Investigating White-Box Attacks for On-Device Models

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

Kui Liu
brucekuiliu@gmail.com
Huawei Software Engineering
Application Technology Lab
China

Hailong Sun
sunhl@buaa.edu.cn
Beihang University
Beijing, China

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

ABSTRACT

Numerous mobile apps have leveraged deep learning capabilities. However, on-device models are vulnerable to attacks as they can be easily extracted from their corresponding mobile apps. Although the structure and parameters information of these models can be accessed, existing on-device attacking approaches only generate black-box attacks (*i.e.*, indirect white-box attacks), which are less effective and efficient than white-box strategies. This is because mobile deep learning (DL) frameworks like TensorFlow Lite (TFLite) do not support gradient computing (referred to as non-debuggable models), which is necessary for white-box attacking algorithms. Thus, we argue that existing findings may underestimate the harmfulness of on-device attacks. To validate this, we systematically analyze the difficulties of transforming the on-device model to its debuggable version and propose a Reverse Engineering framework for On-device Models (*REOM*), which automatically reverses the compiled on-device TFLite model to its debuggable version, enabling attackers to launch white-box attacks. Our empirical results show that our approach is effective in achieving automated transformation (*i.e.*, 92.6%) among 244 TFLite models. Compared with previous attacks using surrogate models, *REOM* enables attackers to achieve higher attack success rates (10.23%→89.03%) with a hundred times smaller attack perturbations (1.0→0.01). Our findings emphasize the need for developers to carefully consider their model deployment strategies, and use white-box methods to evaluate the vulnerability of on-device models. Our **artifacts** [1] are available.

ACM Reference Format:
Mingyi Zhou, Xiang Gao, Jing Wu, Kui Liu, Hailong Sun, and Li Li. 2024. Investigating White-Box Attacks for On-Device Models . In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639144

*Dr. Li Li was a senior lecturer at Monash. He supervised this project for the whole period. Corresponding authors: Li Li.
[1]https://github.com/zhoumingyi/REOM

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639144

1 INTRODUCTION

The number of mobile devices worldwide is continuously growing. The capabilities of those devices also keep increasing, i.e., with powerful Central Processing Units (CPUs) and a large amount of memory, making them suitable for running deep learning (DL) models. Indeed, mobile devices have now become an ideal platform for deploying the DL model. Many intelligent applications have already been deployed on mobile devices [39] and have already benefited millions of users. Though DL models can also be deployed on a cloud platform, data transmission between a mobile device and the cloud may compromise user privacy. Indeed, to achieve high-level security, users' personal data should not be sent outside the device. This could be the reason why more and more DL models are deployed on the device, which has been advertised as one of the most important features by the newly emerged OpenHarmony mobile system [22]), and the corresponding models are often referred to as on-device models.

Unfortunately, such on-device models are directly presented on mobile devices, giving attackers a lot of opportunities to exploit since it is relatively easy to unpack mobile apps to locate the physical models. As a result, on-device models are facing more and more serious security threats. Although on-device models are released to users as black-box ones for preventing potential attacks because attackers cannot obtain gradient information[2] from on-device models (referred to as **non-debuggable models**), they do not fulfill such a purpose in practice. Indeed, as illustrated in Figure 1, attackers still find ways to attack black-box models without accessing their gradient information, e.g., via the so-called **transferable attacks** (referred to as **indirect white-box attack**). They achieve this by first, for target models, identifying debuggable surrogate models that are available to generate attacks. They then exploit surrogate models through white-box strategies. Once the strategies satisfy the attackers' needs, they apply the same strategies to attack on-device models.

In fact, many vulnerabilities of on-device models have already been discovered by our fellow researchers in recent years. For example, Huang et al. [15, 16] propose to achieve the purpose by parsing features of the on-device model to find a surrogate model from the web, which could then be used to launch transferable attacks on mobile models. Cao *et al.* [4] also use surrogate models for attacking mobile models under the black-box setting, albeit by obtaining

[2]Gradient information is considered crucial to implement effective white-box attacks.

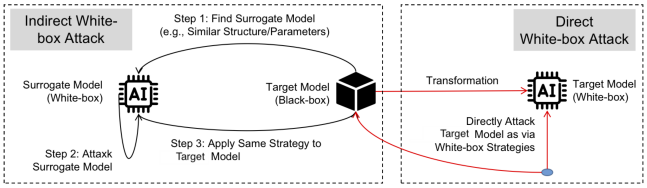

Figure 1: The typical scenarios of evaluating the vulnerability of on-device DL models.

information from mobile models via querying their outputs, and then training a surrogate model using such information.

Unfortunately, the performance of these approaches is highly dependent on the similarity between the surrogate model and the target models. It is often difficult to find an ideal surrogate model that is highly similar to the target, thus affecting the effectiveness of attacks. Since on-device models are directly hosted on devices, we manually look into those models and observe that such on-device models still keep the model's architecture and weights information but cannot be directly accessed. We are, therefore, wondering if it is possible to extract this information so as to allow for evaluating vulnerabilities of on-device models as if they are white-box ones, without generating surrogate models (*cf*. the right part in Figure 1). If that is possible, current evaluation methods underestimate the threats of on-device models as direct white-box attacks are much more effective than black-box attacks (*i.e.*, indirect white-box attacks) [35]. To this end, in this work, we propose a method *REOM* to explore the following research question:

RQ: Can on-device Models Be Directly Attacked via White-box Strategies?

Borrowing the idea of reverse engineering the software artifact, which is often considered a black box as analysts cannot directly access the code, we start by checking if it is possible to reverse engineering on-device models. Our exploitation reveals that it is possible to obtain a white-box version of the target model by first transforming it to an Open Neural Network Exchange (ONNX) model and then transforming it back to debuggable AI models (thanks to the transparency of ONNX).

Towards answering the aforementioned research question, we start by verifying this hypothesis through a preliminary study. Specifically, we focus on transforming TensorFlow-Lite (TFLite in short) models, the most popular on-device models, to ONNX and then PyTorch models, which is the most popular debuggable model format. With 244 TFLite models extracted from 173,905 Google Play apps released in 2021, preliminary experimental results show that such a process is not able to achieve our purpose, i.e., transforming black-box on-device models with white-box versions. Indeed, over 90% of the models cannot be successfully transformed.

We then go one step deeper to understand why the majority of models cannot be transformed by looking into the error messages. Our manual investigation reveals three types of errors in the above model transformation study, namely `Compatibility Errors`, `Not Implemented Errors`, and `Input Type Errors`. To this end, we propose to complement the aforementioned process by automatically correcting those errors. In particular, we present to the community a model transformation framework called *REOM*, which includes dedicated strategies in three modification modules to correct the aforementioned three types of errors, respectively. Experimental results show that our approach is effective by increasing the success rate from 6.6% to 92.6%, with the aforementioned 244 on-device TFLite models. We further demonstrate that the transformed debuggable models and the original on-device model are indeed very similar, with a normalized ℓ_2 output distance less than 0.001 in most cases. Moreover, we also experimentally show that the transformed models can indeed support stronger attacks. Compared with previous methods of generating attacks using surrogate models, attackers can achieve higher attack success rates (10.23%⟶89.03%) with a hundred times smaller attack perturbations (1.0⟶0.01) based on our proposed tool.

