Model-less Is the Best Model: Generating Pure Code Implementations to Replace On-Device DL Models

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

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

Pei Liu
CSIRO's Data61
Clayton, Australia
Pei.Liu@data61.csiro.au

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

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

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

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

ABSTRACT

Recent studies show that on-device deployed deep learning (DL) models, such as those of Tensor Flow Lite (TFLite), can be easily extracted from real-world applications and devices by attackers to generate many kinds of adversarial and other attacks. Although securing deployed on-device DL models has gained increasing attention, no existing methods can fully prevent these attacks. Traditional software protection techniques have been widely explored. If on-device models can be implemented using pure code, such as C++, it will open the possibility of reusing existing robust software protection techniques. However, due to the complexity of DL models, there is no automatic method that can translate DL models to pure code. To fill this gap, we propose a novel method, *CustomDLCoder*, to automatically extract on-device DL model information and synthesize a customized executable program for a wide range of DL models. *CustomDLCoder* first parses the DL model, extracts its backend computing codes, configures the extracted codes, and then generates a customized program to implement and deploy the DL model without explicit model representation. The synthesized program hides model information for DL deployment environments since it does not need to retain explicit model representation, preventing many attacks on the DL model. In addition, it improves ML performance because the customized code removes model parsing and preprocessing steps and only retains the data computing process. Our experimental results show that *CustomDLCoder* improves model security by disabling on-device model sniffing. Compared with the original on-device platform (*i.e.*, TFLite), our method can accelerate model inference by **21.8% and 24.3%** on x86-64 and ARM64 platforms, respectively. Most importantly, it can significantly reduce memory consumption by **68.8% and 36.0%** on x86-64 and ARM64 platforms, respectively.

CCS CONCEPTS

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

KEYWORDS

SE for AI, AI safety, software optimization for AI deployment

ACM Reference Format:
Mingyi Zhou, Xiang Gao, Pei Liu, John Grundy, Chunyang Chen, Xiao Chen, and Li Li. 2024. Model-less Is the Best Model: Generating Pure Code Implementations to Replace On-Device DL Models. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3652119

1 INTRODUCTION

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

However, as such DL models are directly hosted on devices, attackers can easily unpack the mobile Apps, identify DL models through keyword searching, and then extract key information from the DL models. This accessible model key information thus makes it easy to launch attacks or steal the model's intellectual property [43]. To protect on-device models, the most commonly used on-device DL

*Corresponding author.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3652119

model framework, TFLite, converts the general DL model such as TensorFlow and PyTorch models to TFLite models, which disables direct white-box attacks. This is done by disabling the gradient calculation of the on-device models, which is essential for conducting effective white-box attacks. Such models are called non-differential models (*i.e.*, non-debuggable models). Other on-device platforms such as TVM [6] have similar processes. TVM compiles a high-level DL model representation into low-level representations that can be applied to various hardware platforms. It also supports packing such representation into the API library. The low-level information makes it hard for attackers to reverse engineer the on-device model.

However, these on-device platforms still suffer from significant security risks. Recent attack methods [5, 16, 22] can parse model information in the on-device model (*e.g.*, `.tflite`) files or the compiled low-level representation (*e.g.*, from TVM), then reverse engineer them or search for a similar debuggable one. Recent defence approaches propose to obfuscate the information of on-device models [42]. The information inside model files (*e.g.*,.tflite files) is protected using several obfuscation strategies. However, the obfuscated model representation is still directly exposed to threats. Even if the obfuscated information is secured, the sniffing methods [39] can still locate the on-device model and generate black-box attacks [14, 26, 44], which is similar to the attacks for cloud models. **Therefore, as the DL model is easily attacked, the current mainstream model deployment strategy that employs explicit model information is a serious risk for mobile Apps and devices.** In addition, model obfuscation will introduce inevitable overheads (*e.g.*, up to 20% in RAM consumption) as it requires parsing the obfuscated APIs during model inference and injecting extra layers to achieve high obfuscation performance.

Parsing the model file or identifying the DL component is usually required for the initial phase of attacking a DL model. we aim to explore whether we can hide DL components in their on-device deployment environment without introducing overheads to model inference. One approach is to deploy the DL model as a pure code implementation (*e.g.*, C/C++ code), eliminating the need for an explicit model representation that can be easily located and parsed. It is also more efficient than the common model deployment strategies that deploy the library and explicit model representation separately. For instance, *m2cgen* [1] implements different ML algorithms as pure code by translating some ML algorithms to their corresponding pure code implementations. However, this strategy cannot be extended to DL techniques which have diverse architectures and frequently evolving algorithms. Another alternative involves creating pure code implementations for specific DL algorithms such as *llama.cpp* [3]. However, this approach necessitates substantial manual effort for each DL model. Automatically generating code for complex ML algorithms is a difficult research problem. **Therefore, it is still required to design an automatic method that can translate the model inference to pure code implementations for various DL algorithms.**

In this study, we want to answer the following research question: **Can we extract the essential codes from the DL API library and refactor them into an executable program for diverse DL algorithms?** To this end, we propose a novel solution, *CustomDLCoder*, to extract the computing code unit from the DL library and refactor them in an executable program, whose overall architecture is shown in Figure 1. Given a trained on-device DL model and its underlying on-device DL library (TFLite), *CustomDLCoder* produces an executable C++ program that can be deployed on devices, as shown in the right part of Figure 1. *CustomDLCoder* removes the explicit model representations, *i.e.*, computational graph and weights, as illustrated in the left part of Figure 1, hence avoiding the need of model representations for model inference. To achieve this goal, *CustomDLCoder* first parses the DL model, extracts its related computing codes from the TFLite library, configures the extracted codes, and then generates an executable program for model inference. Our method will not affect the model's performance because it has the same computing process as the original model inference. Experiments on 11 representatives on-device DL models (*e.g.*, MobileNet [15], SSD [24], GPT2 [28]) show that *CustomDLCoder* achieves a higher level of security compared to existing on-device protection methods without any overhead. In addition, the program generated by our method only contains the essential computing process for each model, by removing generic DL library steps involved in analyzing the computational graph and its parameters. This results in accelerating model inference (**by 21.8% on x86-64 and 24.3% on ARM64**) compared to existing deployment strategies, and reducing memory consumption (**by 68.8% on x86-64 and 36.0% on ARM64**).

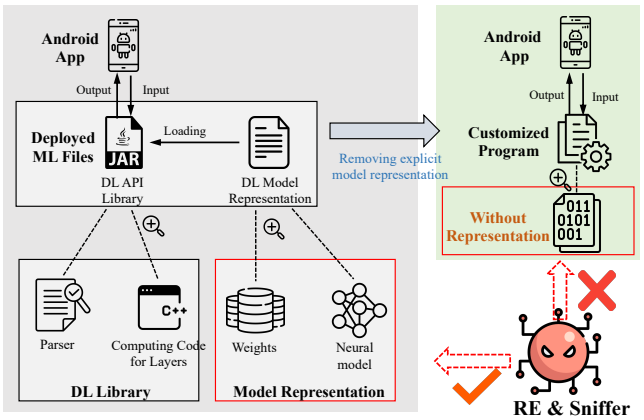

Figure 1: The high-level idea of generating pure code to replace DL model representations. The red block shows the difference between the deployed DL components.

