REDLC: Learning-driven Reverse Engineering for Deep Learning Compilers

Minghui Li[a], Yang Li[a], Hao Han[a*], Xiaopeng Ke[b], Tongyu Wang[a], Fengyuan Xu[b] and Liming Fang[a]
[a]Nanjing University of Aeronautics and Astronautics, Nanjing, China
[b]Nanjing University, Nanjing, China
Email: {leemh, lyang0314, hhan, fangliming}@nuaa.edu.cn, xiaopeng.ke@smail.nju.edu.cn,
tonyspur@163.com, fengyuan.xu@nju.edu.cn

***Abstract*—Deep Learning (DL) compilers such as TVM enable the efficient deployment of diverse DL models on heterogeneous and resource-constrained devices to meet the needs for low latency, privacy protection, and enhanced reliability. However, the booming of on-device DL technology will inevitably attract new types of cybercriminals and industrial spies aiming to steal commercial models. Emerging research focused on model-stealing attacks from the perspective of DL compilers mainly uses heuristic approaches, which do not work well with compiler-optimized models. This work proposes an advanced model-stealing attack pipeline that combines code representation learning and binary analysis to efficiently reverse retrainable DL framework models from TVM-compiled executables. To further improve the accuracy of reversed models, we exploit the computational relationships to correct the prediction of operators in the models using Graph Convolutional Networks. Extensive experiments demonstrate that our approach can recover 18 common DL models with different scales downloaded from Keras repositories with 99% accuracy.**

***Index Terms*—Deep Learning Compiler, Model Stealing Attack, Model Security**

I. Introduction

Due to the emergence of a large number of heterogeneous and resource-limited mobile and edge devices, deploying deep learning models on these devices has become crucial to meet the needs for low latency, privacy protection, and enhanced reliability. As a result, Deep learning (DL) compiler technology that transforms model definitions described in DL frameworks into highly optimized code for various backend hardware has gained significant attention. This compiler-centric deployment paradigm significantly enhances the efficiency of model deployment across devices equipped with CPUs, GPUs, FPGAs, and other AI accelerators [1]. Many companies, such as OctoML, Huawei, OPPO, and Tencent, have already integrated DL compilers (e.g., XLA, TC, Glow, nGraph, and TVM) into their AI development toolchains.

However, the booming development of on-device DL technology has inevitably attracted a new breed of cybercriminals and industrial spies aiming to steal commercial (non-public) models. This poses a critical and urgent demand: *how to protect the intellectual property of DL models deployed on end devices?* In practice, training an efficient DL model often requires substantial human and financial costs. Studies report that training the 11 billion parameter model may require an investment exceeding $1.3 million [2]. An analysis of 46,753 popular apps in the Android App markets revealed that 1,468 used DL models, ranging from face recognition to malware detection. Yet approximately 40% of the apps had no protection for their models [3]. Despite the implementation of encryption measures in other apps, models can be extracted from memory when they are decrypted during execution [4]. Attackers might exploit these extracted models to generate adversarial samples [5], backdoors [6], or to reuse them at minimal cost, for instance, through transfer learning [7], thereby resulting in financial and security implications.

The model stealing attack is not a new threat. A large body of existing work shows how to steal DL models deployed on cloud servers. Approaches such as query-based [8], [9], side-channel-based [10], [11], and adversarial training [6], [12] are effective in obtaining a mimic model. However, these attackers often cannot extract highly accurate models or can only recover a specific type/portion of the model. Thus, the most recent research has focused on the accurate extraction of models via reverse-engineering on-device models generated by DL compilers. Note that directly decompiling binary code fails to recover the high-level descriptions of models because this approach cannot effectively capture their mathematical semantics (i.e., DL operators) [13]. DND [14] and BTD [15] have been proposed to overcome this limitation, but they all **use heuristic methods based on code signatures for identifying DL operators**. In complex environments, the same operator may be compiled into varied codes on heterogeneous hardware. Furthermore, different compile modes (e.g., standalone, bundle, and ahead-of-time) and optimization options (e.g., operator fusion, memory layout transformation, and auto-tuning) may eliminate or change the signature of the compiled code, presenting significant scalability and generality challenges to operator inference.

To address these challenges, we propose a learning-driven model reverse engineering pipeline named REDLC, which can accurately convert compiled model binaries (e.g., via TVM) back to the framework models (e.g., in Keras). In particular, REDLC first analyzes the target binary and extracts the computation graph of the model based on pre-defined rules. Next, we propose the combination of multiple Bidirectional-Long Short-Term Memory (Bi-LSTM) and Multi-Layer Perceptron (MLP) to infer DL operators (including type and attributes) in the graph. For training such a network to learn the inherent

representation of DL operators, we use DL compilers to generate a large amount of compiled codes for each operator with permutated attributes. Additionally, we propose a Graph Convolution Network (GCN)-based algorithm that learns the contextual relationship of multiple operators in the model to correct the inference errors. At last, we conduct two case studies (i.e., generating adversarial samples and backdoors) for reversed models to demonstrate the realistic threats posed by REDLC.

To summarize, our contributions are listed as follows:

1) We propose the first learning-driven model reverse pipeline named REDLC that can convert compiled model binaries back to retrainable models in DL frameworks. The pipeline ranges from the automatic generation of training sets to the inference of computation graphs, operators, tensors, and hyperparameters. These advances also testify to the power of carefully introducing DL technology to the decompilation domain.
2) We propose a two-stage reconstruction methodology to improve the accuracy of reserved DL models. In the first stage, the combination of Bi-LSTM and MLP is used to accurately infer operator types and their attributes. In the second stage, GCN is leveraged to correct the potential misclassification that occurred in the first stage.
3) Extensive experiments were conducted on 18 widely-used DL models, demonstrating the efficacy of REDLC. The model recovery accuracy reaches up to 99%. Additionally, our experiments reveal two significant findings: (i) The reversed models can be leveraged to generate adversarial examples capable of deceiving the original model. (ii) The reversed models can be manipulated by inserting poisoning samples to set backdoor triggers.

II. Background Knowledge

A. Definition

To better describe our attack pipeline, we first explain some key components of the DL models.

- **Computation graph.** A DL model can be represented by a computation graph. It breaks down the complex process of calculation into a series of operations and shows how data flows through the network during training and inference. A computation graph is a Directed Acyclic Graph (DAG), where each node is an operation, a tensor variable, or a constant. Edges indicate the flow of data from one operation to the next and also hold the intermediate values calculated during computation.
- **Operator node.** Operator nodes or operators (e.g., Conv2D) are a type of nodes defined in the computation graph. Each operator acts as a function that takes inputs from other nodes, performs mathematical computations, and outputs results to other nodes.
- **Attributes of operator.** Each operator often comes with its own set of configurable attributes, such as the number of output filters, kernel size, strides, padding type, etc. These attributes are key hyperparameters that influence the behavior of the operation.
- **Parameter node.** Parameter (or tensor) nodes are another type of nodes defined in the computation graph. They hold the model's weights and biases, which are learned during training. In the inference, parameters are fed to an operator node as input.
- **Input node.** Input nodes hold the initial data fed into the model, which are provided by users.
- **Ouput node.** The output node produces the final results of the model.

The objective of REDLC is to recover the computation graph of the model, including the graph structure, all operator nodes with their attributes, and parameter nodes.

B. Model compiling process

DL compilers play a crucial role in translating a high-level model description into an optimized executable that can run efficiently on various hardware platforms. A DL compiler typically comprises two primary components: the *frontend* and the *backend*. The frontend's main task is to receive the original Deep Neural Network (DNN) model defined using frameworks like TensorFlow, PyTorch, or ONNX and convert it into a high-level intermediate representation (IR). The frontend also performs some hardware-agnostic optimizations such as operator fusion, constant folding, layout optimization, cache utilization, and quantization.

