM-IR, which can be directly transformed by OLLVM, Fusor, and Loki. Tigress transforms code at the C code

[1] https://discuss.tvm.apache.org/t/relay-tvmerror-if-is-not-supported/6599

Table 3: Model semantics preservation

model	train(test) / external dataset	train dataset			test dataset					external dataset	
		#samples	accuracy		#samples	accuracy		oor%	$\frac{\delta}{h-l}$	#samples	oor%
			original	*NeuroShield*		original	*NeuroShield*				
mnist	MNIST / EMNIST	60000	99.49%	99.49%	10000	98.90%	98.89%	0.01%	0.18%	100000	0.20%
resnet	CIFAR10 / STL10	50000	90.08%	90.08%	10000	86.16%	86.10%	0.09%	16.79%	5000	0.12%
mobilenet	ImageNet / OpenImages	1281167	82.60%	82.60%	50000	69.13%	69.12%	0.01%	8.82%	100000	0
fasttext	AGNews / DBPedia14	120000	87.92%	87.92%	7600	82.03%	82.00%	0.03%	0.69%	100000	4.82%
esm	UniProtCL / UniProtSS	4674	89.02%	89.02%	520	89.31%	85.96%	4.04%	26.55%	4348	2.48%
albert	SST2 / CustRev	67349	97.97%	97.97%	872	92.66%	92.20%	0.46%	6.45%	3394	3.03%

level. We use llvm-cbe [36] to first lift the compiled model in LLVM-IR to C code, and then apply the Tigress transformation.

***NeuroShield* configuration.** For Mnist, Resnet, Mobilenet and FastText, we set the maximum number of complex operators in a fused function (max_fuse_depth) to 3, as this number gives complicated fusion results and does not cause extensive execution time overhead. Note that complex operators are commonly followed by element-wise math operators — the total number of operators in a fused function can be much larger than 3. For Transformers (ESM and AlBert), we set $max\_fuse\_depth = 2$, as we observed a large overhead with a larger max_fuse_depth. $insert\_depth$ and $insert\_width$ are set to 3 and 2 respectively, which generate sufficient (up to 27) input-dependent control flows for each obfuscated function (as discussed in Section 4.2).

All experiments are run on a machine with an Intel Xeon Gold 6258R CPU, 1024GB memory, and 8TB HDD, running Ubuntu 22.04. In the rest of this section, we use `ff`, `fi`, `cr` to represent Flexible Operator Fusion, Fake Operator Insertion, and Operator Computation Reordering respectively (note that Fake Operator Insertion depends on Flexible Operator Fusion, thus `fi` implies `ff+fi`). Without additional explanation, `all` denotes applying all three *NeuroShield* primitives.

6.1 Model Semantics Preservation

Fake Operator Insertion (`fi`) uses the profiled neuron range $[l, h]$ over the training dataset to construct fake control flows. This introduces potential model functionality inconsistency as unseen inputs may trigger neuron activation values outside the profiled range. To evaluate how *NeuroShield* preserves the original functionality of DNN models, we report the model inference accuracies over the training and testing datasets. For the testing dataset, we compute the percentage of data samples that trigger out-of-range neuron activation, and the smallest $\frac{\delta}{h-l}$ such that expanding the range to $[l-\delta, h+\delta]$ can enclose all activation values. For each model, we also repeat the out-of-range sample analysis on an external dataset that has a similar distribution to the training/testing data. For large external datasets with over 100K samples, we randomly pick 100K samples for evaluation. The corresponding results are shown in Table 3.

Training/Testing Data. *NeuroShield* incurs no out-of-range neuron activation and accuracy loss for the training data that are used for neuron range profiling. For the testing dataset, ESM shows the largest accuracy loss (3.35%), out-of-range rate (4.04%), and required range expansion percentage (26.55%), because its relatively small training dataset fails to adequately cover the input distribution. The other five models have much smaller accuracy loss (0.01% - 0.46%), out-of-range rate (0.01% - 0.46%) and range expansion percentage (0.18% - 16.79%).

External Data. CV models (Mnist, Resnet and Mobilenet) exhibit a small out-of-range rate (0 - 0.20%) over external datasets, while NLP models (FastText, ESM and AlBert) show a higher rate (2.48% - 4.82%). The reason is that different text corpora differ widely in vocabulary, syntax, and sequence length — yielding a much larger input space than images. Such a divergence in external text datasets results in a higher likelihood of out-of-range neuron activation values [79].

Discussion. When applying Fake Operator Insertion (`fi`), we recommend expanding the profiled activation range by about 50% (2x the observed 26.55%) to preserve the model functionality. Since NLP models are more prone to out-of-range activations on unseen data, we recommend training and profiling them with a broader, more representative corpus [22]. Overall, *NeuroShield* preserves most of the semantics for the evaluated DNN models.

6.2 Obfuscation Overhead

In this section, we evaluate the overhead of *NeuroShield* and general obfuscators. Let m_{obf} and m_{ori} be the corresponding metrics (i.e., binary size, execution time, memory usage size) for obfuscated binaries and the original binaries compiled by TVM, respectively. We report the results in times overhead m_{obf}/m_{ori}.

Binary Size. Figure 5 reports the binary size overhead in times (log scale). Compared to other obfuscation schemes, VM-based obfuscation (Tigress, Loki) introduces a significantly large overhead. Tigress incurs at most 47x overhead, with an average of over 23x overhead among six tested models. The overhead for Loki is even higher (up to 188x with an average of around 103x), because it extensively employs Mixed Boolean Arithmetic (MBA) in addition to Virtualization. In comparison, all other non-VM-based obfuscations introduce an average times overhead under 2x.

For non-VM-based obfuscations, Fusor introduces the least binary size overhead (0.53% - 9.63%) as it only inserts opaque predicates for bogus control flow construction. OLLVM has 6.71% - 15.28% overhead for small models (Mnist and FastText), and up to 141.63% overhead for large models. The main binary size overhead for OLLVM comes from the Control Flow Flattening (CFF) obfuscation.