The main contributions of this paper are shown as follows:

- We propose a complete Reverse Engineering framework for On-device Models (*REOM*) to convert the compiled on-device models to their corresponding debuggable version.
- *REOM* can transform the model automatically, which presents the potential to be an essential tool to develop methods for testing the reliability of on-device models.
- Our paper shows attackers can achieve comparable on-devices attack performance with the white-box setting. The current model deployment strategy is at serious risk.
- We provide solutions to defense against reverse engineering based on our observation.

2 BACKGROUND AND RELATED WORK

We now provide the necessary background about on-device DL models, DL model attack,s and the ONNX project.

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 [19], Keras [9], 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, the AI community has 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 [16, 39].

TFLite Models. TFLite models have powerful features for running models on edge devices but they do not provide APIs to access the gradient or intermediate outputs like other TensorFlow or PyTorch models. TensorFlow provides a *TensorFlow Lite Converter* to convert a TensorFlow model into a TensorFlow Lite model. In addition, the models trained by other DL frameworks can also be converted to the TFLite model. For example, PyTorch provides the API to save the model as ONNX format, and then convert the ONNX model to TensorFlow and TFLite model Onnx-tf tool. 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.

ONNX Models. Open Neural Network Exchange (ONNX) is an open format built to represent machine learning models. ONNX defines a common file format to enable developers to use models with a variety of frameworks and tools. ONNX platform has various tools to support the exchange between common neural network model formats (*e.g.*, TensorFlow, PyTorch) and the ONNX model format. For instance, the *tf2onnx* tool can transform the TensorFlow model to the ONNX model accurately. The *onnx2tf* and *onnx2pytorch* transform the ONNX model to corresponding TensorFlow and PyTorch models, respectively. The *onnx2tf* tool only generates a low-level saved model, which can just use the forward inference APIs (*i.e.*, the generated model is not debuggable). Differently, the mechanism of the *onnx2pytorch* is based on a rule list. which defines the relationship between ONNX operators and PyTorch operators. It will first create a model instance by translating the ONNX model based on the rule list and generate a forward function to define the data flow at runtime. The converted PyTorch model is debuggable. Our proposed reverse engineering method based on the ONNX platform will process the non-debuggable components on the unified ONNX level and then convert them to debuggable format. So, it can be applied to multiple on-device formats such as TFLite, ONNX Runtime, and Caffe. In contrast, other transformation pipelines may not easily handle multiple on-device formats and need ad-hoc manners to build different rules for different on-device model formats.

2.2 Adversarial Attacks for DL models

Adversarial attacks add perturbation that can be considered a special noise to the original image to fool the DL models. Adversarial attacks can be categorized into white-box attacks such as gradient-based attacks [10, 13, 20, 26–28, 31], and black-box attacks [3, 6–8, 14, 17, 18, 29]. Gradient descent (GD) is an iterative optimization algorithm, used to find a local minimum/maximum of a given function. This method is commonly used in training DL models. For the gradient-based (white-box) attack [13, 20], they use the gradient to compute the perturbation that can increase the model loss. Query-based black-box attacks [3, 5] estimate the gradients to compute the perturbation, *e.g.*, randomly update the perturbation to estimate the right update direction. White-box attacks have full access to the model structure and its parameters to enable gradient computing. In black-box attacks, only partial information (*i.e.*, model output) about the model is available. Goodfellow *et al.* [13] show that adversarial examples generated by surrogate models [30, 41] can fool the target model. Therefore, for adversarial attacks on devices, Huang *et al.* [16] and Cao *et al.* [4] evaluated the mobile model robustness by generating attacks from surrogate models. However, they heavily rely on the similarity between surrogate models and target models. According to this, we propose the *REOM* to transform the TFLite model to the PyTorch model, to explore the security issue of model deployment.

[3] schema file (The link is too long to display)

3 PRELIMINARY STUDY

Recall that the ONNX platform has provided various tools to support the exchange from neural network models to ONNX models. In this preliminary study, we would like to investigate whether these tools can be leveraged to transform a TFLite model (will be regarded as a Tensorflow model) into a PyTorch model (i.e., TFLite model $\xrightarrow{tf2onnx}$ ONNX model $\xrightarrow{onnx2pytorch}$ PyTorch model).

3.1 Harvesting On-device Models

To conduct this preliminary study, we need to first collect a set of TFLite models that are actually included in Android apps. Since there is no existing dataset containing a set of apps with TFLite models, we have to construct such a dataset from scratch. To this end, we resort to the AndroZoo dataset [21] to first collect a set of real-world Android apps. AndroZoo is by far the largest app set well maintained by researchers and has been widely leveraged by researchers to support various Android-related studies. At the moment, AndroZoo contains over 19 million Android apps. It is extremely time-consuming for us to download and scan all of them to locate TFLite models. For the sake of simplicity, we only focus on the latest Google Play apps (published from 2021 to 2022) to fulfill our study. In total, we have collected 173,905 apps (it takes more than one month). After disassembling the apps, we find 674 of them contain TFLite-related packages (i.e., *org.tensorflow*). All of these apps are regarded as candidates to extract TFLite models. Among 674 apps, we were eventually able to extract 244 TFLite models, which are then taken into account to fulfill our preliminary study.

3.2 Model Transformation Study

Figure 2 illustrates the working process of our preliminary model transformation study. We would like to check if existing tools can be leveraged to transform TFLite models into debuggable PyTorch models. Since ONNX does not directly provide the tool for TFLite models, we naively regard them as Tensorflow models to fulfill this study, which is made up of three steps, as highlighted in Figure 2.

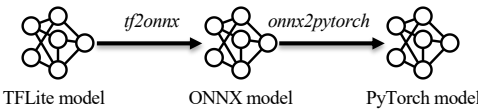

Figure 2: The naive transformation flow from compiled on-device models to debuggable models in our preliminary study.

- ***Step-1: Extractor*** First, we extract TFLite models from Android apps by applying the well-known *apktool* [4] tool to decompile the apps. *Apktool* is one of the most popular tools proposed for reverse engineering Android APKs. After decompilation, we search for `.tflite` files in the decompiled folder.
- ***Step-2: tf2onnx*** After obtaining the TFLite model, we then transform it to the ONNX model. The advantage of achieving the transformation based on the ONNX platform is that: (1) The ONNX model is intended to be easily modified. Adding or removing a layer from an ONNX model requires just one line of code. (2) Our transformation tool is easy to be applied to other on-device

[4] https://ibotpeaches.github.io/Apktool/

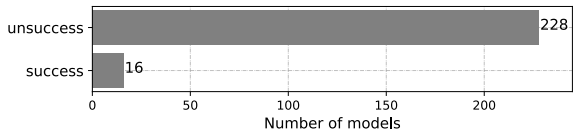

Figure 3: Results of the naive transformation flow.

Table 1: Errors types in the failure cases of the preliminary study. Note that one operator may cause multiple errors.

Errors	Reasons	Count	Related Operators
`Compatibility`	Structure Mismatch	156	Quantization Transformation
`Not Implemented`	Operators Mismatch	100	Quantization Data Processing Computing
	Operators Not Supported	24	Customized Deprecated
`Input Type`	Specification Mismatch	18	Transformation Computing

formats like caffe2 and ONNX Runtime. Specifically, once the on-device caffe2 models are converted to ONNX model, the proposed tool can be applied to convert the ONNX model to a debuggable version.

- ***Step-3: onnx2pytorch*** After we convert the TFLite model to an ONNX model, we then convert the ONNX model to the debuggable DL model, *i.e.*, PyTorch model. Here we choose the PyTorch model as the transformation target because the API library of PyTorch is more stable than that of TensorFlow. In addition, due to its flexibility, it is simpler for us to assemble the parsed information into the debuggable PyTorch model.