The backend is responsible for converting the high-level IR generated by the frontend into a low-level IR tailored to a specific hardware platform. This conversion process involves some hardware-specific optimizations, such as vectorization, loop unrolling, etc. After optimization, the backend generates machine code from the optimized low-level IR. This machine code is a binary file that can be directly executed on the target hardware, containing all the operations and parameters of the target model.

C. Challenges of extracting models from binaries

One extremely challenging aspect of recovering the computation graph of a DL model is accurately identifying all operators and their corresponding attributes. Due to various compiler and optimization choices, the same operator will be converted into different code snippets with no obvious signature. Even with the same compiler and disabled optimization, the same type of operator may also yield different code depending on its attributes. Existing methods attempt to address this issue by finding the algebraic operation abstract syntax tree (AST) of operators as specific templates for matching [14]. Alternatively, some studies extend this approach to additional aspects [15], such as generating symbolic constraints through trace-based symbolic execution and defining constraint patterns based on expert knowledge to infer higher-level information about operators (e.g., dimensions and memory layouts of parameters). However, their drawbacks are as follows:

Extensibility. The implementation of each operator is often influenced by different hardware architectures, compiler optimizations, and variations in input shapes and data layouts.

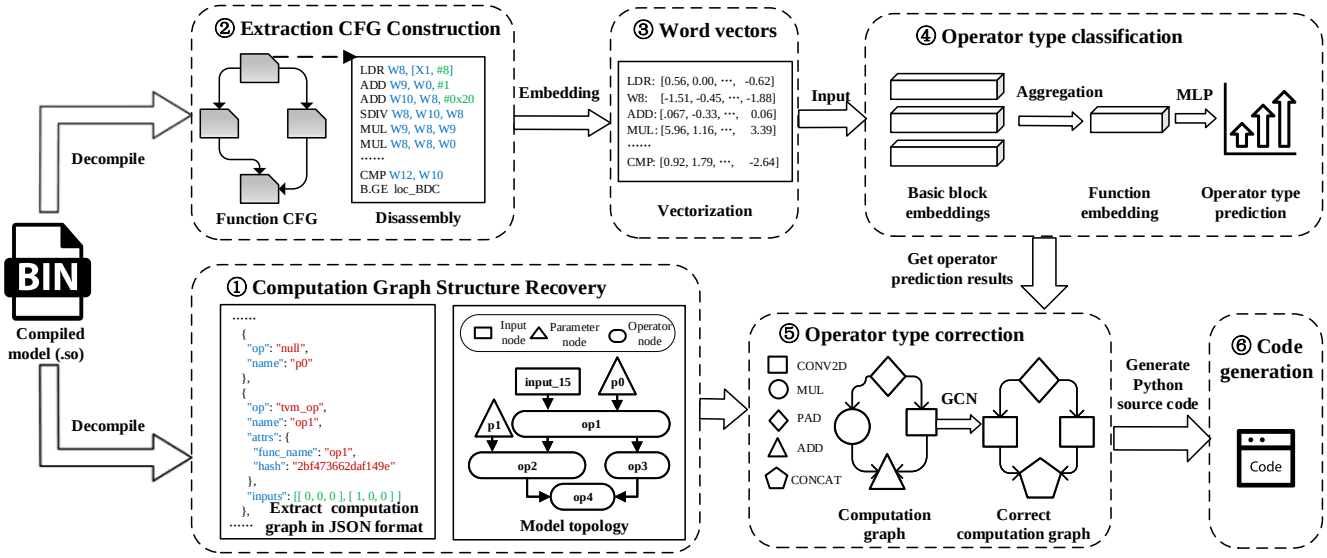

Fig. 1: Overview of REDLC.

Differences in configurations will result in significant variations in the decompiled output. Additionally, general-purpose decompilation tools tend to produce outputs with long loop bodies and excessive bitwise operations. The presence of these factors complicates the design of the template library. The need to manually add scalability support for each newly recognized operator type, even across multiple compiler versions. This leads to low generality. Furthermore, if the compiler doesn't support specific operators in a given framework, template matching will inevitably fail, as demonstrated in [14].

Incompleteness. The effectiveness and comprehensiveness of using AST templates in recognizing operators is heavily constrained by the representational capabilities of algebraic IR, mathematical computation induction, and code structure. Current symbol-execution-based extraction schemes are only capable of capturing the semantics of vectorized mathematical operations, failing when faced with operations like sorting and searching. Additionally, recursive structures pose significant challenges to this form of loop analysis.

Existing no validation after inference. When the computational type of an operator cannot be represented by IR, or when it is a new type not in the template library, there are no auxiliary schemes for judgment. Existing approaches lack the capability to correct these errors from a global view of the computation graph when extraction and recognition errors occur. The usage patterns of subgraphs and relationships between connected operators provide information that can help opportunistically adjust misrecognized operator types and attributes.

III. System Design

Our proposed approach, named REDLC, circumvents the above challenges by creating a scalable database of binary files for each operator layer and training a semantic representation model on decompiled functions. This model is trained with machine code from various hardware platforms and incorporates different compiler scenarios for the same operator type with varying attributes. Thus, our approach facilitates DNN reverse engineering across different hardware platforms and offers greater scalability and recognition capabilities compared to existing methods based on template ASTs. To more accurately restore topological semantics in assembly functions, we propose a novel embedding algorithm that combines syntax-semantic representations with topological semantics for a more precise representation of assembly functions. The overall workflow of REDLC is depicted as follows:

A. Workflow Overview

First, we recover the computation graph structure from the compiled model (Step ① in Fig.1) and identify the operator type by using the semantic characteristics of the corresponding disassembly. To infer the operator type, we find the corresponding function (Step ②) and make use of the Word2Vec [16] technique to transfer all disassembly instructions to numerical vectors (Step ③). We feed these numerical vectors to a Bi-LSTM with MLP and obtain the classification results (Step ④). Then, we try to fix misclassified operator nodes by exploiting the prior knowledge of model structures to increase the total reverse precision. Here, we use GCN to catch the structure information of the computation graph and generate the node embeddings to identify if the node has a wrong prediction (Step ⑤). Last, we generate the trainable model by using the corrected computation graph (Step ⑥).

As shown in Fig.1, our model reverse attack contains two main stages: (a) Computation Graph Recovery; (b) Operator Type Correction. After obtaining the corrected model computation graph, we designed a Code Generation module to convert it to model code.

B. Computation Graph Recovery

We use TVM as an example to show the process of computation graph recovery in REDLC. For a DL model compiled by TVM, the graph is typically stored in a text segment of its binaries (.so file), which can be directly extracted. As shown in Listing 1, the graph is represented in JSON format, containing two key elements: `nodes` and `attrs`. Nodes represent an array of operator nodes, while attrs contain input/output tensor shapes and data types. To analyze the compiled model, we use the disassembly tool IDA-Pro to extract the disassembly code.

```
{
  "nodes": [
    {
      "op": "null", # Input Node
      "name": "input_15",
      "inputs": []
    },
    {
      "op": "null", # Parameter Node
      "name": "p0",
      "inputs": []
    },
    {
      "op": "tvm_op", # Operator Node
      "name": "op1",
      "attrs": {
        "num_outputs": "1",
        "num_inputs": "2",
        "flatten_data": "0",
        "func_name": "op1",
        "hash": "2bf473662daf149e"
      },
      "inputs": [[ 0, 0, 0 ], [ 1, 0, 0 ] ]
    }
    ]
}
```

Listing 1: Computaion graph in JSON format

In Listing 1, the operator node is a dictionary with keys such as `op`, `name`, `attrs`, and `inputs`. Constructing the computation graph involves inferring both node and edge information. Upon extracting the graph structure, we have node information but must still infer all directed edges. Our observation suggests that inputs denote the indices of input operator nodes, aiding in identifying directed edges. Additionally, during graph construction, we identify parameter nodes, which are constant vectors or matrices stored in a fixed format.