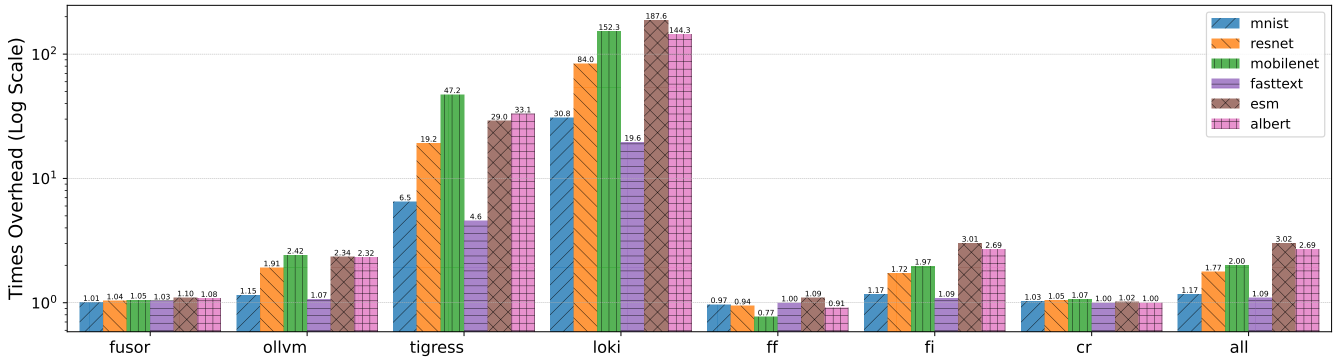


Figure 5: Binary size overhead for each obfuscator (log scale)

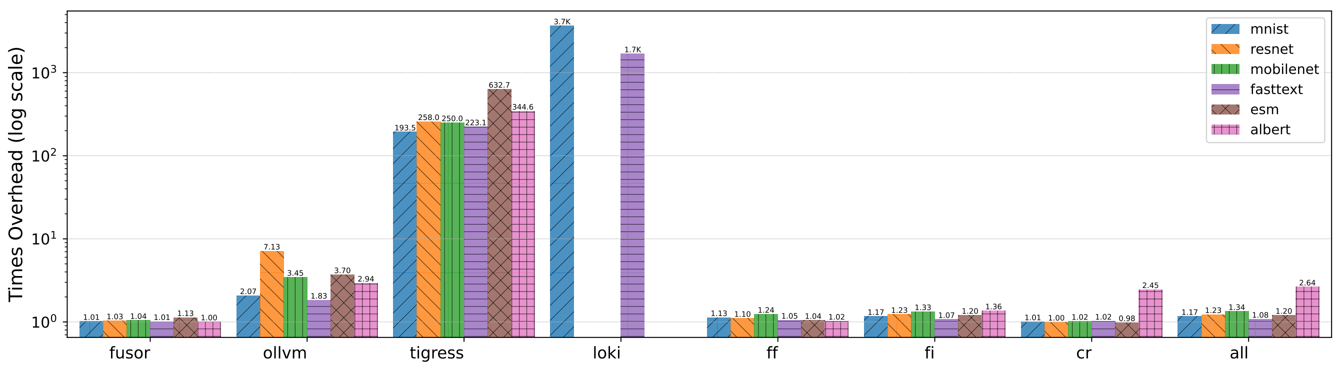


Figure 6: Execution time overhead for each obfuscator (log scale)

The three obfuscation primitives in *NeuroShield* contribute to binary size overhead differently. Flexible Operator Fusion (`ff`) can generally reduce the binary size by employing aggressive operator fusion, which results in a smaller number of compiled functions. Apart from the ESM model, it reduces the binary size from 0.29% to 23.09% for the rest of the five models. The binary size for ESM increases by 9.44%, where flexible operator fusion cannot be effectively applied due to its complicated neural network structure. Operator Computation Reordering (`cr`) has a marginal overhead from 0 to 6.54% as it only switches some nested loop order in binaries. The main binary size overhead of *NeuroShield* comes from Fake Operator Insertion (`fi`) which inserts additional fake operator code and branch conditions into the binaries. Overall, *NeuroShield* has a comparable binary size overhead to OLLVM — the overhead for CV models (16.85% to 100.35%) is slightly smaller, while the overhead for NLP models is larger (9.31% to 201.83%).

Execution Time. We use the single input batch inference time as the execution time for DNN binaries. Values are averaged among 100 random samples for the corresponding datasets for each model. Figure 6 presents the execution time overhead in log scale. Similar to the binary size overhead, the execution time shows a significantly higher overhead for VM-based obfuscators. Tigress results in 194x-633x (with an average of 317x) higher execution time for all six models. Loki presents a much higher execution time for Mnist (over 1700x) and FastText (over 3700x). For other larger models, the binaries cannot finish the execution — the process was killed after running for a few hours because of running out of stack space (stack size limit is set to 1GB using `ulimit -s`). Compared to VM-based obfuscators, other obfuscators exhibit averaged overhead smaller than 3.52x.

Regarding non-VM-based obfuscations, Fusor has a small overhead from 0.24% to 12.77%. OLLVM introduces 82.9% and 107.4% overhead for small models Mnist and FastText respectively, but incurs much higher overhead (up to 7.13x) for large models. As for *NeuroShield*, Operator Computation Reordering (`cr`) has minimal overhead from -1.78% to 2.33% for all five models except AlBert. The overhead for AlBert is 145.2% as the computation reordering for Albert operators causes a much worse cache locality. We thus only applied `ff` and `fi` to Albert as the final *NeuroShield* obfuscation output. For CV models, the major overhead comes from Flexible Operator Fusion (`ff`), which introduces a 10.49% - 23.67% overhead. Fake Operator Insertion (`fi`) increases the overhead to 16.52% - 33.01%. On the other hand, Fake Operator Insertion (`fi`) contributes more to the execution time overhead for NLP models. Flexible Operator Fusion (`ff`) has a base of 2.28%-5.04% overhead, and Fake Operator Insertion (`fi`) increases it to 7.13% - 36.05%. The reason is that CV models are structurally simpler, which gives better opportunities for aggressive fusion. As a result, it is harder for subsequent compilation to optimize the complicated functions that contain the semantics of more fused operators. When all three primitives are applied (except for AlBert to which only `ff` and `fi` are applied), *NeuroShield* introduces 7.8% to 36.1% overhead, which is much smaller than OLLVM.

