Automation Slicing and Testing for in-App Deep Learning Models

Hao Wu  
National Key Laboratory for Novel Software Technology, Nanjing University

Yuhang Gong  
National Key Laboratory for Novel Software Technology, Nanjing University

Xiaopeng Ke  
National Key Laboratory for Novel Software Technology, Nanjing University

Hanzhong Liang  
National Key Laboratory for Novel Software Technology, Nanjing University

Minghao Li  
Harvard University

Yunxin Liu  
Institute for AI Industry Research (AIR), Tsinghua University

Sheng Zhong  
National Key Laboratory for Novel Software Technology, Nanjing University

ABSTRACT

Intelligent Apps (iApps), equipped with in-App deep learning (DL) models, are emerging to offer stable DL inference services. However, App marketplaces have trouble auto testing iApps because the in-App model is black-box and couples with ordinary codes. In this work, we propose an automated tool, ASTM, which can enable large-scale testing of in-App models. ASTM takes as input an iApps and the outputs can replace the in-App model as the test object. ASTM proposes two reconstruction techniques to translate the in-App model to a backpropagation-enabled version and reconstruct the IO processing code for DL inference. With the ASTM’s help, we perform a large-scale study on the robustness of 100 unique commercial in-App models and find that 56% of in-App models are vulnerable to robustness issues in our context. ASTM also detects physical attacks against three representative iApps that may cause economic losses and security issues.

1 INTRODUCTION

Deep learning (DL) technologies have significantly advanced many fields critical to mobile applications, such as image understanding, speech recognition, and text translation [29, 43, 49]. Besides, a lot of research efforts have been put into optimizations of DL latency and efficiency [11, 22, 34, 40], paving the path towards the local intelligent inference on mobile devices like smartphones. Recent study [8, 45, 51] indicates the intelligent Apps (iApps), smartphone Apps using in-App DL models, will be increasingly popular, which is also verified by our own study shown in Section 6.1.

The key difference between an iApp and an ordinary App is a software component performing the local intelligent inference (Figure 1). This component usually consists of two parts, the in-App DL model and IO processing code. The in-App DL model is commonly optimized for easy deployment and speedy inference, and thus gets rid of the backpropagation (BP) ability. The IO processing code is tightly associated with the corresponding in-App model, and it is responsible for both preparing inference inputs and interpreting inference outputs. Incorrect IO processing will impede the success of DL inference.

Such key difference in iApps brings troubles to the App marketplaces’ important routine work — the security-oriented auto testing of Apps released to them. This is because, by nature, the testing philosophy, paradigm, and requirements for neural networks are different from those for software codes [20, 23, 50]. Regarding an in-App DL model, the inference changes to perturbed inputs, including their possible interpretations, are more important than the behavior changes of codes. For example, a road-lane-detection iApp is insecure and should be delisted from App marketplaces if it is too easy to show different lane lines when adversarial perturbations, say a small mark on the road, are applied. Since more and more Apps go intelligence, it is urgent to resolve how to efficiently auto-test massive released iApps in a comprehensive manner from the AI security perspective. However, to remove such “dark cloud” of auto-testing requires a new tool design addressing two challenges.

Model Conversion. BP operations on the DL model are not only inevitable in various model testing methods [15, 16, 31, 38], but also

*Corresponding author. Email: fengyuan.xu@nju.edu.cn

Figure 1: A typical structure of an iApp. Compared to ordinary Apps, the key difference is a software component performing the local intelligent inference. This component consists of a in-App DL model optimized for inference and its paired code for input and output processing. App marketplaces so far do not support the auto testing of this component from the AI security perspective.
critical to the DL interpretability [33, 55]. However, it is difficult to convert an in-App DL model into its white-box counterpart on a BP-enabled test platform, because such a model in App is heavily optimized for inference. For example, the neural operators’ attributes are implicit and hidden in the App codes, and weights quantization makes gradient computations extremely complicated.