As depicted in Algorithm 1, we utilize the op label to identify parameter nodes. This algorithm takes two input variables: node and shape, both extractable from the compiled model. Initially (lines 1-4), we initialize the number of nodes and the graph structure. Subsequently (lines 5-6), we extract the output shape of each node. We treat all Parameter Nodes and Input Nodes as additional nodes (lines 8-11), then initialize all Operator Nodes with the input list and node name (lines 12-14). Finally (lines 15-20), we add all relevant parent nodes for each node and append them to our computation graph.

Sometimes, the operator's name may be replaced by a hash string. Despite the name replacement, we can utilize the `func_name` to locate the corresponding code snippet of the operator in the compiled binaries. In Step ② of Fig. 1, we employ the IDA-Pro tool to search for the function and retrieve the disassembly code of all basic blocks in the CFG.

Algorithm 1 Computaion Graph Generation

Input: nodes, shapes
Output: computation graph

```
1:  N =| nodes |;
2:  graph = Graph();
3:  count = 0;
4:  cNodeDict = Dict();
5:  nToIDict = Dict();
6:  for i ← 1 to N do
7:      nodes[i].output_shape = shapes[i];
8:  end for
9:  for i ← 1 to N do
10:     if nodes[i].op == null' then
11:         AddExtraNode(graph, nodes[i]);
12:         count + = 1;
13:         cNodeDict[count] = nodes[i].name;
14:         nToIDict[nodes[i].name] = count;
15:     end if
16:     inputs = nodes[i].inputs;
17:     cNodeDict[count] = nodes[i].name;
18:     nToIDict[nodes[i].name] = count;
19:     count + = 1;
20:     for j ← 1 to | inputs | do
21:         idx = inputs[j][0];
22:         FName = cNodeDict[nToIDict[nodes[idx].name]];
23:         fatherNode = GetNode(graph, FName);
24:         AddFatherNode(nodes[i], fatherNode);
25:     end for
26:     AddNode(graph, nodes[i]);
27: end for
```

However, we observe that the operator logic can be concealed within the intermediate call. Moreover, we notice that there consistently exists a single, similar basic block across different operators. As shown in Fig. 2, the address **sub_AF4** is the real function address, but IDA-Pro can not recognize it. To obtain the key details of the operator disassembly, we consider all possible indirect calls and add all basic blocks from the CFG of indirect calls. As shown in Fig. 2, the striped node represents the basic block node which has an indirect function call. To obtain more information about all operators and get more accurate inference results, we extend the original CFG to the Extended Function CFG. We add a directional edge on the entry node and return node in the indirect function CFG.

Then, we can obtain a simple computation graph representation, including operator nodes (without knowing the type of operator) and edges.

C. Vectorization of Disassembly

Next, we infer the types of operators through the extended CFG. To make accurate type inferences, we utilize the deep neural network since it has excellent capability for learning code representation. Here, we follow some previous works [17] and regard all instructions as words in NLP. We use

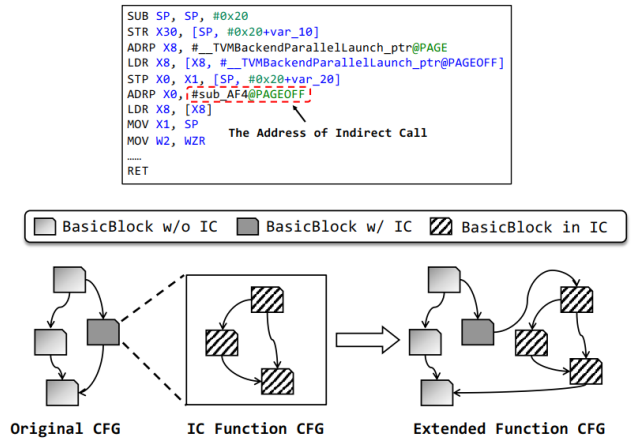

Fig. 2: Extend function CFG. (The IC means the Indirect Call).

the Word2Vec [16] model to get the word embeddings (i.e. the instruction embedding). As shown in Fig. 3, we utilize the CBOW (Continuous Bag Of Words) implementation for our Word2Vec model since it is widely used and has great capability for catching the context information on many NLP tasks. First, we split every single instruction into several instruction tokens and convert all instruction tokens to a random real vector of fixed length. For example, the instruction $(LDRW8, [X1, X8])$ will be split to $LDR, W8, [, X1, X8,]$. For a single basic block, we concatenate every adjacent instruction token and form an array that contains all tokens of the basic block. In the CBOW implementation, there is a sliding window with an odd length, and we denote the center word as W0 (as the ADD in Fig. 3). The neighbor instruction tokens are denoted as $(w_{-(l-1)/2}, ..., w_{-2}, w_{-1}, w_1, w_2, ..., w_{(l-1)/2})$ where l is the total length of the sliding window and $w_i \in R^d$ is a real vector with d dimensions. Our objective is to utilize neighboring instruction tokens to predict the center token. The predicted probability of center token w_0 can be expressed as:

$$p(w_o \mid w_{-(l-1)/2}, ..., w_{-2}, w_{-1}, w_1, w_2, ..., w_{(l-1)/2}) \quad (1)$$

To compute such a probability, we use a single multilayer perceptron (MLP) to estimate:

$$\hat{w_0} = \sigma_2(A_2\sigma_1(A_1W + b_1) + b_2) \quad (2)$$

where $W = [w_{-(l-1)/2}, ..., w_{-2}, w_{-1}, w_1, w_2, ..., w_{(l-1)/2}] \in \mathbb{R}^{(l-1)\times d}$ is the vector of neighbor instruction tokens, $A_1 \in \mathbb{R}^{h\times(l-1)}$, $A_2 \in R^{1\times h}$ are height matrix, and $b_1 \in \mathbb{R}^h$, $b_2 \in \mathbb{R}$ are bias vectors. The target of the Word2Vec model is to maximize the predicted probability of every token. It can be written as:

$$max\mathbb{E}_{w_0\sim D}[p(w_o \mid w_{-(l-1)/2}, ..., w_{(l-1)/2})] \quad (3)$$

As shown in Equation 3, we try to maximize the expectation of all possible center tokens w0 from the training dataset D. We can define the loss as:

$$\mathcal{L} = \frac{1}{\mid D \mid}\sum_{i=1}^{\mid D \mid} \parallel w_i - \hat{w_i} \parallel_2 \quad (4)$$

Here, we apply the Stochastic Gradient Descent (SGD) to optimize all parameters of MLP and adjust all vectors. When the loss is converged, we could get the trained vector for every token as the token embedding.

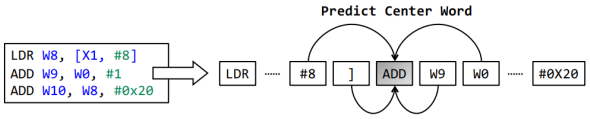

Fig. 3: Word2Vec model on disassembly.

D. Operator Type Classification.

After obtaining the vocabulary of all tokens (i.e., the mapping of the token string and the corresponding embedding), we first construct the basic block embedding by passing the token embeddings to the Bi-LSTM. As shown in step ④ of Fig. 1, we input all token embedding of each basic block to the Bi-LSTM with order and generate the basic block embedding.