The key contributions of this work include:

- We propose a novel solution that can automatically extract the related backend code of specific models and refactor them to executable programs to address the challenge in existing methods;
- Our method can automatically translate various DL algorithms to pure codes, resulting in a higher level of security;
- Our methods can accelerate model inference (**by 21.8% on x86-64 and 24.3% on ARM64**) compared to existing deployment strategies and reduce memory consumption (**by 68.8% on x86-64 and 36.0% on ARM64**).
- We open-sourced our prototype tool *CustomDLCoder* by an GitHub repository: https://github.com/zhoumingyi/CustomDLCoder.

2 BACKGROUND AND RELATED WORKDS

2.1 DL Frameworks

Deep Learning (DL) Frameworks: The open-source community has developed many well-known **DL frameworks** to facilitate users to develop DL models, such as TensorFlow [4], Keras [9], and PyTorch [27]. These frameworks provide standards for developing DL models [12]. PyTorch is one of the latest DL frameworks which has gained academic user popularity for its easy-to-use and high performance. In contrast, TensorFlow is widely used by industry to develop new DL-based systems because it has the most commonly used on-device DL library, Tensor Flow Life (TFLite). TFLite is the most popular library for DL models on smartphones, as it supports various hardware platforms and operation systems.

DL Deployment Strategy: As the training of DL models is intensive in both data and computing, mobile developers often collect data and train their models on the cloud or high-end desktop server prior to App deployment. Developers also need to compile the trained models to be compatible with specific devices (*i.e.*, to produce on-device models) so as to speed up model inference on mobile CPU/GPUs [7, 8, 25]. Developers use a tool (*e.g.*, *TFLite Converter* in TensorFlow) integrated into their DL framework to compile their DL model to an on-device model. It will produce a compiled model that contains model architecture, weights and API library. At App installation time, the compiled models and libraries 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 related APIs in their DL libraries.

2.2 On-device DL Frameworks

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

Traditionally, on-device models are released as **DL files** that are deployed on devices. Mobile app code then accesses these models through a dedicated **DL library**, such as the TFLite library if the AI model is developed using the TFLite framework. Each model file contains two types of information: **computational graph** and **weights**, which record the model's architecture and parameters tuned based on the training dataset, respectively. Such a computational graph is usually a multi-layer neural network. In the network, each layer contains an **operator** that accepts **inputs** (i.e., the outputs of the previous operator), **weights** (i.e., stored in the dedicated file that is pre-calculated in the training phase), and **parameters** (i.e., configuration of the operator. For example, the `conv2d` layer in TFLite requires the parameters of stride size and padding type. Their parameter will affect the outputs of layers.) to conduct the neural computation and outputs the results for the next operator.

As shown in Figure 1, the traditional way of deploying DL models has to put DL model information directly on devices. The DL framework stores the model representations including computational graph and weights in one file (*e.g.*, `.tflite` file for TFLite) or packs them into the library when AI compilers such as TVM [6] are involved). However, these explicit model representations may be extracted and exploited by attackers [5, 16, 22], resulting in security threats to device users.

TFLite models run on the FlatBuffers Platform[2], which is efficient for loading the model and running it using multiple programming languages. It can access serialized data without parsing/unpacking and only needs small computational resources. TFLite uses the `.schema` file to define the data structures. For parsing the model structure and weights from the `.tflite` file, users can use the `.schema` file[3] of TFLite to parse the information on FlatBuffers level and get the JSON file that has detailed information of the `.tflite` model file.

Compared with TFLite, TVM uses low-level representations (such as assembly code) to build the model, and it can then pack the representation into the library. However, those low-level representations still can be parsed by attackers [5]. In addition, the performance of AI compilers like TVM relies on conversion accuracy. However, the conversion from high-level representation to low-level representation usually uses human-defined rules, which are often not stable due to the rapid change in DL frameworks. For instance, developers may have results inconsistency and conversion failure problems in TVM.

2.3 DL Model Attacks

DL models deployed on devices are subject to a range of attacks [16, 22, 26, 37, 40, 44]. These can include tricking the DL model with perturbed inputs into e.g. classifying an image incorrectly; extracting model information to facilitate other attacks; stealing a copy of the model (which may have been very expensive to produce) for use in one's own application; and others. These attacks can be black-box [16, 22, 38] or white-box [40]. Access to DL components and/or access to DL models facilitates these attacks.

2.4 Code and Model Obfuscation

Code Obfuscation: Code obfuscation methods were initially developed for hiding the functionality of malware. The software industry also uses it against reverse engineering attacks to protect code IP [30]. Code obfuscators provide complex obfuscating algorithms for programs like JAVA code [10, 11], including robust methods for high-level languages [33] and machine code level [36] obfuscation. Code obfuscation is a well-developed technique to secure the source code. However, traditional code obfuscation approaches are hard to use to protect on-device models, especially for protecting the structure of DL models and their parameters.

Model Obfuscation: To prevent attackers from obtaining detailed information of deployed DL models, model obfuscation has been proposed. This obfuscates key model information such as the semantic information of model components and model architectures [42]. It is then hard for attackers to obtain the precise trained weights and structure of models from the obfuscated model because the connection between the obfuscated information and the original one is randomly generated. However, this method will introduce inevitable overhead to the model inference. Besides, this method

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

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

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

cannot defend against black-box attacks for ML models, which are also effective in attacking the models, as they do not need the inner information of models.

2.5 Customized DL Programs

Our core idea is to replace DL model representation and generic libraries with pure code implementations (e.g., C++ Code, as shown in Figure 1). This code implementation, we refer to as a **customized DL program**. **Computing code units** define core inference processes for a DL operator. Note that one operator may have multiple computing code units.

We aim to hide the on-device DL representation of an app or device by extracting a DL model computing process into a customized DL program. We generate customized C++ code implementations for the DL model inference that does not have explicit model information (*i.e.*, model, graph, and parameter files), and the model file or explicit model representation is unneeded in deployment. Because the generated code is a pure C++ program, we can use code obfuscation techniques to obfuscate the compiled program to only retain the computing process of the computational graph. This new customized, obfuscated DL program achieves the same objectives as of traditional DL representations + DL library approach. However, it does not need the DL library anymore, and no model files or other representations need to be stored on insecure devices. So, the DL functions are retained but deployed in a more secure manner. In addition, as our generated code keeps the essential computing process and removes other model parsing steps, it can accelerate model inference and reduce memory consumption.

