DynaMO: Protecting Mobile DL Models through Coupling Obfuscated DL Operators

Mingyi Zhou*
Beihang University
Beijing, China
mingyi.zhou@monash.edu

Xiang Gao
Beihang University
Beijing, China
xiang_gao@buaa.edu.cn

Xiao Chen
University of Newcastle
Callaghan, Australia
xiao.chen@newcastle.edu.au

Chunyang Chen
TU Munich
Heilbronn, Germany
chun-yang.chen@tum.de

John Grundy
Monash University
Clayton, Australia
john.grundy@monash.edu

Li Li†
Beihang University, Beijing
Yunnan Key Laboratory of Software Engineering, China
lilicoding@ieee.org

ABSTRACT

Deploying deep learning (DL) models on mobile applications (Apps) has become ever-more popular. However, existing studies show attackers can easily reverse-engineer mobile DL models in Apps to steal intellectual property or generate effective attacks. A recent approach, Model Obfuscation, has been proposed to defend against such reverse engineering by obfuscating DL model representations, such as weights and computational graphs, without affecting model performance. These existing model obfuscation methods use static methods to obfuscate the model representation, or they use half-dynamic methods but require users to restore the model information through additional input arguments. However, these static methods or half-dynamic methods cannot provide enough protection for on-device DL models. Attackers can use dynamic analysis to mine the sensitive information in the inference codes as the correct model information and intermediate results must be recovered at runtime for static and half-dynamic obfuscation methods. We assess the vulnerability of the existing obfuscation strategies using an instrumentation method and tool, *DLModelExplorer*, that dynamically extracts correct sensitive model information (*i.e.*, weights, computational graph) at runtime. Experiments show it achieves very high attack performance (*e.g.*, 98.76% of weights extraction rate and 99.89% of obfuscating operator classification rate). To defend against such attacks based on dynamic instrumentation, we propose *DynaMO*, a Dynamic Model Obfuscation strategy similar to Homomorphic Encryption. The obfuscation and recovery process can be done through simple linear transformation for the weights of randomly coupled eligible operators, which is a fully dynamic obfuscation strategy. Experiments show that our proposed strategy can dramatically improve model security compared with the existing obfuscation strategies, with only negligible overheads for on-device models. Our prototype tool is publicly available at https://github.com/zhoumingyi/DynaMO.

KEYWORDS

SE for AI, AI safety, on-device AI

ACM Reference Format:
Mingyi Zhou, Xiang Gao, Xiao Chen, Chunyang Chen, John Grundy, and Li Li. 2024. *DynaMO*: Protecting Mobile DL Models through Coupling Obfuscated DL Operators. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3694998

1 INTRODUCTION

More and more mobile applications (Apps) and IoT devices are leveraging deep learning (DL) capabilities. Deploying DL models on such devices has gained great popularity as it avoids transmitting data and provides rapid on-device processing. It also enables applications to access their DL model offline. As the computing power of mobile and edge devices keeps increasing, it also reduces the latency of model inference and enables the running of large on-device models.

However, as such DL models are directly hosted on devices, attackers can easily unpack the mobile Apps, identify DL models through keyword searching, extract key information from the DL models, formulate attacks on the models, or even copy them and reuse them in their own software. This accessible model key information thus makes it easy to launch attacks or steal the model's intellectual property [38]. To protect such on-device DL models, TFLite, the most commonly used on-device DL model framework, compiles the general DL model (such as TensorFlow and PyTorch models) to TFLite models, which disables direct white-box attacks. This is done by disabling the gradient calculation of the on-device models, which is essential for conducting effective white-box attacks. Such models are called non-differential models (*i.e.*, non-debuggable models). Such model compilation makes it hard for attackers to reverse engineer the on-device model. However, these on-device platforms still suffer from significant security risks. Recent attack methods [6, 13, 18, 38] can parse model information in

*This work was partially done when Mingyi Zhou was a PhD student at Monash University
†Corresponding author.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3694998

the on-device model (*e.g.*, `.tflite`) files and then reverse engineer them.

Recent model obfuscation approaches propose to use static or dynamic methods to obfuscate the representation of on-device models [3, 36, 37]. Such DL model representations produced by model obfuscation methods cannot be understood by automatic tools or humans, but will not affect the model performance *et al.* [37]. The information inside model representations (*e.g.*,.tflite files) is protected by elaborating static obfuscation strategies, *e.g.*, operator renaming. However, DL models obfuscated by fully static methods must have the same computing process (*i.e.*, the whole data computing process from inputs to outputs) as the original model. The Mindspore platform [3] uses a 'half-dynamic' approach that produces an obfuscated model that defines different computing processes to the original model, requiring additional input arguments from users to restore the correct computing process at the runtime. As developers need the correct input arguments to get the correct model output in mobile Apps, attackers can also find the arguments by unpacking the App and extracting the actual API calling steps [24]. This 'half-dynamic' obfuscation can be considered a special form of static DL model obfuscation.

To analyse and understand the limitations of these current static or half-dynamic obfuscation strategies, we designed a dynamic instrumentation method, *DLModelExplorer*. *DLModelExplorer* can identify the actual inference function of each operator, extract the sensitive data (*e.g.*, model weights), and identify the obfuscating components of the obfuscated DL model through dynamic instrumentation. Our tool can recover nearly all obfuscated information to the original form (*e.g.*, 100% of weights extraction rate and 99.87% operator classification accuracy). This shows a major need to use robust obfuscations that can defend against dynamic instrumentation. We then propose a fully dynamic obfuscation strategy, that we call Dynamic Model Obfuscation and build a tool *DynaMO* to realiase it. *DynaMO* adopts an idea similar to Homomorphic Encryption to randomly sample eligible operators to form obfuscation propagation paths to obfuscate the information from the start of the path and recover the results at the end of the path, thus building a random obfuscation-recovery operator pair. The process better secures the obfuscated model by the intermediate results and model information inside the obfuscation propagation path is obfuscated and will no be recover. The obfuscation and recovery process can be performed through simple linear transformation of the DL model weights, which avoids introducing overhead to the inference process and is hard for attackers to identify. Such dynamic DL model obfuscation can prevent instrumentation methods from collecting correct information about the inference code of each operator. Our experiments show that our method can significantly secure the model information compared with existing mobile DL model obfuscation strategies. Our proposed approach also introduces negligible overheads to the model inference.

The key contributions in this work include:

- We propose an attacking method using dynamic instrumentation to demonstrate key limitations with existing model obfuscation strategies. This can automatically acquire real information from the model inference functions at runtime.
- We analyse the limitations of existing obfuscation methods and propose a novel solution that can defend against the proposed instrumentation. It introduces obfuscation to the intermediate results of model inference.
- We have shown that our *DynaMO* method can significantly increase the obfuscation in the model inference process with only negligible performance and efficiency loss compared with existing obfuscation strategies.
- We open-sourced our prototype tool *DynaMO* [35] in a GitHub repository: https://github.com/zhoumingyi/DynaMO.

In the following sections, we provide a motivation for our work and introduce the basic background of this study. We then conduct an experiment to demonstrate the key limitations of static and half-dynamic obfuscation methods. We propose a new approach and tool, and evaluate this on real-world mobile DL models. We discuss key findings, limitations and future research directions.

2 BACKGROUND AND RELATED WORKS

2.1 Terminology

According to their defending performance against different kinds of attacks, we use static method and dynamic methods to distinguish the different obfuscation methods. We call the existing model obfuscation methods [37] as **static model obfuscation** as they only obfuscate the model representation in the compilation. So, they can just defend against the attacks based on static analysis for on-device models. In contrast, our method is **dynamic model obfuscation** as it can obfuscate the model information that is generated at runtime. Thus, it can defend against dynamic instrumentation.

2.2 DL Frameworks

Deep Learning (DL) Frameworks: The open-source community has developed many well-known **DL frameworks** to facilitate users to develop DL models, such as TensorFlow [5], Keras [7], and PyTorch [22]. These frameworks provide standards for developing DL models [10]. PyTorch is one of the latest DL frameworks which has gained academic user popularity for its easy-to-use and high performance. In contrast, TensorFlow is widely used by industry to develop new DL-based systems because it has the most commonly used on-device DL library, Tensor Flow Life (TFLite). TFLite is the most popular library for DL models on smartphones, as it supports various hardware platforms and operation systems.

