ModelObfuscator: Obfuscating Model Information to Protect Deployed ML-Based Systems

Mingyi Zhou
mingyi.zhou@monash.edu
Monash University
Melbourne, VIC, Australia

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

Jing Wu
jing.wu1@monash.edu
Monash University
Melbourne, VIC, Australia

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

Xiao Chen*
xiao.chen@monash.edu
Monash University
Melbourne, VIC, Australia

Chunyang Chen
chunyang.chen@monash.edu
Monash University
Melbourne, VIC, Australia

Li Li*
lilicoding@ieee.org
Beihang University
Beijing, China

ABSTRACT

More and more edge devices and mobile apps are leveraging deep learning (DL) capabilities. Deploying such models on devices – referred to as on-device models – rather than as remote cloud-hosted services, has gained popularity because it avoids transmitting user's data off of the device and achieves high response time. However, on-device models can be easily attacked, as they can be accessed by unpacking corresponding apps and the model is fully exposed to attackers. Recent studies show that attackers can easily generate white-box-like attacks for an on-device model or even inverse its training data. To protect on-device models from white-box attacks, we propose a novel technique called *model obfuscation*. Specifically, model obfuscation hides and obfuscates the key information – structure, parameters and attributes – of models by *renaming*, *parameter encapsulation*, *neural structure obfuscation obfuscation*, *shortcut injection*, and *extra layer injection*. We have developed a prototype tool *ModelObfuscator* to automatically obfuscate on-device TFLite models. Our experiments show that this proposed approach can dramatically improve model security by significantly increasing the difficulty of parsing models' inner information, without increasing the latency of DL models. Our proposed on-device model obfuscation has the potential to be a fundamental technique for on-device model deployment. Our prototype tool is publicly available at https://github.com/zhoumingyi/ModelObfuscator.

*Dr. Li Li was a senior lecturer at Monash. He supervised this project for the whole period. Corresponding authors: Xiao Chen and Li Li.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598113

CCS CONCEPTS

• **Software and its engineering** → **Software safety**; **Software reliability**.

KEYWORDS

SE for AI, AI safety, model obfuscation, model deployment

ACM Reference Format:
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 (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598113

1 INTRODUCTION

This study focuses on how to defend against software analysis and reverse engineering that collects the inner information of on-device models. These kinds of attacks are far more effective than model-stealing techniques such as substitute model training or side-channel attacks, since on-device models are directly hosted on mobile devices.

Numerous edge and mobile devices are leveraging deep learning (DL) capabilities. Though DL models can be deployed on a cloud platform, data transmission between mobile devices and the cloud may compromise user privacy and suffer from severe latency and throughput issues. To achieve high-level security, users' personal data should not be sent outside the device. To achieve high throughput and short response time, especially for a large number of devices, on-device DL models are needed. The capabilities of newer mobile devices and edge devices keep increasing, with more powerful systems on a chip (SoCs) and a large amount of memory, making them suitable for running on-device models. Indeed, many intelligent applications have already been deployed on devices [52] and benefited millions of users.

Unfortunately, since on-device models are directly hosted on mobile devices, adversaries can easily unpack the mobile apps to locate

the models for security exploitation. Such on-device models are thus facing more serious security threats. To protect the on-device model, on-device DL frameworks like TFLite are released to users as black-box ones, *i.e.*,, the released TFLite models do not support the gradient computing. Since gradient information is considered crucial to implement effective white-box attacks, preventing attackers from obtaining gradient information from on-device models (referred to as non-debuggable models) could enhance model's security. However, it does not fulfill such a purpose in practice, which is shown in Figure 1. Attackers can parse the information of models (*e.g.*, model structures and weights), generate efficient attacks based on the parsed information, and apply the attacks to the target on-device model. For example, Huang *et al.* [17] parse features of the on-device model to find a surrogate model from the web, which can then be used to launch transferable adversarial attacks on mobile models. Li *et al.* [26] developed a method to perform backdoor attacks on mobile models. This method uses reverse engineering to parse the model file and inject a triggering model into the model file. As the attackers can easily obtain the explicit information of on-device models through public APIs or reverse engineering, on-device models are at serious risk that attackers can also perform many other attacks for on-device models, such as model inversion attacks, membership inference attacks, *etc.* [3, 12, 37, 40]. However, to the best of our knowledge, no effective defense method has been proposed to resist reverse engineering for on-device DL models.

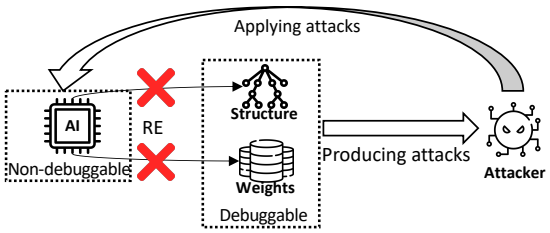


Figure 1: The typical scenarios of attacking on-device models. Our approach can disable the reverse engineering (*i.e.*, RE).

Borrowing the idea of code obfuscation, which is a well-developed approach for hiding key information in software, we propose to obfuscate the information of on-device ML models. As the on-device model interacts with the DL libraries like TFLite at runtime, we also need to obfuscate the information of model files (*e.g.*,.tflite files) and modify the DL libraries to execute the correct forward computation of original models. In this paper, **we propose a novel on-device ML model protection approach based on model obfuscation, which focuses on improving AI safety for disabling the model parsing using software analysis or reverse engineering**. Given a trained model and its underlying DL library (*e.g.*, TFLite), an end2end prototype tool, *ModelObfuscator*, is developed to generate obfuscated on-device model and corresponding DL library. *ModelObfuscator* first extracts the information of the target model and locates the source code in the library used by its layers. It then obfuscates the information of the models and builds a customized DL library that is compatible with the obfuscated model. To achieve this, we design five obfuscation methods, including: (1) *renaming*, (2) *parameter encapsulation*, (3) *neural structure obfuscation*, (4) *random shortcut injection*, and (5) *random extra layer injection*. These obfuscation methods significantly increase the difficulty of parsing the information of the model. Our on-device ML model obfuscation can prevent attackers from reconstructing the model. It is also hard for attackers to transfer the trained weights and structure of models to steal intellectual property using model conversion, because the connection between the obfuscated information and the original one is randomly generated. Experiments on 10 different models show that *ModelObfuscator* can against state-of-the-art model parsing and attack tools with a negligible time overhead and 20% storage overhead. The codes of *ModelObfuscator* are shared at https://github.com/zhoumingyi/ModelObfuscator.

The key contributions in this work include:

- We propose a novel model obfuscation framework to hide the key information of deployed DL models to resist model parsing. It can prevent attackers from generating efficient attacks and stealing the knowledge of on-device models by significantly increasing the cost of attacks.
- We design five obfuscation strategies for protecting on-device models and implement an end2end prototype tool, *ModelObfuscator*. This tool automatically obfuscates the model and builds a compatible DL software library. The tool is publicly available.
- We provide a taxonomy and comparison of different obfuscation methods in terms of effectiveness and overhead, to guide model owners in choosing appropriate defence strategies.

2 BACKGROUND AND RELATED WORK

2.1 On-Device DL Models

DL frameworks. The open-source community has developed many well-known open-source frameworks for DL tasks such as TensorFlow [1], Theano [2], Caffe [20], Keras [6], and PyTorch [32]. These frameworks dominate the development of DL models and set standards for them [11]. PyTorch is one of the latest DL frameworks, and is gaining popularity for its ease of use and its capability to construct the dynamic computational graph, which is now widely used by the academic community. In contrast, TensorFlow is widely used by companies, startups, and business firms to automate things and develop new systems. It has distributed training support, scalable production options, and support for mobile devices. Currently, companies and research teams have made huge efforts to develop open-source on-device frameworks like TensorFlow Lite (TFLite), Caffe2, Caffe, NCNN, and ONNX. As an on-device DL platform, TensorFlow Lite (TFLite) is the most popular framework for DL models on smartphones, as it has GPU support and is optimized for mobile devices [17, 52].