3 *CUSTOMDLCODER* APPROACH

We want to generate pure code implementations of DL inference to replace on-device DL models and libraries. This should both improve their security to attacks and significantly improve their run-time efficiency. Although we can locate the code file for each DL operator easily, this study still has two main technical challenges. (1) First, it is hard to generate customized code implementations for DL models without efficiency loss. This is because the DL library contains various APIs for the same operator and numerous data preprocessing APIs to optimize the inference on different data types and devices. It is hard to exactly extract the optimal and minimized codes for each DL operator. The redundant extracted codes and redundant execution path will cause efficiency loss after converting the original DL model to the customized program. (2) Second, for removing redundant code, we only extract small pieces of code for each operator. That means we will extract a large number of small code fragments for large DL models (*e.g.*, large language model). Besides, for the sake of code performance, the extracted code is highly condensed and often incomplete. It is difficult to configure and assemble complex code pieces into an executable program using automated methods.

To solve the aforementioned problems, we argue we can trace the execution process of DL models in its related API library, extract the related small code units, and refactor them according to their original execution path and data structure. In addition, we propose a dynamic configuration algorithm to configure and assemble the incomplete small code pieces automatically. The overview of our proposed method is shown in Figure 2. It has four main steps: (1) Model Parsing, (2) Computing Unit Extraction, (3) Configuring Data Analysis, and (4) Dynamic Configuration. We will detail our proposed method in the following subsections.

3.1 *CustomDLCoder* Overview

Given a DL model $\mathcal{G}(x)$ and the corresponding DL library, our goal is to extract the complete computing process of the library using the DL model information (*i.e.*, computational graph and weights) and compile it to a customized DL program $C(x)$. Formally the procedure can be denoted as:

$$C(x) = \mathcal{G}(x) \qquad \forall x \in \mathcal{X}, \tag{1}$$

For any inputs x within the input range $\mathcal{X}$ of the task, the compiled executable program will output the same result as the original DL model using its target DL library. Moreover, the extracted program contains the same computing process of $\mathcal{G}$ (*i.e.*, the data flow from the input to the prediction) to maintain the model accuracy but simplify the model parsing and data processing. It thus will run faster than the original model inference using the DL library.

The key objectives of *CustomDLCoder* are to automatically identify the backend computing unit related to the operator in the given computational graph, refactor the extracted code, and then compile the refactored code to an executable program. As the generated code is a pure C++-based program, it can be further obfuscated by existing code obfuscation tools to remove any remaining semantic representations of the computational graph (*e.g.*, function names).

CustomDLCoder carries out the following four steps: 1) *Model Parsing*: This parses the information contained in a DL model. It will analyze two main components, operators and parameters, of the DL computational graph. 2) *Computing Unit Extraction*: This will use the parsed information to identify the computing code unit in the DL library for each operator. These computing functions will be collected into a computing unit set. 3) *Configuring Data Analysis*: After obtaining the computing unit set, we use the parsed parameter information of each operator to configure the data needed for each corresponding computing unit. 4) *Dynamic Configuration*: Finally, this will refactor the collected computing functions and input data and assemble them into a complete C++ program according to the provided DL computational graph. For configurations that cannot be determined by given parameter information, which needs to be determined by the status of intermediate data or device, this module will further search for their configuration. Our proposed code extraction scheme will not affect the model performance and efficiency because the produced code program uses the same computing code to get the output.

3.2 *Model Parsing*

The purpose of this step is to parse the information of DL models so that we can automatically extract the computing process of the given DL model. The DL model file contains two kinds of information: operators and weights tensors. In this module, we first use `Flatbuffer` to reverse engineer the TFLite model file to a JSON file. This contains the high-level representation of the DL model, which is shown in the Parsed Information of Figure 2. Note that the weights tensor of the model file will not be extracted in this

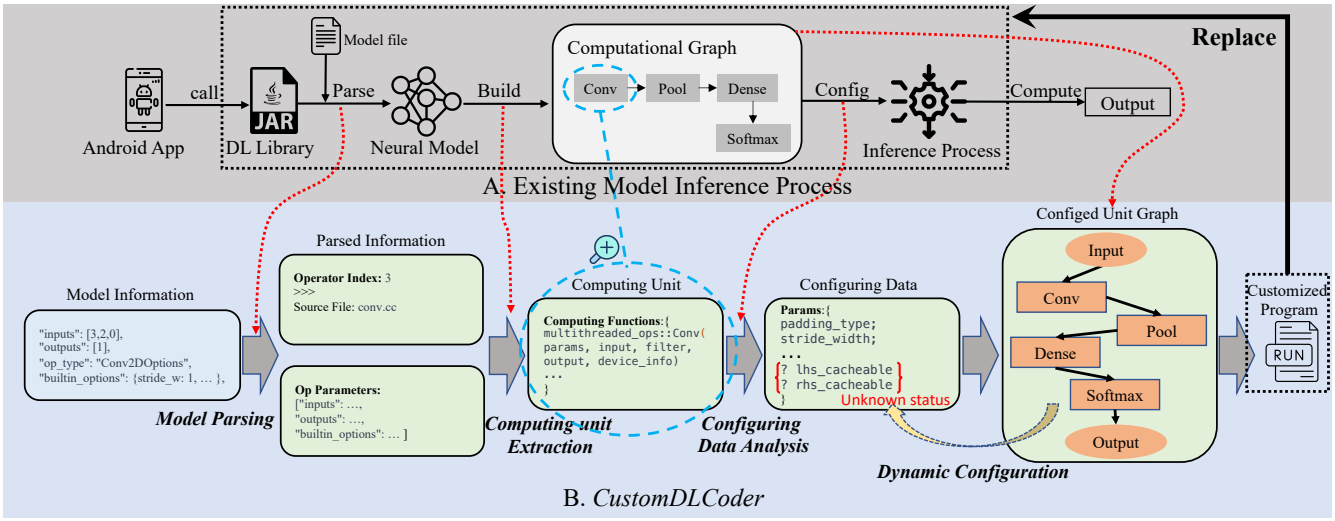

Figure 2: Design Overview of *CustomDLCoder*. The generated program is collected and confined by analyzing the inference process of the original TFLite library.

step. After obtaining this high-level representation, we then use the operator index (that can be obtained from the operator information) to locate the source code for each operator. As shown in Listing 1, TFLite will register all operators in the 'register.cc' file to assign the operator code (*e.g.*, BuiltinOperator_CONV_2D = 3) to TFLite operators or assign a name (*e.g.*, Mfcc) to custom operators as the operator index. Note that the custom operators are implemented by users. The computing code for each operator will be implemented in the Register_{op_name}. Thus, the TFLite interpreter can parse the operators in the computational graph by such operator index. We use the operator index extracted from the model file to collect the source code of operators that will be used in the model inference. For example, the operator code of Conv2D operator in TFLite is 3 (i.e., line 3). We then use this operator code 3 to identify the corresponding operator registration Register_CONV_2D(). After that, we locate the source code, in the conv.cc file. For custom operators, we will use the operator name (*e.g.*, Mfcc) to identify the registration. For example, the custom operator with the name Mfcc will use "Mfcc" as the keyword to search the registration in the 'register.cc' file.

```
enum BuiltinOperator {
 // Operator code
 BuiltinOperator_CONV_2D = 3,
  ...
}

BuiltinOpResolver::BuiltinOpResolver(){
 // The registration for Conv2D operator
 AddBuiltin(BuiltinOperator_CONV_2D, Register_CONV_2D());
  ...
 // The registration for custom operator
 AddCustom("Mfcc", tflite::ops::custom::Register_MFCC());
}
```

Listing 1: The registration of operators in TFLite. The Conv2D is TFLite operator and the Mfcc is a custom operator implemented by users.