Experiment results: In this work, we use all the 244 apps (i.e., 244 TFLite models) identified in Section 3.1 to fulfill this study. Figure 3 summarizes the experimental results. Among the 244 models, only 16 of them can be successfully transformed into PyTorch models. This preliminary approach yields a failure rate of 93.4%, making it impossible to be adopted in practice to achieve our purpose, i.e., automatically transforming TFLite models to debuggable ones.

Error types: We then go one step further to check why some models can be successful while majorities cannot. The preliminary transformation approach fails with three types of errors: (1) `Compatibility Error` (156), (2) `Not Implemented Error` (124), and (3) `Input Type Error` (18). `Compatibility Error` appears when the model structure is not compatible with the debuggable model format. For the `Not Implemented Error`, it appears when the ONNX model has operators that are not in the transformation rule list. For the `Input Type Error`, it appears when the *onnx2pytorch* assigns wrong inputs and parameters to the layer. The detailed analysis can be found in Section 4.

Overall, our preliminary study shows that existing tools cannot achieve the purpose of transforming TFLite models into debuggable models. We, therefore, argue that there is a strong need to invent new approaches to address this challenge. Motivated by this evidence, we design and implement in this work a prototype tool called *REOM*, which aims at transforming on-device models by resolving the aforementioned errors.

> The fact that less than 10% of TFLite models can be automatically transformed to debuggable PyTorch models by existing approaches. It shows that there is a strong need to invent new approaches to achieve the purpose.

4 APPROACH

We now detail our approach proposed to transform on-device TFLite models into debuggable PyTorch models. Before presenting our method, we first analyze the errors (see Table 1) in existing tools:

- **`Compatibility - Structure Mismatch`:** To accelerate the computation on mobile devices equipped with mobile CPU and mobile GPU, compiled on-device models are optimized to contain some different data types and model structures with debuggable models. For example, `float32` will be converted to `uint8` when compiling the debuggable model to the on-device model. When converting the on-device model back to ONNX models, some extra operators (*e.g.*, quantization operators, transformation operators) will be created to make it compatible with the ONNX data types and structures. Since the extra structures are non-debuggable, the structure mismatch will unfortunately result in the failure of the transformation. **In our approach, we propose the *Pruning Module* to resolve this problem.**
- **`Not Implemented - Operator Mismatch`:** Some compiled operators of on-device models are optimized for mobile computing. The optimized operators transformed from TFLite model are not compatible with the debuggable model format. For example, when the data type of the on-device operator is `uint8` (optimized data type for on-device model), this operator will not be debuggable because DL frameworks like PyTorch and TensorFlow do not have debuggable API for `uint8`. In addition, TFLite defines many unique operators (*i.e.*, mismatched operators for debuggable models) supported by its corresponding library. However, such unique operators are not supported by other DL frameworks (*e.g.*, PyTorch, TensorFlow). **Therefore, in this work, we propose the *Translation Module* to bridge the mismatched operators.**
- **`Not Implemented - Operators Not Supported`:** Those customized **operators are not supported** by other DL models, resulting in TFLite cannot be directly transformed. For example, if developers want to implement an advanced Convolutional operator in their model but this advanced function is not supported by the current version of the on-device DL framework, they could add their customized C/C++ implementation of the advanced function to their TFLite model. Besides, developers can name this customized operator and define the interface freely. So, other DL frameworks cannot identify this customized operator because it is not included in the operator library. This problem also exists on the deprecated operators of TFLite models. Fortunately, on-device operators are usually a subset of the debuggable operators' library. **Therefore, we propose *Auto-matching Module* to identify the equivalent operators in the debuggable model format to fulfill the transformation.**
- **`Input Type - Specification Mismatch`:** The specifications of some operators (*e.g.*, computing operators, transformation operators) may vary in different model formats, such as the order of

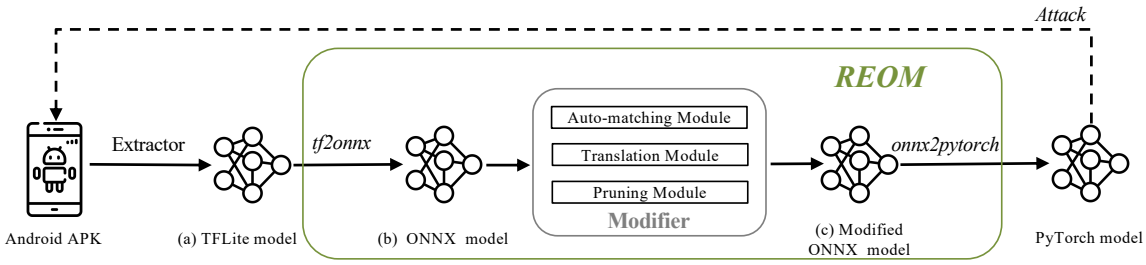

Figure 4: The overview working process of *REOM*. We use the (a), (b), and (c) to define the state of the model in *REOM*.

inputs and the range of parameters. Therefore, the setting of the converted debuggable operators may be wrong. **Because these errors need to be resolved in an ad-hoc manner, we omit them in this paper.**

Therefore, we design and implement in this work a prototype tool called *REOM*, which aims at removing the aforementioned errors while maximizing the similarity between the on-device model and the converted debuggable model. Figure 4 presents the overall working process of our proposed *REOM*, which essentially contains four steps to achieve its purpose, *i.e.*, transforming a TFLite model into a debuggable PyTorch model for security exploitation. The four steps are ① Extractor, ② *tf2onnx*, ③ Modifier, and ④ *onnx2pytorch*. The details of Extractor and *tf2onnx* can be found in Section 3.2.

4.1 Modifier

We now detail the Modifier of *REOM* with the three modules to resolve the aforementioned problems, respectively.

Pruning Module: To solve the structure mismatch issue, we propose Pruning Module. Before presenting technique details, we first show an example of the structure mismatch in Figure 5. When the on-device model (Figure 5(a)) tries to convert to the debuggable format, some non-debuggable components are not compatible with the debuggable format. As shown in green areas of Figure 5(b), those non-debuggable components need to be processed before connecting with other debuggable components. Therefore, the "extra" part will be produced, but it will confuse debuggable DL libraries because they do not consider this special case in many functions like gradient computing. For example, in Figure 5, the type of weights in `FullyConnected` layer is `uint8`, which cannot be handled by ONNX operators and debuggable model format. To address it, the `uint8` tensor needs to be transformed by the formula: $y = (x - y_0) \times y'$, where the y_0 and y' are stored in the model files and are the zero-point and scale parameters of this `uint8` tensor. After converting to ONNX format, it needs to attach an extra new operator `DequantizeLinear` to achieve the above transformation.