Table 4: Memory size overhead

model	param. size (bytes)			workspace size (bytes)		
	original	*NeuroShield*	overhead	original	*NeuroShield*	overhead
mnist	23,984	24,132	0.617%	54,272	61,080	12.54%
resnet	44,676,648	44,677,236	0.00132%	1,048,576	1,310,720	25.00%
mobilenet	13,951,488	13,952,788	0.00932%	11,239,424	13,396,992	19.20%
fasttext	84,473,024	84,473,104	0.0000947%	114,000	117,300	2.89%
esm	30,082,056	30,086,404	0.0145%	6,063,616	6,521,224	7.55%
albert	46,550,052	46,560,612	0.0227%	5,189,632	7,376,828	42.15%

Memory Size. Compared to general obfuscators, *NeuroShield* introduces additional memory overhead due to Fake Operator Insertion (`fi`). Firstly, we need more parameters to store ranges and indices for neurons that are used to construct fake operators. Note that the parameters for the fake operators do not introduce additional overhead because we reuse the parameters for true operators as mentioned in Section 4.2. Secondly, the output size of fake operators may not exactly match the size of true operators. This may result in more workspace memory usage because we need to make sure the output size of a fake operator is not less than the output size of its corresponding true operator.

Table 4 shows the overhead of parameter size and workspace memory size for *NeuroShield*. It results in a marginal overhead (at most 0.62%) because only a single neuron in an operator is needed to construct fake control flows. The neuron ranges and index size are negligible compared to the parameter size of the whole model. The majority of additional memory comes from the increase of workspace size, with an overhead of 2.89% - 42.15%.

6.3 Resilience against Existing Reverse Engineering Attacks

To compare the effectiveness of general obfuscators and *NeuroShield* on DNN binaries, we evaluate them against two state-of-the-art DNN decompilers, DnD and BTD. Note that the BTD attack is the extended version we mentioned in Section 3.2. Table 5 reports the attack time and the number of operator functions reconstructed by DnD and BTD on compiled DNN models under different obfuscation techniques. We group the results for three CV models (Mnist, Resnet, Mobilenet) and NLP models (FastText, ESM, Albert) together for better presentation. We only evaluate CV models for DnD as it only supports CV models in its evaluation dataset. For Loki's obfuscated models, only Mnist and FastText can finish execution. As BTD is based on dynamic analysis, we only evaluate the two models under Loki's obfuscation for BTD.

The upper part of Table 5 presents the attack results for DnD. Among general obfuscators, Fusor is ineffective against DnD with 100% operator functions semantics being recovered. The reason is that the inserted opaque predicates do not change the overall nested loop structures, which is the fundamental information leveraged by DnD to infer the operator types and attributes. Other three general obfuscators acheive full protection against DnD, as they drastically change the control flow structures of binaries. For Tigress and Loki,

Table 5: Evaluation on existing DNN binary reverse engineering attacks. For the BTD attack, we use a thread pool with 20 threads to parallelize the attack for all functions in each obfuscator-model combination.

attack	obfuscation	model(s)	func reconstruction			
			#func	#recon.	percent	time(s)
DnD	original	CV	82	82	100.00%	8023.99
	fusor	CV	82	82	100.00%	10349.70
	ollvm	CV	82	0	0.00%	161859.85
	tigress	CV	82	0	0.00%	31.89
	loki	CV	82	0	0.00%	27.33
	cr	CV	82	65	79.27%	7344.36
	ff	CV	33	4	12.12%	11693.53
	fi	CV	33	0	0.00%	11156.50
	all	CV	33	0	0.00%	13820.01
BTD	original	CV	82	82	100.00%	**2307.96**
		NLP	464	464	100.00%	**6134.82**
	fusor	CV	82	82	100.00%	2794.23
		NLP	464	464	100.00%	7063.31
	ollvm	CV	82	82	100.00%	38688.70
		NLP	464	464	100.00%	54949.21
	tigress	CV	82	82	100.00%	578763.65
		NLP	464	440	94.83%	441964.83
	loki	Mnist	5	5	100.00%	12595.44
		FastText	5	5	100.00%	3145.58
	cr	CV	82	82	100.00%	**6321.34**
		NLP	464	464	100.00%	**25778.41**
	ff	CV	33	4	12.12%	2498.10
		NLP	426	265	62.21%	3782.66
	fi	CV	33	1	3.03%	2619.58
		NLP	426	201	47.18%	5620.18
	all	CV	33	1	3.03%	6153.58
		NLP	426	201	47.18%	16701.23

the attack stopped in about 30s because DnD failed to retrieve the control flow graph of binaries under VM-based obfuscation.

Regarding *NeuroShield*, 17 operators after Operator Computation Reordering (`cr`) cause DnD to mispredict their attributes, resulting in a 79.27% recovery rate. Under Flexible Operator Fusion (`ff`), only 4 operators are successfully recovered by DnD. They are operators related to shortcut structures in Resnet and Mobilenet that cannot be fused with other operators to form complicated semantics. Fake Operator Insertion (`fi`) further protects these operators by introducing additional fake operator semantics, which alters the main control flow structures in the binaries.

The lower section of Table 5 summarizes the results of the BTD attack. Obfuscators that change the static control flows (OLLVM, Tigress and Loki) are not resilient against BTD, as BTD is based on dynamic analysis. As a result, BTD successfully recovers all operators except for 24 operators in the Tigress-obfuscated transformers (ESM and AlBert), where the dynamic instruction log size exceeds our 1TB limit. For the result of Fusor, although it introduces additional input-dependent control flows, they are based on unsatisfiable opaque predicates. As a result, BTD can still recover 100% function semantics because any random input can trigger the true execution path.

For *NeuroShield*, Operator Computation Reordering (`cr`) is not resilient against BTD, as it only changes the static loop structures.