2.3 On-device DL Frameworks

On-device DL Frameworks: TensorFlow provides a tool *TensorFlow Lite Converter*[1] to convert TensorFlow models into TFLite models. A compiled TFLite model can then be run on mobile and edge devices. However, it does not provide APIs to access the gradient or intermediate outputs like other DL models.

Traditionally, on-device models are released as **DL files** that are deployed on devices. Mobile app code then accesses these models through a dedicated **DL library**, such as the TFLite library if the AI model is developed using the TFLite framework. Each model file contains two types of information: **computational graph** and **weights**, which record the model's architecture and parameters

[1] https://www.tensorflow.org/lite/convert/index

tuned based on the training dataset, respectively. Such a computational graph is usually a multi-layer neural network. In the network, each layer contains an **operator** that accepts **inputs** (i.e., the outputs of the previous operator), **weights** (i.e., stored in the dedicated file that is pre-calculated in the training phase), and **parameters** (i.e., configuration of the operator. For example, the `conv2d` layer in TFLite requires the parameters of stride size and padding type. Their parameters will affect the outputs of layers.) to conduct the neural computation and outputs the results for the next operator.

2.4 DL Model Attacks

DL models deployed on devices are subject to a range of attacks [13, 18, 21, 32, 34, 39]. These can include tricking the DL model with perturbed inputs into, e.g., classifying an image incorrectly; extracting model information to facilitate other attacks; stealing a copy of the model (which may have been very expensive to produce) for use in one's own application; and others. These attacks can be black-box [13, 18, 33] or white-box [34]. Access to DL components and/or access to DL models facilitates these attacks.

2.5 Code Obfuscation

Code obfuscation methods were initially developed to hide the functionality of malware. The software industry also uses it against reverse engineering attacks to protect code IP [25]. Code obfuscators provide complex obfuscating algorithms for programs like JAVA code [8, 9], including robust methods for high-level languages [28] and machine code level [31] obfuscation. Code obfuscation is a well-developed technique to secure the source code. However, solely relying on traditional code obfuscation approaches cannot effectively protect on-device models, especially in terms of protecting the structure of DL models and their parameters.

2.6 Model Obfuscation

To prevent attackers from obtaining detailed information on deployed DL models, model obfuscation has been proposed. This obfuscates model representations such as the weights and model architectures [37]. The *renaming* and *parameter encapsulation* can prevent most model parsing or reverse engineering methods from extracting the key information (*e.g.*, weights and computation graph) of the deployed model. In the scenarios where computational costs are critical, the *neural structure obfuscation* and *shortcut injection* do not introduce any additional overhead as they just add misleading information to the model representation but will not modify any inference process. For structure obfuscation, developers can use *shortcut injection* and *extra layer injection*. These methods can increase the difficulty of understanding the model structure of the deployed models.

The most vulnerable obfuscation methods are *neural structure obfuscation* and *shortcut injection*. These do not change any inference process but just use to mislead the attacker when parsing the model information. For *neural structure obfuscation*, attackers can analyse the real data flow in the model inference process to obtain the real neural architecture from the shape of intermediate data. TFLite actually provides an official API to get such information. For *shortcut injection*, we can slightly modify the model like the paper in [18] to remove each shortcut and check whether the model inference works well. We can identify the obfuscating shortcut that will not actually be used in the inference. However, we cannot do the same process for *extra layer injection* because the obfuscating extra layer actually participates in the model inference (although it will not affect the model output). Therefore, in our study, we mainly focus on analysing the robustness of three obfuscation methods (shown in Figure 1): *i.e.*, *renaming*, *parameter encapsulation*, and *extra layer injection*.

2.7 Motivation for Our Work

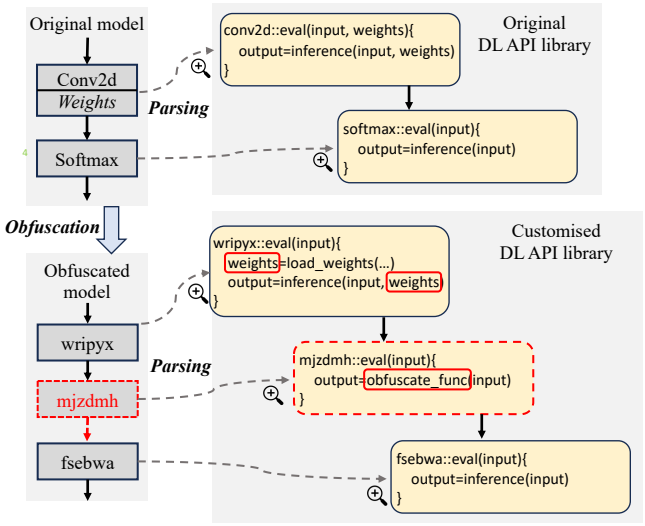

Figure 1: Demonstration of existing model obfuscations [37]. Here, existing model obfuscation hides the weights of `conv2d` operator, renames the original operator name to random strings (`conv2d` → `wripyx`), and injects an extra obfuscating operator (*i.e.*, `mjzdmh`). The customised DL API library is generated to execute the inference of the obfuscated model. The function `{OP_NAME}::eval` is the code implementation of the operator's forward inference. The extra operator `mjzdmh` only has an obfuscating function `obfuscate_func` to copy the input value to the output.

Statically obfuscated DL model representations on mobile devices are still directly exposed to threats. As shown in Figure 1, an original DL API library will use the operator's name to locate the code implementation of the operator's forward inference to build a correct function call graph. We refer to this as the ***inference code of DL operators*** in this paper. Although the name of the obfuscated DL operators is randomly generated by *renaming*, *e.g.*, `conv2d` → `wripyx`, attackers can still use the operator's name to locate the inference code for each operator. For example, when the DL library gets an operator's name `{OP_NAME}`, it will use the function `{OP_NAME}::eval` to perform the forward inference of the operator.

Existing model obfuscation strategies use static or half-dynamic ways to obfuscate model representation. To produce correct model outputs, each obfuscated information or its produced results needs

to be recovered in the inference code of operators at runtime. **Therefore, attackers can use dynamic analysis to extract the correct model representation from the DL model's inference code at runtime.** For example, in Figure 1, attackers can hook a data collection function to the `wripyx::eval` to obtain the correct model weights. In addition, they can modify the `obfuscate_func` function at runtime to identify whether an operator is an extra obfuscating operator (the extra operator should not affect the model output [37]).

2.8 Research Questions

To assess and enhance the robustness of existing mobile DL model protections, this study aims to address the following key research questions:

- **RQ1 - What are the limitations of existing model obfuscation methods?**
- **RQ2 - How can we better defend DL models against dynamic instrumentation attacks?**
- **RQ3 - How efficient is our proposed obfuscation strategy?**

3 RQ1: MODEL DEOBFUSCATION

In this study, we use the most commonly used on-device model on Android, TFLite models, as our target.

3.1 Threat Model

The on-device DL models and their corresponding third-party API library used for model inference are packed into mobile applications on devices like Android. In the model deobfuscation process, attackers need to unpack the mobile application using reverse engineering tools like *apktool* [2] to get the on-device DL models and related API libraries. The DL model representation is usually stored in a separate model file such as `.tflite` files for TFLite models. As the existing commonly used DL platforms are usually open-sourced like TensorFlow, PyTorch, and ONNX, even if the on-device model has been obfuscated, attackers can obtain the name of DL operators, such as a random string, used in the model inference for parsing the computational graphs. Thus, attackers can use the collected names of each operator to identify the actual inference function of each operator in the API library, because DL libraries use the operator name to determine the actual inference function at runtime.

```
void {op_name}::Eval {
  // Create essential data for the operator
  TfLiteTensor* input = create_input(input_data);
  TfLiteTensor* weights = create_weights(weights_data);
  ConvParams params = create_params(paras_data);
  TfLiteTensor* output;
  // The computing function for operator
  multithreaded::Conv(params, input, weights, output);
}
```

Listing 1: Simplified backend inference function of `Conv2d` operator for running on a multithreaded CPU.