Unfortunately, the TFLite-converted ONNX model with such an extra branch is still not compatible with the debuggable model format like PyTorch. To address this problem, the Pruning Module is proposed to correct the mismatched structure so that it will be compatible with the debuggable format. To remove the mismatched structure, we first find the suspect non-debuggable extra operators using pruning rules. Specifically, we analyze the ONNX operator library to identify the operators that can be used to transform the model weights (*e.g.*, DequantizeLinear, Reshape, Transpose). These operators are potential extra operators. The complete pruning rules

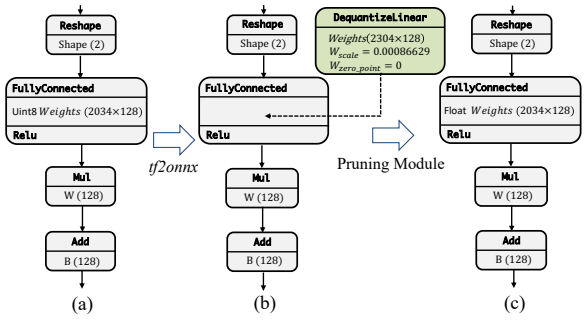

Figure 5: Demonstration of the structure mismatch. (a) The TFLite model. (b)The TFLite-converted ONNX model. (c) The ONNX model modified by our Pruning Module.

list can be found in our code repository. If an operator conforms to our definition of extra operators (*i.e.*, the operator is in the pruning list, uses the fixed tensor data as its input, and produces output data as the next operator's weights), we will remove the extra operator, and compute the corresponding transformed weights for debuggable models, *e.g.*, using the transformation formula of the extra operator $y = (x - y_0) \times y'$, which is shown in Figure 5(c), to remove the non-debuggable branch.

Translation Module: Translation Module is used to solve the operator mismatch issue. Different DL libraries have different specifications (*e.g.*, interface, parameters, and algorithm principle) for the equivalent operators, especially for the on-device DL library and debuggable library. Some on-device operator (*e.g.*, `QuantizeLinear`) is not compatible with the debuggable format because DL libraries do not provide support for those on-device-related operators in their debuggable platform. So we cannot directly map the operator (*i.e.*, mismatched operators) from the on-device form to the debuggable version like existing tools to achieve our purpose.

To address this problem, the Translation Module translates the mismatched operators (*e.g.*,`SpaceToDepth`, `QuantizeLinear` in TFLite→ONNX→PyTorch) into several basic operators that are supported by debuggable formats. For example, the formula of `QuantizeLinear` in TFLite and ONNX can be presented as:

$$y = \delta(\frac{x}{y'} + y_0) \tag{1}$$

where the y_0 and y' are the zero-point and scale parameters of the operator, respectively. Note that, the x/y' is float division. The δ is a saturation parameter that saturates the value to [0, 255], and then converts its data type to `uint8`. To make it debuggable and compatible with PyTorch, the `QuantizeLinear` operator can be

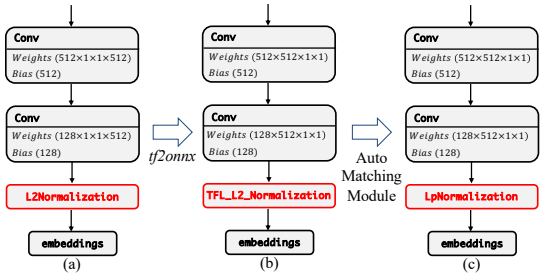

Figure 6: Demonstration of non-supported operators. (a) The TFLite model. (b) The TFLite-converted ONNX model. (c) The ONNX model modified by our Auto-matching Module.

Algorithm 1 Auto-matching

Input: supported debuggable operators list L, input operator o_x, similarity threshold α, random inputs $\mathbf{I}$

Output: syntactic matched debuggable operator o_y

```
If o_x not in L :
    For i in range(lenth(L)) :
        compute the D_ro^i by o_x and L[i] using Equation 3
    end For
    Sort D_ro = [D_ro^1, ..., D_ro^lenth(L)] in descending order
    For D_ro^y in D_ro :
        If ||f_ox(I) − f_oy(I)||_2 ≤ α :
            return o_y
```

divided into two separate operators `Add` (matrix addition) and `Div` (matrix division). Those two operators support the computation of float values. The formula can be shown as:

$$y = \texttt{Clip}(\texttt{Add}(\texttt{Div}(x, y'), y_0)), \tag{2}$$

where the `Clip` clip all elements of input into the range [0, 255] when the target data type is `uint8`. To automatically translate the mismatched operators, we analyze the operator list of the DL platform where we want to process the issue (*e.g.*, ONNX in our study), and identify which operator does not have the matched debuggable operators. Then, we create a translation list that defines the mapping from mismatched operator list L_o^T to the debuggable equivalence list $\widehat{L_o^T}$. Note that we build translation rules for all operators in the translation list. If the Translation Module finds an operator that is in the translation list, it will replace the mismatch operator with the equivalent operator combination.

Auto-matching Module: Auto-matching Module is used to handle the customized or deprecated operators that are not supported by other debuggable model formats. This is because mobile DL developers sometimes use customized implementations on DL libraries to achieve their purpose. Sometimes they don't use the latest version of DL libraries to build their model, some operators are deprecated in the latest versions when we try to transform them. To enhance the exception-handling ability of our method, we introduce the Auto-matching Module. The example (`TFL_L2_NORMALIZATION`) is shown in Figure 6. Unfortunately, PyTorch does not support the `TFL_L2_NORMALIZATION`, hence does not support the transformation of this operator.

To this end, we propose a three-step syntactic matching approach to find an equivalent supported operator to replace the non-supported operator, which is shown in Algorithm 1.

First, when our proposed tool *REOM* finds the non-supported operator (ONNX operators in our tool), which is not in the operator list of other DL model formats, the Auto-matching Module will compute the distance between the supported debuggable operators list L and the non-supported operator. The similarity between the `op_types` of the non-supported operator and the supported debuggable operator can be obtained as follows:

$$D_{ro} = \frac{2K_m}{|S_1| + |S_2|} \tag{3}$$

where the D_{ro} is the similarity metric and $0 \leq D_{\text{ro}} \leq 1$. The S_1 and S_2 are the keyword string of the non-supported operator and the supported debuggable operator, respectively. K_m is the number of matched characters.

Second, we rank the supported operators list L as the distance between the non-supported operator and the supported operator. Then we can find the most similar supported operator o_y with the non-supported operator.

Third, the Auto-matching Module will replace the unsupported operators with the most similar supported operator o_y. Then, it calculates the function similarity between the unsupported operators and o_y by comparing the output difference of original on-device models (TFLite model in our study) and the modified model (ONNX model in our study) with the same inputs $\mathbf{I}$ (the number of inputs is 100 in our experiments). If the l_2 difference $||f_{o_x}(\mathbf{I}) - f_{o_y}(\mathbf{I})||_2$ between the non-supported operator and o_y is smaller than a threshold α (we set it to 0.1 in default), the o_y is the matched operator.

For example, as shown in the (b)→(c) of Figure 6, the customized (non-supported) operator of the TFLite-converted ONNX model (`TFL_L2_NORMALIZATION` in Figure 6 (b)) is converted to equivalent ONNX operator (`LpNormalization` in Figure 6 (c)) with the Auto-matching Module.

4.2 Converting to The Debuggable Model

After modifying the converted ONNX model, it will save the modified model as the new `.onnx` file. Then, we use the *onnx2pytorch* tool to load the structure information and parameter of the ONNX model and assemble them into the Python code using PyTorch API. It is also worth noting here that the generated debuggable model will share the same structure and parameters as the on-device model extracted from real-world Android apps. Consequently, the two models should share the same capabilities. They should also share the same attacking surfaces. In other words, the attack scenarios applicable to the debuggable PyTorch model could be directly applied to attack the on-device TFLite model (indicated via dotted line in Figure 4). We present the experimental results in Section 5.

5 EVALUATION

Towards checking if our research objective is achieved, we propose to answer the following three key research questions.

- **RQ1:** How effective is our approach in achieving automated model transformation?
- **RQ2:** How accurate is the transformation approach?