Here, we give the formalization of that process. We denote the i-th basic block as $B_i = [t_1, t_2, ..., t_{|B_i|}]$ where $t_j \in \mathbb{R}_t^d$ is the j-th token embedding and d_t is the dimension of the token embedding. For the operator function F, it has multiple basic blocks, and we denote that function as $F = [B_1, B_2, ..., B_{|F|}]$. We denote the Bi-LSTM as the function Bi-LSTM$(\bullet) \in \mathbb{R}_o^d$ where d_o id the dimension of the output vector. As for a basic block B_i, the Bi-LSTM's output can be written as:

$$Bi - LSTM(t_1, t_2, ..., t_{|B_i|}) = v_{B_i} \quad (5)$$

Then, we can obtain all basic block embeddings $(v_{B_1}, v_{B_2}, ..., v_{B_{|F|}})$ and make aggregation on them as:

$$v_F = Aggr(v_{B_1}, v_{B_2}, ..., v_{B_{|F|}}) \quad (6)$$

where $Aggr \in \mathbb{R}^{d_F}$ is the Aggregation function and d_F is the dimension of output operator function embedding. In our implementation, the aggregation is mean operation as $Aggr(x_1, ...x_n) = \frac{1}{n}\sum_i x_i$.

We feed the operator function embedding to an MLP to make a classification of operator type. It can be written as:

$$MLP(v_F) = Sigmoid(H_2RELU(H_1v_F + b_F)^{(1)}) + b_F^{(2)}) \in \mathbb{R}^C \quad (7)$$

where C is the number of operator types. H_1 and H_2 represent the trainable weight matrix. The output of MLP is passed to a softmax layer and gets the probability vector. For each operator function, we analyze their basic block information and finally select the top-1 result as our basic classification result.

To train such a classification model, we use a cross-entropy loss function to optimize the parameters. The loss function can be written as:

$$\mathcal{L}_{cls} = \sum_{F\in\mathcal{F}}\sum_{i=1}^{C} -(y_{F,i}log(MLP(v_F, i)) + (1 - y_{F,i})(1 - \log(MLP(v_F, i)))) \quad (8)$$

Where the $\mathcal{F}$ is our operator dataset, F is a single operator from the dataset, $y_{F,i} \in 0, 1$ is a binary variable which represents if the operator F belongs to the class i, v_F is our output function embedding through our Bi-LSTM model,

TABLE I: The number of computation graph nodes from different models.

Model Name	Number of Nodes	Model Name	Number of Nodes
MobileNet	506	Xception	807
ResNet50	1085	DenseNet101	2250

and $MLP(v_F, i)$ is our output classification probability of the class i.

E. Operator Type Correction

Through the Computation Graph Recovery step, we can obtain a computation graph with nodes, which have been assigned an operator type. However, it is hard to make sure that all nodes have the correct operator type. As shown in Fig. 1, we misclassify the CONCAT operator to the ADD operator (100% accuracy is difficult to achieve in practice). These failure cases have extreme influences on the code generation since these wrong nodes may yield the incorrect output shape or even can not accept the output from previous operator nodes. For example, the 2D convolution operator CONV2D has one input. If this operator is misclassified to the MULTIPLY operator, the computing process will stop at that operator due to the mismatched input shape. The correctness of all operator types is so important that we need to check the type carefully. However, there are numerous operator nodes in a DL model, and it is hard to check every node manually, as shown in Table I.

To handle the above obstacle, we try to exploit the knowledge of common model structures and fix some wrong nodes. If there just is one node that is not mismatched, we could find it based on the original structure. We want to automatically catch these patterns with some prior knowledge. Then we can find out some nodes, which do not match the common graph pattern. In other words, we want to find these abnormal nodes in the computation graph.

We propose a correction algorithm to correct the node type based on the Graph Convolution Network (GCN) [18] since the GCN can capture the structure information, and it performs well in many graph problems.

As shown in Fig. 4, we first generate the computation graph $G =< V, E >$ where V is the vertex set and E is the directional edge set. $V = (v_1, v_2, ..., v_n)$ contains all nodes of the computation graph. These nodes have three types: (a) Operator Node, (b) Input Node, and (c) Parameter Node. The operator node represents the operator of the model such as CONV2D, ADD, and DENSE. The input node represents the placeholder of the input data to the model. The parameter node represents the weight data of the operator. For example, the CONV2D operator always has two parameter nodes (kernel weights and bias weights) that point to the operator node. Here, we let the type of the input node as the 'INPUT'. Also, we set the type of all parameter nodes as the 'Params'. As for the operator nodes, we use their prediction results to set their types. For these different node types, we assign a random vector with a fixed length to these types before the training step. Therefore, our vertex set is a $n \times d_v$ matrix where n is the node number and d_v is the length of the random vector.

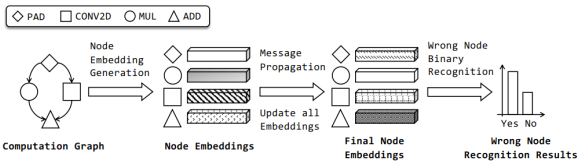

Fig. 4: Wrong type classification by using GCN.

We denote the degree matrix of the computation graph as $D \in \mathbb{R}_{n\times n}$ and denote the adjacent matrix as $A \in \mathbb{R}_{n\times n}$. Our GCN process can be expressed as:

$$\begin{aligned} h^{t+1} =&((1-\alpha)(D^{-1/2}(A+I)D^{1/2}h^{(0)} + \alpha h^{(0)})) \\ &((1-\beta)I + \beta W) \end{aligned} \tag{9}$$

where $h^{(0)} = V \in \mathbb{R}^{n\times d_v}$ is the initial embedding matrix, I is the identity matrix, and W is the weight matrix. The Equation 10 represents the process of the message propagation (i.e., The vertex embedding will be propagated to its neighbor nodes, and all nodes will merge these embedding from their neighbors). After the message propagation, we can obtain the embedding vector of every node. Then, we feed these node embedding vectors to MLP and get binary results. Those binary results can show nodes that have the wrong prediction of operator types.

To train such a correct GCN model, we need to generate some training data. In this paper, we collect 38 popular models and use 19 model computation graph structures to train our GCN model. For each model computation graph, we randomly change the operator type of some nodes with a fixed ratio setting. We regard these changed nodes as wrong operator nodes and add labels to them. We use the cross-entropy loss to optimize the network parameters. The loss function can be written as:

$$\begin{aligned} \mathcal{L}_{gcn} = \frac{1}{\mid \mathcal{G} \mid} \sum_{g\in\mathcal{G}} \sum_{v\in g} &-(y_v \log(f_{gcn}v))) \\ &+(1-y_v)\log(1-f_{gcn}(v)) \end{aligned} \tag{10}$$

where G represents our graph dataset, g is a single computation graph, v is a node of the computation graph, $y_v \in 0, 1$ is an indicator value for the changed nodes (i.e., $y_v = 1$ when the node v is changed), and $f_{gcn}(v) \in \mathbb{R}$ is our GCN model. Our GCN model outputs a probability of being a wrong node for each operator node.

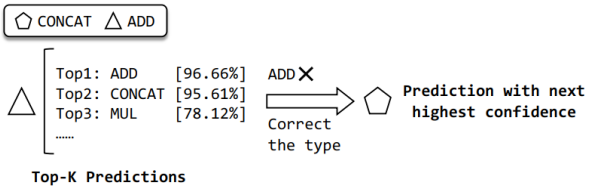

Fig. 5: Operator type correction.

After we obtain these binary results, we set the operator type prediction to the next prediction result for every wrong node. Here, the next prediction result means the predicted type that has the next highest confidence score in all types. As shown in

TABLE II: Operator attribute information in Keras. (The symbol '-' in the column 'Related Clues' means we cannot directly infer their attributes.)