Deployed DL models. As the training of DL models is intensive in both data and computing, mobile developers often collect the data and train their models on the computing server prior to app deployment. Developers also need to specifically compile the trained models to be compatible with devices (*i.e.*, on-device models) so as to speed up the model inference on mobile CPU/GPUs [4, 5, 30]. At app installation time, the trained models are deployed, along with the app code itself, in the installation package of an app. At runtime, apps perform the inference of DL models by invoking

application programming interfaces (APIs) of DL frameworks, and subsequently achieve AI capabilities.

2.2 TFLite Models

TFLite models have powerful features for running models on edge devices but it does not provide APIs to access the gradient or intermediate outputs like other TensorFlow or PyTorch models. TensorFlow provides a *TensorFlow Lite Converter*[1] to convert a TensorFlow model into a TensorFlow Lite model. TFLite models run on the FlatBuffers Platform[2]. FlatBuffers is an efficient cross-platform serialization library for multiple programming languages. It has several advantages for employing the model on mobile devices that it can access serialized data without parsing/unpacking and only needs small computational resources. TFLite uses a schema file to define the data structures. For parsing the model structure and weights from the `.tflite` file, we can use the schema file [3] of TFLite to parse FlatBuffers and get the JSON file that contains detailed information of the `.tflite` file.

2.3 Existing Model Parsing and Defenses

The key information in a deployed DL model can be parsed by three kinds of approaches: (1) Parsing the model's weights through queries [45, 54]. Attackers use samples to query the outputs of the target model. Then, they can train a similar model with target model through the samples and outputs. (2) Parsing the model's architecture by side-channel attacks[15, 25, 49]. Attackers can estimate the architecture of the neural networks by the side channel information (*e.g.*, latency and DRAM accesses). (3) Parsing the entire model from devices using software analysis or reverse engineering [26, 46]. Although many attacks have been proposed to extract DL models, it is hard for adversaries to precisely reconstruct DL models that are identical to the original ones using queries or side-channel information. These attacks cannot access the inner information of the model, meaning that they are black-box attacks. Among these three approaches, reconstructing DL models using queries or side-channel information is hard to obtain identical models to the original ones. Since, the first two approaches cannot access the inner information of models, they only enable black-box attacks. In contrast, since on-device models are delivered in mobile apps and hosted on mobile devices, adversaries can easily unpack the mobile apps to parse the original models for security exploitation, which enables white-box attacks. As shown in previous work, white-box attacks can achieve a much higher success rate than black-box attacks [53]. Apart from being attacked, deploying DL models on users' devices may cause intellectual property leakage [39].

Existing model defense methods can be categorized into two different groups: (1) defend against query-based attacks, and (2) defend against side-channel attacks. For defending against query-based attacks, some studies [21, 29, 31, 41, 42, 51] propose different strategies to degenerate the effectiveness of query-based model extraction. While other studies [41, 42, 51] propose methods to train a simulating model, which has similar performance to the original model, but is more resilient to query-based attacks. For securing the AI model at the side-channel level, a recent work modifies the CPU and Memory costs to resist the model extraction attacks [25]. However, existing defenses have not yet been aware of model parsing using software analysis or reverse engineering, which is more harmful than other kinds of model parsing for deployed DL models. Our method is proposed to defend against the parsing using software analysis or reverse engineering.

2.4 Reverse Engineering Tools for DL Models

We want an approach that is highly resistant to model parsing that uses software analysis or reverse engineering tools. Existing model parsing methods using reversing engineering fall into three categories: (1) model format conversion – existing model conversion tools access the structure and weights of non-debuggable on-device models through public APIs, and then assemble them into a new model with a different format. (2) model parsing in the buffer – the on-device model format TFLite loads the data on FlatBuffer [4]. Attackers can then extract the model structure and weights in FlatBuffer [26]. (3) finding a similar differentiable model from the Internet by comparing the features among models – the current mainstream on-device model format TFLite does not support some advanced functions like auto-differentiation to protect the deployed model. But attackers like the App Attack can find a differentiable model with similar structure and weights from the Internet [16].

2.5 Code Obfuscation

Code obfuscation methods are initially developed for hiding the functionality of the malware. Then, the software industry also uses it against reverse engineering [36]. They provide complex obfuscating algorithms for programs like JAVA code [7, 8], including robust methods for high-level languages [47] and machine code level [50] obfuscation. Code obfuscation is a well-developed technique to secure the source code. However, traditional code obfuscation approaches are hard to protect on-device models, especially for protecting the structure of the models and their parameters. In this work, inspired by traditional code obfuscation, we propose a novel model obfuscation approach to obfuscate the model files and then produce a corresponding DL library for them.

3 THE *MODELOBFUSCATOR* SOLUTION

In this section, we will introduce the threat model in this study and show the detail of the proposed *ModelObfuscator*.

3.1 ML Platform and Threat Model

On-device ML model platform. We chose the TensorFlow Lite (TFLite) platform to demonstrate our model obfuscation approach. TFLite is currently the most commonly used on-device DL platform. The steps to produce TFLite models are shown in the top half of Figure 2. Usually, DL developers use TensorFlow application programming interfaces (APIs) to define and train TensorFlow (TF) models. The trained TF models are then compiled into a TFLite models. Note that TFLite has different implementations with TF, and their operators (*i.e.*, layers) may not be compatible. Therefore, when compiling TFLite models, the TFLite library will check their

[1] https://www.tensorflow.org/lite/convert/index
[2] https://google.github.io/flatbuffers/
[3] schema file (The link is too long to display)
[4] https://google.github.io/flatbuffers/

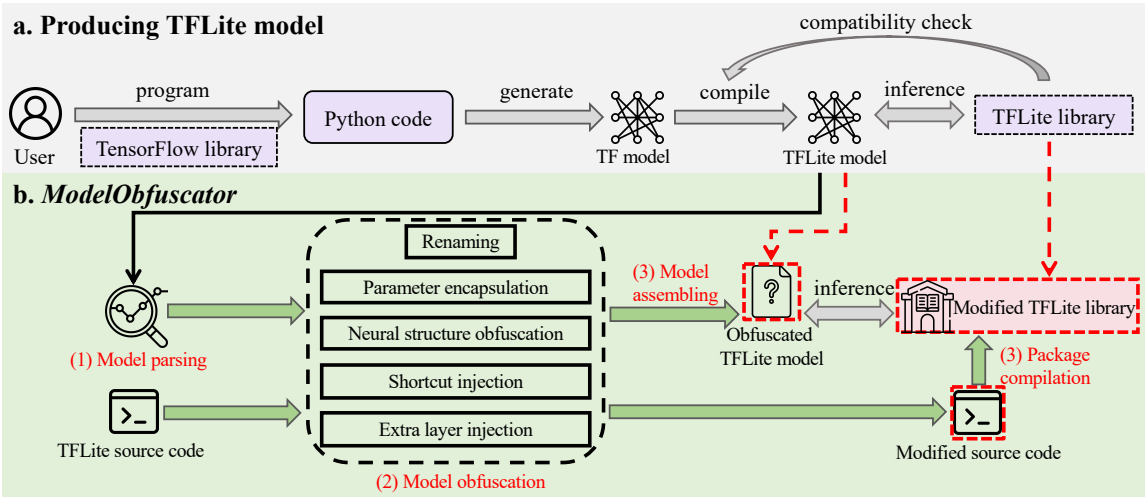


Figure 2: The working process of our *ModelObfuscator* on-device DL model obfuscation tool. 'a': the process of producing TFLite model. 'b': the framework of the proposed *ModelObfuscator*.

compatibility. Once the compatibility check passes, the compiled TFLite model can run on devices using the TFLite library.