Reliable Slicing. A converted white-box model cannot properly run on the testing platform without the cooperation of its own IO processing code in the App. Therefore, the App marketplace needs to precisely slice out this part of the released App code and re-use it during the testing. However, this code slicing should be efficient and accurate with an extremely high success rate, so that the massive testing of released iApps are able to be supported with no manual efforts involved. For example, IO processing often needs to perform assignment operations between data objects and arrays, which is hard to be tracked accurately in general cases.

Existing testing tools cannot meet the above needs. First, existing model conversion tools, e.g., MMdnn [36] and ONNX-based [4] tools, can translate trainable DL models between frameworks, but they cannot translate inference-only models into corresponding BP-enabled versions. Second, the dynamic code slicing [10] needs human involvement, while the success rate of static code slicing is not high. We find that the Android slicing tool Jicer [41] cannot extract runnable IO processing codes for model testing purpose.

In this work, we propose an automated tool, ASTM, which can enable large-scale testing of In-App DL Models for App marketplaces. ASTM can successfully extract the in-App DL models from massive real-world iApps and its IO processing code, and then prepares the ready-to-test versions of in-App DL models with IO processing code for various model assessments. The whole procedure of ASTM is efficient without developers’ cooperation like model information or source code. The outputs are capable of BP operations which are important to many model assessments and interpretation methods. ASTM can help App marketplaces to auto-test iApps in practice from the AI security perspective.

The key designs of ASTM are two reconstruction techniques, i.e., precise code reconstruction and BP-enabled model reconstruction. The code reconstruction adopts a precise Android static slicing, which can produce runnable processing codes by fully considering the App’s control flow, invoking context, field access, and branch selection during the slicing. It also proposes an IO-processing-oriented code generation to ensure that sliced statements can execute in the same order as they are in the original iApp. The BP-enabled model reconstruction can convert an inference-only model into a white-box one, which is used for comprehensive testing. It first abstracts the framework-independent computation procedure and then utilizes a rule-based operator’s attributes completion of recovering the stripped information for model reconstruction.

We implement the ASTM and test in-App models at scale. We collect about 15k unique Apps from five marketplaces. Following the in-App model finding method proposed by the work [51], we find there are 3,064 iApps equipped with 800 unique in-App DL models. Then we test 100 unique in-App models with popular DL frameworks in terms of robustness. We find 56% of the in-App models are vulnerable to robustness issues (Section 6.3). ASTM also detects physical attacks against three representative in-App models. Details of the measurement are presented in Section 6.

We highlight the key contributions as follows:

1. ASTM is the first effort to enable in-App DL model testing at scale. It can automatically reconstruct the IO processing code and BP-enabled DL models for powerful white-box testing techniques. ASTM is fully-automatic and works without the iApp provider’s cooperation.
2. ASTM proposes two novel reconstruction techniques. The code reconstruction is a precise Android static slicing to produce runnable IO processing code. The model reconstruction rebuilds the BP-enabled model from its inference-only version by establishing equivalent calculation and information completion.
3. We perform robustness assessment on 100 unique commercial in-App models through ASTM. We also successfully detect physical adversarial attacks against commercial iApps, which may cause economic losses and serious security issues.

2 BACKGROUND AND RELATED WORKS

2.1 In-App DL Models

The iApps equipped with DL models are emerging. The in-App models have been studied preliminarily. The work [51] is the first work to keep an eye on iApps and proposes a way to find DL models and frameworks in an App. Then the work [45] investigates how iApp providers protect in-App DL models. The work [8] and work [54] perform comprehensive studies on DL model inference performance. The work [24] studies the robustness of the in-App DL models.

Existing works do not propose techniques for automated reconstruction of IO processing code in the iApp. These works either do not need the IO processing code or reconstruct the processing code manually [8, 51]. These works can also not directly reconstruct the BP-enabled model from the in-App model to perform the model assessment. For example, the work [24] assesses the model’s robustness by utilizing the adversarial attacks’ transferability.