Table 2: Transformation performance of the proposed *REOM*.

Error Types	Reasons	Count	Success	Fail
`Compatibility`	Structure Mismatch	156	156	0
`NotImplemented`	Operator Mismatch	100	100	0
	Operator Not Supported	24	6	18
`Input Type`	Specification Mismatch	18	18	0

Table 3: Comparison between different α values of Algorithm 1.

Operator Not Supported	0	0.001	0.01	0.1	100
Success Cases	0	0	2	6	24
Fail Cases	24	24	22	18	0

- **RQ3:** Can on-device models be directly attacked via *REOM*-based white-box strategies?

Dataset Construction In the evaluation section, we use the same dataset construction strategy in Section 3.1 and answer the research question using *REOM*.

5.1 RQ1: Effectiveness

In this part, we use all 244 apps (*i.e.*, 244 TFLite models) to fulfill our study. **As shown in Figure 3, among the 244 models, only 16 of them can be successfully transformed into PyTorch models by existing tools. This baseline approach yields a failure rate of 93.4%**, making it impossible to be adopted in practice to achieve our purpose, *i.e.*, automatically transforming TFLite models to debuggable ones.

In contrast, ***REOM* is able to successfully transform 226 of them, giving a success rate of 92.6%**. Note that the 16 models that can be handled by existing tools do not have the non-debuggable component. For those models, the debuggable models produced by our method are the same as the models generated by existing tools. Table 2 further breaks down the detailed results with respect to the three types of issues summarized previously. Note that a given TFLite model may encounter several errors. Hence, the total number of errors (*i.e.*, 289) is slightly larger than the number of TFLite models. All the failure cases are caused by the Operator Not Supported issue, for which the Auto-matching Module cannot find an existing ONNX operator that is equivalent to the customized or deprecated TFLite operator.

Observant readers may have noticed that our approach has taken the parameter α to determine whether the newly generated ONNX operator (because of non-supported TFLite operators) should be accepted in the Auto-matching Module. We now go one step deeper to evaluate the sensitivity of this parameter. As shown in Table 3, when α is set to be 0.1 (the default value), 6 of 24 models with custom non-supported operators can be successfully converted (18 failures). When decreasing this threshold, the failure rate will increase. In the worst case, when α is set to zero, none of the non-supported operator problems can be resolved. However, in another extreme setting, when setting the α to be 100, all the non-supported operators can be resolved, *i.e.*, mapped to newly generated operators that are accepted by the debuggable model format. Subsequently, all the TFlite models can be successfully converted. However, such transformation will not make much sense as the transformed models may not perform the same as their source models. In this work, the default α value 0.1 is set based on our empirical experience, under which the output difference between the original TFLite model and its PyTorch counterpart can be controlled within a distance of 0.1.

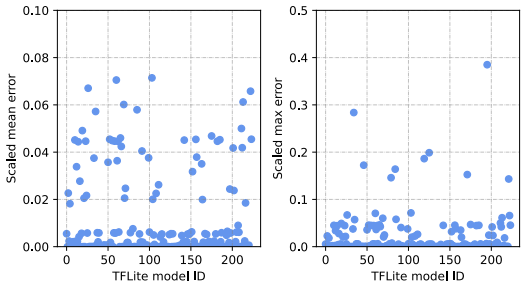

Figure 7: The scaled difference for models from TFLite model to the converted model. The x-axis refers to the ID of the on-device models. Here we plot the transformation difference of all collected models.

Table 4: Demonstration of which operator of the TFLite model will affect the transformation accuracy. We use the (min, max) to define the typical difference range, where the min and max are the minimal and maximal transformation differences for the TFLite model with the specific operator.

Category	Operator	Difference Range
Computing Difference	`DequantizeLinear` `QuantizeLinear`	(0.001, 0.01)
API Difference	`Resize` `Upsample`	(0.001, 0.05)

> **Answer to RQ1:** The proposed *REOM* can successfully transform over 90% of TFLite models to debuggable models. Two out of the three modifier modules have achieved 100% correctness, while our best-effort attempts implemented in the remaining module have also been demonstrated to be useful.

5.2 RQ2: Accuracy

We then compare the output similarity between the transformed models and source models to evaluate the accuracy. Given a pair of models (*i.e.*, a TFLite model and its PyTorch counterpart), with the same inputs, the accuracy of our approach is evaluated based on the similarity of the outputs yielded by the two models. The similar the results are, the higher the accuracy will be. In practice, we use the collected 244 on-device models as the test set. We generate 100 random inputs as the specification of the TFLite model and compare the outputs between the TFLite models and their debuggable PyTorch versions. However, the output ranges of the on-device models are different. To standardize the output difference between the two models, we use the scaled mean transformation difference, which can be calculated as follows:

$$d = \frac{1}{rk}\sum_{i=1}^{k}|\boldsymbol{y_i} - \hat{\boldsymbol{y}}_{\boldsymbol{i}}|, \tag{4}$$

Table 5: Classification accuracy of on-device models and converted debuggable models on test images. Each app has a different test dataset. The dataset list can be found in the shared code repository. The TFLite models are collected from the work [16] and the TensorFlow Hub.

	Models	Fruit	Skin cancer	object	Sign language	Plant	Cassava disease	Plant disease	Insect	Bird
Accuracy	*REOM*	100.00%	80.51%	70.59%	98.71%	95.00%	91.76%	93.20%	96.48%	92.07%
	Source model	100.00%	80.60%	70.62%	98.24%	95.08%	93.12%	93.20%	96.48%	92.50%
	Difference	0.00%	0.09%	0.03%	0.47%	0.08%	1.36%	0.00%	0.00%	0.43%

where d represents the difference between the PyTorch model and the TFLite model. The $\boldsymbol{y_i}$ and $\hat{\boldsymbol{y}}_{\boldsymbol{i}}$ are the outputs of the TFLite model and the converted PyTorch model, respectively. k is the element number of the $\boldsymbol{y_i}$. It means we calculate the average difference for each output data point. For example, if the output is an image, the d is the average difference for each pixel. r is the range of the source on-device model's output. For example, if the data type of output data is `Uint8`, the output range r is $255 - 0 = 255$. However, we cannot know the actual output range of the TFLite model when the data type is `Float32`. To estimate the output range, we use the $r = \max(\hat{\boldsymbol{y}}_{\boldsymbol{i}}) - \min(\hat{\boldsymbol{y}}_{\boldsymbol{i}})$ as the output range, where the max and min are the functions to compute the maximal and minimal value of a vector of matrix, respectively. Therefore, **the estimated difference in our experiments may be lower than the actual difference because the output range in our calculation is smaller than the actual value**. Similarly, to compute the scaled maximal transformation difference, the formula is shown as follows:

$$d = \frac{1}{r}\max(|\boldsymbol{y} - \hat{\boldsymbol{y}}|). \tag{5}$$

The result is highlighted in Figure 7. The transformed PyTorch model generally has a very small difference compared with the TFLite model. **Most cases have a difference of less than 0.001. Some cases have a difference from 0.001 to 0.08**, which is also small. For the cases where the transformation errors are larger than 0.1, the output debuggable model may have a large difference from the source model. It means attackers cannot achieve the white-box attacks based on our method for these cases. However, it will still outperform the black-box attacks used in the existing attacking evaluation studies [15, 16]. In addition, our method will not affect the overall accuracy of models (*cf*. the accuracy difference of converted models and source models in Table 5). **It demonstrates that the converted debuggable models have very similar accuracy to the source on-device models.** The difference exists because the on-device models usually have 8-bit precision but PyTorch debuggable formats only support 16 or 32-bit precision.