Threat model. The on-device model is usually saved as a separate file (e.g., .tflite file) and packed into the app package. Attackers can either download the target app from the app markets (e.g., Google Play and iOS App store) or extract the app package file (e.g., APK file for Android, and IPA file for iOS) from the hosting devices. These app package files can then be decompiled by off-the-shelf reverse-engineering tools (e.g., Apktool [5] and IDA Pro [6]) to get the original DL model file. Although many on-device DL platforms do not support some advanced functions like backpropagation, attackers can assemble the model architecture and weights into a differentiable model format [17], or they can use software analysis methods to generate attacks for the target model [16, 26]. In this paper, we will obfuscate the information of the model and underlying DL library to prevent the software analysis and model conversion tools from getting model's key information.

We analyzed the current mainstream DL platforms (*i.e.*, TensorFlow and PyTorch) and identified the following two main observations. First, these platforms are open-source and provide a set of tools to build the library (*e.g.*, TFLite library) from the source code. Second, they officially support customized operators (*e.g.*, neural layers). Specifically, on top of these DL platforms, users could implement customized layers in C/C++ and compile customized layers to executable files (`.so` file in TensorFlow). Then, users can use these customized layers via high-level Python interfaces. Those features enable us to design obfuscation techniques to obfuscate model and DL library code together. The bottom half of Figure 2 shows the overview of our model obfuscation framework. Specifically, *ModelObfuscator* has three main steps: (1) model parsing, (2) model obfuscation, and (3) model assembling & library recompilation. In the following subsections, we detail the proposed *ModelObfuscator*.

[5] https://ibotpeaches.github.io/Apktool/
[6] https://hex-rays.com/ida-pro/

3.2 Model Parsing

The first step of *ModelObfuscator* model obfuscation is to parse the deployed model to extract its key information. *ModelObfuscator* first extracts the structure information of each layer, including the name of layers (*e.g.*, `Conv2D`) and model structures (including model's input, output, layer ID and *etc.*). The extracted structure information is depicted using JSON format, which can be visualized as Figure 3 (a). It includes all neural layers and their spatial relationships. Then, *ModelObfuscator* extracts the parameter of each layer, such as the layer's configurations and weights. Moreover, *ModelObfuscator* identifies the source code used by each layer by referring to the underlying libraries. The identified source code includes relevant packages and functions of the TFLite layers.

3.3 Model Obfuscation

After extracting the on-device ML model information and its corresponding source codes, *ModelObfuscator* will obfuscate the model as well as the source codes. *ModelObfuscator* uses five obfuscation strategies: *renaming*, *parameter encapsulation*, *neural structure obfuscation*, *shortcut injection*, and *extra layer injection*. We describe the implementation details of each of these obfuscation strategies in the following subsections.

Renaming. The most straightforward obfuscation strategy is the renaming of a layer. Usually, the layer's name contains important information, which is the function of this layer. For instance, "Conv2DOptions" indicates a 2D convolution layer. Such information is useful for attackers to reconstruct the model to generate white-box attacks or to obtain a similar surrogate model to conduct effective black-box attacks. To hide such important information, we randomly modify each layer's name. The ① of Figure 3 is an example of an obfuscated `Conv2D` layer. *ModelObfuscator* automatically replaces the real name with the random meaningless string `Tozwyu`. Note that the same layers in the TFLite model will have different random names. Meanwhile, *ModelObfuscator* creates a copy

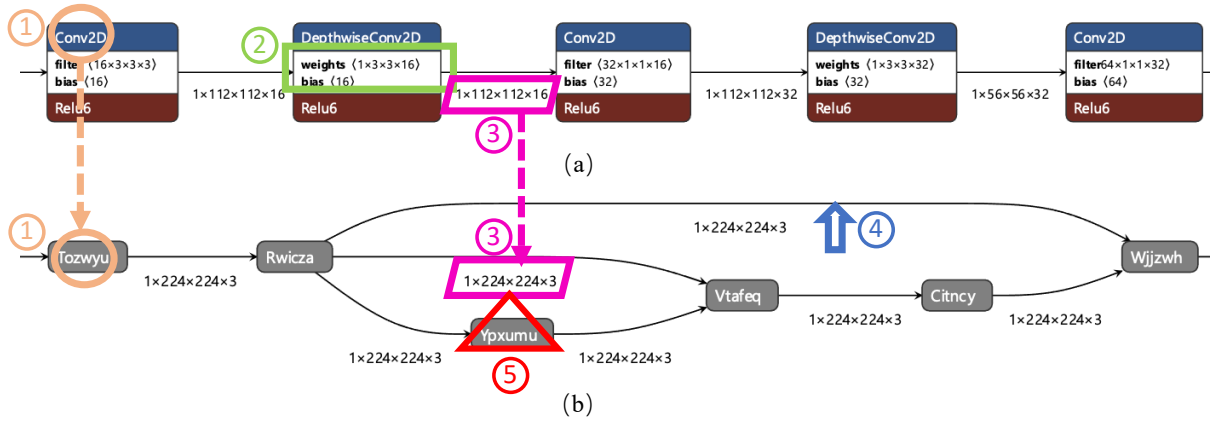


Figure 3: Example of five obfuscation strategies for the TFLite model obtained from a real-world fruit recognition app. (a) part of the original model. (b) the corresponding obfuscated model. '①': *renaming*. '②': *parameter encapsulation*. '③': *neural structure obfuscation*. '④': *shortcut injection*. '⑤': *extra layer injection*. This visualization is generated by Netron [35], which is a well-known visualization platform for neural networks.

of Conv2D's source code and replaces the layer name (*i.e.*, Conv2D) in the source code with Tozwyu. Note that we modify the duplicate of Conv2D in case the modification affects other parts of the TFLite library. After adding modified source codes to the TFLite project, the recompiled TFLite library will recognize obfuscated layers as custom layers and correctly execute them at runtime.

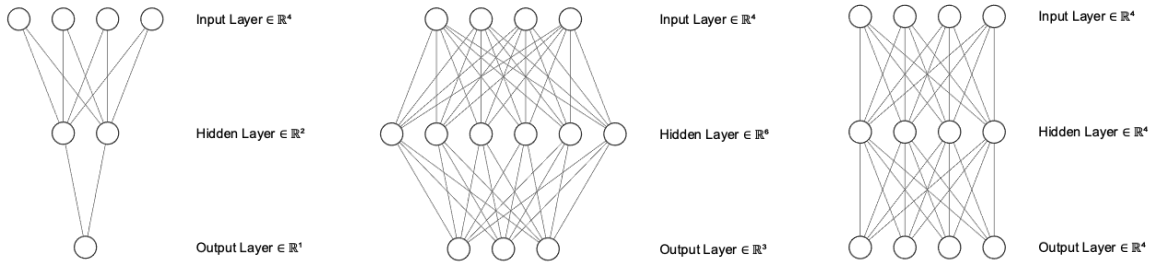
Parameter encapsulation. Existing TFLite models have two main assets: model structure and parameters. The parameter can be obtained in the training period. Given an input, a TFLite model will compute the results using the input tensor and parameters that are stored in the model file. As we discussed above, parameter exposure is very dangerous. An adversary could use the parameter information to perform many kinds of white-box attacks, *e.g.*, adversarial attacks and model inversion attacks. Besides, an adversary can guess the function of this layer according to the shape of the parameter, because different layers have different numbers of parameters (*e.g.*, two in the convolution layer) and different shapes of parameters (*e.g.*, (3, 3, 64) in the convolution layer).