2.2 Android Slicing

Programing slicing is a technique to extract statements that may affect a given statement (stmt) and a set of values (V) through data dependency and control dependency analysis. The given statement and the set of values are referred as a slicing criterion <stmt, V>. There are two well-known frameworks in the Android analysis scenarios, i.e., WALA [44] and Soot [32]. The frameworks support the basic functionalities and programable interfaces to build slicing algorithms. The code reconstruction of ASTM is based on Soot because Soot has better support for Dalvik bytecode, and many Androids analysis tools are developed based on Soot.

The work [10] designs a human-involved Android dynamic slicing tool by which users first perform App instrumentation, manually run the instrumented App, and dynamically collect logs to perform Android slicing. AppSlicer [12] is another activity-level App dynamic slicing technique. It first triggers all activities in a simulator, records the code invoked by each activity, and slices the target activity according to the recorded content. Jicer [41] is a work to perform general Android static slicing. It aims at slicing the code of any App with any slicing criterion. However, Jicer fails to
slice the processing code from a commercial iApp because it lacks the processing code oriented design.

The slicer’s accuracy depends on its sensitivity regarding various program features [41]. A flow-sensitive slicer takes control dependencies into full consideration. Fully considering from which context a method is called makes the slicer context-sensitive. Handling data dependencies between usages of the same field in different methods makes a slicer field-sensitive. When handling conditional statements, a path-sensitive slicer must consider slicing which branch.

The code reconstruction of ASTM is a fully-automatic precise Android static slicing without human involvement. In addition, our reconstruction tool is flow-, context-, field-, path-sensitive.

2.3 Adversarial Attacks

The adversarial attacks have been widely studied in both the AI and security communities. DL models have been demonstrated that they are vulnerable to adversarial examples [46]. The adversarial examples are inputs to DL models that have been intentionally optimized to cause models to make a mistake. Specifically, given a DL model \( f_\theta(\cdot) \) with parameters \( \theta \) and an input \( x \) with a ground truth label \( y \), an adversarial example \( x' \) is produced by optimization, which is closed to \( x \). \( x' \) is able to cause the DL model to make an incorrect prediction as \( f_\theta(x') \neq y \) (untargeted attacks), or \( f_\theta(x') = y^* \) (targeted attacks) for some \( y^* \neq y \).

The attack scenarios can be classified by the amount of knowledge the adversary has about the model, i.e., white box attack [38] and black box attack [13]. In the white box scenario, the adversary fully knows the model, including model type, model architecture, and values of all parameters. The adversary can perform gradient-based attacks on the model. In the black box scenario, the adversary has limited information about the model. The adversaries can only perform the attack by probing and observing the output.

The adversarial attacks can be classified into digital attacks [21] and physical attacks [19] according to how the adversary modifies the input data. In a digital attack, the adversary has direct access to the actual data fed into the model. The adversary can modify each bit of the input data. In a physical attack, the adversary does not have direct access to the digital representation of the input data. The adversary can place objects in the physical environment seen by the camera.

In our work, we assess the robustness of the in-App models through the white box digital attacks. We also perform white box physical attacks on three representative in-App models.

3 OVERVIEW

In this section, we first define the security model and design requirements of ASTM. Then we present the high-level design of ASTM and introduce the two key reconstruction techniques.

3.1 Problem Overview

The security issues of in-App models are able to cause the iApp to behave abnormally. And the in-App model may even become a “protective umbrella” for the iApp’s malicious behavior. Therefore, App markets have a strong need to test the in-App models in massive iApps.

3.1.1 Security Model. The iApp may be developed by careless or malicious developers. Those careless developers may use an iApp model that has not undergone comprehensive testing. Those malicious developers even launch active attacks by poisoning the training data. We do not directly handle the scenario where the iApp is packed. If App markets want to test the in-App model in a packed iApp, they can first use previous works [17, 52, 53] to unpack the iApp, and then use ASTM to prepare the IO processing code and BP-enabled model.

3.1.2 Requirements. Given the challenges discussed in Section 1, we summarize three requirements to perform comprehensive testing on in-App models.

First, the model before and after reconstruction should be equivalent. The model reconstruction mainly performs two things. One is porting the in-App model to a DL framework that supports training. The other is removing the obstacles that hinder gradient computation, such as undoing weights quantization and replacing the inference-only operations. The reconstructed model and the in-App model should be perfectly equivalent in computational procedure and trainable parameters so that testing on the reconstructed model is equivalent to testing on the in-App model.