After identifying the inference function of each DL operator, attackers do not need to understand all the compiled code used for the operator inference in the DL API library, which is a very complex task. But as the source code of TFLite and its related tools are open-source, for example, they can identify the API of loading the `weights_data` into the memory and be used to build the corresponding TFLite tensor (*i.e.*, `TfLiteTensor* weights` in Listing 1). Note that the construction processes of `weights` are the same in different DL operators. Thus, attackers can synthesise test samples and use dynamic instrumentation analysis to hook a data collection function in the inference code to get the real model information.

3.2 *DLModelExplorer*

Existing model obfuscation methods can obfuscate multiple kinds of model information such as the functionality of operators (*i.e.*, operator name), model weights, neural architectures, and the spatial relationship among operators. It is hard for attackers to parse the model representation without any auxiliary information. However, the model representations must store the names of the operators to provide the information on which inference codes need to be used to produce the model outputs. Attackers can use the operator's name (even if it is obfuscated) to identify the inference code of operators, thus attackers can use dynamic instrumentation and analysis to extract the correct model information inside the inference code.

Overview. The overview of our proposed method *DLModelExplorer* is shown in Figure 2. (1) Our method *DLModelExplorer* first parses the obfuscated model and determines which inference function will be used at runtime for each operator. (2) Then, our proposed tool can use the instrumentation method to hook a data collection function to the construction process of model weight data. This function can automatically extract the weights of each operator and save them into a separate file. (3) Next, our tool will automatically modify the actual inference function for each operator to filter out which operator is the obfuscating operator. The obfuscating operator produces the output that is equal to the input or modifies the output but will not change the output of the next operator. (4) Finally, after getting the weights and the real computation graph of the on-device models. Our tool will then analyse the relation between inputs, outputs, and weights to infer the functionality (*i.e.*, the real name) of operators. We implement *DLModelExplorer* using dynamic program instrumentation framework ***Pin*** [4].

Model Parsing. We first write a script to automatically synthesise the inputs as the specification and feed them into the on-device model. Thus we can use dynamic instrumentation and analysis to parse the model. We need to collect the names of operators (obfuscated as random strings) in the DL model. Our tool traces the execution process of the model inference, and then identifies the executed API functions that contain the operator's name. In the implementation of TFLite, each operator's source code will have four basic functions: `prepare` (pre-allocate the intermediate data), `init` (initialize the essential data), `free` (remove the data after the inference), `eval` (the computing code of inference). The source codes of each function will be written in a C++ `workspace` named as the operator name. Therefore, we can use the operator's name in model representations to identify the corresponding workspace and locate the `eval` function which has the code implementation of model inference. The identified inference code will be used in the dynamic instrumentation to deobfuscate the model information.

Hook Adding. After identifying the core inference function of each operator, *DLModelExplorer* extracts the model weights of each

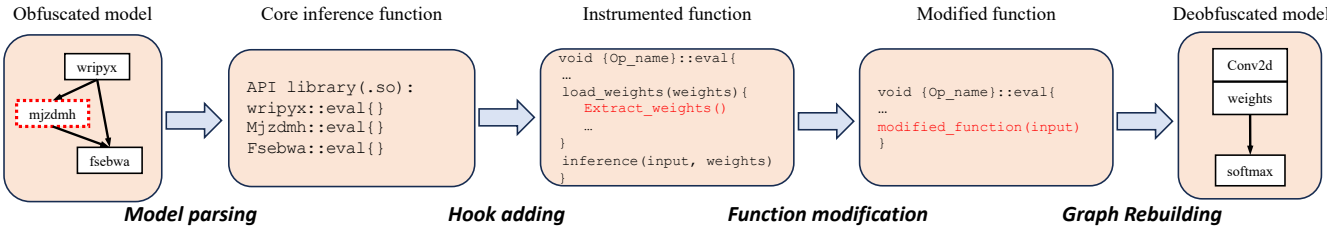

Figure 2: Overview of our proposed model deobfuscation method *DLModelExplorer*. The `mjzdmh` operator with a red dotted block is an extra obfuscating operator.

operator. Although the weights can be obfuscated (*e.g.*, store them in different places and assemble them before inference), the inference function needs the correct value to perform the right DL model inference computing process. Therefore, *DLModelExplorer* will search for the weight tensor construction function in the inference function of the DL API library using dynamic analysis, to find the place where the weights data is loaded into the memory and can be extracted. As the TFLite and its related tools are open-sourced, we can get the keyword of the weight tensor construction function. *DLModelExplorer* will identify the tensor construction step of the API library through keyword searching. If the related construction function starts to be executed in the inference process of operators, our *DLModelExplorer* will automatically hook a data extraction function into the start of the construction to parse the input data of the construction function and store the extracted weights data in a separate file. Note that, *DLModelExplorer* will also extract the parameter of the operator if it has, such as the padding value and stride size for `conv2d` operator. The parameter is stored as a `struc` data `params` in C++ source codes of TFLite. The process of extracting parameters is similar to model weights, so we omit the implementation details here. Then, the instrumented API library will automatically extract and save the real weights used in the model inference at runtime.

Function Modification. Another key aspect of on-device DL models is their computational graph. This can be obfuscated by extra obfuscating operator injection. Such extra obfuscating operators are used to participate in the inference process but will not affect the final results. For example, they can just produce the same output as the input, or produce an output out of the range of the input shape of the next operator which will not affect the results of the next operator. In the function modification process, *DLModelExplorer* will first monitor the inference process at runtime. As shown in Listing 2, if a function `{op_name}::eval()` starts to execute, our *DLModelExplorer* will dynamically insert a value-copy function to the inference function, *i.e.*, the `eval()` function, and add a stop signal of the inference function. For instance, the TFLite inference function will finally `return` a `TFLite_OK` (*i.e.*, equal to 0) to stop the execution and jump to the inference process of the next operator. We use the value-copy function as the modified computing function which will produce the same output as the input. Next, *DLModelExplorer* will compute the output difference between the instrumented API library and the original library for each operator under the same input. If the two outputs are not identical, it means the modified operator is a valid operator that belongs to the original model. Otherwise, the modified operator is an extra obfuscating operator because the modified function will not affect the model outputs.

```
void {op_name}::Eval {
  // Create essential data for the operator
  TfLiteTensor* input = create_input(input_data);
  TfLiteTensor* output;
  // Copy the value of input to output
  copy_value(input, output);
  return TFLite_OK;
  // The original implementation of the operator
  ...
}
```

Listing 2: Modified inference function of DL operator

Graph Rebuilding. After removing the extra obfuscating operators and extracting the real model weights for each operator, the only obfuscated information we left is the functionality of the operator, *i.e.*, the name of the operators. For the operators with weights, the functionality of operators can be inferred by the data shape (*i.e.*, input, output, and weights). For example, the output shape of `conv2d` operator can be found as follows:

$$H_{\text{out}} = \left\lfloor \frac{H_{\text{in}} + 2 \times \text{P}[0] - \text{D}[0] \times (\text{W\_Size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$
$$W_{\text{out}} = \left\lfloor \frac{W_{\text{in}} + 2 \times \text{P}[1] - \text{D}[1] \times (\text{W\_Size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor \quad (1)$$

Where the H_{in} and H_{out} are the dimensions in height of input data and output data, respectively. The P (*i.e.*, padding), D (*i.e.*, dilation), and stride are the parameters (*i.e.*, the setting) of the `conv2d` operator. W_Size is the shape of the weights. In commonly used DL models, there are only a limited number of operators with weights like `conv2d`, `depthwise_conv2d`, `fullyconnected`. These operators have different output shapes for the same input and weights shape. In our *DLModelExplorer*, it will use the output size to identify the name of operators according to the predefined size transformation rules of potential operators (`conv2d`, `depthwise_conv2d`, and `fullyconnected`), *e.g.*, Equation (1). For the operators without weights (*e.g.*, `softmax`, `relu`, and `add`), *DLModelExplorer* will extract the input and output data of each obfuscated operator at runtime. Then, it uses the input to test different operators in a pre-collected list. Note that to produce this operator list, we can collect the operators from the TFLite operator list. Because the parameter data of these operators have various construction processes, we cannot extract the parameter data for all operators. So, to develop a robust analysing method, we do not collect the parameter data to

Table 1: Deobfuscation performance of the proposed *DLModelExplorer*. WER, WEA, OCA, NIR, and SS are the five metrics used to measure deobfuscation performance.