To hide key model parameter information, we instead encapsulate parameters into their corresponding generated custom source codes of the obfuscated layer. For example, for a simple one-layer feed-forward neural network, the output can be computed by $\boldsymbol{Y} = \sigma(\boldsymbol{W}^T\boldsymbol{X} + \boldsymbol{b})$, where $\boldsymbol{X}$, $\boldsymbol{W}$, and $\boldsymbol{b}$ is input tensor, parameter of the layer, and bias, respectively. σ is the activation for neural nodes. For *ModelObfuscator* obfuscation, the network layer can be disguised as $\boldsymbol{Y} = g(\boldsymbol{X})$, where g is an unknown function. We then implement the correct computation (*i.e.*, g) in the generated custom TFLite source code, which we then obfuscate. At runtime, function g will be invoked to achieve the computation from $\boldsymbol{X}$ to $\boldsymbol{Y}$. Now an adversary is unable to extract the key parameter information from our obfuscated model. For example, as shown in the ② of Figure 3, the explicit parameter information has been removed from the on-device DL model file. Furthermore, the implementation of g in the compiled DL library can be obfuscated using transitional and well-proven code obfuscation strategies [9]. Thus, adversaries will be hard to identify key model parameters by reverse engineering the compiled library.

Neural structure obfuscation. Just obfuscating layer names and parameters is not enough, since an adversary may still infer the function of each layer according to the model structure. For instance, Figure 4(a) presents the structure of the neural network, where the input, hidden and output layers include four, two, and one node, respectively. Attacker could search for a surrogate model according to the neural architecture. To solve this problem, *ModelObfuscator* uses *neural structure obfuscation* to obfuscated neural architecture with the goal of confusing the adversary. We propose two strategies for network structure obfuscation: *random* and *align-to-largest*. Given a model with output shapes $vs. = (s_0, \cdots, s_n)$, where s_n refers to the number of dimensions for the n-th channel, the random strategy generates a random shape $\mathbf{r} = (\mathrm{r}_0, \cdots, \mathrm{r}_n)$ of the output for each layer. Figure 4(b) shows an obfuscated model of Figure 4(a) using random strategy. Second, the align-to-largest strategy finds the largest output shape $vs.'$ and then fills the output shapes of other layers to the size of $vs.'$. Figure 4(c) shows such an obfuscated model, where the output shapes of each layer are filled up to (4). Note that this will not affect the performance of models because the modified TFLite library will not compute the output using the provided neural structure information.

Shortcut injection & extra layer injection. Neural structure obfuscation changes the network structure by inserting new nodes, but the spatial relationships of original layers remain the same. Therefore, even with the above three obfuscation strategies, the attacker can still infer node information by analyzing the spatial relationships of runtime data (*e.g.*, actual input-output values of each node). To further obfuscate the model structure, hence, *ModelObfuscator* applies two more strategies: shortcut and extra layer injection. The injected shortcut and extra layers would destroy the original spatial relationships of TFLite models.

To automatically inject random shortcuts, *ModelObfuscator* first randomly selects a shortcut pair $(\mathrm{r}_1, \mathrm{r}_2)$. The output index of r_1-th



(a) Orginal neural nodes (b) Random neural nodes (c) Unchanged neural nodes

Figure 4: Neural structure obfuscation for a simple feed-forward neural network.

layer are then added to the input list of r_2-th layer. For example, as shown in the ④ of Figure 3 (b), we add a shortcut between the layer 'Rwicza' and 'Wjjzwh' by adding the output index of 'Rwicza' to the input list of 'Wjjzwh'. Second, to inject extra layers, just like the shortcut injection, *ModelObfuscator* randomly picks a layer pair (r'_1, r'_2). The input of the extra layer is the output node list of r'_1-th layer, and its output is added to the input list of r'_2-th layer. For instance, an extra layer Ypxumu (as shown in the ⑤ of Figure 3) is injected between the 3-rd and 5-th convolution layers.

Note that the injected shortcut and layers will not affect the prediction results of the deployed model because the modified TFLite library will ignore the obfuscation part (*e.g.*, Ypxumu layers) in model inference. Specifically, the extra layer $Y = f(X)$ just needs to create the output with a specific shape to confuse the adversary. In addition, extra layer injection will not significantly increase the latency of on-device models because the extra layer does not need many computational resources.

Combing our five obfuscation strategies. To apply the proposed five on-device ML model obfuscation strategies sequentially, we designed an automatic model obfuscation tool *ModelObfuscator*. *ModelObfuscator* can automatically obfuscate an on-device ML model and produce compatible API libraries (*e.g.*, JAVA, Python) that can use it. Figure 3 shows an example, where Figure 3(a) is the original model from a real fruit recognition app, while Figure 3(b) shows an example of the obfuscated model. We just add one shortcut and one extra layer to it. Note that developers can add more shortcuts and extra layers to achieve high obfuscation performance in practice. The structure and parameters of the obfuscated model are quite different from that of the original model, which will make the obfuscated model hard to analyze. To prevent an adversary from parsing the obfuscated model through reverse engineering the modified TFLite library and customized layers, we can use code obfuscation strategies, a well-developed technology to make the code unreadable and unable to be reverse engineered. Code obfuscation is a well-developed technology, so we do not show the detail in our paper. The combination of model and code obfuscation would hide the key information of models.

3.4 Model Assembling & Library Recompilation

After obtaining the obfuscated model and modified TFLite source codes, *ModelObfuscator* assembles the new obfuscated model using the obfuscated model structure 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 and code library. To further prevent attackers from parsing the model by reverse engineering the modified TFLite library if it is possible, developers can use code obfuscation, which is a well-developed technique, to obfuscate the source code of the TFLite library.

4 *MODELOBFUSCATOR* EVALUATION

This work aims to use our proposed model obfuscation method to secure the deployed neural networks. To determine if this objective has been achieved, we evaluate *ModelObfuscator* by answering the following key research questions:

- **RQ1:** How effective is *ModelObfuscator* at obfuscating on-device ML models?
- **RQ2:** What is the overhead of obfuscated on-device ML models?
- **RQ3:** What is the impact of parameter sensitivity of *ModelObfuscator* on obfuscated model overhead?
- **RQ4:** How effective is *ModelObfuscator* in defending against on-device ML model parsing?

In this section, we show the evaluations of the proposed *ModelObfuscator* in terms of effectiveness, efficiency, and reliability. Our experiment settings are shown as follows:

Dataset. To evaluate *ModelObfuscator*'s performance on models with various structures for multiple tasks, we collected 10 TFLite models including a fruit recognition model, a skin cancer diagnosis model, MobileNet [14], MNASNet [43], SqueezeNet [19], EfficientNet [44], MiDaS [33], LeNet [24], PoseNet [22], and SSD [27]. The fruit recognition and skin cancer diagnosis model were collected from Android apps (see the provided code repository). The other models were collected from the TensorFlow Hub [7]. All of these models can be found in our provided code repository.

Experimental Environment. *ModelObfuscator* is evaluated on a workstation with Intel(R) Xeon(R) W-2175 2.50GHz CPU, 32GB RAM, and Ubuntu 20.04.1 operating system.

4.1 RQ1: Obfuscation Effectiveness

In addressing this RQ we aim to demonstrate the effectiveness of *ModelObfuscator*. Attackers can either extract ML model information from the model file [26] or analyze the structure of models to

[7] https://tfhub.dev/

Table 1: Comparison between original model files and obfuscated model files. 'Operator codes': type of the neural layer in this model. 'Tensors': the tensor that should be allocated in the memory. 'Operators': each neural layer. 'OR': obfuscation rate for original components. Note that the *renaming* and *parameter encapsulation* will increase and decrease the number of the corresponding model component, respectively.