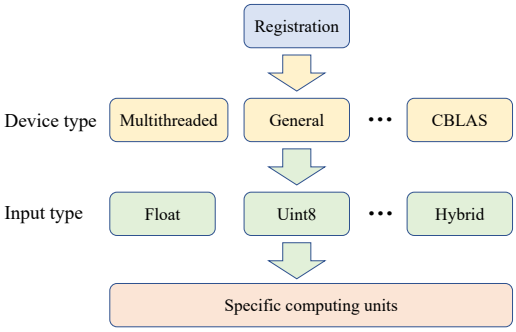

Figure 3: Structure of the operator source code. TFLite implements different code units for computing the output in different situations. It will parse the device and input information to choose the computing unit.

3.3 *Computing Unit Extraction*

After we collect the source code for each operator used in the DL model inference, we have all the backend computing code from the model input to the model output. However, the operator source code needs to support different hardware platforms and different data types (*e.g.*, float, uint8). For a given TFLite model, the model inference only uses part of the collected codes. We can remove the unneeded part of the collected code to improve the code efficiency.

The function structure in an operator is shown in Figure 3. The operator registration will first parse the device information to choose a specific registration entity. Each operator usually has several registration entities including a multithread-optimized entity, a general entity, and other entities that can be executed on special hardware platforms. Each entity has four functions: Init, Free, Prepare, and Eval. The TFLite interpreter will use Init and Free to initialize the data and delete the data in memory. When the TFLite interpreter loads the operator, Prepare function will be used

Algorithm 1 Computing Unit Extraction

Input: Device info d, tensor info t_n, computational graph $\mathcal{G}$, operator's source code $\mathbf{U}_n = \{U_n^{i,j}\}$, where n is the id of operators.
Output: Extracted computing unit collection $\overline{\mathbf{U}}$.
Notation: the n-th operator O_n, data type dt.
1 : Initialize $\overline{\mathbf{U}}$
2 : **For** O_n in $\mathcal{G}$ **do** :
3 : Parse the data type dt in t_n
4 : Extract the source code $\mathbf{U}_n$
5 : $\overline{u}_n = U_n^{dt,d}$
6 : $\overline{\mathbf{U}}$.append($\overline{u}_n$)
7 : **end For**
8 : **Return** $\overline{\mathbf{U}}$

to load and prepare the configurations (*e.g.*, output size) for this operator. `Eval` function computes the output using the given input, operator weights, and operator parameters. In the `Eval` function, it has different implementations for different data types. Thus, the source code of operators can be considered a set of computing code units that support different hardware platforms and different data types, which is shown in Figure 3. Note that the computing code unit in TFLite has data manipulation in the matrix form, not the complex TFLite tensor form.

Therefore, we use computing unit extraction to obtain the computing process of each operator in the computational graph, as shown in Algorithm 1. First, we load the information of operators by parsing the computational graph and obtain the source code of each operator in the *Model Parsing* module. As we mentioned before, the source code of operators can be considered as a set of computing units that support different hardware devices and different data types. We then use the information about the target device and data type to identify the computing unit that will be used in the model inference. Finally, we iterate over all operators in the computational graph, and do the same extraction process above to get the computing unit set $\overline{\mathbf{U}}$. However, the $\overline{\mathbf{U}}$ is not a complete program. These computing units are like puzzle pieces, and we need to assemble these units into complete code implementations of model inference in subsequent modules. This is very hard to solve in normal programs. However, the computing code of DL libraries like TFLite only has a limited kind of configuration, which enable us to use rules to configure the code correctly.

```
void EvalFloat {
  // Create essential data for the computing unit
  TfLiteTensor* input = create_input();
  TfLiteTensor* weights = create_weights();
  ConvParams params = create_params();
  // initialize the output tensor
  TfLiteTensor* output;
  // The called API for this computing unit
  multithreaded::Conv(params, input, weights, output);
}
```

Listing 2: Simplified computing unit of `Conv2d` operator when the data type is float and the hardware device supports the multithread-optimized implementation.

3.4 *Configuring Data Analysis*

Each computing unit has two kinds of data that need to be created to obtain its output: tensor (*e.g.*, the input and weights) and parameter. These data can be referred to as the configuration for the computing unit. As we extract the separate computing unit from each operator, we need to create such data to execute the DL model inference. We use the `EvalFloat` unit in `conv2d` operator as an example, as illustrated in Listing 2.

In TFLite, the tensor and parameter data are designed as `structs` in C++. Our *CustomDLCoder*'s *Model Parsing* module will parse the operator information and tensor information. The weights tensor is easy to create, we just need to use the corresponding construction method to create it using the parsed tensor information. For the input tensor, we can get specifications of the input tensor from the parsed tensor information, and obtain the tensor value from the previous operator of the computational graph. We can then use them to construct the input data. For parameter data in TFLite (*e.g.*, `ConvParams params` as shown in Listing 2), this is produced by analyzing the operator information (*e.g.*, the `"builtin_options"` in Figure 2).

To create the parameter data automatically, we build a mapping function $f(p)$ for each operator, which is collected from the source code of TFLite. TFLite needs to use such functions to configure the parameters correctly in model inference. Creating the parameter data can then be formulated as `params` $= f_n(p)$, where n is the operator index. We can use this method to create the most required data for each computing unit.

However, as the examples shown in Figure 2 (*i.e.*, `lhs_cacheable` of the configuring data), some members in parameter `struct` data are determined by the device status and tensor status, which are not provided in the model representation. These members are referred to as unknown status. Therefore, we introduce a *Dynamic Configuration* method to solve this problem.