Table 6: Function path resolving results. We use a thread pool with 20 threads to parallelize the attack for all functions in each obfuscator-model combination.

obfuscation	model	#func	#resol.	percent	time(s)
fusor	mnist	5	5	100.00%	4351.24
	resnet	22	17	77.27%	432000.00
	mobilenet	55	55	100.00%	186643.39
	fasttext	5	5	100.00%	183.92
NeuroShield	mnist	2	0	0.00%	432000.00
	resnet	12	1	8.33%	432000.00
	mobilenet	19	0	0.00%	432000.00
	fasttext	1	0	0.00%	432000.00

However, the time required to recover the function semantics increases by around 3x for CV (from 2307.96s to 6321.34s) and 4x for NLP models (6134.82 to 25778.41). This is because `cr` reorder the nested loop structure in a way that the computation is evenly distributed across all output elements, thus BTD needs to log more instructions until the final computation on an output element is finished. Operator Flexible Fusion (`ff`) exhibits markedly different function recovery rates on CV versus NLP models. The recovered functions are related to the connection structure in neural networks. They either combine the results from multiple operators (e.g., `Add` or `Multiply` of two outputs from upstream), or appear in a shortcut path with simple semantics (e.g., a single `Conv2D` in a CV model's residual connection). They are hard to be fused to other operators to form complicated semantics. The recovery rate for NLP models is much higher (62.21% vs. 12.12%) because the neural network structures for NLP models are more complicated. Fake Operator Insertion (`fi`) further lowers the recovery rates to 47.18% and 3.03% respectively, by inserting fake operators and causing the random input from BTD to trigger wrong execution paths. When `all` *NeuroShield*'s obfuscation primitives are applied, the operator recovery rate is the same as `fi`, but the attack time increases by around 3x because `cr` causes BTD to log and analyze more instructions.

6.4 Strength of Inserted Control Flow

To demonstrate the resilience of the input-dependent control flow introduced by *NeuroShield*, we measure how long symbolic execution takes to resolve the constraints added by Fusor and Fake Operator Insertion (`fi`). We firstly use a random input to identify the input and output buffer of a function dynamically (similar to BTD), mark the elements in the input buffer as symbolic, and run symbolic execution using angr [2] for 5 days. Table 6 reports, for each of the four models (Mnist, Resnet, Mobilenet, and FastText), the number of functions whose execution paths were completely resolved. With Fusor obfuscation, constraints are successfully resolved for all functions except 5 in ResNet. In contrast, angr only fully resolves the constraints of a single function under Fake Operator Insertion (`fi`). This is because Fusor's floating-point opaque predicates rely on simple equality checks (e.g., "`fv1==fv2`"), making their true branches trivial to detect and prune. Conversely, Fake Operator Insertion (`fi`) generates its constraints using floating-point less-than comparisons, resulting in two satisfiable branches.

7 Discussion

Extension to other DNN Compilers. We implement *NeuroShield* on TVM, which provides an expressive enough, multi-level IR infrastructure (Relay IR and TIR) that we can operate on to encode our obfuscation schemes. However, the design of multi-level IR abstraction for DNN compilation is not unique to TVM. For instance, OpenXLA [51] adopts a similar Graph-Level IR (StableHLO [59]) for computational graph representation, and a low-level IR (the affine dialect [65] of MLIR [30]) for representing nested loop implementation of operators. We believe our obfuscation schemes can be extended to such DNN compilers with similar IR abstractions.

Difficulties of valid input inferring. One potential strategy to circumvent the Fake Operator Insertion obfuscation scheme is to generate valid model inputs, which trigger the true execution path of the obfuscated operator function. However, attackers rarely have detailed knowledge of the input distribution due to data privacy concerns — model users usually perform the input preprocessing (eg., normalization/tokenization) and even execute the first few layers of inference locally [10, 33, 38], before sending the data out for DNN inference. In these scenarios, attackers cannot easily figure out the valid input distribution due to the lack of knowledge about the model's normalization parameters/tokenization schemes, or visibility into the upstream operators. Consequently, unless attackers can first reconstruct the model's architecture through binary analysis, inferring valid inputs via black-box queries remains significantly more challenging compared to white-box scenarios, where the model architecture is known [58, 81].

Strength of proposed obfuscation schemes. As discussed in Section 4.2, Flexible Operator Fusion (`ff`) and Fake Operator Insertion (`fi`) construct comparatively large obfuscated operator functions, which consist of the semantics of around 10 operators and up to 27 satisfiable execution paths. Directly recovering the semantic of the whole function by dynamically observing input/output samples (e.g., through program synthesis [7, 31]) is challenging given the large tensor input/output size and complicated fused operator semantics. Another potential attack against `ff` and `fi` involves extracting individual operator's sub-routines from the fused functions. However, separating sub-routines without explicit function boundaries information is a well-known hard problem for both static and dynamic analysis [14, 16, 80]. Existing works [6, 16] rely on binary matching techniques that target well-known library functions (e.g., libc), while DNN operators are domain-specific and typically do not resemble standard inline functions commonly found in traditional binaries.

In this paper, we use loop reordering (`cr`) to evenly distribute computation across the output elements of each operator, making it more difficult for existing DNN binary decompilers (DnD and BTD) to quickly infer the semantics of a single output. Experiments in Section 6.3 show that `cr` alone is not resilient against existing reverse engineering attacks although it can increase the attack time. Evaluating the resilience of other loop transformations (e.g., unrolling, tiling) remains an open direction for future work.

Obfuscation Overhead. Although *NeuroShield* introduces a modest execution time overhead (7.8% - 36.1%), its binary size overhead can be up to 201.8%. Prior work has shown that protecting a small subset of critical DNN operators can protect essential model functionality with significantly lower overhead [39]. Since the primary source of *NeuroShield*'s binary size overhead is Fake Operator Insertion, this cost can be mitigated by applying `fi` only to fused functions containing such critical operators.

Obfuscation of Complicated DNN Structures. A comparatively large portion of operators in complex neural network structures (e.g., 47.18% operators in transformers) remain unprotected by *NeuroShield*. These operators are related to connection structures and can not be easily fused. Extending Flexible Operator Fusion to support a wider range of fusion patterns is a promising direction that we leave for future work.

8 Related Work

DNN Binary Reverse Engineering. Attackers can reverse engineer DNN model structures from DNN binaries through static analysis [74] or dynamic analysis [35]. DnD [74] utilizes the static control flow information in DNN binaries to reconstruct the loop structures of DNN operators, which are then lifted to a customized AST representation to recover the high-level semantics of DNN operators. BTD [35] performs symbolic execution on dynamic execution traces of DNN operators to reconstruct their semantics.

End-to-end DNN model fidelity attacks. Another category of DNN model stealing attack targets the model's high-level functionality by treating it as a black box, e.g., querying it to train a surrogate model [24, 67]. Fake Operator Insertion provides a certain extent of protection for this kind of attack — only valid input will trigger the model's true behavior. Attackers could still potentially train a high-fidelity surrogate model given sufficiently extensive queries, but with higher computational resources for model training.