	Example	Number	OR
Operator codes			
Original	"deprecated_builtin_code": 3	50	**100%**
Obfuscated	"deprecated_builtin_code": 32, "custom_code": "Fvsfxf",	510	
Tensors			
Original	"shape": [1,224,224,3], "name": "conv2d_1/Relu", · · · "shape": [500,3200], "name": "dense_1/kernel/transpose",	1345	**100%**
Obfuscated	"shape": [1,224,224,3], "name": "Fvsfxf",	534	
Operators			
Original	"inputs": [3,2,0], "outputs": [1], "op_type": "Conv2DOptions", "builtin_options": {stride_w: 1, · · ·},	510	**100%**
Obfuscated	"inputs": [0], "outputs": [1], "op_type": "Fvsfxf", "custom_options": {84,0,1,3,1, · · ·},	510	

collect similar models from the web for security exploitation [16, 17]. Thus, we demonstrate the effectiveness of *ModelObfuscator* in obfuscating model files and structures.

Model File Obfuscation. In this experiment, *renaming*, *parameter encapsulation*, and *neural structure obfuscation* approaches of *ModelObfuscator* are employed. To demonstrate the effectiveness of these approaches, we use *Flatc*[8], an existing python tool to parse the information of TFLite models in the buffer.

A comparison of file contents between the original models and *ModelObfuscator* obfuscated models is shown in Table 1. The results include the summary of all 10 models. For operator codes (types of the neural layer), *ModelObfuscator* creates different obfuscated operator names (e.g., layer names) for each neural layer. As shown in the first section of Table 1, the original model assigns the same name to the same type of layers (*i.e.*, 510 layers are assigned with 50 names), while the obfuscated models assign random names to each of the individual layer (i.e., 510 different operator codes with random names). This can confuse attackers in inferring the functionality of layers by hiding the relationship between certain layers.

For tensors, as shown in the second section of Table 1, the original model file contains the exact shape and name of each allocated tensor. The allocated tensors include the inputs, outputs, and weights of each layer. It also provides the index of layers' weights. In contrast, the obfuscated model has a fixed tensor shape (`[1,224,224,3]` in the example) and random names. In addition, it does not have information on layers' weights. So, the obfuscated model file only has 534 allocated tensors. Attackers cannot extract the weights and output shape of each layer from model files.

[8] https://google.github.io/flatbuffers/flatbuffers_guide_use_python.html

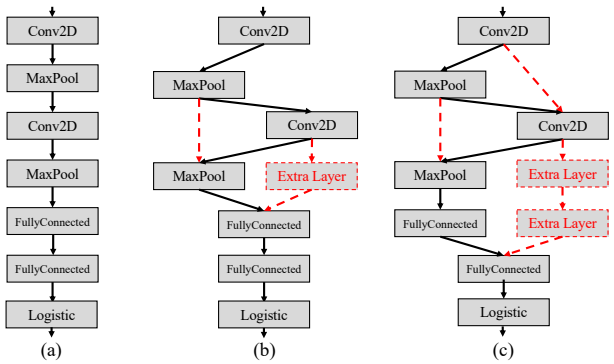


Figure 5: Example of structure obfuscation for LeNet. (a) Original model. (b) Obfuscated model with one extra shortcut & layer. (c) Obfuscated model with two extra shortcuts & layers. Red dotted line and red dotted block represent extra shortcuts and extra layers, respectively.

For operators, as shown in the third section of Table 1, the original model file contains detailed information about each layer, including the inputs, outputs, type, and settings ('`builtin_options`'). In contrast, the obfuscated models exclude the weights (`inputs 3` and `2` in this example) of layers in the input list, hide the operator types, and randomize the settings. Thus, attackers cannot get the details of layers. Overall, these three obfuscation strategies (*i.e.*, *renaming*, *parameter encapsulation*, and *neural structure obfuscation*) can effectively obfuscate the model file. ***ModelObfuscator* can confuse attackers, especially those using automatic tools or reverse engineering [26], to extract model information.**

Model Structure Obfuscation. While we can obfuscate the model file component, attackers still can parse the model using model structure. In order to mitigate against such attacks, we conducted a second experiment to showcase the effectiveness of *ModelObfuscator* in obfuscating model structures, which only utilizes the *shortcut* and *extra layer injection*. We use the Propagation Graph Kernel method [38] to calculate the structure similarity between graphs (*i.e.*, original model structure and obfuscated model structure), which is shown in Table 2. We utilize the structure similarity between the models to demonstrate the effectiveness of our approach in obfuscating the model structure. This is because the model structure can be represented as a directed graph, as shown in Figure 5, with each layer functioning as a node in the graph.

Table 2 and 3 show the structure difference between the original model and the obfuscated model will lay on the normal range of the difference (*i.e.*, 0.50-0.95) between original models when the number of extra shortcuts & extra layers is 20. **Our results show that attackers will find it hard to use structure similarity to identify the correct similar models (*i.e.*, a model that has similar structure and parameters) from the web when the model information – structure and parameters – has been obfuscated by *ModelObfuscator*.** In addition, the more the number of extra shortcuts and extra layers, the higher the effectiveness of structure obfuscation. When the model has higher complexity (*e.g.*, MiDaS), *ModelObfuscator* needs more number of extra shortcuts

Table 2: Structure similarity between original and obfuscated structures. We use the Propagation Kernel algorithm to calculate the similarity between two graphs [38]. '(n_1, n_2)': obfuscated models with n_1 shortcuts and n_2 extra layers. The range of similarity is [0,1]. Note that we only use the *extra shortcuts & extra layer injection* in this experiment.

(n_1, n_2)	Fruit	Skin	MobileNet	MNASNet	SqueezeNet	EfficientNet	MiDaS	LeNet	PoseNet	SSD	**Average value**
(10, 10)	0.78	0.81	0.80	0.95	0.89	0.93	0.97	0.60	0.81	0.95	**0.86**
(20, 20)	0.53	0.61	0.64	0.82	0.75	0.84	0.95	0.59	0.65	0.89	**0.74**
(30, 30)	0.48	0.59	0.51	0.76	0.63	0.77	0.92	0.59	0.64	0.81	**0.67**

Table 3: Structure similarity between the original models. '① - ⑩': Fruit, Skin, MobileNet,MNASNet, SqueezeNet, EfficientNet, MiDaS, LeNet, PoseNet, SSD.

	①	②	③	④	⑤	⑥	⑦	⑧	⑨	⑩
①	1.0	0.99	0.98	0.80	0.66	0.80	0.66	0.18	0.97	0.67
②	0.99	1.0	0.99	0.81	0.68	0.82	0.68	0.18	0.97	0.69
③	0.98	0.99	1.0	0.82	0.67	0.83	0.68	0.18	0.97	0.68
④	0.80	0.81	0.82	1.0	0.63	0.99	0.94	0.18	0.79	0.58
⑤	0.66	0.68	0.67	0.63	1.0	0.69	0.57	0.20	0.69	0.63
⑥	0.80	0.82	0.83	0.99	0.68	1.0	0.93	0.18	0.81	0.66
⑦	0.66	0.68	0.68	0.94	0.57	0.93	1.0	0.12	0.68	0.50
⑧	0.18	0.18	0.18	0.18	0.20	0.18	0.12	1.0	0.55	0.62
⑨	0.97	0.97	0.97	0.79	0.69	0.81	0.68	0.55	1.0	0.83
⑩	0.67	0.69	0.68	0.58	0.63	0.66	0.50	0.62	0.83	1.0

& layers to generate obfuscated models that have low structure similarity with original models.

> Overall, the *ModelObfuscator* can effectively obfuscate both the data and structures of on-device ML models.