Second, the processing code reconstruction should be precise and runnable. We show a case\(^1\) in Figure 2 to demonstrate the effect of processing code on the correctness of DL inference. Before being fed into the DL model, the user input needs to be resized in a preset way. If the resize operation is omitted or substituted by a random resize operation, the inference result is wrong. The reconstruction should be precise so that processing operations are not missed. At the same time, the reconstruction should be practical so as to ensure that it can produce runnable results.

Third, the proposed tool should be automatic. To enable App markets to test in-App models at scale, the model and code reconstructions should be fully automatic without human involvement. And in our scenario, the reconstructions should be done without the cooperation of iApp developers.

\(^1\)The DL model is taken from an iApp, whose bundle ID is org.prudhvianddheeraj.lite.example.detection.
3.2 ASTM Design Overview

In this section, we introduce the ASTM workflow and explain how ASTM meets the requirements discussed in Section 3.1 in brief.

We first used the work [51] to determine whether an App contains in-App models. If an iApp is found, we then perform the proposed code reconstruction and model reconstruction to build the test object.

During the code reconstruction, ASTM takes as input a released iApp and outputs runnable IO processing codes. It first finds the invoking statement of the DL inference framework through static program analysis. Note that we do not need to reconstruct the DL inference framework because it will be replaced by a ready-made framework that supports BP.

Then, ASTM performs precise Android static slicing on the released iApp to extract the IO processing code. The slicing starts at the DL framework’s invoking statement, and it performs backward slicing to extract all statements that determine the value used by the invoking statements and performs forward slicing to extract all statements that use the value defined by the invoking statements.

Next, ASTM generates Python code that can run on PCs based on the sliced IO processing code for model testing. Generating Python code from the sliced APK is feasible because the sliced codes are responsible for IO processing, which is not coupled with the Android platform and system services.

During the model reconstruction, ASTM takes as input an in-App model and outputs its BP-enabled version. ASTM first extracts the computation procedure of the in-App model and eliminates the factors hindering the gradient calculation by undoing the quantization, removing inference-specific operations, and so on.

Then, ASTM utilizes a set of carefully elaborate rules to rebuild the stripped information during the deployment-oriented conversion. The stripped information is mainly the attributes of the operations, such as the filter numbers and kernel size of a convolution operation. Finally, with the rebuilt computation procedure and stripped attributes, ASTM reconstructs the BP-enabled model with a widely-used DL framework Keras².

Combining the code reconstruction and model reconstruction, our ASTM can produce a testing object, equivalent to the iApp’s DL inference module, that can be assessed by various powerful testing techniques. In the next section, we will detail the ASTM’s design. And in Section 6, we will show how to use ASTM to test the in-App models of commercial iApps at scale.

---

2https://keras.io/
to the CG (e.g., Edge ❷ and Edge ❸ in Figure 4), and 2) represents dependency between the statements that read or write the same field in an class (e.g., Edge ❹ in Figure 4).

We denote the built graph as the **slicing basis** because it is able to represent data and control dependencies between statements in an iApp. The data dependency and control dependency between two statements are denoted as $\text{←}_d$ and $\text{←}_c$, respectively.

As for the data dependency, if $\text{use}(\text{stmt}_1) \cap \text{def}(\text{stmt}_2) \neq \emptyset$, $\text{stmt}_1 \text{←}_d \text{stmt}_2$. The data dependencies we consider can be classified into two types. The first type of data dependency is brought by the def-use relationship of local variables in a single function (Edge ❷ in Figure 4). The second type of data dependency is brought by the def-use relationship of filed variables between functions (Edge ❹ in Figure 4).

We consider three types of control dependency. The control dependency in a single function is brought by the procedure branching statement. If $\text{stmt}_1$ can determine whether $\text{stmt}_2$ is executed, $\text{stmt}_2 \text{←}_c \text{stmt}_1$ (Edge ❸ in Figure 4). The control dependency between functions occurs when a function is called (Edge ❶ in Figure 4) or exits (Edge ❹ in Figure 4).