Attribute	Type	Range	Operators	Related Clues
padding [conv]	string	'same', 'valid'	conv2/3d	In&Out Shape
padding [pad]	int	[1, max_size]	pad	In&Out Shape
filter	int	[1, max_size]	conv2/3d	Output Shape
strides	Tuple	[1, max_size]	conv2/3d	In&Out Shape
kernel_size	int	[1, max_k]	conv2/3d	Weight Shape
depth_multiplier	int	[1, max_d]	conv2d [dw]	Weight Shape
units	int	[1, max_u]	dense	Weight Shape
use_bias	bool	{True, False}	conv2/3d, dense	Child Nodes
size [upsampling]	int	[1, max_size]	upsampling2/3d	In&Out Shape
interpolation	string	'bilinear', 'nearest'	upsampling	-
pool_size	int	[1, max_size]	max_pool2/3d	In&Out Shape
pool_size	int	[1, max_size]	avg_pool2/3d	In&Out Shape
cropping	int	[0, max_size]	strided_slice	In&Out Shape
center, scale	bool	{True, False}	BN	Graph Structure
axis	int	[0, channels]	concatenate	In&Out Shape
max_value, beta, ...	float	[0, +inf]	Activation Func	-
kernel_initializer, ...	string	{'kaiming', 'xavier', ...}	conv2/3d	-

Fig. 5, the operator ADD has been judged as the wrong node by our GCN model. This node has many prediction results with different confidence scores and the ADD type is the top-1 result. To correct that type, we just select the top-2 result as its type prediction and replace the top-1 result.

For every wrong node, we change their type prediction and we can construct a new corrected computation graph. To achieve higher precision of the operator type prediction, we propose the iterative correction algorithm to strengthen our GCN-based correction. We input the corrected computation graph to our GCN again and get the node predictions. Once our GCN predicts a node as a wrong node, we correct the type of these wrong nodes and construct a new computation graph. For example, our GCN finds two wrong nodes in the first correction process in Fig. 6 and we correct their type. In the second correction, we find the MUL node is the wrong node and correct its type to CONV2D. After the second correction, we can obtain a final corrected computation graph (i.e., all nodes are classified as the correct node by GCN).

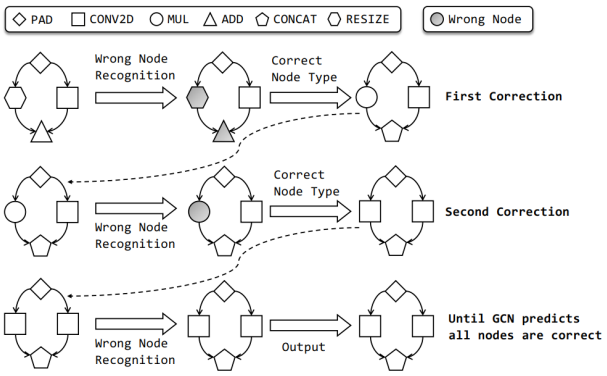

Fig. 6: Iterative operator type correction.

F. Code Generation

To obtain a complete model suitable for training, we must generate the corresponding code from the corrected computation graph. Due to space constraints, we present the code generation algorithm. Algorithm 2 outlines our code generation process. Firstly, we include all necessary import statements in the code results, such as importing Keras and NumPy libraries. As the computation may include NOP nodes that don't impact the graph, we eliminate them and establish new connections between their parent and child nodes. However, some NOP nodes serve as RESHAPE operators. To ensure correct output shapes for all nodes, we examine the input and output shapes of each NOP node and introduce additional RESHAPE nodes as needed to rectify any discrepancies.

Algorithm 2 Code Generation

Input: graph
Output: code

```
code = [];
code.append(importCodes);
graph = cleanNOP(graph);
for node in graph do
    if notParamOrInput(node) then
        node.attr = computeAttr(node)
    end if
end for
graph = MergeNodes(graph);
for node in graph.nodes do
    code.append(genInitCode(node))
end for
sortedNodeList = topologySort(graph);
for node in sortedNodeList do
    code.append(genForwardCode(node))
end for
code.append(returnCode);
code.append(weightLoadingCode);
for node in graph.nodes do
    if hasWeight(node) then
        code.append(genWeightInitCode(node))
    end if
end for
```

Attributes inference. After eliminating all NOP nodes, we analyze the input, output, and parameter tensor shapes to compute attributes for nodes with settings (e.g., strides of the convolution operator). Subsequently, we merge specific basic operator nodes into higher-level operators. For example, the Batch Normalization operator may be decomposed into multiple basic operator nodes (e.g., ADD, MUL, ...) within a defined graph structure. To capture such a specific subgraph structure, we utilize the VF2 algorithm [19] to match the subgraph in the whole corrected computation graph since VF2 is an efficient subgraph isomorphism algorithm in large graphs.

Then, we generate the initialization code for every operator. These initialization codes contain the definition and attribute assignment. During the initialization code generation step, we assign unique variable names to distinct operators. For operators of the same type, we append an ordinal index to differentiate them as shown in Fig. 7. Every operator node has a unique name that contains the type name and the order

index (e.g., $mp2d\_4$ represents the 4th MAXPOOLING2D operator). We use this unique variable name to generate the forward codes of the whole model. (Note that, we use specific rules to directly infer the attributes of all operators as shown in Table II)

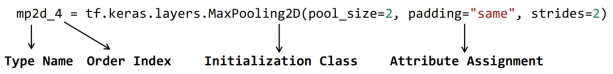

Fig. 7: Operator initialization.

To deal with the dependencies in operator nodes during the network forward process, we use the topology sort on all nodes and get a sorted node list. We generate the forward codes by the order in the sorted node list. When we visit a node in this sorted node list, we get the input variable name list from the output name of its father nodes.

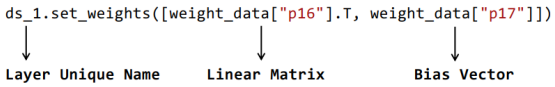

Fig. 8: A code example of the weight assignment.

At the end, we generate the weight loading code (Note that we need to extract the weight dictionary from the .so file first). We apply the pickle python library to load the dictionary from the extracted .pkl weight dictionary file. For different operators that need the weight initialization, we utilize the fixed rules to assign the weight to them. For example, we assign the weight to the dense layer as shown in Fig. 8. Note that the dense operator also has a no-bias mode, and we need to check if there is a BiasAdd node in the child nodes of the dense operator nodes.

IV. Implementation and Evaluation

A. Implementation

REDLC is primarily written in Python with about 5.5K LOC. We select TVM as the DL compiler and use the Keras as the DL model frontend. The current implementation is able to decompile 64-bit executables compiled by TVM in the ELF format on x86 platforms. All our experiments were conducted on a 64-bit server running Ubuntu 20.04 with 12th Gen Intel(R) Core(TM) i7-12700KF 3.60 GHz, 32GB memory, 2TB hard drive, and NVIDIA GeForce RTX 3080 GPU. The TVM [20] version is 0.8, paired with LLVM 11. For reverse engineering, we used IDA Pro 7.5.

B. Experimental Methodology

With the above implementation, we evaluated REDLC by answering the following questions:

- **RQ1**: does REDLC have sufficient accuracy in recognizing operator types and attributes to enable the new paradigm of model stealing attack?
- **RQ2**: can REDLC correct errors of operator inference to effectively recover the entire computation graph of common DL models?
- **RQ3**: can REDLC improve the state-of-the-art (SOTA) model decompilation methods?
- **RQ4**: does REDLC pose serious threats in practice, and what consequences can be derived from our attack?

In particular, we first evaluate the accuracy of computation graph recovery and operator classification, followed by a comparison with SOTA deep learning decompilers. Our experimental setup is as follows:

*1) **Dataset**:* To train a classification model for identifying the operator type, we randomly generate all Keras operators that can be supported by TVM with different attribute settings and compile these codes with TVM under default settings. We randomly generated a total of 30,600 Keras operator samples, which were consolidated into the Keras layers shown in Table III. Through TVM compilation, all frontend operators were transformed into Relay operators. The generated Keras operators were compiled into a total of 59,400 Relay operators. The data of each operator is divided into the training set and the test set according to the ratio of 7:3.

To train the correction model, we gather 38 popular models, allocating 19 models for training the correction GCN model and the remaining 19 models for testing. We use 70% of the data to train our classification operators. We set the learning rate, training epochs, and optimizer as 0.0001, 200, and Adam, respectively. As for the correction model, we set the learning rate, flip ratio, and training epochs as 0.01, 0.1, and 200, respectively.