Code Obfuscation. Code Obfuscation is a technique used to make software binaries harder to understand and analyze. This is achieved by altering the binary code in a way that preserves its original functionality but conceals its logic and intent from human readers or automated reverse engineering attacks. General obfuscation schemes include Opaque Predicates [11, 42, 52, 75], Control Flow Flattening [8, 70], Virtualization [1, 4, 9, 19, 27, 28, 32, 56, 61, 71, 72, 76], and Mixed Boolean Arithmetic [56, 82]. These obfuscation schemes are ineffective against existing reverse engineering attacks on DNN binaries due to certain unique characteristics of DNN binaries.

DNN Structure Obfuscation. DNN Structure Obfuscation alters the neural network structures without changing their functional semantics. Common transformations include: (1) decomposing a single operators into multiple smaller operators that together perform the same computation; (2) Inserting pairs of redundant operators that can cancel out the effects of each other. DNN Structure Obfuscation can be used for model configuration file obfuscation [81], watermark removal [78] and defend against side-channel-based model extraction attacks [34]. These constructions can also be used for DNN binary obfuscation. However, the attacker can still recover a DNN model that has the same functionality, but with more complicated structures compared to the original DNN model. In contrast, *NeuroShield* hardens DNN binaries in a way that makes DNN binary reverse engineering attacks ineffective.

Hardware-based Model Protection. Hardware-based methods offer protection for DNN models through approaches such as: (1) using hardware-secured secret keys to obfuscate model weights [21, 40, 62]; (2) executing DNN models within Trusted Execution Environments (TEEs) [39, 41]. These approaches require specific hardware support, while *NeuroShield* uses binary obfuscation to protect model structures and is more generic.

9 Summary

We present *NeuroShield*, an implementation of three distinct obfuscation techniques tailored for DNN binaries: (1) Flexible Operator Fusion, (2) Fake Operator Insertion, and (3) Operator Computation Reordering. Experiments show that *NeuroShield* provides strong protection against existing reverse engineering attacks while introducing a reasonable overhead.

References

[1] Bertrand Anckaert, Mariusz H. Jakubowski, and Ramarathnam Venkatesan. 2006. Proteus: virtualization for diversified tamper-resistance. In *Proceedings of the Sixth ACM Workshop on Digital Rights Management, Alexandria, VA, USA, October 30, 2006*, Moti Yung, Kaoru Kurosawa, and Reihaneh Safavi-Naini (Eds.). ACM, 47–58. doi:10.1145/1179509.1179521

[2] Angr. [n. d.]. https://github.com/angr.

[3] Apache TVM. [n. d.]. https://tvm.apache.org/.

[4] Amir Averbuch, Michael Kiperberg, and Nezer Jacob Zaidenberg. 2013. Truly-Protect: An Efficient VM-Based Software Protection. *IEEE Syst. J.* 7, 3 (2013), 455–466. doi:10.1109/JSYST.2013.2260617

[5] Sébastien Bardin, Robin David, and Jean-Yves Marion. 2017. Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 633–651. doi:10.1109/SP.2017.36

[6] BinDiff. [n. d.]. https://github.com/google/bindiff.

[7] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 643–659. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/blazytko

[8] Jan Cappaert and Bart Preneel. 2010. A general model for hiding control flow. In *Proceedings of the 10th ACM Workshop on Digital Rights Management, Chicago, Illinois, USA, October 4, 2010*, Ehab Al-Shaer, Hongxia Jin, and Marc Joye (Eds.). ACM, 35–42. doi:10.1145/1866870.1866877

[9] Xiaoyang Cheng, Yan Lin, Debin Gao, and Chunfu Jia. 2019. DynOpVm: VM-Based Software Obfuscation with Dynamic Opcode Mapping. In *Applied Cryptography and Network Security - 17th International Conference, ACNS 2019, Bogota, Colombia, June 5-7, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11464)*, Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung (Eds.). Springer, 155–174. doi:10.1007/978-3-030-21568-2_8

[10] Hao-Jen Chien, Hossein Khalili, Amin Hass, and Nader Sehatbakhsh. 2023. Enc2: Privacy-Preserving Inference for Tiny IoTs via Encoding and Encryption. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking, ACM MobiCom 2023, Madrid, Spain, October 2-6, 2023*, Xavier Costa-Pérez, Joerg Widmer, Diego Perino, Domenico Giustiniano, Haitham Al-Hassanieh, Arash Asadi, and Landon P. Cox (Eds.). ACM, 35:1–35:16. doi:10.1145/3570361.3592501

[11] Christian S. Collberg, Clark D. Thomborson, and Douglas Low. 1998. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, David B. MacQueen and Luca Cardelli (Eds.). ACM, 184–196. doi:10.1145/268946.268962

[12] MLC LLM: Universal LLM Deployment Engine With ML Compilation. [n. d.]. https://llm.mlc.ai.

[13] Saumya K. Debray and Jay Patel. 2010. Reverse Engineering Self-Modifying Code: Unpacker Extraction. In *17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA*, Giuliano Antoniol, Martin Pinzger, and Elliot J. Chikofsky (Eds.). IEEE Computer Society, 131–140. doi:10.1109/WCRE.2010.22

[14] Florian Deissenboeck, Lars Heinemann, Benjamin Hummel, and Stefan Wagner. 2012. Challenges of the Dynamic Detection of Functionally Similar Code Fragments. In *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012*, Tom Mens, Anthony Cleve, and Rudolf Ferenc (Eds.). IEEE Computer Society, 299–308. doi:10.1109/CSMR.2012.38

[15] Zizhuang Deng, Kai Chen, Guozhu Meng, Xiaodong Zhang, Ke Xu, and Yao Cheng. 2022. Understanding Real-world Threats to Deep Learning Models in Android Apps. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 785–799. doi:10.1145/3548606.3559388

[16] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 303–317. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/egele

[17] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Efficient Multi-Objective Neural Architecture Search via Lamarckian Evolution. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=ByME42AqK7

[18] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural Architecture Search: A Survey. *J. Mach. Learn. Res.* 20 (2019), 55:1–55:21. http://jmlr.org/papers/v20/18-598.html

[19] Hui Fang, Yongdong Wu, Shuhong Wang, and Yin Huang. 2011. Multi-stage Binary Code Obfuscation Using Improved Virtual Machine. In *Information Security, 14th International Conference, ISC 2011, Xi'an, China, October 26-29, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 7001)*, Xuejia Lai, Jianying Zhou, and Hui Li (Eds.). Springer, 168–181. doi:10.1007/978-3-642-24861-0_12