After building the slicing basis, ASTM prepares the slicing criterions for the following App slicing (Step ❶ in Figure 3). Due to different inference tasks and different engineering implementations, the entry point and the exit point of IO processing lack uniform characteristics among different iApps. Therefore, it is difficult for us to locate the entry point and exit point of the IO processing as the slicing criterions. In contrast, the DL framework’s invoking interfaces, which are used to load DL models and perform universal DL computation, have significant static characteristics [54]. ASTM utilizes these invoking interfaces to prepare slicing criterions for the forward and backward slicing, respectively. The backward slicing criterions consist of statements that invoke the DL framework and the values used by the corresponding statement. The goal of backward slicing is to extract all statements that determine the used values. The forward slicing criterions consist of statements that invoke the DL framework and the values defined by the corresponding statement. The goal of forward slicing is to extract all statements that use the defined values.

To find the invoking interfaces of DL framework, we improve the idea used by existing works [45, 51]. If the iApp utilizes an open-sourced DL framework, e.g., TFLite, with well-documented interfaces, we can directly locate statements that invoke the interfaces in slicing basis as the slicing criterions. We find using the invoking interfaces’ parameters and return values and the availability of the invoking methods is robust enough to determine the invoking interfaces when the iApp is protected with string-based obfuscation.

### 4.2 Code Slicing

ASTM proposes an Android slicing technique to extract the user data pre-processing and inference output post-processing code (Step ❶ in Figure 3). The slicing technique can perform bidirectional slicing starting from the found slicing criterions.

We denote the slicing criterion as $<\text{stmt}, V>$. The $\text{stmt}$ in the slicing criterion denotes the invoking statement of DL framework, which is the slicing start point. $V$ represents variables used by $\text{stmt}$ when performing the backwards slicing, i.e., $V = \text{use} (\text{stmt})$. $V$ represents variables defined by $\text{stmt}$ when performing the forwards slicing, i.e., $V = \text{def} (\text{stmt})$. For example, `int[] r = function(String p1, byte[] p2, long[] p3)` is a DL framework invoking statement, denoted as `stmt criterion`. Its def values and use values are `(r)` and `(p1, p2, p3)`.

The slicing algorithm is shown in Algorithm 1. The algorithm takes as input the slicing basis and extracts statements belonging to the processing code by analyzing the data and control dependency. Recall that the slicing basis is the App-level CFG with global variable dependency built in Section 4.1. Each edge of the slicing basis graph represents the data or control dependency between two statements.

When performing backward slicing, if the statement to analyze has a data or control dependency with the sliced statement (line 11), the statement will be added to the sliced results (line 12). And the values defined by the newly-sliced statement can be removed from the slicing criterion, and the values used by the newly-sliced statement should be added to the slicing criterion (line 13). Then the backward slicing goes on (line 14).

When performing forward slicing, if the statement to analyze does not have a data or control dependency with the sliced statement (line 21–22), the values defined by the statement will be removed from the slicing criterion (line 23). If the statement to analyze has a dependency on the sliced statement, the value defined by the statement should be added to the slicing criterion (line 24). In addition to continuing the forward slicing, we should also perform the backward slicing to determine the values used by the newly-added statement (line 29–30, 32).

The slicing ends when all the values can be determined (line 2–3). Or there are no more statements to analyze (line 7–8). The `pred of()` (line 6) and `succ of()` (line 16) represent the predecessor and successor of the given statement on the slicing basis.

### 4.3 Code Generation

After slicing all statements, ASTM generates the executable python code for the following DL model assessment (Step ❷ in Figure 3). The code generation is feasible because the sliced code is for data processing in the DL inference scenario, which is not Android platform-specific. The generation procedure consists of two steps, i.e., statement translation and statement ordering.