2) IDA-Pro Parsing: In parsing the IDA-Pro disassembly results, we clean the results and construct the token list for training or inference. In particular, We clean all annotations after ';' and remove all page information that has '@PAGE' or '@PAGEOFF'. We replace all addresses with the same token 'addr' to avoid the Out-Of-Vocabulary problem of Word2Vec. We also replace all big numbers which are larger than 2^{15} with the same token 'num'. For all numbers less than 2^{15}, their respective binary vectors are utilized as their word vectors. Choosing 2^{15} is based on its practical limit for binary representation, which can reduce vocabulary size and ensure model efficiency without losing important information.

3) Inference Detail: First, we extract all the function disassembly from the compiled model by using the IDA-Pro. Then we use the pre-trained word2vec vocabulary and convert all instructions to corresponding word vectors. For unseen instructions, we use the zero vector. In the correction process, we don't flip any labels. Empirical Data shows that setting the maximum correction to 9 can provide optimal balance.

*4) **Performance Metrics**:* We quantify operator classification accuracy using precision (the number of true positives), recall, and F1-score. Moreover, assessing the overall accuracy of model recovery is a key metric. To achieve this, we specifically evaluate the accuracy of the operator correction module to determine its contribution to the overall model recovery process. The processing time for model recovery acts as a metric for evaluating the attack efficiency of our proposed approach. We also measured how much the accuracy of the reverse-engineered model decreases after subsequent derived

TABLE III: The count information of our Keras Layer Data

Keras Layer Name	Count	Keras Layer Name	Count
Activation	5800	Conv2D	1000
SeparableConv2D	1200	Cropping2D	1200
UpSampling3D	900	GlobalMaxPooling2D	900
GlobalAveragePooling2D	900	GlobalMaxPooling3D	900
Multiply	800	Minimum	800
MaxPooling2D	800	Maximum	800
AveragePooling3D	800	AveragePooling2D	800
Add	800	BatchNormalization	800
Concatenate	800	Flatten	800
DepthwiseConv2D	1200	Conv3D	1000
GlobalAveragePooling3D	900	Subtract	800
MaxPooling3D	800	Dot	800
Average	800	Dense	800

attacks to estimate the potential threat level that REDLC could pose in real-world scenarios.

C. *RQ1: Correctness and Comprehensiveness*

We wrote scripts to randomly generate Keras operators with different attributes. To adapt to the TVM compilation pipeline and cover various operator configurations, we opted not to use public datasets. We use precision, recall, and F1-score to measure the effect of the individual operator classification. The results are shown in Table IV. The precision and recall of individual operator classification categories can reach more than 99%, and the F1-score can reach more than 96.45%.

TABLE IV: The result of individual operator classification. P means precision, R means recall, and F means F1-score. 'amp3d' means adaptive_max_pool3d. 'aap3d' means adaptive_avg_pool3d. 'gap2d' means global_avg_pool2d. 'gmp2d' means global_max_pool2d.

Operator	P	R	F	Operator	P	R	F
conv2d	1.000	1.000	1.000	dense	1.000	0.990	0.990
relu	1.000	1.000	1.000	avg_pool2d	1.000	0.990	0.990
bias_add	0.992	0.990	0.960	max_pool2d	0.999	0.970	0.970
abs	0.999	1.000	0.960	'aap3d'	1.000	0.970	0.980
'amp3d'	1.000	0.990	0.990	add	0.992	0.930	0.960
avg_pool3d	1.000	1.000	0.990	batch_flatten	1.000	1.000	1.000
batch_matmul	1.000	1.000	1.000	cast	1.000	1.000	0.990
clip	1.000	1.000	1.000	conv3d	1.000	1.000	0.990
divide	1.000	1.000	1.000	exp	1.000	1.000	1.00
expand_dims	1.000	1.000	0.990	'gap2d'	1.000	1.000	0.980
'gmp2d'	1.000	1.000	0.990	greater	1.000	0.970	0.980
leaky_relu	1.000	1.000	1.000	log	1.000	1.000	1.000
max_pool3d	1.000	1.000	0.980	maximum	1.000	1.000	1.000
minimum	0.999	0.990	0.980	multiply	0.998	0.980	0.990
negative	1.000	1.000	1.000	pad	1.000	1.000	1.000
softmax	1.000	1.000	1.000	sqrt	1.000	1.000	1.000
strided_slice	1.000	0.990	1.000	subtract	0.999	0.980	0.990
transpose	0.999	1.000	1.000	upsampling	1.000	1.000	1.000
upsampling3d	1.000	1.000	1.000	concatenate	1.000	1.000	0.990

Precision of Reversed Model. As shown in Table V, we test the compiled models, which are pre-trained at the ImageNet dataset with the corresponding test dataset.

Answer of RQ1: We compare our reversed models with these compiled models on the same test dataset. The experiment result shows that our reversed model has the same performance as the compiled models.

TABLE V: Model Performance of compiled model and reversed model

Model	Rev Acc	Ori-Top1	Ori-Top5	Rev-Top1	Rev-Top5
Xception	100%	77.524	93.752	77.524	93.752
DenseNet121	100%	71.560	90.646	71.560	90.646
ResNet50	100%	68.084	88.346	68.266	88.408
ResNet50V2	100%	65.920	86.908	65.920	86.908
MobileNet	100%	68.360	88.250	68.360	88.250

D. *RQ2: Correction Performance and Model Reverse*

We utilize a 2-layer Bi-LSTM to extract embeddings from each basic block and set the output dimension as 200 for each direction, resulting in a 400-dimensional final output vector. For operator classification, we employ a 2-layer MLP. In the GCN-based Correction, we choose GCN2CONV for its superior performance, setting the hidden layer's output embedding dimension as 200, $\alpha = 0.1$, $\theta = 0.5$, and the number of layers as 3. Subsequently, we apply a 2-layer MLP to identify incorrect predictions, constituting a binary classification task.

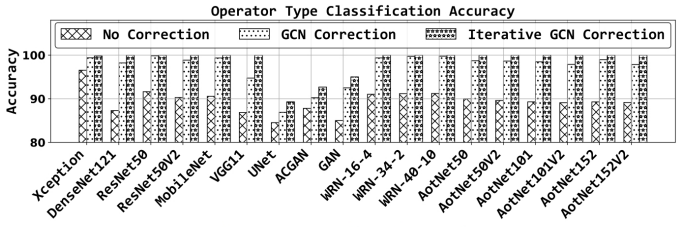

Fig. 9: Operator type classification accuracy.

Fig. 9 illustrates the accuracy of model reverse. Following correction, the accuracy of most networks, including Xception, achieves 100%. It means that all misidentified operators are successfully rectified. Through our investigation, it is discovered that the operators add, max_pool2d, and 'gmp2d' are misidentified. The add operator can be rectified by adjusting tensor dimensions and confidence. As for the other two pooling operators, their misidentification can be corrected by modifying input and output tensor dimensions. The accuracy of DenseNet operators' identification increased to 100%. Observations reveal that all errors associated with add and bias_add in these models are rectified. However, there still exist operators that cannot be rectified, such as cases where relu is misidentified as cast. The cast operator is unique to TVM's NN library, since relu and cast have no tensor parameters and do not affect input tensor dimensions, they cannot be repaired through model reconstruction of the computation graph. These models heavily utilize relu activation functions, and a portion of relu is misidentified as a clip. Although clip and relu have similar functionalities, clip has hyperparameters that can impact reversed model performance.

Attack Success Rate. We perform three attacks on 18 popular DL models.The 'REDLC without Correction' means we only use the classification model without any correction. The 'REDLC + GC' means we use the classification model and only apply one correction. The 'REDLC + IGC' means we perform the classification with our iterative correction method.