Metric	Fruit	Skin	MobileNet	MNASNet	SqueezeNet	EfficientNet	MiDaS	LeNet	PoseNet	SSD	**Average value**
WER	100%	100%	100.0%	98.15%	96.30%	98.04%	95.10%	100.0%	100%	100%	**98.76%**
NIR	100%	100%	96.77%	98.41%	97.50%	100%	91.18%	100%	100%	100.0%	**98.39%**
OCA	100%	100%	100%	98.88%	100%	100%	100%	100%	100%	100%	**99.89%**
WEE	2.8×10^{-5}	1.3×10^{-5}	2.6×10^{-5}	3.0×10^{-5}	3.6×10^{-5}	2.3×10^{-6}	1.9×10^{-5}	8.8×10^{-7}	2.5×10^{-5}	2.5×10^{-5}	$\mathbf{2.1 \times 10^{-5}}$
SS	1.0	1.0	0.97	0.98	0.97	1.0	0.91	1.0	1.0	1.0	**0.98**

predict the functionality of operators without weights. We use the input-output relationship to guess the parameter of each candidate operator and compute the output using the guessed parameter. If the test operator produces the same output as the obfuscated operator of the on-device model, the name of the test operator is the name of the obfuscated operator as they share the same functionality. Thus, our proposed *DLModelExplorer* can parse the functionality of each valid operator with a random obfuscating name.

3.3 Deobfuscation performance

3.3.1 ***Dataset.*** To evaluate *DLModelExplorer*'s performance on deobfuscating on-device DL models with various structures for multiple tasks, we use the 10 TFLite models, as used in [37]. These include a fruit recognition model, a skin cancer diagnosis model, MobileNet [12], MNASNet [26], SqueezeNet [15], EfficientNet [27], MiDaS [23], LeNet [17], PoseNet [16], and SSD [20]. The fruit recognition and skin cancer diagnosis models are collected from Android apps. The other models were collected from the TensorFlow Hub [2].

3.3.2 ***Results.*** We first evaluate the deobfuscation performance of *DLModelExplorer*. Existing model obfuscation methods can obfuscate multiple model information that includes weights, the functionality of each operator, and computational graphs. **We use five deobfuscation metrics to measure the performance of *DLModelExplorer*, including Weights Extraction Rate (WER), Weights Extraction Error (WEE), Operator Classification Accuracy (OCA), Name Identification Rate (NIR), and Structure Similarity (SS).** Note that the two metrics WEE and SS represent the global performance in weight and structure extraction.

The formula of WER is $\mathbf{WER} = n/m$, where n is the number of extracted weights with an element-wise maximal error of the output difference less than 1×10^{-4} compared with the original weights. The original weights are extracted by official APIs provided by TFLite. We choose 1×10^{-4} as the threshold because we consider the error in computing and extraction (The error in extracting the original weights of the model using TFLite APIs is within 5 decimal places.). m is the total number of weights that need to be extracted. The WEE is used to measure the average difference of the extracted weights and original weights, which can be formularized by $\mathbf{WEE} = \frac{1}{N}\sum_1^N \max(|W'_n - W_n|)$, where W'_n and W_n are the extracted weights and the original weights of the $n-$th operator, respectively. Note that we only compute the WEE for the successfully extracted weights (the maximal error of the output difference less than 1×10^{-4} compared with the original weights).

[2]https://tfhub.dev/

The **OCA** is a binary classification metric. We only classify whether an operator is a valid operator or an obfuscating extra operator. For NIR, it can be computed by $\mathbf{NIR} = i/j$, where i is the number of successfully identified operators, and j is the total number of operators that have been classified as valid operators. We collect an operator list (See our code repository), and identify which operator in the list the renamed operator is equal to. For SS, if one operator is misclassified in NIR or OCA, it can be considered an error point. Then we use the number of error points divided by the number of operators to compute the SS.

The deobfuscation results are shown in Table 1. The *DLModelExplorer* achieves 98.76% of Weights Extraction Rate (WER), 99.89% of Operator Classification Accuracy (OCA), 98.39% of Name Identification Rate (NIR), and 0.98 pf Structure Similarity (SS). That means our instrumentation analysis method can effectively deobfuscate the on-device model. In addition, our method achieves low Weights Extraction Error (WEE).

3.3.3 ***Analysis.*** Our results show that existing mobile DL model obfuscation methods are not robust. The reason is we need to deploy the model representation on mobile devices to guarantee the DL API library performs the right inference computations. Existing model obfuscation strategies need to restore the correct information and execute the correct inference process for each operator at runtime, thus enabling the attacking exploitation based on the dynamic instrumentation. In addition, *DLModelExplorer* only needs a maximum of $2 \times n$ (some operators don't have any weights) times of inference to collect the information of an n-layer model. In our experiments, attackers only need 5-20 minutes to finish all steps for one model. Therefore, existing obfuscation methods are very vulnerable to dynamic instrumentation-based attacks. This shows that we need to develop a new approach to defence against attacks based on dynamic instrumentation.

> RQ1: Our *DLModelExplorer* can effectively extract the sensitive information of obfuscated models generated by existing obfuscation strategies. Existing obfuscation methods are not robust for the instrumentation-based attack.

4 RQ2: DYNAMIC OBFUSCATION STRATEGY

4.1 Approach Requirements

Despite using *renaming*, *parameter encapsulation*, and *extra layer injection* static obfuscation methods, RQ1 results show that attackers can use dynamic instrumentation to extract correct DL model

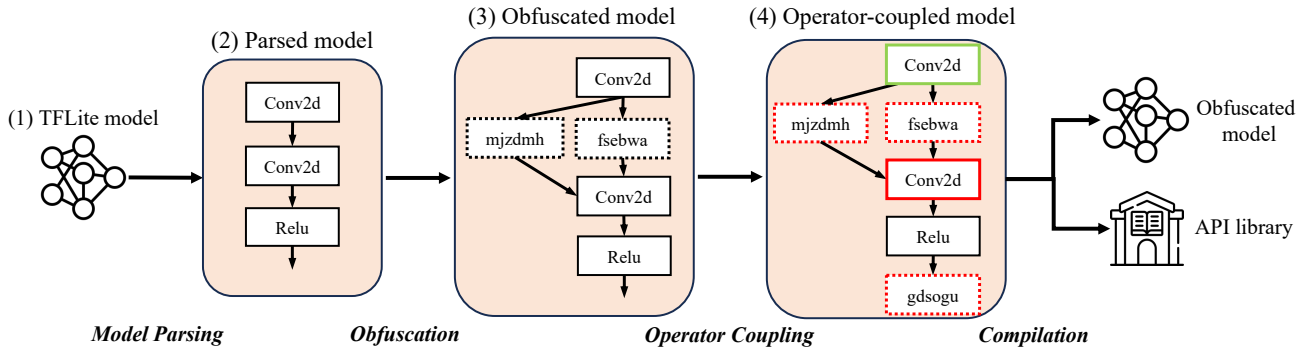

Figure 3: Overview of our proposed *DynaMO*. The dotted block denotes the extra obfuscating operator. The green block denotes the selected operator in weight transformation obfuscation. The red block (both dotted and solid) denotes the eligible coupled operators that can be used to recover the correct results. Note that we do not obfuscate the names of valid operators to demonstrate the process clearly.

information at runtime. Although there are many existing methods, such as various cryptographic techniques, that might be used to protect a DL model, it is hard to use them to protect on-device models, as such a model is very sensitive to computational cost. In addition, we should avoid requiring additional hardware support, as existing mobile platforms like Android cannot provide specific needed hardware/software environments to millions of Apps.