[20] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. 2023. TensorIR: An Abstraction for Automatic Tensorized Program Optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 804–817. doi:10.1145/3575693.3576933

[21] Brunno F. Goldstein, Vinay C. Patil, Victor da Cruz Ferreira, Alexandre Solon Nery, Felipe M. G. França, and Sandip Kundu. 2021. Preventing DNN Model IP Theft via Hardware Obfuscation. *IEEE J. Emerg. Sel. Topics Circuits Syst.* 11, 2 (2021), 267–277. doi:10.1109/JETCAS.2021.3076151

[22] Suchin Gururangan, Ana Marasovic, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A. Smith. 2020. Don't Stop Pretraining: Adapt Language Models to Domains and Tasks. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (Eds.). Association for Computational Linguistics, 8342–8360. doi:10.18653/V1/2020.ACL-MAIN.740

[23] IDA Pro. [n. d.]. https://hex-rays.com/ida-pro/.

[24] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. 2020. High Accuracy and High Fidelity Extraction of Neural Networks. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 1345–1362. https://www.usenix.org/conference/usenixsecurity20/presentation/jagielski

[25] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM - Software Protection for the Masses. In *1st IEEE/ACM International Workshop on Software Protection, SPRO 2015, Florence, Italy, May 19, 2015*, Paolo Falcarin and Brecht Wyseur (Eds.). IEEE Computer Society, 3–9. doi:10.1109/SPRO.2015.10

[26] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. 2007. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*. 46–53.

[27] Kaiyuan Kuang, Zhanyong Tang, Xiaoqing Gong, Dingyi Fang, Xiaojiang Chen, and Zheng Wang. 2018. Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling. *Comput. Secur.* 74 (2018), 202–220. doi:10.1016/J.COSE.2018.01.008

[28] Kaiyuan Kuang, Zhanyong Tang, Xiaoqing Gong, Dingyi Fang, Xiaojiang Chen, Heng Zhang, Jie Liu, and Zheng Wang. 2017. Exploit dynamic data flows to protect software against semantic attacks. In *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation, SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI 2017, San Francisco, CA, USA, August 4-8, 2017*. IEEE, 1–6. doi:10.1109/UIC-ATC.2017.8397540

[29] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=H1eA7AEtvS

[30] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 2–14. doi:10.1109/CGO51591.2021.9370308

[31] Jaehyung Lee and Woosuk Lee. 2023. Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Term Rewriting. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM, 2351–2365. doi:10.1145/3576915.3623186

[32] Jae-Yung Lee, Jae Hyuk Suk, and Dong Hoon Lee. 2018. VODKA: Virtualization Obfuscation Using Dynamic Key Approach. In *Information Security Applications - 19th International Conference, WISA 2018, Jeju Island, Korea, August 23-25, 2018, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 11402)*, Brent ByungHoon Kang and Jin Soo Jang (Eds.). Springer, 131–145. doi:10.1007/978-3-030-17982-3_11

[33] Ang Li, Jiayi Guo, Huanrui Yang, Flora D. Salim, and Yiran Chen. 2021. DeepObfuscator: Obfuscating Intermediate Representations with Privacy-Preserving Adversarial Learning on Smartphones. In *IoTDI '21: International Conference on Internet-of-Things Design and Implementation, Virtual Event / Charlottesville, VA, USA, May 18-21, 2021*. ACM, 28–39. doi:10.1145/3450268.3453519

[34] Jingtao Li, Zhezhi He, Adnan Siraj Rakin, Deliang Fan, and Chaitali Chakrabarti. 2021. NeurObfuscator: A Full-stack Obfuscation Tool to Mitigate Neural Architecture Stealing. In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2021, Tysons Corner, VA, USA, December 12-15, 2021*. IEEE, 248–258. doi:10.1109/HOST49136.2021.9702279

[35] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, Xiaofei Xie, and Lei Ma. 2023. Decompiling x86 Deep Neural Network Executables. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association. https://www.usenix.org/conference/usenixsecurity23/presentation/liu-zhibo

[36] LLVM C backend. [n. d.]. https://github.com/JuliaHubOSS/llvm-cbe.

[37] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. 2018. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 120–131.

[38] Peihua Mai, Ran Yan, Zhe Huang, Youjia Yang, and Yan Pang. 2024. Split-and-Denoise: Protect large language model inference with local differential privacy. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net. https://openreview.net/forum?id=bZ4fzw1iz7

[39] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Soteris Demetriou, Ilias Leontiadis, Andrea Cavallaro, and Hamed Haddadi. 2020. DarkneTZ: towards model privacy at the edge using trusted execution environments. In *MobiSys '20: The 18th Annual International Conference on Mobile Systems, Applications, and Services, Toronto, Ontario, Canada, June 15-19, 2020*, Eyal de Lara, Iqbal Mohomed, Jason Nieh, and Elizabeth M. Belding (Eds.). ACM, 161–174. doi:10.1145/3386901.3388946

[40] Alireza Mohseni, Mohammad Hossein Moaiyeri, Abdolah Amirany, and Mohammad Hadi Rezayati. 2024. Protecting the Intellectual Property of Binary Deep Neural Networks With Efficient Spintronic-Based Hardware Obfuscation. *IEEE Trans. Circuits Syst. I Regul. Pap.* 71, 7 (2024), 3146–3156. doi:10.1109/TCSI.2024.3397925

[41] Myungsuk Moon, Minhee Kim, Joonkyo Jung, and Dokyung Song. 2025. ASGARD: Protecting On-Device Deep Neural Networks with Virtualization-Based Trusted Execution Environments. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/asgard-protecting-on-device-deep-neural-networks-with-virtualization-based-trusted-execution-environments/

[42] Fukutomo Nakanishi, Giulio De Pasquale, Daniele Ferla, and Lorenzo Cavallaro. 2020. Intertwining ROP Gadgets and Opaque Predicates for Robust Obfuscation. *CoRR* abs/2012.09163 (2020). arXiv:2012.09163 https://arxiv.org/abs/2012.09163

[43] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 58. doi:10.1145/3458817.3476209

[44] Amazon SageMaker Neo. [n. d.]. https://aws.amazon.com/sagemaker/neo.

[45] O-MVLL. [n. d.]. https://github.com/open-obfuscator/o-mvll.