4.2 RQ2: Obfuscation Overhead

Ideally, applying *ModelObfuscator* to an on-device ML model should not affect the prediction accuracy of the original models, while still providing sufficient defense against model attacks. To this end, we apply all five proposed obfuscation strategies to each model and compare the prediction results based on 1,000 randomly generated inputs. The obfuscation error is calculated as $||\boldsymbol{y}-\boldsymbol{y'}||_2$, where $\boldsymbol{y}$ and $\boldsymbol{y'}$ is the output of original models and obfuscated models, respectively. Note that the number of extra layers and shortcuts is set to 30 in *shortcut injection & extra layer injection*. In this experiment, **the obfuscation error of the proposed model obfuscation method is 0 for all 10 on-device models. It shows that our obfuscation strategies have no impact on the prediction results of the original models**.

In order to evaluate the efficiency impact of our obfuscation strategies on ML models, we conducted experiments to measure the runtime overhead of each model in our dataset. We measured both the time and memory overhead of the proposed obfuscation method under various settings, based on 1,000 randomly generated instances. The results of these experiments are presented in Tables 4 and 5, which respectively report the time overhead and memory overhead of the obfuscated models. We also include the time and memory consumption of the original models as the baseline. As shown in Table 4, even though extra layers are injected into the obfuscated model, ***ModelObfuscator* obfuscated models incur a negligible time overhead** (*i.e.*, approximately 1% on average for the most time-consuming obfuscation). The differences between using various obfuscation settings are also not significant. Because the *parameter encapsulation* will remove some data processing steps in the source code of APIs, the basic obfuscation ('(0,0)' in Table 4) may reduce the latency of TFLite models.

The memory overhead for *ModelObfuscator* obfuscated models is shown in Table 5. To eliminate the impact of different memory optimization methods, we use peak RAM usage where the model preserves all intermediate tensors. We argue that even with 20 extra shortcuts and 20 extra layers in our experiment, which provides sufficient protection to original models, **the memory overhead of obfuscated ML models is acceptable (approximately 20%)**.

Considering that the models are deployed on mobile devices that have limited storage space, we also present the size differences of the modified TFLite software library and the obfuscated models. Note that the size change is caused by creating additional `.so` files to support the inference of the obfuscated layer. Hence, the size difference will be similar with different implementations (*e.g.*, Python, Java, etc.). Table 6 shows the size change to the TFLite Python library and the TFLite models after applying *ModelObfuscator* obfuscation strategies. We use all obfuscation strategies and the number of extra shortcuts and extra layers is 30. Our results show that **the library size change is mainly caused by the *renaming* method**, because it will create a new API for the renamed layer in the TFLite library. In addition, the *extra layer injection* will not increase the size of the library, although it will create a new API to support the extra layer. Because the extra layer just has a simple function, the effect of the *extra layer injection* in size is negligible. Our obfuscation strategies **significantly reduce the size of the TFLite model file** because it only keeps the obfuscated minimal structure information. However, **the size of the TFLite library is significantly increased**, and increases the size of the application deployed on the mobile device.

> The time overhead of *ModelObfuscator* on on-device ML models is negligible. In addition, the memory overhead on obfusacted models is acceptable (approximately 20%) when the model has sufficient protection. However, *ModelObfuscator* will increase the size of the TFLite library.

4.3 RQ3: Parameter Sensitivity

In this study, our research question investigates the overhead of obfuscated on-device models in various settings. Our proposed tool, *ModelObfuscator*, includes two hyper-parameters: the number of

Table 4: Time overhead (seconds per 1000 inputs) of the original model and obfuscated model. We use five obfuscation strategies. We set the number of extra shortcuts and extra laters to 20. '+' and '-' refer to the increase and decrease, respectively.

	Fruit	Skin	MobileNet	MNASNet	SqueezeNet	EffcientNet	MiDaS	LeNet	PoseNet	SSD	**Average value**
Original	30.3	98.5	62.1	72.5	33.4	81.2	346.6	1.5	130.4	231.2	**108.8**
Obfuscated	30.2	98.7	63.2	74.3	34.6	82.9	349.2	1.6	130.4	237.1	**110.2**
Difference	-0.1	+0.2	+1.1	+1.8	+1.2	+1.7	+2.6	+0.1	0.0	+5.9	**+1.4**

Table 5: Overhead of the model obfuscation on random access memory (RAM) cost (Mb per model). We use five obfuscation strategies. 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	18.8	49.5	33.1	46.5	41.3	51.9	237.1	2.8	43.4	97.8	**62.2**
Obfuscated	30.1	65.6	50.5	56.6	52.6	66.3	254.5	3.1	61.6	107.6	**74.8**
Difference	+11.3	+16.1	+17.4	+10.1	+11.3	+14.4	+17.4	+0.3	+18.2	+9.8	**+12.6**

Table 6: Size change (Mb) of the TFLite library and models after the obfuscation. The size of the obfuscated model is reduced to a few Kb. The original library size is 183 Mb.

	Fruit	Skin	MobileNet	MNASNet	SqueezeNet	EffcientNet	MiDaS	LeNet	PoseNet	SSD
Library (renaming)	+8.7	+31.9	+19.6	+38.7	+8.8	+40.6	+126.1	+11.6	+9.7	+52.1
Library (all strategies)	+8.7	+31.9	+19.6	+38.7	+8.8	+40.6	+126.1	+11.6	+9.7	+52.1
Model	-5.5	-16.9	-10.3	-17.5	-5.0	-18.6	-66.3	-6.5	-5.0	-27.5
Total	+3.2	+15.0	+9.3	+21.2	+3.8	+22.0	+59.8	+5.1	+4.7	+24.4

Table 7: Time consumption (seconds per 1000 inputs) of obfuscated models in different settings. '(n_1, n_2)': obfuscated models with n_1 shortcuts and n_2 extra layers.

($n1$, $n2$)	Fruit	Skin	MobileNet	MNASNet	SqueezeNet	EffcientNet	MiDaS	LeNet	PoseNet	SSD	**Average**
(0, 0)	30.4	98.7	61.1	70.4	33.5	82.3	346.4	1.5	130.7	232.1	**108.7**
(10, 0)	30.4	98.7	61.0	70.7	33.4	82.1	346.5	1.7	130.2	232.1	**108.7**
(20, 0)	30.5	98.6	61.4	70.4	33.6	82.4	346.4	1.6	130.6	232.3	**108.8**
(30, 0)	30.7	97.9	62.3	71.7	33.1	82.0	346.9	1.5	130.2	231.0	**108.7**
(0, 10)	30.2	98.6	61.1	74.1	34.9	82.3	346.7	1.6	130.9	231.6	**109.2**
(0, 20)	30.6	98.3	63.3	74.1	36.8	81.3	348.2	1.6	130.5	231.3	**109.6**
(0, 30)	30.4	98.5	63.2	74.6	34.5	82.8	349.0	1.8	130.4	236.7	**110.2**

additional shortcuts and the number of extra layers, which are used to regulate the effectiveness of obfuscating model structures. As demonstrated in Tables 7 and 8, we conduct ablation studies to examine the impact of these hyper-parameters on the overhead of obfuscated models. The notation (n_1, n_2) in the table represents the number of shortcuts (n_1) and extra layers (n_2) applied. When $(0, 0)$ is used, it indicates that only basic obfuscations (*i.e.*, *renaming*, *parameter encapsulation*) and *neural structure obfuscation* are employed. Our results in Table 7 indicate that **these two parameters have little effect on the time overhead**. However, as shown in Table 8, **the RAM cost of obfuscated models increases with the number of extra layers**.

Hence, it is worthwhile to consider the trade-off between the obfuscation complexity and the memory overhead.