To disable instrumentation-based analysis, as used in *DLModel-Explorer*, we need to not only provide the obfuscating information to the attackers but also feed obfuscating information to the inference code of operators. However, it is hard to feed the obfuscating information to the inference process but finally restore the results to the correct one, as well as add enough randomness to the process. For example, a straightforward strategy is transforming the weights or outputs of one operator and performing the inverse process in the next operator (similar protection has been used in existing DL platforms like [3]). Such an obfuscation can be easily identified because it requires the next operator to perform a reverse transformation after transforming the weights in the previous operator (it has a different inference pattern than the normal DL model). Attackers can also use dynamic instrumentation to get the details of the result recovery process in the next operator and then deobfuscate the weights of the operator. If the weights of a valid operator i (the inference formula is $X_{i+1} = f(W_i^T X_i)$) are obfuscated by aW_i, the correct results of the operator need to be recovered by performing $\frac{1}{a}X_{i+1}$ in the start of the next operator. Attackers can identify the result-recovering process as the inference code needs to be modified to recover the correct results (adding another step $\frac{1}{a}X_{i+1}$). Then, they can use dynamic instrumentation to obtain the recovering parameter $\frac{1}{a}$, and deobfuscate the weights of the operator $\frac{1}{a}(aW_i) = W_i$.

4.2 The Dynamic Model Obfuscation Process

Our solution to defend against dynamic instrumentation adopts a concept akin to Homomorphic Encryption – that is, we use obfuscated information (*e.g.*, weights, architecture) to perform DL model inference to get *obfuscated results*. These results can then be recovered by a *deobfuscation step* after the computing process of several operators. To do this, we perform simple linear transformations for model weights in randomly picked operators. Then, we randomly choose eligible operators (including the extra obfuscating operators), which may not be the directly connected next operator. The extra obfuscating operators can be considered a special linear operator whose weight data is an identity matrix. It injects the corresponding inverse transformation in the weights of chosen eligible operators to recover the model results. This approach prevents attackers from using dynamic instrumentation in actual inference code functions of each operator to obtain the sensitive model information because the relation between the representation obfuscation and information recovery at runtime does not have a fixed pattern. It then becomes very hard to reverse engineer the steps of weight transformation and corresponding result recovery, even if attackers can parse one of these. In addition, extra obfuscating operators can be added to contribute to the model output, *i.e.*, recover the correct intermediate results using a linear transformation like other normal operators. Attackers cannot use dynamic code modification to identify such extra obfuscating operators.

The key steps of our proposed method, Dynamic Model Obfuscation (*DynaMO*), are outlined in Figure 3. We integrate our obfuscation strategy into the existing model obfuscation process. Our tool *DynaMO* first parses the on-device TFLite model. It then obfuscates it using existing model obfuscation methods, producing obfuscated operators. Compared with the existing obfuscation process, our method has one more step for coupling these obfuscated DL model operators. It uses a fully dynamic obfuscation method to compute intermediate results using obfuscated weights from a selected operator, and then recovers the correct results at the coupled operator. To achieve this, *DynaMO* obfuscates the weights of the selected operator like existing methods (*i.e.*, using simple linear transformation), but recovers the correct results by another weight transformation. That means *DynaMO* does not need to add an additional step to recover the results which is easy to identify. Attackers cannot identify it unless they know the original weights.

Unlike existing strategies, the choice of coupled operators has multiple possibilities. For example, it can obfuscate the weights of the green `conv2d` operator in Figure 3, and choose the `mjzdmh` and `fsebwa` or following `conv2d` operators as the coupled operators.

This can further increase the difficulty for attackers to identify the operator pair (*i.e.*, selected operator and coupled operator), even if attackers manage to identify one of them. In addition, previous extra obfuscating operator injection methods can only inject obfuscating operators that will not change the final output to the computation graph. In our *DynaMO*, the extra obfuscating operators (*e.g.*, `mjzdmh` and `fsebwa`) perform a simple linear computation, and can affect the final output when they are chosen as the selected operator or the coupled operator without significantly increasing the computational overhead. Thus, it is hard for the dynamic instrumentation and analysis proposed in *DLModelExplorer* to identify the extra obfuscating operator as it performs a similar linear computation process to other operators.

In summary, we propose a fully dynamic obfuscation strategy. It can obfuscate the intermediate results and model information without performance loss. It also increases the randomness in the obfuscation process, including the choice of the selected operator and the coupled operator, with negligible overhead compared with existing obfuscation methods. Thus, *DynaMO* can significantly increase the difficulty of the model deobfuscation. Note that we do not need to obfuscate all weights of on-device models if attackers cannot identify which weight data is obfuscated.

4.3 Obfuscation

The first step is obfuscating the on-device DL model using the existing obfuscation methods, including *Renaming*, *Weights encapsulation*, *Neural architecture obfuscation*, *Extra layer injection*, and *Shortcut injection*. The *Neural architecture obfuscation* and *Shortcut injection* are compatible with our strategy, but they will not affect the outcome of our method whether they exist in the obfuscation process or not. Therefore, we omit them in our analysis and visualization (*i.e.*, Figure 3). After parsing and obfuscating the on-device model using an existing tool like ModelObfuscator [37], we obtain the obfuscated DL model, shown in Figure 3.

4.4 Obfuscation Coupling

Notations: *DynaMO* will then perform obfuscation coupling to introduce the obfuscation to the intermediate results of the computing process. As we mentioned in the Overview section, consider a linear operator, we can obfuscate the weight of the selected operator using a linear transformation, which can be formulated as:

$$X_{n+1} = aW_n^T X_n + ab_n \tag{2}$$

where X_n is the input of the n-th operator of the DL model, x_{n+1} is the output of the n-th operator and also the input of $(n+1)$-th operator. W_n, b_0 are the weights of the operator. In this linear transformation, we use the product of a scale value a and original weights W_n, b_0 as the new weights. We can use such linear transformation to obfuscate the weights of any linear operators, including `FullyConnected`, `Conv2D`. The formula for `Conv2D` is different from the Equation 2, but it has similar properties. So, we use it to present all linear operators. **Note that, the extra injecting operator in existing model obfuscations also can be represented as a linear operator, where the W_n is equal to an identity matrix I_n, and the b_n is a zero vector.** Thus, the weights of the selected operator can be obfuscated. Next, we need to find the coupled operator to perform another linear transformation to the weights to recover the correct results. To show the process clearly, we first introduce the Coupled Weight Transformation rule for linear operators, which is shown as follows:

Lemma 4.1. ***(Coupled Weight Transformation on linear model)**: A sub-network f consists of multiple linear layers $\{L_1, L_2, \cdots, L_n\}$ and the i-th layer is $L_i : X_i = W_{i-1}^\top X_{i-1} + b_{i-1}$ where $i \in [1, n]$. The output of the sub-network f w.r.t. to the input X_0 would be $f(X_0)$. If W_1 is transformed to aW_1, W_n is transformed to $\frac{1}{a}W_n$, and b_i is transformed to ab_i for $i \in [1, n-1]$, then the transformed network f^s w.r.t. to the input X_0 would be $f^s(X_0)$ and we have $f^s(X_0) = f(X_0)$.* ***The proof can be found in our code repository.***

Explanation of Lemma 4.1: Through the Lemma 4.1, if we obfuscate the weights by a linear transformation (*i.e.*, aW_0, ab_0) in the first `conv2d` (the selected operator) with a green box (shown in Figure 3 (4)), We can recover the corrected results by performing a linear transformation (*i.e.*, $\frac{1}{a}W_n$, $\frac{1}{a}b_n$, where n denotes the operator ID of the coupled operator) in the coupled operator. The coupled operator could be {`mjzdmh`, `fsebwa`} as the output of the selected operator (the green `conv2d`) are shared to both `mjzdmh` and `fsebwa`, or the coupled operator could be the second `conv2d` with a red box. The coupled operator can not be `gdsogu` in Figure 3, as the previous operator `relu` is not a linear operator. So the coupled Weight Transformation rule will not exist for it.