#### 4.3.1 Statement Translation

The code slicing works on an intermediate representation (IR) provided by Soot 5. Most of the IR code, e.g., assignment statement and invoking statement, can be easily translated into python code. The IR code needing elaborate processing is the conditional statement, i.e., if statement, and jump statement, i.e., goto statement. This is because the loop structure is also represented as the combination and nesting of the conditional statement and jump statement. We should build the loop structure, especially the loop condition and loop body, to ensure the correctness of data processing.

ASTM adopts the following rule to identify the loop structure. We denote a sliced statement as `stmt` and the CFG of the function that `stmt` belongs to as `G`. The `stmt`’s direct successor statement is

---

5ASTM utilizes Jimple, which is the widely-used intermediate language in Android App analysis.
Algorithm 1: Pseudo code of performing slicing, denote as $\text{slice}()$.

Data: The App-level CFG with global variable dependency, $G$; The slicing criterion, $\langle \text{stmt}_m, \text{stmt}_n \rangle$; The slicing direction, $D$.

Result: The slicing results: $S_{res}$.

1. 
   begin
   2. if $\text{stmt}_n = \emptyset$ then
   3. \hspace{1em} return $S_{res}$
   4. end

5. if $D = \text{"backward"}$ then
   6. $\text{stmt}_n = G.\text{pred}_o(\text{stmt}_m)$
   7. if $\text{stmt}_n = \text{NULL}$ then
   8. \hspace{1em} return $S_{res}$
   9. \hspace{1em} $V_d = \text{def}(\text{stmt}_n)$
   10. \hspace{1em} $V_a = \text{use}(\text{stmt}_n)$
   11. if $V_d \cap V_a \neq \emptyset$ then
   12. \hspace{1em} $S_{res} = \text{add}(\text{stmt}_n)$
   13. \hspace{1em} $V_{sc} = (V_c - V_d) \cup V_a$
   14. \hspace{1em} $S_{res} = \text{add}(\text{slice}(G_b, \langle \text{stmt}_m, V_{sc} \rangle, \text{"backward"}))$
   15. if $D = \text{"forward"}$ then
   16. \hspace{1em} $\text{stmt}_n = G.\text{succ}_o(\text{stmt}_m)$
   17. if $\text{stmt}_n = \text{NULL}$ then
   18. \hspace{1em} return $S_{res}$
   19. \hspace{1em} $V_d = \text{def}(\text{stmt}_n)$
   20. \hspace{1em} $V_a = \text{use}(\text{stmt}_n)$
   21. if $V_d \cap V_a \neq \emptyset$ then
   22. \hspace{1em} if $V_d \cap V_a \neq \emptyset$ then
   23. \hspace{2em} $V_{sc} = V_a - V_d$
   24. \hspace{2em} $S_{res} = \text{add}(\text{slice}(G_b, \langle \text{stmt}_m, V_{sc} \rangle, \text{"forward"}))$
   25. else
   26. \hspace{1em} $S_{res} = \text{add}(\text{stmt}_n)$
   27. \hspace{1em} $V_{sc} = V_a \cup V_d$
   28. \hspace{1em} $S_{res} = \text{add}(\text{slice}(G_b, \langle \text{stmt}_m, V_{sc} \rangle, \text{"forward"}))$
   29. \hspace{1em} $V_{sup} = V_a - V_a$
   30. \hspace{1em} if $\text{stmt}_n = \text{NULL}$ then
   31. \hspace{2em} return $S_{res}$
   32. \hspace{2em} $S_{res} = \text{add}(\text{slice}(G_b, \langle \text{stmt}_m, V_{sup} \rangle, \text{"backward"}))$

According to ASTM’s design, these organized functions have explicit invoking relationships or implicit data dependencies with at least one of the rest functions. Note that the data dependency is brought about by reading or writing the same field. We group these functions according to whether there is a direct or indirect invoking relationship between them. For example, if $f_A() \rightarrow f_B() \rightarrow f_C()$, then $f_A()$, $f_B()$, and $f_C()$ are organized into one function group. We denote the function without any caller in each function group as head functions. In the above example, $f_A()$ is a head function. Now, we only need to order these head functions, and the rest functions in the function groups will automatically be invoked when the head function is called.