3.5 *Dynamic Configuration*

As the parameter `struct` data has limited potential choices. A naive way to create configurations for unknown status is searching for an optimal setting in the whole space. However, this simple approach will usually be computationally infeasible when we search for an optimal configuration combination in the large DL model. This is because the large model may has too many operators that need to be configured. Therefore, we analyze the aforementioned status-related configuration and propose a *Dynamic Configuration* method to automatically configure each computing unit in a computational graph. This module will produce the complete computing code for the given TFLite DL model. The process of *CustomDLCoder*'s *Dynamic Configuration* is shown in Algorithm 2.

After we obtain the computing unit set from the *Computing Unit Extraction* and configure its known data using the *Configuring Data Analysis*, we then collect any unknown status for all operators in the computational graph. After that, we use status check rules to find eligible configurations from all possibilities of configurations. The status check rules are designed to reduce the search space. According to our analysis of collected DL models, we define two status check rules: 1) the unknown status in two computing units should be the same when the two computing units have the same

Algorithm 2 *Dynamic Configuration*

Input: extracted computing unit $\overline{\mathbf{U}} = \{\overline{u}_n\}$, computational graph $\mathcal{G}$.
Output: a complete program C
Notation: the n-th operator O_n, the number of operators s
1 : Initialize a unknown status set **M**
2 : **For** O_n in $\mathcal{G}$ **do** :
3 : find the unknown status M_n in $\overline{u}_n$
4 : **M**.append(M_n)
5 : **end For**
6 : **For** $\{m_0, ..., m_s\}$ in $\{M_1, ..., M_s\}$ **do** :
7 : **If** check_rules($\{m_0, ..., m_s\}$):
8 : Configure the $\overline{\mathbf{U}}$ using $\{m_0, ..., m_s\}$ to get C
9 : **If** $\|C(x) - \mathcal{G}(x)\|_2 < \delta$
10 : **Break**
11 : **Return** C

operator information (*i.e.*, name and known parameters). This is because, in DL models, the same kind of operators in one model usually have the same data type and configurations. 2) The same kind of data (*i.e.*, input, weights, `Conv2d` parameter) should have the same unknown status. This is because TFLite usually gets the same system and tensor status information for the same kind of data. The two rules can significantly reduce the search space because DL models usually are a combination of basic model blocks. For example, a well-known model architecture ResNet is built with a lot of similar residual blocks, which only have limited kinds of operators and data. If the configuration is eligible, we assemble the computing unit set to a complete code implementation for the model inference. We iterate over all eligible unknown configurations until the output difference between the generated program returns and the original model inference is lower than a prefixed value δ under the same input. The output difference can be calculated by:

$$diff = \|C(x) - \mathcal{G}(x)\|_2 \tag{2}$$

where C is produced computing program, $\mathcal{G}$ is the original model inference. x is the input that is in the eligible range of this model. We use the l_2-norm to measure the distance between two outputs. In experiments, we use 100 random inputs to compute the output difference. When the generated program has a similar output to the original model inference, the generated program can be used to replace the model file and library in the deployment environment.

3.6 Compilation

Finally, the generated C++ program from *CustomDLCoder* embodies the complete computing process as the original model inference, without needing the DL model file or representations. Because this generated code is a pure C++ program, we can use existing code obfuscation techniques to obfuscate the executable program to remove the semantic information and just keep its computing process. The DL component in the compiled program is thus hard to be identified and its DL model information cannot be easily decompiled by attackers. Code obfuscation is a well-developed technique, we omit discussing it in this study.

4 EVALUATION

Our *CustomDLCoder* aims to provide better security and performance for on-device deployed DL solutions. To determine if this has been achieved, we evaluate *CustomDLCoder* by answering the following key research questions:

- **RQ1:** How accurate is *CustomDLCoder* in transforming on-device models to executable programs?
- **RQ2:** How effective is *CustomDLCoder* in defending against DL model information parsing and extraction?
- **RQ3:** How efficient is the program generated by *CustomDLCoder* at model inference compared with original TFLite?
- **RQ4:** What is the memory cost of code generated by *CustomDLCoder* compared with original TFLite?
- **RQ5:** Is the code produced by *CustomDLCoder* better than other model deployment strategies.

Dataset. To evaluate *CustomDLCoder*'s performance on models with various structures for multiple tasks, we use the DL model collected in Kaggle Model Hub[4] and Huggingface[5] to evaluate our proposed method including the fruit recognition model, the skin cancer diagnosis model, MobileNet [15], MNASNet [31], SqueezeNet [19], EfficientNet [32], MiDaS [29], Lenet [21], PoseNet [20], SSD [24], and GPT-2 [28], respectively.

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

Setting. For comparing the latency between our method and others, we run the model for one sample to compute the time consumption because some DL platforms may have optimization methods for the input with multiple samples, *i.e.*, keep the essential intermediate data in memory. To accurately compute the time consumption, we repeat the process 1,000 times and compute the average latency on x86-64 platforms. For computing the memory usage, we use the *Valgrind Massif*[6]. *Valgrind* is a powerful instrumentation framework for memory profiling. *Massif*, a *Valgrind* tool, can be used to measure memory usage accurately.

Baseline. As our proposed *CustomDLCoder* extracts the code from the TFLite platform, we use the TFLite platform as the baseline, *i.e.*, using `.tflite` model file and the API library compiled by `CMake`, because *CustomDLCoder* also uses *CMake* to produce the executable program. The compilation process can be found on the TFLite official documents[7]. In addition, we also use other model deployment strategies (*i.e.*, TVM, ONNX-Runtime) to show the efficiency of our method in compiling the DL models.

4.1 RQ1: Transformation Accuracy

We need to first evaluate the compilation correctness of our generated customized DL code. Table 1 summarises the evaluation of our methods on compilation correctness. If the compilation has errors (*i.e.*, the conversion is not correct), the performance of DL models

[4] https://www.kaggle.com/models
[5] https://huggingface.co/models
[6] https://valgrind.org/docs/manual/ms-manual.html
[7] https://www.tensorflow.org/lite/guide/build_cmake

Table 1: The maximal translation error ($\times 10^{-5}$) of our proposed *CustomDLCoder* and other model deployment strategies on `x86-64` platform. *CustomDLCoder* program, TVM model, and ONNX-Runtime model are all converted from the TFLite model.

	Model name	Fruit	Skin cancer	MobileNet	MNASNet	SqueezeNet	EfficientNet	MiDaS	Lenet	PoseNet	SSD	GPT-2	Average
error	TVM	0.40	0.13	0.21	1.03	0.17	0.07	73.24	0.05	4.20	1.86	34.33	10.52
	ONNX-Runtime	52.72	38.31	0.01	0.51	0.23	36.89	101.23	0.01	11.7	1.62	4.6×10^5	4.2×10^4
	CustomDLCoder	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

Table 2: The success number of existing attacking methods to parsing the information of on-device models. '√': this information collection method cannot extract information for all models. '-': not applicable for this method. We use '✗' to show that TVM can be attacked by reverse engineering the low-level representation of computation graphs with NNReverse [5]. However, this method is only effective for the TVM. We omitted it from our experiments and simply used its results.