We can follow the Coupled Weight Transformation rule to search the operator pair on linear models. Although the DL model usually has non-linear operators like `relu`, we can still find many eligible coupled operator pairs for the non-linear model unless all linear operators are followed by a non-linear operator. However, the most commonly used on-device DL architecture Convolutional Neural Networks (CNNs) usually have many nonlinear operators, *i.e.*, some `conv2d` operators followed by a `relu` operator in the Convolutional Neural Network, and the extra obfuscating operator cannot be injected between the *conv2d* and the followed `relu` operator as they are fused as one operator in TFLite to increase the inference efficiency. Thus, searching for the potential coupled operator will usually fail if we choose the *conv2d* as the selected operator to obfuscate. Therefore, to increase the utility of our method and extend the Coupled Weight Transformation rule for the propagation path has the nonlinear operator, we define a Coupled Weights Obfuscation rule for non-linear models, which is shown as follows:

Theorem 4.2. ***(Coupled Weights Obfuscation)**: We consider a general non-linear $ReLU_\beta$ layer (*e.g.*, `relu6` operator):*

$$ReLU_\beta(x) = \begin{cases} \beta, & if \quad x \geq \beta; \\ x, & else\ if \quad 0 < x < \beta; \\ 0, & otherwise. \end{cases}$$

If W_i and b_i in the sub-network f are scaled to aW_i and ab_i for $i \in [1, n]$ with $0 < a < 1$, respectively, to get the transformed sub-network f^s. Then, $ReLU_\beta(\frac{1}{a}I_{n+1} \times ReLU_\beta(f^s(X_0))) = I_{n+1} \times ReLU_\beta(f(X_0))$. ***The proof can be found in our code repository.***

Explanation of Theorem 4.2: When we choose the first `conv2d` operator with a green box (see Figure 3), the transformation propagation can be extended to the next extra obfuscating operator

Algorithm 1 : Obfuscation Coupling

Input: computational graph $\mathcal{G}$, the number of obfuscation pairs n.
Output: obfuscation-coupled graph $\mathcal{G}$
1 : Initialize a coupled operators set **S**, and a transformation parameter set **A**
2 : **For** m in range$(1, n)$ **do** :
3 : random choose an eligible selected operator O in $\mathcal{G}$
4 : find the eligible coupled operator set $\overline{\mathbf{O}}$
5 : **if** $\overline{\mathbf{O}}$ is not empty:
6 : random choose the coupled operator $\overline{O}$ from $\overline{\mathbf{O}}$
7 : **S.append**$(O, \overline{O})$
8 : $a =$ **random**$(0, 1)$
9 : **A.append**(a)
10 : **For** $(O, \overline{O})$ in **S** :
11 : perform transformation for the propagation path$[O, \overline{O}]$
12 : **Return** $\mathcal{G}$

connected to the first `relu` in the transformation path, *i.e.*, `gdsogu` operator that performs $\text{ReLU}_\beta(\frac{1}{a}I_{n+1} \times \text{ReLU}_\beta(f^s(X_0)))$, where the I_{n+1} is the weights of the operator and is an identity matrix. Specifically, *DynaMO* will perform a linear transformation to the weights (aW_0, ab_0) as it is the selected operator. Then, except for choosing {`mjzdmh`, `fsebwa`} or `conv2d` as the coupled operator (as shown in the explanation of Lemma 4.1), we can choose the `gdsogu` operator as the coupled operator by transforming the weights (*i.e.*, $\frac{1}{a}W_4$ and $\frac{1}{a}b_4$, where W_4 is an identity matrix and b_4 is a zero vector). Note that here we need to add a fused `relu` computation at the end of the inference process of `gdsogu` (*i.e.*, the outer `ReLU` function of $\text{ReLU}_\beta(\frac{1}{a}I_{n+1} \times \text{ReLU}_\beta(f^s(X_0)))$). By adding many extra obfuscating operators to the computational graph, *DynaMO* can find more eligible transformation pairs for CNNs, thus improving the obfuscation performance.

Algorithm. Our *DynaMO* will first use existing obfuscation methods [37] to obfuscate the model representation. Then, it will perform the Obfuscation Coupling algorithm, which is shown in Algorithm 1. Note that our dynamic obfuscation strategy uses linear transformation, it supports obfuscating the same operators several times. So, *DynaMO* will randomly sample n obfuscation pairs (we set the n to the total number of operators) from $\mathcal{G}$ based on the Lemma 4.1 and Theorem 4.2. Note that the selected operator and the coupled operator may include multiple operators. For example, as shown in Figure 3, if the green `conv2d` is chosen as the selected operator, the coupled operator can be `mjzdmh` and `fsebwa` as they are both the next operator of the `conv2d`. Finally, *DynaMO* will follow the rule defined in Lemma 4.1 and Theorem 4.2 to obfuscate the weights through a linear transformation. Such transformation is hard to detect unless attackers know the original weights.

4.5 Compilation

After obtaining the obfuscated model and the modified source code of the API library using an existing DL obfuscation tool [37], *DynaMO* then assembles the new obfuscated model generated by its dynamic obfuscation strategy in Python. Then, it recompiles the modified TFLite library to support the newly generated obfuscated model. The newly generated obfuscated model and library can be packaged into mobile apps or embedded device software to replace the original unobfuscated model or obfuscated model generated by existing obfuscation strategies.

4.6 *DynaMO* Effectiveness

Note that in our evaluations, we set the number of obfuscation pairs n in Algorithm 1 to the total number of operators. The settings are the same as the evaluation in section 3. **Note that we set the number of extra obfuscating operators to 30 in evaluating the effectiveness of *DynaMO*.**

4.6.1 ***Output Difference.*** We need to first evaluate the performance loss of our proposed obfuscation strategy. Table 2 summarises the evaluation of our methods on performance loss compared with existing model obfuscation [37]. The maximal element-wise error can be formulated as:

$$\theta = (\max_{i=1}^{N} |C(x_i) - \mathcal{G}(x_i)|) \div \max_{i=1}^{N} |\mathcal{G}(x_i)| \tag{3}$$

where $\mathcal{G}$ is the model results before obfuscation. C refers to the output of obfuscated models. N denote the total number of output elements (the output is usually a matrix or vector). We use 100 inputs to compute the maximal element-wise error and divide the error by the maximal value of the original output to get the scaled maximal element-wise error. Our *DynaMO* method only has negligible errors, because *DynaMO* theoretically uses the same computing process as the original model inference. But the weights transformation process in our *DynaMO* will have an inevitable computing error that is very small.

4.6.2 ***Resilience to Attack.*** The effectiveness evaluation of our *DynaMO* is shown in Table 1. Compared with the existing model obfuscation method (see Table 1), the *DLModelExplorer* significantly reduce the performance of attacks based on dynamic instrumentation. Note that we only introduce a maximum of 30 obfuscating operators to the obfuscated model. Because *DLModelExplorer* will classify all operators (including valid operators and obfuscating operators) to valid operators, *DLModelExplorer* achieves 60.04% of Operator Classification Accuracy (OCA). It only obtains 0.5% of true negative rate that shows the performance of correctly identifying the obfuscating operator. Our method can obfuscate the intermediate results, the *DLModelExplorer* only achieves 57.60% of the operator's name identification rate (NIR). Our *DynaMO* also achieves high performance on weights obfuscation, attackers can only correctly extract 52.52% of model weights. In global metrics (*i.e.*, WEE and SS), the extracted weights have high errors (*i.e.*, 0.78) compared with the results of existing obfuscation methods (*i.e.*, 2.1×10^{-5}). Our method also can significantly decrease the structure similarity (SS) from 0.98 to 0.58. The results show the dynamic instrumentation analysis method cannot effectively deobfuscate the models generated by our proposed dynamic obfuscation strategy. Note that The model representations are randomly obfuscated by our method, it is hard for attackers to identify which operators have been obfuscated. Therefore, we do not need to apply our *DynaMO* to obfuscate all model weights, operators, and intermediate results to reduce the performance loss.