TABLE VI: Runnable and Performance Checking Results for Reversed Models. The symbol '✓' means that the reversed model has no errors and the same performance as the corresponding compiled model. The symbol '✗' means the reversed model has errors, and we can not run it correctly.

Method \ Model	Xception	DenseNet121	ResNet50	ResNet50V2	MobileNet	VGG11	UNet	ACGAN	WRN-16-4
REDLC without Correction	✗	✗	✗	✗	✗	✗	✗	✗	✗
REDLC + GC	✗	✗	✗	✗	✓	✗	✗	✗	✗
REDLC + IGC	✓	✓	✓	✓	✓	✓	✗	✗	✓
	AotNet50	AotNet50V2	AotNet101	AotNet101V2	AotNet152	AotNet152V2	GAN	WRN-34-2	WRN-40-10
REDLC without Correction	✗	✗	✗	✗	✗	✗	✗	✗	✗
REDLC + GC	✗	✗	✗	✗	✗	✗	✗	✗	✗
REDLC + IGC	✓	✓	✓	✓	✓	✓	✗	✓	✓

TABLE VII: Comparison with SOTA model. '-' means limited by conversion or analysis conditions. 'O' indicates that the operator is not included in the supported classification list. 'L' means the workload requirement is light. 'H' means heavy. '✓' and '✗' represent the presence and absence of specific characteristics in various studies, respectively.

			DND [14]	BTD [15]	REDLC
Operator ACC		Upsampling	-	O	100%
		batch_flatten	-	O	100%
		'amp3d'	-	O	100%
		avg_pool3d	-	O	100%
		conv3d	-	O	100%
		max_pool3d	-	O	100%
		'aap3d'	-	O	100%
		upsampling3d	-	O	100%
Workload		Injective	L	L	L
		Reduction	L	L	L
		Complex-out	H	H	L
Characteristic	Methodology	Template-based	✓	✓	✗
		DL-based	✗	✗	✓
	Correction Model	Heuristic-based	✓	✓	✗
		GCN-based	✗	✗	✓
	Compilation Mode	Graph-Executor	✗	✓	✓
		AOT	✓	✗	✗

As shown in Table VI, we successfully reverse about 83% of all compiled models under the 'REDLC + IGC' attack.

Ablation Study of Our Attack Design. As shown in Table VI and Fig. 9, all reversed models need 100% reverse accuracy to run correctly and keep the same performance as the corresponding compiled model.

Answer of RQ2: These experiments present that our attack (Classification + Iterative GCN Correction) achieves high reverse precision which is so important for running the reversed model correctly.

E. RQ3: Comparison

We compared our REDLC with DnD and BTD, which are SOTA DL decompilers. We conducted comparisons in terms of classification accuracy, scheme characteristics, and operator workload requirements. As shown in Tablee VII, we find that the REDLC framework comprehensively outperforms the other two template-based frameworks in terms of operator support and workload requirements. DnD is engineered with a focus on the ARM architecture, and its support for operators and networks outside of its specified framework is very limited. BTD performs well with networks such as ResNet, MobileNet, and VGG, but shows poor support for networks handling three-dimensional data (e.g., 3D images or videos), primarily due to the high cost of trace logging and taint analysis for operators like conv3d. REDLC overcomes these limitations. Moreover, both of these SOTA models rely on AST templates to identify operator types and attributes. However, operators with varying attributes can exhibit different implementations across diverse compilers and optimization settings. Manually crafting templates for each variation is insufficiently advanced and impractical. Furthermore, developing new patterns for different compilation scenarios is time-consuming, particularly for operators with complex loop structures like convolutions, which can take several days to complete [15]. In contrast, REDLC achieves significantly lower overhead, requiring only 391.77 seconds on average, as shown in Fig. 10. This efficiency is due to its approach of learning feature relationships between instructions, providing a more generalized solution.

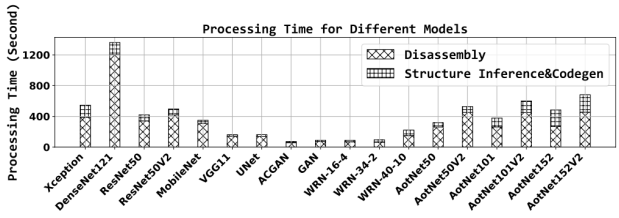

Fig. 10: Processing time of the model reverse attack. The time scale for REDLC is in seconds. BTD [15] takes several minutes to hours and can extend to several days when new complex operators are introduced.

Processing Time. As shown in Fig. 10, our attack just needs 391.77 seconds on average. In our attack process, the disassembly process always takes much time with a comparison of our structure inference and the code generation.

Answer of RQ3: Compared to similar approaches, we maintain a leading edge in both restoration accuracy and attack duration, suggesting that deep learning-driven model reverse attack pipelines represent a highly promising implementation method for this field in the future.

F. Case Study (RQ4): Derived Attacks

1) Adversarial Attack: We perform the PGD Attack on five models (Xception, DenseNet121, ResNet50, ResNet50V2, and

MobileNet) that are pre-trained on the ImageNet dataset. We let $\varepsilon = 0.3$ and the iteration number as 10. We randomly select 100 images from the ImageNet test set and generate the adversarial examples by using the reversed models. Then, we test the accuracy of the compiled model under these adversarial examples. As shown in Table VIII, the compiled models' top-1 accuracy drop is about 88.65% on average and the top-5 accuracy drop is about 80.785% on average.

TABLE VIII: Model Performance under Adversarial Attack

Model	Attack	Top1-Acc [Before Attack]	Top5-Acc [Before [Attack]	Top1-Acc [After Attack]	Top5-Acc [After Attack]
Xception	PGD-10	0.79	0.95	0.03	0.04
DenseNet121	PGD-10	0.69	0.91	0.03	0.06
ResNet50	PGD-10	0.70	0.92	0.28	0.67
ResNet50V2	PGD-10	0.67	0.87	0.02	0.07
MobileNet	PGD-10	0.71	0.91	0.04	0.04

2) Backdoor Attack: We perform the Hidden Trigger Backdoor Attack on five models that are pre-trained on the CIFAR-10 dataset. We construct a dataset with 2500 samples and generate 75 poison samples. We fine-tuned five reversed models with our poison dataset. As shown in Table IX, the backdoor attack has about 24.34% attack success rate on average. Meanwhile, the backdoor attack causes an acceptable 2.34% accuracy drop.

TABLE IX: Model performance under the Backdoor Attack

Model	Ori Top-1 ACC	Top1-Acc [After Backdoor]	Poison Succ
Xception	0.8133	0.7954	0.3300
DenseNet121	0.7207	0.6899	0.4640
ResNet50	0.6470	0.6329	0.1360
ResNet50V2	0.6369	0.6871	0.1620
MobileNet	0.6150	0.6019	0.1250

3) Increasing the attack surface: Our research delves into the risks of deploying deep learning (DL) models in real-world scenarios. Experimental results demonstrate that attackers can employ reverse engineering to fully access models across various applications. Furthermore, the recovered models can be exploited for adversarial or backdoor attacks, often resulting in more severe consequences than attacks on the original models. However, existing studies (e.g., [6], [21]) often lack the prior knowledge needed to address situations where models are accessed as white boxes through attack vectors. Our research showcases that reverse engineering and deep learning-based model recovery attacks can successfully lead to direct white-box attacks on models, thus increasing the success rate of these attacks. This implies that existing studies may underestimate the associated risks.

Answer of RQ4: This implies that our model reverse attack holds deeper exploratory value at the extension attack level, underscoring that this type of attack is extremely dangerous and warrants thorough investigation by researchers.