	Model conversion tool				Reverse Engineering	Feature analyzing	Searching
	TF-ONNX	TFLite2ONNX	TFLite2TF	ONNX2TF	Reverse Engineering [5, 22]	Smart App Attack [16]	DL Sniffer [39]
Original TFLite (without defense)	11	10	10	-	11	8	11
Model Obfuscation[42]	√	√	√	-	√	√	11
TVM	-	-	-	-	✗	√	11
ONNX-Runtime	-	-	-	10	10	8	11
CustomDLCoder	-	-	-	-	√	√	√

will lose. The maximal compilation error can be formulated as:

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

where $\mathcal{G}$ is the model inference before compilation. C refers to compiled program. We use 100 inputs to compute the maximal compilation error. Our *CustomDLCoder* method does not have any compilation errors as *CustomDLCoder* uses the same computing process as the original model inference.

In comparison, TVM and ONNX-Runtime will have slight compilation errors because they rely on conversion rules to compile a high-level representation to a low-level representation, which will cause inevitable compilation errors. As per our observation, for the model with a more complex architecture, TVM and ONNX-Runtime models will have more compilation errors, as the error for each operator will be accumulated in the next computing.

> RQ1 Answer: *CustomDLCoder* will not introduce any compilation errors to the generated program and has better compilation correctness than other deployment strategies.

4.2 RQ2: Resilience to Attacks

We use the potential attacks mentioned in [42] to evaluate the effectiveness of *CustomDLCoder* in concealing information in DL models. In addition, we use the *DL sniffer* to show the performance of deployment strategies in defending against keyword searching [39]. We use methods to collect model information as follows:

(1) **Model format conversion:** Existing model conversion tools first parse the model's structure and weights, and then assemble them into a new model with different formats. We utilize four tools, namely TF-ONNX [2], TFLite2ONNX [34], TFLite2TF [17], and ONNX2TF [18] to evaluate the performance of different methods. If a tool can convert the model to the target model format, we consider it a successful model information extraction. Note that the results may change for different versions of tools. Conversely, if the defence method is effective, these tools will be unable to extract the model information.

(2) **Reverse engineering for in-model information:** A TFLite model can be parsed by the `Flatbuffer`. The other model representation types can also be collected. We refer to a study in [5, 22], in which the researchers attempt to extract a model's structure and weights inside the model file.

(3) **Finding a similar model from the Internet:** This involves comparing the features among models. Attackers can use the App Attack to find a debuggable model with a similar structure and weights from the Internet [16]. For the App Attacking method, if it can correctly identify the model with defences that have the same model structure as the original one on TensorFlow Hub (8 models in our test set are collected from the TensorFlow Hub), we consider it a successful model extraction.

(4) **Ientifying DL model and model components using keyword searching:** This includes searching for operators (*e.g.*, '`conv2d`') and weights in model representations (*e.g.*, '`.tflite`' file). We adopt the method in [39] to try to identify the model representation in App packages. In our test set, two models were originally collected from the Android Apps. For other models, we randomly pack them into the public Andoird Apps as the test set. In experiments, we use *Apktool* to unpack App packages and use the DL sniffer to search the DL-related component.

Our attack results are shown in Table 2. Model obfuscation was recently developed to obfuscate on-device DL model information. It can successfully defend against all attacking methods except for the searching technique. This is because it will use the conventional model-library interaction for model inference. Such interaction needs a model file (*i.e.*, '`.tflite`') to store the computational graph. Thus, attackers can use the keyword of model format to search for

Table 3: The latency (ms per input) compared with the original TFLite library. *CustomDLCoder* programs are converted from the TFLite model. The lower is better. 'x86-64': the experiments on the Ubuntu workstation. 'ARM64': the experiments on the Xiaomi 11 Pro smartphone with Android 13.

	Model name	Fruit	Skin cancer	MobileNet	MNASNet	SqueezeNet	EfficientNet	MiDaS	Lenet	PoseNet	SSD	GPT-2	Average
x86-64	TFLite (Baseline)	33.0	98.6	60.4	83.8	58.9	99.1	484.8	2.9	116.6	227.9	578.8	167.7
	CustomDLCoder	**32.6**	**91.9**	**54.6**	**70.8**	**50.2**	**84.9**	**317.0**	**2.0**	**110.3**	**195.9**	**433.0**	**131.2**
ARM64	TFLite (Baseline)	13.1	42.9	26.9	34.8	52.9	40.0	406.8	5.1	**42.7**	96.1	530.2	117.4
	CustomDLCoder	**12.7**	**35.1**	**22.8**	**27.8**	**52.4**	**33.8**	**272.1**	**1.4**	44.0	**84.9**	**391.6**	**88.9**

the corresponding model file. However, our *CustomDLCoder* can compile the DL model to an executable program, which does not use any semantic representations to store the model information. Therefore, the parsing and reverse engineering tool cannot be applied to the proposed deployment method. In addition, our method can use traditional existing code obfuscation methods to enhance the model security because the program generated by our tool is the complete C++ code. In comparison, other AI compiler methods like TVM cannot achieve this. TVM needs files to store its model representation (*i.e.*, computational graph) and weights using low-level representations or such information can be extracted from the program because the TVM program needs to interact with the representation of the computation graph and weights. Although TVM uses low-level representations, such low-level representations can be converted to the corresponding high-level representation [5]. It cannot resist such reverse engineering tools and search by keywords. The ONNX-Runtime uses the ONNX model that can be converted from TFLite models. However, ONNX models can be also converted to the TFLite model. So, the ONNX-Runtime platform can be attacked by the methods that are effective for TFLite models.

> RQ2 Answer: Programs produced by *CustomDLCoder* can mitigate attacking methods that use semantic information to identify DL components. It provides the deployed DL model a higher level of security compared with existing deployment methods by encoding model file information in generated code and obfuscating this generated code.

4.3 RQ3: Efficiency of Generated ML Code

Our proposed *CustomDLCoder* extracts the computing code from the TFLite library. It reduces the computational complexity because *CustomDLCoder* removes some inference-unrelated steps (*e.g.*, analyzing the computational graph and operator parameters) during code generation. However, TFLite models use `Flatbuffer` to save and load the model. Parsing and loading the file using `Flatbuffer` is very fast and can increase the efficiency of the model inference. The source code of TFLite is optimized for running the model on `Flatbuffer`. We need to experiment to show the inference time efficiency of our generated DL model code is at least as good.