Table 2: The scaled maximal element-wise error of our proposed *DynaMO* compared with the existing model obfuscation method ModelObfuscator [37].

Model name	Fruit	Skin cancer	MobileNet	MNASNet	SqueezeNet	EfficientNet	MiDaS	Lenet	PoseNet	SSD	Average
ModelObfuscator	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	**0.0**
DynaMO	1.4×10^{-7}	2.0×10^{-7}	4.9×10^{-9}	5.9×10^{-8}	3.8×10^{-9}	1.2×10^{-8}	3.8×10^{-7}	4.8×10^{-7}	4.3×10^{-8}	6.5×10^{-8}	$\mathbf{1.4\times10^{-7}}$

Table 3: Performance of *DynaMO* in defending against the instrumentation attack *DLModelExplorer*. 'TN': True negative (*i.e.*, correct identification for obfuscating operators) rate of operator classification. 'Difference': the average value difference between the attacking performance based on *DynaMO* (this table) and existing obfuscation methods (Table 1). 'WER, NIR, OCA, SS': the lower is better. 'WEE': the higher is better.

Metric	Fruit	Skin	MobileNet	MNASNet	SqueezeNet	EfficientNet	MiDaS	LeNet	PoseNet	SSD	**Average value**	**Difference**
WER	55.17%	51.72%	35.71%	50.00%	62.96%	68.63%	56.94%	25.00%	59.38%	59.72%	**52.52%**	**46.24** ↓
NIR	52.54%	51.67%	49.18%	70.45%	74.70%	73.81%	71.26%	33.33%	26.19%	72.82%	**57.60%**	**40.80%** ↓
OCA	52.54%	51.67%	50.82%	70.79%	73.26%	73.81%	79.31%	29.63%	57.89%	80.65%	**60.04%**	**37.85%** ↓
TN (OCA)	0%	0%	0%	0%	0%	0%	0%	5.00%	0%	0%	**0.5%**	N/A
WEE	0.61	1.46	2.70	1.25	0.32	0.33	0.14	0.04	0.75	0.24	**0.78**	**0.78** ↑
SS	0.53	0.52	0.49	0.70	0.75	0.74	0.71	0.33	0.26	0.73	**0.58**	**0.40** ↓

Table 4: The ability to resist the reverse engineering for on-device models. '√': this model parsing method cannot extract information for all models. 'Model conversion': TF-ONNX [1], TFLite2ONNX [29], and TFLite2TF [14]. 'Parsing in buffer': [18]. 'Feature analyzing': [13].

	Model conversion	**Parsing in buffer**	**Feature analyzing**
ModelObfuscator	√	√	√
DynaMO	√	√	√

In addition, we follow the setting in [37] to check whether our proposed obfuscation strategy can still be robust for the reverse engineering of an on-device model or not. The results are shown in 4. It shows our proposed strategy will not degrade the model resistance against normal reverse engineering methods.

> RQ2: Our *DynaMO* is effective in defending against existing model parsing methods and dynamic instrumentation attack method *DLModelExplorer*, with only negligible performance loss.

5 RQ3: EFFICIENCY OF *DYNAMO*

As *DynaMO* introduces additional computations via its extra obfuscating operators, it is important to evaluate the influence of such additional computation on inference efficiency. In order to evaluate the efficiency impact of our obfuscation strategies on ML models, we conducted experiments to measure new obfuscated ML model runtime overhead.

Experimental Environment: The efficiency of *DynaMO* is evaluated on a workstation with Intel(R) Xeon(R) W-2175 2.50GHz CPU, 32GB RAM, with Ubuntu 20.04.1 operating system and a Xiaomi 11 Pro smartphone with Android 13 OS.

Time Overhead: We measured the time overhead of *DynaMO* and existing DL model obfuscation strategies [37], based on 5,000 randomly generated instances. The results of these experiments are presented in Tables 5, which report the time overhead of the obfuscated models. As shown in Table 5, even though additional computations are introduced into the extra obfuscating operators, ***DynaMO* obfuscated models incur a negligible time overhead compared with existing model obfuscation strategy**. This is because the obfuscating operators just perform simple linear transformations, which have low complexity compared with the whole model inference process.

RAM Overhead: We measured the RAM overhead of both obfuscation strategies. The memory overhead for *DynaMO* obfuscated models is shown in Table 6. To eliminate the impact of different memory optimization methods, like the study [37], we use peak RAM usage where the model preserves all intermediate tensors. **Our method only introduces negligible overheads compared with the existing model obfuscation strategies**. Because our obfuscation strategy only transforms the value of the weights.

> RQ3: The time and RAM overhead of *DynaMO* are both negligible, while RQ2 shows that our proposed *DynaMO* strategy significantly improves the security level of the ob-device models.

6 LIMITATIONS

Our proposed *DLModelExplorer* and *DynaMO* are designed for on-device TFLite models. Although our strategy can be adapted to other DL platforms, we do not know if there are unsupported DL model constructs our approach may not be able to support.

We have not evaluated our *DynaMO* with real-world mobile ML developers. We have not evaluated our *DynaMO* with side-channel information (*e.g.*, RAM, CPU usage) to reconstruct the model [11, 19,

Table 5: Time overhead (seconds per 1000 inputs) of *DynaMO* compared with existing obfuscation method [37] on `x86-64` and `ARM64` platforms. We set the number of extra layers to 20 for both obfuscation strategies.

	Fruit	Skin	MobileNet	MNASNet	SqueezeNet	EffcientNet	MiDaS	LeNet	PoseNet	SSD	**Average value**
`x86-64`											
original	28.6	86.7	53.3	69.4	37.5	99.7	309.1	2.1	114.3	208.6	**100.9**
ModelObfuscator	28.4	87.4	55.6	71.2	39.7	101.1	321.7	2.9	117.8	211.1	**103.7**
DynaMO	29.1	87.7	54.9	71.5	40.3	102.4	329.2	2.9	117.9	212.6	**104.9**
`ARM64`											
	Fruit	Skin	MobileNet	MNASNet	SqueezeNet	EffcientNet	MiDaS	LeNet	PoseNet	SSD	**Average value**
original	14.2	45.5	28.1	36.0	53.2	41.1	380.9	4.6	43.8	92.3	**73.9**
ModelObfuscator	14.3	45.9	28.6	36.2	53.7	41.1	381.4	4.9	44.1	92.4	**74.3**
DynaMO	14.2	46.2	28.8	36.5	53.8	41.4	385.3	4.9	44.7	92.6	**74.8**

Table 6: Overhead of *DynaMO* on random access memory (RAM) cost (Mb per model) compared with existing obfuscation method [37]. To eliminate the influence of other processes on the test machine, we show the increment of RAM usage.

	Fruit	Skin	MobileNet	MNASNet	SqueezeNet	EffcientNet	MiDaS	LeNet	PoseNet	SSD	**Average value**
Original	13.5	34.6	23.9	37.7	34.4	43.4	253.8	6.2	38.8	97.8	**58.4**
ModelObfuscator	22.2	54.9	38.0	56.6	47.9	59.6	278.9	6.8	55.4	107.6	**72.8**
DynaMO	26.9	58.5	39.2	56.3	48.6	61.1	281.4	9.3	58.5	108.3	**74.8**

30]. Our proposed *DynaMO* may not be effective for them because the obfuscated model has a similar resource consumption pattern to the original models. However, our method is designed for and very effective for resisting the attacks based-on program analysis.

Our dynamic obfuscation method *DynaMO* will cause a very slight performance loss for on-device models. In addition, our proposed *DynaMO* will introduce very small overheads to the model inference similar to the existing model obfuscation strategies.

Threats to Validity. For internal threats to validity, the efficiency evaluation results (RQ3) may be affected by other services running in the experimental environments (i.e., Ubuntu server, Xiaomi 11 Pro). For external threats to validity, as the DL techniques and AI compilers may change a lot, our tool needs continuous updates in the future.