Besides, we analyze which operator of the TFLite model will affect the transformation difference. In Table 4. We find two main reasons that can affect the transformation difference. One is the computational difference. Another one is the API difference. For the computational difference, the converted PyTorch model runs on the `float32` data type. However, when the TFLite model has some operators like `DequantizeLinear` and `QuantizeLinear`, it will cause the computational difference between the TFLite model and the converted PyTorch model. For the API difference, some TFLite APIs and PyTorch APIs are fundamentally different. For example, the `Resize` API of TFLite and PyTorch will use a basic `Interpolate` operation. It determines how to compute the value of resized tensors. However, TFLite has more methods to implement the `Interpolate` operation. If the `Interpolate` operation of TFLite layers is not supported by PyTorch, *REOM* will use a substitute `Interpolate` operation to execute the `Resize`. It will cause the output of PyTorch models to be slightly different from the output of TensorFlow models.

> **Answer to RQ2:** The proposed *REOM* approach can achieve high accuracy of the transformation. The performances of the generated PyTorch models are generally very similar to their original on-device TFLite models, which enables security exploitation in the white-box setting.

5.3 RQ3: Supporting White-box Attacks

We evaluate the attacking performance of *REOM* to check whether attackers can directly perform white-box attacks for on-device models. We choose nine TFLite classification models of the work [16] and the TensorFlow Hub to answer this research question. We choose these models because we can find large-scale public datasets (see our code repository) to evaluate the attack success rate to show the effectiveness. For the fruit app, we identify 848 images whose categories correspond to the task scope of models and then use these images as the test set. For other apps, we randomly sample 10000 images from the large-scale datasets as test sets.

Then, we evaluate the attacking performance of the on-device model using the proposed *REOM*. In [4, 16], they focus on generating adversarial attacks by the surrogate model to mislead the target on-device model in the black-box setting. The performance of adversarial attacks generated by the surrogate model relies on the similarity between the target model and the surrogate model. Our study proposes a method for transforming the compiled TFLite model into a debuggable model, eliminating the need to search for or train a surrogate model in order to achieve white-box-like attack performance. Therefore, we will go in-depth into the on-device adversarial attacks and show how our tool can be a general method to evaluate the robustness of the on-device model.

We calculate the attack success rate (*i.e.*, fooling rate) by $p = \frac{n}{m}$, where n and m are the number of successful adversarial examples and the number of images that can be correctly classified by the model, respectively. Note that we only perform attacks on the image which is correctly classified by the target model. n is the number of successful adversarial examples. For **non-targeted attacks**, the attack succeeds when the target model outputs the wrong labels for the inputs. For **targeted attacks**, the attack succeeds when the target model outputs a specific wrong label. Generally, targeted attacks are more difficult to produce than non-targeted attacks.

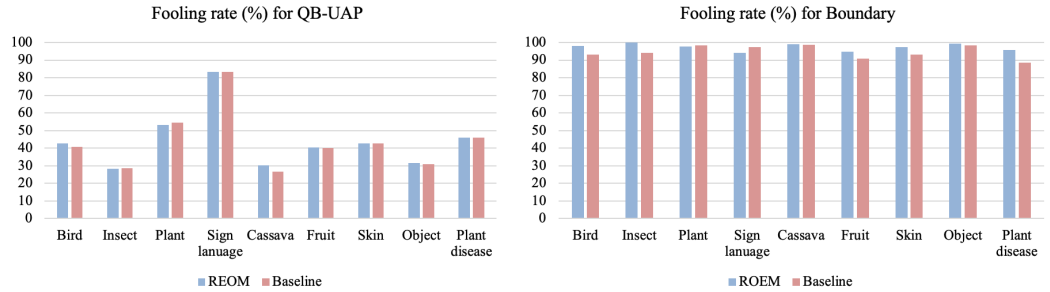

Figure 8: The Fooling rate of the debuggable model converted by *REOM* and the source on-device models using black-box attack methods. It shows that converted models have a similar fooling rate on black-box attacks with source models.

Table 6: Fooling rate (%) of non-targeted and targeted attacks using white-box attack methods. 'ℓ_2': the ℓ_2 distance (perturbation magnitude). 'BIM': Basic Iterative Method [20]. 'PGD': Projected Gradient Descent for generating attacks [26]. '*REOM*': we generate attacks on the converted PyTorch model and transfer the adversarial example to attack the target on-device model. 'Baseline': we collect the pre-trained model like the works [15, 16] and fine-tune the model on our collected dataset like the work [4], then transfer the attacks to attack the target on-device model.

Non-targeted Attack										
		Fruit	Skin cancer	ImageNet	Sign language	Plant	Cassava disease	Plant disease	Insect	Bird
ℓ_2	Attack	BIM|PGD	BIM|PGD	BIM|PGD	BIM|PGD	BIM|PGD	BIM|PGD	BIM|PGD	BIM|PGD	BIM|PGD
0.01	*REOM*	2.90|2.90	1.91|1.91	89.77|89.03	0.60|0.60	72.51|72.31	57.89|57.97	0.75|0.80	41.61|41.82	56.08|56.00
	Baseline	0|0	0|0	4.40|4.56	0|0	2.67|2.64	1.18|1.31	0|0	0.78|0.62	1.32|1.30
0.1	*REOM*	28.12|28.12	20.70|20.58	96.14|96.14	31.89|31.05	80.53|80.40	67.20|67.11	14.75|14.79	53.73|53.77	68.37|67.99
	Baseline	0.22|0.22	0.31|0.31	5.73|5.62	2.98|2.74	19.74|19.59	8.37|8.37	0.36|0.36	1.82|1.86	4.51|4.51
1.0	*REOM*	99.78|99.55	100.00|100.00	99.65|99.65	100.00|100.00	99.96|99.92	99.87|99.87	100.00|100.00	96.31|96.27	99.49|99.49
	Baseline	3.79|4.02	2.67|3.02	10.78|10.23	12.93|12.42	41.42|41.40	30.82|30.86	1.96|1.96	24.32|23.89	27.68 | 27.72
Targeted Attack										
ℓ_2	Attack	BIM|PGD	BIM|PGD	BIM|PGD	BIM|PGD	BIM|PGD	BIM|PGD	BIM|PGD	BIM|PGD	BIM|PGD
0.01	*REOM*	0|0	1.36|1.36	3.97|3.97	0.04|0.08	0.86|0.90	19.09|19.09	0.12|0.12	0.27|0.27	1.02|0.90
	Baseline	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0
0.1	*REOM*	1.61|1.61	10.79|10.79	14.94|15.93	3.63|3.63	3.63|3.67	28.08|28.12	1.09|1.09	1.06|1.13	3.64|3.71
	Baseline	0|0	0.76|0.76	0|0	0|0	0|0	2.04|2.11	0|0	0|0	0|0
1.0	*REOM*	86.18|86.41	99.88|99.88	98.21|98.41	89.74|89.74	96.09|95.90	92.56|92.61	90.12|90.34	72.13|71.89	96.17|96.13
	Baseline	1.32|1.37	4.13|4.15	0.02|0.02	2.95|2.95	3.72|3.68	10.54|10.59	1.27|1.23	2.94|2.88	0.64|0.68