To show the performance of *CustomDLCoder* generated DL code, we compare our method with the original TFLite platform. The results are shown in Table 3. Note that we do not use any additional optimization for each deployment method, and we use multi-thread mode on the `x84-64` platform. The model runs faster on ARM64 because TFLite optimizes the model performance on ARM64, and we run the model using Python on `x86-64`. In contrast, we build executable programs using C++ to generate samples to compute the inference time. Our method can accelerate model inference by 21.8% and 24.3% on x86-64 and ARM64 platforms, respectively. Our *CustomDLCoder* generated code has the lowest average inference time because it is much more efficient on the largest neural networks (*i.e.*, MiDaS and GPT-2). TFLite achieves the fastest model inference in the tiny model PoseNet. This is because loading and configuring the model information costs most of the time in model inference for some operators, and the `Flatbuffer` is efficient in doing it. **In most cases, our proposed *CustomDLCoder* achieves better time efficiency than the original TFLite platform because it only contains essential output computing code to execute the model inference**.

> RQ3 Answer: Our *CustomDLCoder* can accelerate the model inference compared with the original TFLite model. In addition, the program generated by our approach runs much faster than TFLite models for large neural networks.

4.4 RQ4: Size and Memory Effciency

CustomDLCoder will produce an executable program that is deployed on devices. We need to evaluate the size of the programs generated by our method to show the efficiency of *Computing Unit Extraction* steps. The results are shown in Table 4. Our *CustomDLCoder* will produce programs that are much smaller than the original TFLite platform, especially for small DL models (*i.e.*, Lenet, Fruit, MobileNet). For large models, the model weights consume most of the size of the deployment files so *CustomDLCoder* files and original TFLite files do not have much difference.

For memory consumption, our *CustomDLCoder* generated program is theoretically much better than the original TFLite model, as the program produced by *CustomDLCoder* removes model parsing and data preparation. TFLite library needs to load the computation graph and use intermediate data to configure their operators. A random-access memory (RAM) consumption comparison among them is shown in Table 5. Note that we do not use any memory optimization methods for all deployment methods.

The customized DL program generated by our method is much more efficient in memory consumption than its original TFLite DL model in all cases. In this experiment, the program generated by our *CustomDLCoder* can significantly reduce memory consumption by 68.8% and 36.0% on x86-64 and ARM64 platforms, respectively,

Table 4: The size (Mb) of deployed components for different deployment methods on the `ARM64` platform. TFLite components contain the model file and the compiled API library (be *CMake*). Our *CustomDLCoder* only has an executable file that also compiled by *CMake*.

Model name	Fruit	Skin cancer	MobileNet	MNASNet	SqueezeNet	EfficientNet	MiDaS	Lenet	PoseNet	SSD	GPT-2	Average
TFLite (Baseline)	58.6	70.0	63.4	70.6	58.1	71.7	119.4	59.6	25.6	58.1	378.4	94.0
CustomDLCoder	**15.0**	**26.4**	**19.3**	**25.9**	**14.2**	**28.3**	**74.4**	**14.5**	**12.2**	**35.2**	**308.9**	**52.2**

Table 5: The RAM consumption (Mb) of model inference for different deployment methods. *CustomDLCoder* programs are converted from the TFLite model. The lower is better. Note that we use the peak memory cost to show the results.

	Model name	Fruit	Skin cancer	MobileNet	MNASNet	SqueezeNet	EfficientNet	MiDaS	Lenet	PoseNet	SSD	GPT-2	Average
x86-64	TFLite (Baseline)	20.41	45.84	31.55	39.80	29.26	43.97	220.22	12.40	25.60	72.33	418.33	87.25
	CustomDLCoder	**8.89**	**11.95**	**10.47**	**9.65**	**17.74**	**11.95**	**94.30**	**5.96**	**14.73**	**19.01**	**94.99**	**27.24**
ARM64	TFLite (Baseline)	15.35	32.89	21.47	29.74	27.55	32.07	219.61	13.29	20.16	48.37	483.96	85.86
	CustomDLCoder	**11.98**	**24.45**	**17.23**	**23.30**	**13.94**	**25.78**	**99.18**	**11.32**	**13.64**	**40.94**	**322.85**	**54.96**

and does not increase the time taken. These benefits will help small memory edge devices to deploy larger and more models on-device.

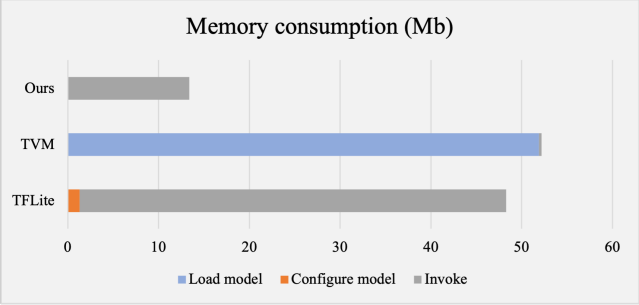

Figure 4: The pattern of memory allocation in different deployment methods on the Skin diagnosis model. A complete model inference process: load model → configure model → invoke (compute the output).

The three typical memory allocation patterns for different deployment methods are shown in Figure 4. TVM allocates memory mainly on the model loading step, which means it will allocate data (*e.g.*, wights) in memory before the model inference and keep them in the memory. For TFlite, it allocates some tensors when configuring the computing function. However, our method only allocates the memory in the invoking step, *i.e.*, computing the outputs, because it does not parse the model representation and only allocates tensors in the computing process.

> RQ4 Answer: The C++ program produced by *CustomDLCoder* uses much less memory than the original TFLite model in all cases. It is especially memory efficient on the large language model example, *i.e.*, GPT-2.

4.5 RQ5: Comparison with Other Strategies

The RQ3 and RQ4 show that the generated program runs much more efficiently than the original model inference of TFLite. In this research question, we would like to compare our tool and other model deployment strategies like TVM and ONNX-Runtime to show the effectiveness of our *CustomDLCoder* method. Although TVM supports packing the model information into the API library, they both need the model representation to execute the model inference and not pure code implementation. We have compared the compilation error of the proposed method, TVM, and ONNX-Runtime in RQ1.

For latency, our *CustomDLCoder* slightly outperforms or underperforms the other model deployment strategies. It does significantly better on the GPT-2 LLM (see Table 6, *Latency* row).

For memory, our method performs significantly better than other strategies, *i.e.*, reduces the RAM consumption by more than 80% on `x86-64` platform (see Table 6, *RAM* row). Programs generated by our *CustomDLCoder* approach are very memory-efficient. For example, on a large language model ML example (*i.e.*, GPT-2), it consumes 10 times less RAM than other strategies. On all others, it ranges from 1.4 times less to 6 times less.

> RQ5 Answer: *CustomDLCoder* significantly outperforms other compilation strategies, *i.e.*, TVM and ONNX-Runtime, in terms of RAM consumption. In addition, our method achieves comparable performance in terms of latency.

5 DISCUSSION

In this section, we will discuss how to maintain our tool, the meta-model of our method, and its limitations.