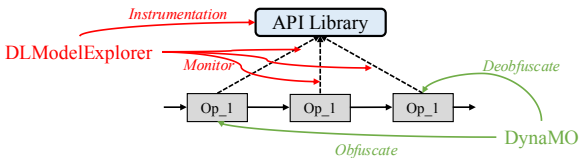

Figure 4: Meta-model our method.

Meta-Model. Although we focus on the TFLite because it's the most commonly used DL platform on mobile devices, especially for Android, our methods are also general for other DL compilers. We propose two methods: *DLModelExplorer* and *DynaMO*. *DLModelExplorer* will monitor the APIs (DL compilers are usually open-sourced) that allocate the model weights to the memory and obtain the intermediate results by instrumentation to predict the operator's type. *DynaMO* only uses linear weight transformations to obfuscate and restore the model's outputs. The meta-model of our study is shown in Figure 4.

7 CONCLUSION

We analyzed the risk of deep learning models deployed on mobile devices and the vulnerabilities of the existing static and half-dynamic DL model obfuscation strategies. We showed that attackers can still extract information from these obfuscated models using dynamic instrumentation-based techniques. To address this vulnerability, we propose a novel dynamic model obfuscation strategy and tool, *DynaMO* to better secure mobile device deployed DL models. *DynaMO* can couple the obfuscated DL model operators to increase the randomness of the obfuscation process without significant overhead. In addition, we provide a theoretical guarantee to achieve such dynamic obfuscation without model performance loss. We developed a prototype tool *DynaMO* to automatically obfuscate TFLite DL models using our proposed obfuscation strategy. Experiments show that our method is effective in resisting the model parsing tools and the proposed dynamic instrumentation attack without performance sacrifice. In the future, we will optimize the obfuscation process and our prototype tool to further increase the efficiency of our method and tool.

ACKNOWLEDGEMENTS

This work is partially supported by the Open Foundation of Yunnan Key Laboratory of Software Engineering under Grant No.2023SE102. Zhou is supported by a Faculty of IT PhD scholarship. Grundy is supported by ARC Laureate Fellowship FL190100035.

REFERENCES

[1] 2022. *tf2onnx - Convert TensorFlow, Keras, Tensorflow.js and Tflite models to ONN.* https://github.com/onnx/tensorflow-onnx

[2] 2024. *Apktool: A tool for reverse engineering Android apk files.* https://ibotpeaches.github.io/Apktool/

[3] 2024. *Mindspore.* https://www.mindspore.cn/lite/docs/en/r1.7/use/obfuscator_tool.html

[4] 2024. *Pin: A Dynamic Binary Instrumentation Tool.* https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html

[5] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[6] Simin Chen, Hamed Khanpour, Cong Liu, and Wei Yang. 2022. Learning to reverse dnns from ai programs automatically. In *AAAI Conference on Artificial Intelligence*.

[7] François Chollet et al. 2018. Keras: The python deep learning library. *Astrophysics source code library* (2018), ascl–1806.

[8] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. A taxonomy of obfuscating transformations.

[9] Christian Collberg, Clark Thomborson, and Douglas Low. 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 184–196. https://doi.org/10.1145/268946.268962

[10] Malinda Dilhara, Ameya Ketkar, and Danny Dig. 2021. Understanding Software-2.0: A Study of Machine Learning library usage and evolution. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 4 (2021), 1–42. https://doi.org/10.1145/3453478

[11] Vasisht Duddu, Debasis Samanta, D Vijay Rao, and Valentina E Balas. 2018. Stealing neural networks via timing side channels. *arXiv preprint arXiv:1812.11720* (2018).

[12] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[13] Yujin Huang and Chunyang Chen. 2022. Smart App Attack: Hacking Deep Learning Models in Android Apps. *IEEE Transactions on Information Forensics and Security* 17 (2022), 1827–1840.

[14] Katsuya Hyodo. 2022. *tflite2tensorflow.* https://github.com/PINTO0309/tflite2tensorflow

[15] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[16] Alex Kendall, Matthew Grimes, and Roberto Cipolla. 2015. Posenet: A convolutional network for real-time 6-dof camera relocalization. In *Proceedings of the IEEE international conference on computer vision*. 2938–2946. https://doi.org/10.1109/iccv.2015.336

[17] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. https://doi.org/10.1109/5.726791

[18] Yuanchun Li, Jiayi Hua, Haoyu Wang, Chunyang Chen, and Yunxin Liu. 2021. Deeppayload: Black-box backdoor attack on deep learning models through neural payload injection. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 263–274. https://doi.org/10.1109/icse43902.2021.00035

[19] Sihang Liu, Yizhou Wei, Jianfeng Chi, Faysal Hossain Shezan, and Yuan Tian. 2019. Side channel attacks in computation offloading systems with gpu virtualization. In *2019 IEEE Security and Privacy Workshops (SPW)*. IEEE, 156–161.

[20] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. 2016. Ssd: Single shot multibox detector. In *European conference on computer vision*. Springer, 21–37.

[21] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. 2017. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*. 506–519.

[22] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[23] René Ranftl, Katrin Lasinger, David Hafner, Konrad Schindler, and Vladlen Koltun. 2020. Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer. *IEEE transactions on pattern analysis and machine intelligence* 44, 3 (2020), 1623–1637. https://doi.org/10.1109/tpami.2020.3019967

[24] Pengcheng Ren, Chaoshun Zuo, Xiaofeng Liu, Wenrui Diao, Qingchuan Zhao, and Shanqing Guo. 2024. 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*. 1–13.

[25] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2016. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Computing Surveys (CSUR)* 49, 1 (2016), 1–37. https://doi.org/10.1145/2886012

[26] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2820–2828. https://doi.org/10.1109/cvpr.2019.00293

[27] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*. PMLR, 6105–6114.

[28] Chenxi Wang. 2001. *A security architecture for survivability mechanisms.* University of Virginia.

[29] Zhenhua Wang. 2021. *tflite2onnx - Convert TensorFlow Lite models to ONNX.* https://github.com/jackwish/tflite2onnx

[30] Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. 2020. Leaky dnn: Stealing deep-learning model secret with gpu context-switching side-channel. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 125–137. https://doi.org/10.1109/dsn48063.2020.00031

[31] Gregory Wroblewski. 2002. General method of program code obfuscation. (2002).

[32] Jing Wu, Munawar Hayat, Mingyi Zhou, and Mehrtash Harandi. 2024. Concealing Sensitive Samples against Gradient Leakage in Federated Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 21717–21725.

[33] Jing Wu, Mingyi Zhou, Shuaicheng Liu, Yipeng Liu, and Ce Zhu. 2020. Decision-based universal adversarial attack. *arXiv preprint arXiv:2009.07024* (2020).

[34] Chaoning Zhang, Philipp Benz, Adil Karjauv, Jae Won Cho, Kang Zhang, and In So Kweon. 2022. Investigating Top-k White-Box and Transferable Black-box Attack. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 15085–15094.

[35] Mingyi Zhou, Xiang Gao, Xiao Chen, Chunyang Chen, John Grundy, and Li Li. 2024. *DynaMO: Protecting Mobile DL Models through Coupling Obfuscated DL Operators (0.1).* https://doi.org/10.5281/zenodo.13762398

[36] Mingyi Zhou, Xiang Gao, Pei Liu, John Grundy, Chunyang Chen, Xiao Chen, and Li Li. 2024. Model-less Is the Best Model: Generating Pure Code Implementations to Replace On-Device DL Models. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 174–185. https://doi.org/10.1145/3650212.3652119

[37] Mingyi Zhou, Xiang Gao, Jing Wu, John 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* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1005–1017. https://doi.org/10.1145/3597926.3598113

[38] Mingyi Zhou, Xiang Gao, Jing Wu, Kui Liu, Hailong Sun, and Li Li. 2024. Investigating White-Box Attacks for On-Device Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[39] Mingyi Zhou, Jing Wu, Yipeng Liu, Shuaicheng Liu, and Ce Zhu. 2020. Dast: Data-free substitute training for adversarial attacks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 234–243.