We first demonstrate how is the similarity of the converted model and the source on-device model (*i.e.*, baseline) on black-box attacks, which is shown in Figure 8. We use two different black-box attack methods, one individual method Boundary [3] that needs to generate different attack perturbations for each input and one universal method QB-UAP [38] that produces a universal perturbation for all inputs. They do not need gradient information of the attacked model to evaluate the robustness of models. Here we set the ℓ_2 attack distance to 15. The hyper-parameters of the attack method in our experiments are the same as the parameters in the original paper. **We find the converted model has a similar black-box attack performance to the source on-device model.**

Then, we use the method proposed by Huang *et al.* [16] to collect similar pre-trained models with the target mobile models from the TensorFlow Hub based on the structure and weights similarity, and then fine-tuned them on the training set as the surrogate model. We choose the attacks generated by the surrogate model as the baseline. For *REOM*, we transform the TFLite model into the PyTorch model as the surrogate model. Then, we compare the fooling rate between our method and the baseline. We use two well-known white-box attack methods, BIM [20] and PGD [26], to generate the attack. They are the most common methods used in the robustness evaluation for DL models [42]. We set the number of iterations to around 500. The step size of each iteration is set to around 0.0001, 0.001, and 0.05 for the perturbation distance ℓ_2 are 0.01, 0.1, and 1.0, respectively.

As the results are shown in Table 6, if we use the proposed tool to get the converted PyTorch model as the surrogate, the attack performance will significantly increase compared with that leveraging conventional transfer attacks. The debuggable models can indeed support stronger attacks. Compared with the baseline, **attackers can achieve higher attack success rates (10.23%→89.03% in ImageNet apps) with a hundred times smaller attack perturbations (1.0→0.01) based on the proposed *REOM* framework.** The visualization of different perturbation distances is shown in Figure 9. REOM-based attacks can achieve high attack performances using small perturbations that are imperceptible to humans. These results show that the converted model can be considered the debuggable version of the source model for security exploitation.

Figure 9: The visualization of different perturbation distances for the image classification model.

Answer to RQ3: The proposed *REOM* approach is indeed useful for helping security analysts evaluate the security of on-device TFLite models. Experimental results demonstrate the converted model can be considered as the debuggable version of the source model for security exploitation. On-device models can indeed be directly attacked via REOM-based white-box strategies.

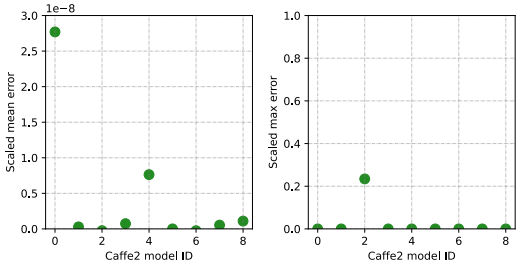

Figure 10: The scaled difference between the Caffe2 models and the converted debuggable model. Our tool can transform 9 of 10 Caffe2 models. The x-axis refers to the ID (0-8) of the Caffe2 models.

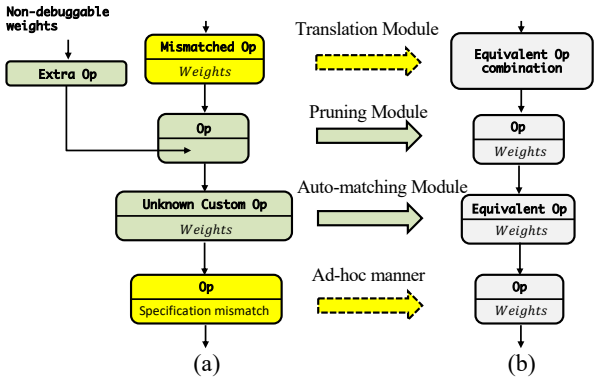

Figure 11: The meta-model of how our *REOM* solves the four problems. (a) general representation of non-debuggable models. (b) general representation of transformed debuggable models.

6 DISCUSSION

In this section, we will discuss the genericity, other properties of our method, and potential defense strategies.

Generalizability of Our Approach: Although our method is designed for the most commonly used on-device model format TFLite, our approach should also work for other formats because the *Modifier* handles the non-debuggable component in the ONNX level, which has a unified model representation. we believe our approach should also work for transforming other on-device formats like Caffe2 to the debuggable model format. To experimentally validate this, we collect 10 Caffe2 models from the Caffe2 model zoo [34] to evaluate the effectiveness of *REOM* on the other on-device format. By default, only two out of the ten models can be successfully handled by the existing toolchain. We then integrate our approach into the process by applying our *Modifier* to automatically modify the intermediate ONNX model generated by the built-in conversion tool of *Caffe2*. Now, 9 of 10 *Caffe2* models can be transformed into PyTorch models. All the converted models have a scaled difference less than $3 \times 10^{-}8$, as shown in Figure 10. It shows *REOM* is indeed generic and can handle multiple on-device formats. However, if the on-device model cannot be converted to the ONNX model, our method does not work. As for our future work, we commit to evaluating and enhancing the generalizability of our approach with more on-device models.

Although our method is generic, some parts (*e.g.*, translation rule list) of our approaches may need to be updated frequently to support the future versions of DL model formats or adapt to other model formats. As shown in Figure 11, The green arrow means this problem exists because of fundamental differences between the non-debuggable models and debuggable models, *i.e.*, having a clear definition and can be solved by a unified solution. The dotted yellow arrow means this problem exists because of occasional differences between the non-debuggable models and debuggable models, *e.g.*, the solution may be various in different versions of DL libraries. For example, some operators are debuggable in this version but may be non-debuggable in the next version.

In addition, we use adversarial attacks to evaluate the effectiveness of our method. The process of producing other kinds of attacks [12, 37] based on the proposed *REOM* is similar. Attackers need to first get the debuggable model from the on-device model and then generate other attacks. However, the attack generation for different kinds of attacks may be different. For example, model inversion attacks [12] use the debuggable model to find the input that can produce the same output as the source on-device model.

On Increasing the Attack Surface of on-device models: Existing studies [15, 23, 36] for evaluating the robustness of on-device models cannot access them as white-box ones. We conduct this study to explore the real risks of DL models on devices. It indicates attackers can fully access the on-device model through reverse engineering for most real-world Apps. Our experimental results demonstrate that the converted debuggable version of on-device models can indeed have a similar prediction performance compared with the original model. This result strongly supports our hypothesis that it is indeed possible to conduct direct white-box attacks for target on-device models. More importantly, the Direct white-box method using *REOM* can significantly increase the attacking performance, and achieve higher attack success rates (10.23%→89.03%) with a hundred times smaller attack perturbations (1.0→0.01). So, existing studies [15, 23, 36] for evaluating the robustness of on-device models usually miss the fact that attackers can bridge the gap by reverse engineering. Our paper leverages empirical software engineering methods to reveal the real risk of on-device models, which is underestimated by the existing studies.

Enabling the white-box testing: Our study enables direct testing on deployed DL models like TFLite models. Although there are many white-box testing methods to evaluate the DL models [25, 33], these methods are designed for debuggable DL models. However, the deployed model may not be debuggable (or differentiable), such as the TFLite model. Existing white-box testing strategies, which are more efficient than black-box ones, cannot be directly applied to the on-device models. Therefore, our contribution lies in enabling direct white-box testing of compiled DL models.