4.4 RQ4. Resilience to Attacks

To demonstrate the effectiveness in concealing information of models, we use three distinct model parsing methods:

(1) The first method involves model format conversion. Existing model conversion tools utilize public APIs to access a model's structure and weights, and then assemble them into a new model with different formats. We utilize three tools, namely TF-ONNX [10], TFLite2ONNX [48], and TFLite2TF [18], to evaluate the performance of *ModelObfuscator*. If a tool can convert the model format, we consider it a successful model information extraction. Conversely, if *ModelObfuscator* is effective, these tools will be unable to extract the model information.

(2) The second method involves reverse engineering in the buffer. The on-device model format TFLite loads data on buffer [9]. We

[9] https://google.github.io/flatbuffers/

Table 8: Random access memory (RAM) cost (Mb per model) of obfuscated models in different settings. '(n_1, n_2)': obfuscated models with n_1 shortcuts and n_2 extra layers.

($n1$, $n2$)	Fruit	Skin	MobileNet	MNASNet	SqueezeNet	EffcientNet	MiDaS	LeNet	PoseNet	SSD	**Average**
(0, 0)	18.9	53.3	33.1	49.4	45.5	56.4	244.8	2.9	40.6	99.9	**64.5**
(10, 0)	18.8	53.2	33.3	49.7	45.8	56.8	244.8	2.8	40.5	99.8	**64.6**
(20, 0)	18.9	53.5	32.8	49.7	45.9	56.7	244.3	2.8	40.5	99.9	**64.5**
(30, 0)	18.7	53.4	32.6	49.5	46.0	55.8	244.6	2.9	40.3	100.2	**64.4**
(0, 10)	25.9	58.8	42.5	53.6	49.2	69.5	248.1	2.9	55.0	107.5	**71.3**
(0, 20)	30.0	67.3	50.4	56.7	52.0	66.2	253.1	3.1	63.9	110.0	**75.2**
(0, 30)	35.9	81.9	55.2	57.3	71.6	73.0	267.7	3.1	73.4	123.8	**81.9**

Table 9: The success number of existing reverse engineering methods to extract the information of on-device models with different obfuscation strategies. '√': this model parsing method cannot extract information for all models.

	Model conversion tool			**Parsing model in the buffer**	**Feature analyzing**
	TF-ONNX	TFLite2ONNX	TFLite2TF	Reverse Engineering in FlatBuffer [26]	Smart App Attack [16]
Without obfuscation	10	9	9	10	8
Renaming	√	√	√	√	8
Parameter encapsulation	√	√	√	√	2
Neural structure obfuscation	√	√	√	√	8
Shortcut injection	√	√	√	√	8
Extra layer injection	√	√	√	√	8
ModelObfuscator	√	√	√	√	√

refer to a study by Li et al. [26], in which the researchers attempt to extract a model's structure and weights in FlatBuffer. If *ModelObfuscator* is effective, the FlatBuffer extractor will be unable to parse the data of the obfuscated model and reverse it to the original one.

(3) The third method involves finding a similar differentiable model from the Internet by comparing the features among models. Although the mainstream on-device model format TFLite does not support some advanced functions like auto-differentiation to protect the deployed model, attackers such as the App Attack can find a differentiable model with a similar structure and weights from the Internet [16]. For the App Attacking method, if it can correctly identify the obfuscated model that has the same model structure as the original one on TensorFlow Hub, we consider it a successful extraction of the model information.

In this experiment, we evaluated the effectiveness of the proposed obfuscation strategies, including *renaming*, *parameter encapsulation*, *neural structure obfuscation*, and *shortcut & extra layer injection*, on the original models. Table 9 shows the results of our evaluation, where we used three different model parsing methods to attempt to extract information from the TFLite models. **Our proposed *ModelObfuscator* successfully prevented all existing model parsing tools and methods from obtaining the information of TFLite models.** In addition, we found that parameter encapsulation alone was able to prevent the App Attack from finding surrogate models on six of the models. However, applying other obfuscation strategies separately did not confuse the App Attack, as it uses the features of the model structure and parameters simultaneously to analyze the model. Therefore, **to defend against the App Attack, we need to obfuscate the model structure and parameters simultaneously by combining the proposed obfuscation methods.**

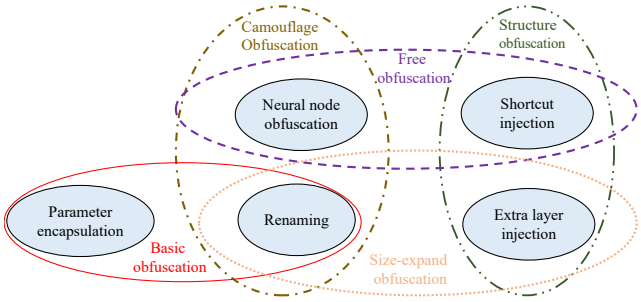


Figure 6: Taxonomy of model obfuscation strategies.

> The proposed *ModelObfuscator* is effective in defending against all three types of model parsing methods, thus providing robust protection for deployed models.

5 DISCUSSION

In this section, we will present the taxonomy of model obfuscation strategies and the limitation of our methods. In addition, we show the possible way to extract information from the obfuscated model.

5.1 Taxonomy of Model Obfuscation Strategies

In this paper, we proposed five different model obfuscation strategies. Figure 6 shows a preliminary taxonomy of the different model

obfuscation methods and the best practice for model deployment. First, developers can use the *renaming* and *parameter encapsulation* to prevent most model parsing or reverse engineering tools from extracting the information of the deployed model. In the scenarios where computational costs are critical, developers can use the *neural structure obfuscation* and *shortcut injection*, as they do not introduce any additional overhead. For structure obfuscation, developers can use *shortcut injection* and *extra layer injection*. These methods can significantly increase the difficulty of understanding the model structure of the deployed models. In addition, developers can use *renaming* and *neural structure obfuscation* to disguise the deployed model, which can mislead the attacker into choosing the wrong architecture to produce a surrogate model. If the size of the app package is critical (e.g., deploying the model on devices with limited storage), developers need to carefully consider the trade-off between the number of obfuscated layers leveraging *extra layer injection* with *renaming*. The reason is that if *renaming* is used to create different obfuscated layers for each layer, *ModelObfuscator* needs to create the corresponding APIs in the library to support obfuscated layers, hence increasing the library size.

5.2 *ModelObfuscator vs.* White-box Gradient Attacks

In Section 4.4, we show that our method can prevent existing reverse engineering methods from extracting the information of on-device models. However, what will happen if attackers directly perform white-box gradient attacks on the obfuscated model? In this subsection, we investigate the attacking performance of white-box gradient attacks on obfuscated on-device models, and analyze the factor that disables such kinds of attacks. We use three well-known white-box gradient attack methods FGSM [13], BIM [23], and PGD [28] to evaluate the resilience of our method to this kind of attack. We use the Foolbox tool [34] to generate these attacks. The results are shown in Table 10. We cannot directly produce attacks for on-device models because the TFLite model format does not support gradient computing. But we can first convert the TFLite model to the PyTorch model using model conversion tools. The on-device models without obfuscation can be successfully converted to the PyTorch model, and the white-box gradient attack methods can successfully generate attacks for the converted PyTorch model. However, as we mentioned in Section 4.4, the model conversion tools cannot transform obfuscated models into PyTorch models. When we directly use the obfuscated models as the input of the attack-generating functions, the attack methods cannot generate attacks for the obfuscated model. This is because (1) first, the white-box attack methods cannot parse the information of the obfuscated model, and (2) second, they cannot compute the gradient information of on-device models. Thus, we can conclude our method can prevent attackers from generating white-box gradient attacks using existing tools (*e.g.*, model conversion tools and attack-generating tools).

Table 10: Evaluation using white-box gradient attacks. '√': the attack method cannot generate attacks. '×': the attack method can generate attacks.