V. Discussion

Other Compilers: The essence of our work is also applicable to other DL compilers such as XLA and Glow. Typically, enabling this functionality (reconstructing models and inferring operators) for a specific compiler merely requires training REDLC using binary files compiled by such compilers. For instance, with GLOW, heuristic methods can be employed to infer operator layer dimensions and extract the CFG from the generated binary files containing computation logic and parameters. This process is similar to the analysis described in Section III-B, albeit with different specifics. Beyond this, the inference and correction parts can fully utilize the REDLC methodology. Due to space constraints, the decompilation analysis for GLOW is not included within the scope of this paper. We chose to focus on TVM for technical discussion because of its representativeness in architectural design, customizability, flexibility, and compatibility. Moreover, decompiling TVM binary files is more challenging as it produces fully standalone executables, whereas some other compilers (e.g., XLA) generate executables that are linked to kernel libraries, thereby simplifying the reconstruction process.

Potential defense strategies: Strategies such as fuzzing and obfuscation are meaningful but come with trade-offs. Applying fuzzing to models can increase inference time or lead to higher memory consumption on devices. If obfuscation is applied to the computational graph, the mathematical correctness of operator calculations and the effects of compiler optimizations (like the removal of redundant operators) must be considered. Certain defense strategies centered on memory access patterns, such as oblivious shuffle, address space layout randomization, and dummy memory accesses, have gained popularity. However, these methods can significantly reduce performance, especially for models with many operators that frequently access memory. Another important question is how much the inference accuracy of a model protected by a defense framework decreases after a model extraction attack. This could offer valuable insights into the effectiveness of defense strategies against such attacks.

Extending to more architectures: REDLC exposes a new and practical attack surface by enabling the recovery of complete DNNs from accessible executable DNN files. However, the backend compilation platform is currently limited, supporting only binary reverse engineering with x86 as the backend. We plan to add support for additional backends and architectures in the future.

Vendor support: Nvidia offers a disassembler [22], but certain device architectures and ISAs are not fully disclosed, complicating further analysis. The lack of vendor support makes porting work challenging for some DL accelerators. Moreover, executable files generated by DL compilers for GPUs and CPUs differ, and some compilers lack mature technical support, presenting challenges for current extension research.

VI. Related Work

A. Query-based Model Extraction Attack

Traditionally, attackers focused on stealing models deployed on remote servers. In this traditional method, attackers exploit vulnerabilities such as inference APIs or side-channel attacks

to extract information about the model. When the attackers use remote deep learning servers, they use API to extract the model in a black box scenario. They can use collected samples to query the target model to reconstruct it [23], or use synthetic samples to steal the information of target models [24]. Correia et al. [9] propose that model extraction attacks require multiple queries and computational resources for training the substitution models. However, the accuracy of the corresponding recovered model still remains unacceptable.

B. Reverse Engineering-based Model Extraction Attack

There has been significant attention from researchers on utilizing reverse engineering techniques to steal local models. Xu et al. [4] conducted extensive research on mobile DL models. Wu et al. [14] introduced a DL decompiler, DND, which extracts DL model operators through symbolic execution and loop analysis. Liu et al. [15] proposed BTD, which like the DND approach, utilizes heuristic methods to infer deep learning model operators, employs dynamic analysis to reveal network architectures, and uses symbolic execution to aid in inferring the dimensions and parameters of DNN operators. However, relying on template matching to identify and recover operators limits scalability.

VII. Conclusion

In this paper, we study a new attack surface on compiled DL models and propose an attack pipeline to reverse DL models. We exploit the disassembly information to reconstruct the complete model computation graph with learning methods. Furthermore, we design a GCN-based correction method to improve the precision of the inference for the model details. We also present the adversarial example attack and the backdoor attack by using these reversed models on corresponding compiled models. The results of these two attacks present that our reversed models can cause significant threats.

Acknowledgment

This work is supported by the National Key RD Program of China (No.2021YFB3100700), the National Natural Science Foundation of China (No. U22B2029, 62272228)

References

[1] B. Olney, "Secure reconfigurable computing paradigms for the next generation of artificial intelligence and machine learning applications," Ph.D. dissertation, University of South Florida, 2023.

[2] O. Sharir, B. Peleg, and Y. Shoham, "The cost of training nlp models: A concise overview," *arXiv preprint arXiv:2004.08900*, 2020.

[3] P. Ren, C. Zuo, X. Liu, W. Diao, Q. Zhao, and S. Guo, "Demistify: Identifying on-device machine learning models stealing and reuse vulnerabilities in mobile apps," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[4] M. Xu, J. Liu, Y. Liu, F. X. Lin, Y. Liu, and X. Liu, "A first look at deep learning apps on smartphones," in *The World Wide Web Conference*, 2019, pp. 2125–2136.

[5] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *2017 IEEE symposium on security and privacy*. IEEE, 2017, pp. 3–18.

[6] Y. Li, J. Hua, H. Wang, C. Chen, and Y. Liu, "Deeppayload: Black-box backdoor attack on deep learning models through neural payload injection," in *2021 IEEE/ACM 43rd International Conference on Software Engineering*. IEEE, 2021, pp. 263–274.

[7] Y. Huang, H. Hu, and C. Chen, "Robustness of on-device models: Adversarial attack to deep learning models on android apps," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 2021, pp. 101–110.

[8] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction APIs," in *25th USENIX security symposium*, 2016, pp. 601–618.

[9] J. R. Correia-Silva, R. F. Berriel, C. Badue, A. F. De Souza, and T. Oliveira-Santos, "Copycat cnn: Stealing knowledge by persuading confession with random non-labeled data," in *2018 International joint conference on neural networks*. IEEE, 2018, pp. 1–8.

[10] X. Hu, L. Liang, S. Li, L. Deng, P. Zuo, Y. Ji, X. Xie, Y. Ding, C. Liu, T. Sherwood *et al.*, "Deepsniffer: A dnn model extraction framework based on learning architectural hints," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 385–399.

[11] Y. Zhu, Y. Cheng, H. Zhou, and Y. Lu, "Hermes attack: Steal DNN models with lossless inference accuracy," in *30th USENIX Security Symposium*, 2021.

[12] X. Qi, T. Xie, R. Pan, J. Zhu, Y. Yang, and K. Bu, "Towards practical deployment-stage backdoor attack on deep neural networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 13 347–13 357.

[13] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations." in *NDSS*. Citeseer, 2015.

[14] R. Wu, T. Kim, D. J. Tian, A. Bianchi, and D. Xu, "DnD: A Cross-Architecture deep neural network decompiler," in *31st USENIX Security Symposium*, 2022, pp. 2135–2152.

[15] Z. Liu, Y. Yuan, S. Wang, X. Xie, and L. Ma, "Decompiling x86 deep neural network executables," in *32nd USENIX Security Symposium*, 2023, pp. 7357–7374.

[16] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv:1301.3781*, 2013.

[17] W. Guo, D. Mu, X. Xing, M. Du, and D. Song, "DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis," in *28th USENIX Security Symposium*, 2019, pp. 1787–1804.

[18] M. Chen, Z. Wei, Z. Huang, B. Ding, and Y. Li, "Simple and deep graph convolutional networks," in *International conference on machine learning*. PMLR, 2020, pp. 1725–1735.

[19] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 26, no. 10, pp. 1367–1372, 2004.

[20] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "TVM: An automated End-to-End optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation*, 2018, pp. 578–594.

[21] Y. Sang, Y. Huang, S. Huang, and H. Cui, "Beyond the model: Data pre-processing attack to deep learning models in android apps," in *Proceedings of the 2023 Secure and Trustworthy Deep Learning Systems Workshop*, 2023, pp. 1–9.

[22] C. binary utilities, 2021, https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html.

[23] A. S. Rakin, M. H. I. Chowdhuryy, F. Yao, and D. Fan, "Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories," in *2022 IEEE symposium on security and privacy*. IEEE, 2022, pp. 1157–1174.

[24] X. Yuan, L. Ding, L. Zhang, X. Li, and D. O. Wu, "Es attack: Model stealing against deep neural networks without data hurdles," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 6, no. 5, pp. 1258–1270, 2022.