Potential defense strategies: The observation from the evaluation of our method shows reverse engineering the on-device model relies on the effectiveness of transformation rules. At the moment, The proposed REOM can successfully transform over 90% of TFLite models into debuggable models as almost all on-device operators (226 cases) can be reverse-engineered into debuggable ones. There are indeed some corner cases for which *REOM* fails because of customized operators (18 cases). Except for this, we believe there might be more options for defending against direct white-box attacks [24]. One possible approach developers could consider is to split models, e.g., by defining multiple sub-models in training and then compiling them into different model files. Attackers need to understand the source code (the code in apps is usually obfuscated) to know how to assemble them. However, this method may increase the inference time of on-device models because it needs to load and parse the model twice. Another approach could be to implement model obfuscation [40], e.g., by replacing the keyword of the `conv2d` operator to a random string, and build a compatible customized TFLite library. However, this method may increase memory and time consumption because it needs to parse the obfuscated information.

7 THREATS TO VALIDITY

We now discuss the potential threats to the validity of this work.

First, our proposed conversion tool *REOM* is based on the ONNX platform, and we evaluate its performance on TFLite models and caffe2 models. However, some on-device model formats may have a higher level of security (*e.g.*, do not use high-level representations like TVM models), which may disable the model parsing based on the operator-to-operator transformation rule list, including our approach proposed in this work. In this case, those on-device models are safe. However, reverse engineering methods may overcome this problem by modifying the conversion rules, *e.g.*, building a mapping list from ONNX operators to TVM model representations.

Second, the development of the DL library is in rapid change. It may affect the performance of reverse engineering when the library updates the model format or conducts other major evolutionary changes. In such a case, the reverse engineering tool may fail to convert on-device models to debuggable versions. Therefore, we argue that there is a strong need for our approach to be aware of the evolution of given DL frameworks.

8 CONCLUSION

This study evaluates the importance of developing a reverse engineering tool that can transform the TFLite model into the debuggable PyTorch model. Such transformation can enable attackers to perform direct white-box attacks for evaluating the vulnerability of on-device models. To achieve this, we propose a *REOM* framework to transform the on-device model into the PyTorch model. Our proposed *REOM* has three steps: (1) first, we use the *tf2onnx* tool to convert the TFLite to the ONNX model. (2) Second, we propose a three-module modifier, which has Pruning Module, Translation Module, and Auto-matching Module. It can modify the ONNX model to make it compatible with the debuggable PyTorch format. (3) Finally, the modified ONNX model can be successfully transformed into the PyTorch model by the *onnx2pytorch* tool. Experiments show the *REOM* can effectively transform most TFLite models to PyTorch models, with small transformation differences compared with the original TFLite model. Then, we test our method on adversarial attacks and find that on-device models can be directly attacked via white-box strategies. The current model deployment strategy is at serious risk. it enables attackers to perform white-box attacks on on-device models. In future works, we will comprehensively analyze the security and privacy issues of on-device models using the proposed *REOM*.

9 DATA AVAILABILITY

We provide an archival repository of our artifact by Software Heritage, which links to our Github repository. We also provide a Docker Image for users to reproduce our results.

ACKNOWLEDGMENTS

This work is partially supported by the Open Foundation of Yunnan Key Laboratory of Software Engineering under Grant No.2023SE102, by the National Natural Science Foundation of China under Grant No.62202026 and No.62172214, and by Guangxi Collaborative Innovation Center of Multi-source Information Integration and Intelligent Processing.

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] Wieland Brendel, Jonas Rauber, and Matthias Bethge. 2017. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. *arXiv preprint arXiv:1712.04248* (2017).

[4] Hongchen Cao, Shuai Li, Yuming Zhou, Ming Fan, Xuejiao Zhao, and Yutian Tang. 2021. Towards Black-box Attacks on Deep Learning Apps. *arXiv preprint arXiv:2107.12732* (2021).

[5] Jianbo Chen, Michael I. Jordan, and Martin J. Wainwright. 2019. HopSkipJumpAttack: A Query-Efficient Decision-Based Attack. arXiv:1904.02144

[6] Jianbo Chen, Michael I Jordan, and Martin J Wainwright. 2020. Hopskipjumpattack: A query-efficient decision-based attack. In *2020 ieee symposium on security and privacy (sp)*. IEEE, 1277–1294.

[7] Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh. 2017. Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*. ACM, 15–26.

[8] Minhao Cheng, Thong Le, Pin-Yu Chen, Jinfeng Yi, Huan Zhang, and Cho-Jui Hsieh. 2018. Query-efficient hard-label black-box attack: An optimization-based approach. *arXiv preprint arXiv:1807.04457* (2018).

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

[10] Francesco Croce and Matthias Hein. 2020. Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks. In *International conference on machine learning*. PMLR, 2206–2216.

[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.

[12] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. 2015. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. 1322–1333.

[13] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and harnessing adversarial examples. In *ICLR*.

[14] Chuan Guo, Jacob Gardner, Yurong You, Andrew Gordon Wilson, and Kilian Weinberger. 2019. Simple Black-box Adversarial Attacks. In *International Conference on Machine Learning*. 2484–2493.

[15] 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.

[16] 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.

[17] Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. 2018. Black-box Adversarial Attacks with Limited Queries and Information. In *ICML*. 2142–2151.

[18] Andrew Ilyas, Logan Engstrom, and Aleksander Madry. 2018. Prior convictions: Black-box adversarial attacks with bandits and priors. *arXiv preprint arXiv:1807.07978* (2018).

[19] 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.

[20] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. 2017. Adversarial examples in the physical world. *International Conference on Learning Representations (ICLR)* (2017).

[21] Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé F Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. 2017. Androzoo++: Collecting millions of android apps and their metadata for the research community. *arXiv preprint arXiv:1709.05281* (2017).

[22] Li Li, Xiang Gao, Hailong Sun, Chunming Hu, Xiaoyu Sun, Haoyu Wang, Haipeng Cai, Ting Su, Xiapu Luo, Tegawendé F Bissyandé, et al. 2023. Software Engineering for OpenHarmony: A Research Roadmap. *arXiv preprint arXiv:2311.01311* (2023).

[23] 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.

[24] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. 2022. Deep learning for android malware defenses: a systematic literature review. *Comput. Surveys* 55, 8 (2022), 1–36.

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

[26] 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

[27] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. 2017. Universal adversarial perturbations. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1765–1773.

[28] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. 2016. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2574–2582.

[29] Konda Reddy Mopuri, Phani Krishna Uppala, and R Venkatesh Babu. 2018. Ask, acquire, and attack: Data-free uap generation using class impressions. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 19–34.

[30] 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.

[31] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. 2016. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 372–387.

[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] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*. 1–18.

[34] Orion Reblitz-Richardson, Lu Fang, Bram Wasti, and Aaron Markham. 2019. Caffe2 Model Zoo. https://github.com/facebookarchive/models.

[35] Ishai Rosenberg, Asaf Shabtai, Yuval Elovici, and Lior Rokach. 2021. Adversarial machine learning attacks and defense methods in the cyber security domain. *ACM Computing Surveys (CSUR)* 54, 5 (2021), 1–36.

[36] Ye Sang, Yujin Huang, Shuo Huang, and Helei Cui. 2023. Beyond the Model: Data Pre-processing Attack to Deep Learning Models in Android Apps. *arXiv preprint arXiv:2305.03963* (2023).

[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.

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

[39] 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.

[40] 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

[41] 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.

[42] Shuai Zhou, Chi Liu, Dayong Ye, Tianqing Zhu, Wanlei Zhou, and Philip S Yu. 2022. Adversarial attacks and defenses in deep learning: From a perspective of cybersecurity. *Comput. Surveys* 55, 8 (2022), 1–39.