5.1 Integrating Our Method into Existing Tools

We developed our method based on current function call process and backend APIs of TFLite. We may need to update some parts of our method, *e.g.*, code extraction and configuration, to support new versions of TFLite. Our approach would be better if integrated into existing on-device DL libraries such as TFLite and other AI compilers. Our prototype tool cannot currently support all DL models. However, the on-device DL libraries have similar inference

Table 6: The model inference time (*i.e.*, latency), and RAM consumption (Mb) of our proposed *CustomDLCoder* and other model deployment strategies on `x86-64` platform. *CustomDLCoder* program, TVM model, and ONNX-Runtime model are all converted from the TFLite model.

	Model name	Fruit	Skin cancer	MobileNet	MNASNet	SqueezeNet	EfficientNet	MiDaS	Lenet	PoseNet	SSD	GPT-2	Average
Latency	TVM	**28.1**	82.4	61.3	85.0	69.9	82.2	447.6	26.6	103.3	207.7	468.3	151.4
	ONNX-Runtime	31.6	**64.9**	59.8	76.4	54.1	**79.7**	**285.6**	27.1	**89.5**	**176.3**	984.3	175.4
	CustomDLCoder	32.6	91.9	**54.6**	**70.8**	**50.2**	84.9	317.0	**2.0**	110.3	195.9	**433.0**	**131.2**
RAM	TVM	35.50	61.93	47.05	65.09	40.78	70.31	179.62	37.85	43.97	93.61	1115.14	162.80
	ONNX-Runtime	25.97	53.81	42.31	48.91	25.80	48.68	154.25	27.00	42.82	76.48	1248.91	163.18
	CustomDLCoder	**8.89**	**11.95**	**10.47**	**9.65**	**17.74**	**11.95**	**94.30**	**5.96**	**14.73**	**19.01**	**94.99**	**27.24**

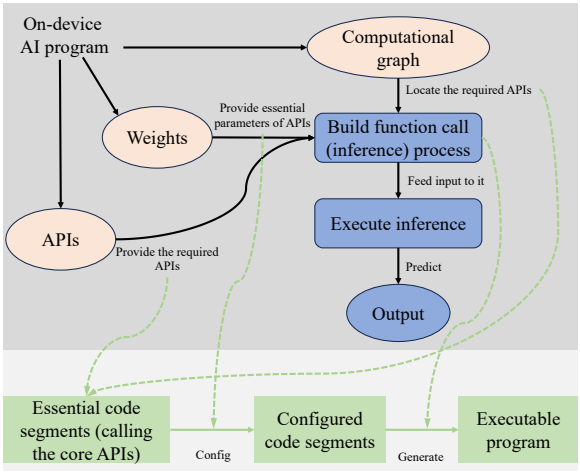

Figure 5: Meta-model our method. Model representations including computational graphs and weights can be stored as a separate file or be integrated into the API library. Because AI platforms usually are open-sourced, the source code of API libraries can be collected from the Internet.

workflows. We provide a design meta-model (see Figure 5) that developers can use to integrate our research ideas into their public tools. In existing on-device AI programs, the computational graph will be parsed to locate the required APIs in the library. Then, the model weights will be loaded into the computing function to build a complete inference process. After obtaining the complete inference process, the input data will be fed to it to get outputs. To generate pure code implementations for on-device model inference, the essential code segments (usually C/C++ or assembly codes) can be located by tracing the function call when building the function call process in AI programs. Each code segment contains the complete computing process from input to output of one specific operator in the on-device models. Next, the model weights can be extracted and parsed from the model representation, and they can be used to configure the extracted codes, *e.g.*, creating essential variables or instants for the C++ functions. The process of generating the complete program based on the configured code segments is then similar to building an inference process in AI programs, *i.e.*, call the functions that include each operator's code segment as the order of operators in the on-device model.

5.2 Limitations

Our *CustomDLCoder* is designed for on-device platforms like TFLite. Our methods may not be effective on other DL platforms without an optimized on-device implementation, as the extracted code will be inefficient if the DL library implements training-related code in operators. However, users can use model conversion tools to convert other models to TFLite models and then use our tool.

We have not evaluated our *CustomDLCoder* with real-world mobile ML developers. We have not evaluated our method with side-channel information (*e.g.*, RAM, CPU usage) to reconstruct the model [13, 23, 35]. Our method may be effective for them because the generated code has a different RAM usage pattern from the original model (see Figure 4). However, our generated program has the same CPU usage pattern as the original model.

6 CONCLUSION

In this paper, we analyze the key security issues in existing model deployment strategies. Attackers can identify the DL components (*e.g.*, models, libraries, programs) on devices, and generate effective attacks against these DL models. We propose a method *CustomDLCoder* to extract the computing codes of the DL model, refactor the extracted codes, and compile them into an executable program. Our proposed *CustomDLCoder* has four steps including *Model Parsing*, *Computing Unit Extraction*, *Configuring Data Analysis*, and *Dynamic Configuration*. Our *CustomDLCoder* will generate complete custom programs of model inference without the need for model representations. Our experiments show that our method not only achieves a higher level of security compared with existing methods but also can accelerate the model inference and reduce memory usage. In future, we will analyze the source code of the DL library to optimize extracted code to improve efficiency of the executable program.

7 DATA-AVAILABILITY STATEMENT

We provide a GitHub repository of our artifact [41]. It has the instruction of installing the dependency and testing our tool.

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 ARC Laureate Fellowship FL190100035.

REFERENCES

[1] 2022. *m2cgen*. https://github.com/BayesWitnesses/m2cgen

[2] 2022. *tf2onnx - Convert TensorFlow, Keras, Tensorflow.js and Tflite models to ONN*. https://github.com/onnx/tensorflow-onnx

[3] 2024. *llama.cpp*. https://github.com/ggerganov/llama.cpp

[4] 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.

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

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[7] 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

[8] 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

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

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

[11] 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

[12] 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

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

[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] 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).

[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.

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

[18] Katsuya Hyodo. 2023. *ONNX2TF*. https://github.com/PINTO0309/onnx2tf

[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] 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

[21] 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

[22] 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

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

[24] 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.

[25] 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

[26] 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.

[27] 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).

[28] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[29] 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

[30] 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

[31] 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

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

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

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

[35] 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

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

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

[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] 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.

[41] Mingyi Zhou, Xiang Gao, Pei Liu, John Grundy, Xiao Chen, Chunyang Chen, and Li Li. 2024. *CustomDLCoder: Generating Pure Code Implementations to Replace On-Device DL Models (0.1)*. https://doi.org/10.5281/zenodo.10897855

[42] 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

[43] Mingyi Zhou, Xiang Gao, Jing Wu, Kui Liu, Hailong Sun, and Li Li. 2024. Investigating White-Box Attacks for On-Device Models. *arXiv preprint arXiv:2402.05493* (2024).

[44] 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.

Received 16-DEC-2023; accepted 2024-03-02