To order the head functions, we propose a ‘Write-before-Read’ principle by fully considering the field dependencies among the functions. The principle consists of two rules.

R1: Function groups that do not read any field can be arranged in any order.

R2: For any field, the function group that writes a field should rank before the function group that reads that field.

ASTM first records the filed variable reads and writes of each function group, denoted as $\text{grp}_R$ and $\text{grp}_W$, respectively. The subscript $f$ indicates the function group. Then we arrange the head functions whose function group’s $\text{grp}_R$ is empty in arbitrary order (R1). Once a head function is organized, ASTM updates $\text{grp}_R$ of each of the rest function groups by $\text{grp}_R = \text{grp}_R - \text{grp}_W$. The ASTM will repeatedly order function groups whose $\text{grp}_R$ is empty (R2) until all function groups are arranged.

5 BP-ENABLED MODEL RECONSTRUCTION

The in-App DL model and its framework are primarily designed for efficient inference. Most on-device DL frameworks do not support gradient computation and backpropagation. However, these training-specific computations are the foundation of the white box DL model assessment. Given that all in-App models are converted or compiled from the BP-enabled models, ASTM proposes a BP-enabled model reconstruction technique for better assessment performance.

The model reconstruction technique takes as input an inference-only DL model and produces its corresponding BP-enabled version. It can port the DL model from the inference-only DL framework (e.g., TFLite) to a DL framework that supports model training (e.g., TensorFlow). Recall existing converters have poor support for converting the inference-only model to a BP-enabled one.

DL model can be viewed as a computational graph that defines how to process the inference input. The computational graph is a directed graph to represent the DL computation procedure. The computational graph has three kinds of nodes, i.e., operator node, parameter node, and input node. Operator nodes represent the basic mathematical computation in the DL model, such as convolution, dense, padding, multiply, etc. The operator nodes are characterized by operator type and operator attributes. Parameter nodes represent the BP-enabled parameters of the neural network, e.g., weights and bias. Parameter nodes, input nodes, and operator nodes can feed their value (i.e., tensor) into other operator nodes. Operator nodes compute the output given values for its inputs.
| Operator Type       | Attribute  | Needed Information | Computation Rule |
|---------------------|------------|--------------------|------------------|
| [DepthwiseConv2D]   | filters    | output_shape       | output_shape[-1] |
|                     | kernel_size[0] | weight_shape     | weight_shape[1]  |
|                     | kernel_size[1] | weight_shape     | weight_shape[2]  |
| strides[0]          | input_shape | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[1]  |
| strides[1]          | input_shape | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[2]  |
| padding             | input_shape | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[1]  |
|                     | strides    | output_shape      | output_shape[-1] |
|                     | output_shape| weight_shape     | weight_shape[2]  |
|                     | padding    | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[1]  |
|                     | strides    | output_shape      | output_shape[-1] |
|                     | output_shape| weight_shape     | weight_shape[2]  |
|                     | padding    | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[1]  |
|                     | strides    | output_shape      | output_shape[-1] |
|                     | output_shape| weight_shape     | weight_shape[2]  |
|                     | padding    | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[1]  |
|                     | strides    | output_shape      | output_shape[-1] |
|                     | output_shape| weight_shape     | weight_shape[2]  |
|                     | padding    | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[1]  |
|                     | strides    | output_shape      | output_shape[-1] |
|                     | output_shape| weight_shape     | weight_shape[2]  |
|                     | padding    | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[1]  |
|                     | strides    | output_shape      | output_shape[-1] |
|                     | output_shape| weight_shape     | weight_shape[2]  |
|                     | padding    | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[1]  |
|                     | strides    | output_shape      | output_shape[-1] |
|                     | output_shape| weight_shape     | weight_shape[2]  |
|                     | padding    | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[1]  |
|                     | strides    | output_shape      | output_shape[-1] |
|                     | output_shape| weight_shape     | weight_shape[2]  |
|                     | padding    | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[1]  |
|                     | strides    | output_shape      | output_shape[-1] |
|                     | output_shape| weight_shape     | weight_shape[2]  |
|                     | padding    | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[1]  |
|                     | strides    | output_shape      | output_shape[-1] |
|                     | output_shape| weight_shape     | weight_shape[2]  |
|                     | padding    | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[1]  |
|                     | strides    | output_shape      | output_shape[-1] |
|                     | output_shape| weight_shape     | weight_shape[2]  |
|                     | padding    | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[1]  |
|                     | strides    | output_shape      | output_shape[-1] |
|                     | output_shape| weight_shape     | weight_shape[2]  |
|                     | padding    | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[1]  |
|                     | strides    | output_shape      | output_shape[-1] |
|                     | output_shape| weight_shape     | weight_shape[2]  |
|                     | padding    | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[1]  |
|                     | strides    | output_shape      | output_shape[-1] |
|                     | output_shape| weight_shape     | weight_shape[2]  |
|                     | padding    | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[1]  |
|                     | strides    | output_shape      | output_shape[-1] |
|                     | output_shape| weight_shape     | weight_shape[2]  |
|                     | padding    | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[1]  |
|                     | strides    | output_shape      | output_shape[-1] |
|                     | output_shape| weight_shape     | weight_shape[2]  |
|                     | padding    | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[1]  |
|                     | strides    | output_shape      | output_shape[-1] |
|                     | output_shape| weight_shape     | weight_shape[2]  |
|                     | padding    | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[1]  |
|                     | strides    | output_shape      | output_shape[-1] |
|                     | output_shape| weight_shape     | weight_shape[2]  |
|                     | padding    | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[1]  |
|                     | strides    | output_shape      | output_shape[-1] |
|                     | output_shape| weight_shape     | weight_shape[2]  |
|                     | padding    | output_shape      | output_shape[-1] |
|                     | kernel_size | weight_shape     | weight_shape[1]  |
|                     | strides    | output_shape      | output_shape[-1] |
|                     | output_shape| weight_shape     | weight_shape[2]  |
|                     | padding    | output_shape      | output_shape[-1] |