[46] The Tigress C Obfuscator. [n. d.]. https://tigress.wtf.

[47] OctoAI. [n. d.]. https://github.com/octoml.

[48] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. 2019. How to kill symbolic deobfuscation for free (or: unleashing the potential of path-oriented protections). In *Proceedings of the 35th Annual Computer Security*

Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019, David Balenson (Ed.). ACM, 177–189. doi:10.1145/3359789.3359812
[49] On-device Gemini Nano in Chrome. [n. d.]. https://developer.chrome.com/docs/ai.
[50] Open Neural Network Exchange. [n. d.]. https://onnx.ai/.
[51] OpenXLA. [n. d.]. https://openxla.org/.
[52] Andre Pawlowski, Moritz Contag, and Thorsten Holz. 2016. Probfuscation: An Obfuscation Approach Using Probabilistic Control Flows. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9721)*, Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez (Eds.). Springer, 165–185. doi:10.1007/978-3-319-40667-1_9
[53] PyTorch. [n. d.]. https://pytorch.org/.
[54] Alexander Rives, Joshua Meier, Tom Sercu, Siddharth Goyal, Zeming Lin, Jason Liu, Demi Guo, Myle Ott, C. Lawrence Zitnick, Jerry Ma, and Rob Fergus. 2021. Biological structure and function emerge from scaling unsupervised learning to 250 million protein sequences. *Proc. Natl. Acad. Sci. USA* 118, 15 (2021), e2016239118. doi:10.1073/PNAS.2016239118
[55] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. *CoRR* abs/1805.00907 (2018). arXiv:1805.00907 http://arxiv.org/abs/1805.00907
[56] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. 2022. Loki: Hardening Code Obfuscation Against Automated Attacks. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 3055–3073. https://www.usenix.org/conference/usenixsecurity22/presentation/schloegel
[57] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *31st IEEE Symposium on Security and Privacy, SP 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 317–331. doi:10.1109/SP.2010.26
[58] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership Inference Attacks Against Machine Learning Models. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 3–18. doi:10.1109/SP.2017.41
[59] StableHLO. [n. d.]. https://github.com/openxla/stablehlo.
[60] Relay Operator Strategy. [n. d.]. https://tvm.apache.org/docs/arch/relay_op_strategy.html.
[61] Jae Hyuk Suk and Dong Hoon Lee. 2020. VCF: Virtual Code Folding to Enhance Virtualization Obfuscation. *IEEE Access* 8 (2020), 139161–139175. doi:10.1109/ACCESS.2020.3012684
[62] Yulian Sun, Vedant Bonde, Li Duan, and Yong Li. 2025. Obfuscation for Deep Neural Networks Against Model Extraction: Attack Taxonomy and Defense Optimization. In *Applied Cryptography and Network Security - 23rd International Conference, ACNS 2025, Munich, Germany, June 23-26, 2025, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 15827)*, Marc Fischlin and Veelasha Moonsamy (Eds.). Springer, 391–414. doi:10.1007/978-3-031-95767-3_15
[63] Zhichuang Sun, Ruimin Sun, Long Lu, and Alan Mislove. 2021. Mind Your Weight(s): A Large-scale Study on Insufficient Machine Learning Model Protection in Mobile Apps. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael D. Bailey and Rachel Greenstadt (Eds.). USENIX Association, 1955–1972. https://www.usenix.org/conference/usenixsecurity21/presentation/sun-zhichuang
[64] TensorFlow Lite. [n. d.]. https://www.tensorflow.org/lite/guide/.
[65] The MLIR Affine Dialect. [n. d.]. https://mlir.llvm.org/docs/Dialects/Affine/.
[66] Themida. [n. d.]. https://oreans.com/themida.php.
[67] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2016. Stealing machine learning models via prediction {APIs}. In *25th USENIX security symposium (USENIX Security 16)*. 601–618.
[68] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo García Bringas. 2015. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 659–673. doi:10.1109/SP.2015.46
[69] VMProtect. [n. d.]. https://vmpsoft.com/products/vmprotect.
[70] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. 2000. Software tamper resistance: Obstructing static analysis of programs. (2000).
[71] Huaijun Wang, Dingyi Fang, Guanghui Li, Na An, Xiaojiang Chen, and Yuanxiang Gu. 2014. TDVMP: Improved Virtual Machine-Based Software Protection with Time Diversity. In *Proceedings of the 3rd ACM SIGPLAN Program Protection and Reverse Engineering Workshop 2014, PPREW 2014, January 25, 2014, San Diego, CA, USA*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 4:1–4:9. doi:10.1145/2556464.2556468
[72] Huaijun Wang, Dingyi Fang, Guanghui Li, Xiaoyan Yin, Bo Zhang, and Yuanxiang Gu. 2013. NISLVMP: Improved Virtual Machine-Based Software Protection. In *Ninth International Conference on Computational Intelligence and Security, CIS 2013, Emei Mountain, Sichan Province, China, December 14-15, 2013*. IEEE Computer Society, 479–483. doi:10.1109/CIS.2013.107
[73] Qingfeng Wang, Hao Liang, Yawen Wang, Genlin Xie, and Benwei He. 2022. Design and Implementation of Code Obfuscation Based on Floating-Point Operations. In *2022 IEEE 8th International Conference on Computer and Communications (ICCC)*. 1646–1650. doi:10.1109/ICCC56324.2022.10065748
[74] Ruoyu Wu, Taegyu Kim, Dave (Jing) Tian, Antonio Bianchi, and Dongyan Xu. 2022. DnD: A Cross-Architecture Deep Neural Network Decompiler. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 2135–2152. https://www.usenix.org/conference/usenixsecurity22/presentation/wu-ruoyu
[75] Hui Xu, Yangfan Zhou, Yu Kang, Fengzhi Tu, and Michael R. Lyu. 2018. Manufacturing Resilient Bi-Opaque Predicates Against Symbolic Execution. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*. IEEE Computer Society, 666–677. doi:10.1109/DSN.2018.00073
[76] Chao Xue, Zhanyong Tang, Guixin Ye, Guanghui Li, Xiaoqing Gong, Wei Wangg, Dingyi Fang, and Zheng Wang. 2018. Exploiting Code Diversity to Enhance Code Virtualization Protection. In *24th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2018, Singapore, December 11-13, 2018*. IEEE, 620–627. doi:10.1109/PADSW.2018.8644535
[77] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 674–691. doi:10.1109/SP.2015.47
[78] Yifan Yan, Xudong Pan, Mi Zhang, and Min Yang. 2023. Rethinking White-Box Watermarks on Deep Learning Models under Neural Structural Obfuscation. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 2347–2364. https://www.usenix.org/conference/usenixsecurity23/presentation/yan
[79] Linyi Yang, Yaoxian Song, Xuan Ren, Chenyang Lyu, Yidong Wang, Jingming Zhuo, Lingqiao Liu, Jindong Wang, Jennifer Foster, and Yue Zhang. 2023. Out-of-Distribution Generalization in Natural Language Processing: Past, Present, and Future. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 4533–4559. doi:10.18653/V1/2023.EMNLP-MAIN.276
[80] Peihua Zhang, Chenggang Wu, Mingfan Peng, Kai Zeng, Ding Yu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. 2023. Khaos: The Impact of Inter-procedural Code Obfuscation on Binary Diffing Techniques. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization, CGO 2023, Montréal, QC, Canada, 25 February 2023- 1 March 2023*, Christophe Dubach, Derek Bruening, and Ben Hardekopf (Eds.). ACM, 55–67. doi:10.1145/3579990.3580007
[81] Mingyi Zhou, Xiang Gao, Jing Wu, John C. Grundy, Xiao Chen, Chunyang Chen, and Li Li. 2023. ModelObfuscator: Obfuscating Model Information to Protect Deployed ML-Based Systems. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 1005–1017. doi:10.1145/3597926.3598113
[82] Yongxin Zhou, Alec Main, Yuan Xiang Gu, and Harold Johnson. 2007. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *Information Security Applications, 8th International Workshop, WISA 2007, Jeju Island, Korea, August 27-29, 2007, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 4867)*, Sehun Kim, Moti Yung, and Hyung-Woo Lee (Eds.). Springer, 61–75. doi:10.1007/978-3-540-77535-5_5