	FGSM	BIM	PGD
Without obfuscation	×	×	×
ModelObfuscator	√	√	√

5.3 How to Parse Obfuscated Models?

When ModelObfuscator obfuscates the model, it will create a cache file to guide the tool to generate a compatible DL library. Attackers cannot automatically extract the obfuscated model unless they obtain the cache file from the developer's computer (which is unlikely to happen). Generally speaking, attackers must use reverse engineering to get source codes from the compiled library, which consists of binary files, and it will cost significant manual effort to understand the obfuscated model.

5.4 Limitations

Although the proposed model obfuscation does not introduce significant computational overhead, it will increase the size of the modified TFLite library. This is because we need to provide support for the new obfuscations made. For a huge network like a 1000-layer network deployed model (although it is unlikely to find a such deployed model in the real world), the size of the modified TFLite library will significantly increase if we rename every layer. As a result, the app package also increases as the modified TFLite library will also need to be deployed on the device. Besides, our method cannot defend against query-based attacks as they do not need the inner information of models. But they are less effective than white-box attacks, which can be disabled by the proposed *ModelObfuscator*.

6 CONCLUSION

In this work, we analyzed the risk of deep learning models deployed on mobile devices. Attackers can extract information from the deployed model to perform white-box-like attacks and steal its intellectual property. To this end, we proposed a model obfuscation framework to secure the deployed DL models. We utilized five obfuscation strategies to obfuscate the information of deployed models, *i.e.*, *renaming*, *parameter encapsulation*, *neural structure obfuscation*, *shortcut & extra layer injection*. We developed a prototype tool *ModelObfuscator* to automatically obfuscate the TFLite model and produce a compatible library with the model. Experiments show that our method is effective in resisting the model parsing tools without performance sacrifice. Although the *ModelObfuscator* will increase the library size, considering the acceptable time & memory overhead required and the extra security it achieves, we believe our method has the potential to be an essential step for security-sensitive smart apps. In future works, optimizing model obfuscation to reduce the library size is worthwhile to be explored.

ACKNOWLEDGMENTS

Zhou is supported by a Monash Graduate scholarship. Grundy is supported by ARC Laureate Fellowship FL190100035. Gao is supported by the National Natural Science Foundation of China under Grant No (62202026). Also, our sincere gratitude goes to Jiawei Wang, a PhD student at Monash, for providing invaluable help in solving problems of TFLite modification.

REFERENCES

[1] 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.

[2] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* (2016), arXiv–1605.

[3] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. 2017. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526* (2017).

[4] Zhenpeng Chen, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, Tao Xie, and Xuanzhe Liu. 2020. A comprehensive study on challenges in deploying deep learning based software. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 750–762. https://doi.org/10.1145/3368089.3409759

[5] Zhenpeng Chen, Huihan Yao, Yiling Lou, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, and Xuanzhe Liu. 2021. An empirical study on deployment faults of deep learning based mobile applications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 674–685. https://doi.org/10.1109/icse43902.2021.00068

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

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

[8] 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

[9] Christian S. Collberg and Clark Thomborson. 2002. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on software engineering* 28, 8 (2002), 735–746. https://doi.org/10.1109/tse.2002.1027797

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

[11] 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

[12] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Gong. 2020. Local Model Poisoning Attacks to {Byzantine-Robust} Federated Learning. In *29th USENIX Security Symposium (USENIX Security 20)*. 1605–1622.

[13] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*.

[14] 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).

[15] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, et al. 2020. 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*. ACM, 385–399. https://doi.org/10.1145/3373376.3378460

[16] 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. https://doi.org/10.1109/tifs.2022.3172213

[17] Yujin Huang, Han Hu, and Chunyang Chen. 2021. 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 (ICSE-SEIP)*. IEEE, 101–110. https://doi.org/10.1109/icse-seip52600.2021.00019

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

[19] 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).

[20] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. 675–678.

[21] Sanjay Kariyappa and Moinuddin K Qureshi. 2020. Defending against model stealing attacks with adaptive misinformation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 770–778.

[22] 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

[23] Alexey Kurakin, Ian J Goodfellow, and Samy Bengio. 2018. Adversarial examples in the physical world. In *Artificial intelligence safety and security*. Chapman and Hall/CRC, 99–112. https://doi.org/10.1201/9781351251389-8

[24] 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

[25] Jingtao Li, Zhezhi He, Adnan Siraj Rakin, Deliang Fan, and Chaitali Chakrabarti. 2021. NeurObfuscator: A Full-stack Obfuscation Tool to Mitigate Neural Architecture Stealing. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 248–258. https://doi.org/10.1109/host49136.2021.9702279

[26] 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

[27] 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.

[28] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards Deep Learning Models Resistant to Adversarial Attacks. In *International Conference on Learning Representations(ICLR)*. https://openreview.net/forum?id=rJzIBfZAb

[29] Mantas Mazeika, Bo Li, and David Forsyth. 2022. How to steer your adversary: Targeted and efficient model stealing defenses with gradient redirection. In *International Conference on Machine Learning*. PMLR, 15241–15254.

[30] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 883–898. https://doi.org/10.1145/3453483.3454083

[31] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. 2019. Prediction poisoning: Towards defenses against dnn model stealing attacks. *arXiv preprint arXiv:1906.10908* (2019).

[32] 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).

[33] 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

[34] Jonas Rauber, Wieland Brendel, and Matthias Bethge. 2017. Foolbox: A Python toolbox to benchmark the robustness of machine learning models. In *Reliable Machine Learning in the Wild Workshop, 34th International Conference on Machine Learning*. http://arxiv.org/abs/1707.04131

[35] Lutz Roeder. 2017. *Netron, Visualizer for neural network, deep learning, and machine learning models.* https://doi.org/10.5281/zenodo.7109451

[36] 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

[37] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *2017 IEEE symposium on security and privacy (SP)*. IEEE, 3–18. https://doi.org/10.1109/sp.2017.41

[38] Giannis Siglidis, Giannis Nikolentzos, Stratis Limnios, Christos Giatsidis, Konstantinos Skianis, and Michalis Vazirgiannis. 2020. GraKeL: A Graph Kernel Library in Python. *Journal of Machine Learning Research* 21, 54 (2020), 1–5.

[39] Zhichuang Sun, Ruimin Sun, Long Lu, and Alan Mislove. 2021. Mind your weight (s): A large-scale study on insufficient machine learning model protection in mobile apps. In *30th USENIX Security Symposium (USENIX Security 21)*. 1955–1972.

[40] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199* (2013).

[41] Kálmán Szentannai, Jalal Al-Afandi, and András Horváth. 2019. Mimosanet: An unrobust neural network preventing model stealing. *arXiv preprint arXiv:1907.01650* (2019).

[42] Kálmán Szentannai, Jalal Al-Afandi, and András Horváth. 2020. Preventing Neural Network Weight Stealing via Network Obfuscation. In *Science and Information Conference*. Springer, 1–11.

[43] 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

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

[45] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2016. Stealing machine learning models via prediction {APIs}. In *25th USENIX security symposium (USENIX Security 16)*. 601–618.

[46] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.

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

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

[49] 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

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

[51] Hui Xu, Yuxin Su, Zirui Zhao, Yangfan Zhou, Michael R Lyu, and Irwin King. 2018. Deepobfuscation: Securing the structure of convolutional neural networks via knowledge distillation. *arXiv preprint arXiv:1806.10313* (2018).

[52] Mengwei Xu, Jiawei Liu, Yuanqiang Liu, Felix Xiaozhu Lin, Yunxin Liu, and Xuanzhe Liu. 2019. A first look at deep learning apps on smartphones. In *The World Wide Web Conference*. 2125–2136. https://doi.org/10.1145/3308558.3313591

[53] 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.

[54] 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. https://doi.org/10.1109/cvpr42600.2020.00031

Received 2023-02-16; accepted 2023-05-03