A Conv2D Concretization

Algorithm 2 Conv2D Concretization

Input: $s_i, s_o, ICand$
Output: $ishape = (in_1, in_2, in_3, in_4)$,
$wshape = (w_1, w_2, w_3, w_4)$,
$oshape = (o_1, o_2, o_3, o_4)$

1: $(w_3, w_4) \leftarrow$ a random element in $\{(2,2), (3,3), (5,5), ...\}$
2: **if** $rand()\%2 \neq 0$ **and** $ICand \neq \emptyset$ **then**
3: $\quad (in_1, in_2, in_3, in_4) \leftarrow$ a random element in $ICand$
4: **else**
5: $\quad (in_1, in_2, in_3, in_4) \leftarrow$ GetRandShape$(s_i, 4)$
6: **end if**
7: **if** $in_3 \leq w_3$ **or** $in_4 \leq w_4$ **then**
8: $\quad$ **goto** Line 1
9: **end if**
10: $o_3 \leftarrow in_3 - w_3 + 1$
11: $o_4 \leftarrow in_4 - w_4 + 1$
12: $o_1 \leftarrow in_1$
13: **if** $o_1 * o_3 * o_4 \geq s_o$ **then**
14: $\quad$ **goto** Line 1
15: **end if**
16: **if** $rand()\%2 \neq 0$ **then**
17: $\quad o_2 \leftarrow \lceil s_o/(o_1 * o_3 * o_4) \rceil$
18: **else**
19: $\quad o_2 \leftarrow \lfloor s_o/(o_1 * o_3 * o_4) \rfloor$
20: **end if**
21: $w_1 \leftarrow o_2$
22: $w_2 \leftarrow in_2$

23: **procedure** GetRandShape(s, n)
24: $\quad res \leftarrow ()$
25: $\quad$ **while** $n \neq 0$ **do**
26: $\quad\quad$ **if** $s = 1$ **then**
27: $\quad\quad\quad res.append(1)$
28: $\quad\quad\quad n \leftarrow n - 1$
29: $\quad\quad\quad$ **continue**
30: $\quad\quad$ **end if**
31: $\quad\quad F \leftarrow$ non-trivial divisors set of s
32: $\quad\quad$ **if** $F \neq \emptyset$ **then**
33: $\quad\quad\quad d \leftarrow$ a random element in F
34: $\quad\quad\quad res.append(d)$
35: $\quad\quad\quad s \leftarrow s/d$
36: $\quad\quad\quad n \leftarrow n - 1$
37: $\quad\quad$ **else**
38: $\quad\quad\quad s \leftarrow \lfloor sqrt(s) \rfloor^2$
39: $\quad\quad$ **end if**
40: $\quad$ **end while**
41: $\quad$ **return** res
42: **end procedure**

The input of `Conv2D` has shape (bt, ic, mh, mw) — mh and mw represent the height and width of a 2D feature map, ic is the number of input feature map and bt is batch size. The parameter *weight* has shape (oc, ic, fh, fw), where fh and fw represent the size of 2D filter that will be applied to the input 2D feature map, oc is the number of output feature map. Accordingly, the output has shape $(bt, oc, mh - fh + 1, mw - fw + 1)$.

Algorithm 2 (in Appendix A) shows the concretization strategies of a `Conv2D` operator, which takes three inputs (1) size s_i of upstream unified output; (2) size s_o of real operator output; (3) *ICand*, the shape set of upstream real and fake operators.

The steps for Algorithm 2 are as follows. ❶: select filter size $fh \times fw$ from a set of general settings, i.e. $\{2 \times 2, 3 \times 3, 5 \times 5, ...\}$ (Line 1); ❷: obtain input shape either from the candidate input shape set *ICand* or generate a random 4D tensor shape by calling procedure `GetRandShape`, with equal probabilities (Line 2-5). In brief, `GetRandShape` generates a dimension by randomly taking a non-trivial divisor of the remaining size s (Line 31-36). If there is no non-trivial divisor, s is set to the greatest square number that is less than s (Line 38); ❸: compute the last two dimensions of the output tensor, if they are not valid, go back to ❶ (Line 7-12); ❹: make sure the three determined dimension is valid, i.e., their product does not exceed s_o; if not valid, go back to ❶ (Line 13-14); otherwise, set the remaining dimension such that the size of output tensor is either the smallest number greater than s_o or the largest number less than s_o, each with equal probability (Line 17-20).