Table 1: Rules for computing the operators’ attributes. The first column is the operator types; The second column is operator attributes to rebuild; The third column is the information of the corresponding operator in the original model; The last column is the rule to compute the operator’s attributes with the original DL model’s information.

The model reconstruction consists of three key steps. ASTM first extracts the structure of the computational graph. The structure conveys the data dependencies between the nodes and the type of the operator nodes. Then ASTM computes the values of operator attributes and the values of the parameter nodes. Finally, ASTM generates the model code from the constructed computational graph. **Graph Structure Extraction.** Before extracting the structure, we define a representation for the DL model’s computational graph. We build a directed graph, which can represent the data dependency among nodes. The input node is characterized by input shape and output shape. The operator node is characterized by input shapes, output shapes, operator types, and operator attributes.

Output shapes and values characterize the parameter node. ASTM’s graph representation shares the same design with the computational graph of the widely-used Keras, in terms of operators and parameters.

First, ASTM parses the in-App DL model according to get the information about model structures, operators, and parameters. Then, ASTM uses the parsed information to rebuild the structure of the computational graph with our representation. The rebuilt computational graph conveys the completed computation procedure, and types of operator nodes are determined. **Node Value Computation.** After getting the graph structure, ASTM builds the values of operator attributes and the values of...
parameter nodes. Note that operator attributes represent the operator configuration. For example, the convolution’s attributes contain the kernel size, strides, padding strategy, and output channel. The parameter nodes’ values are constants used by operators (e.g., the weights of the convolution operator), and the values are determined during training.

The parameter values in different DL frameworks are represented as tensors with different axis orders. Our representation shares the same axis order with Keras. So, we can compute the parameter values by converting the axis of the in-App model’s parameter values. Some of the parameter values, e.g., weights of convolution operators, are quantified. ASTM undoes the quantification according to the quantization rules. For example, we use $v = s \times (q - z)$ to undo the TFLite’s quantization, where the $v$ is the reversed weights, $q$ is the quantized value, $s$ denotes the scale coefficient, and $z$ denotes the zero point value.

As for the attribute values, ASTM proposes a rule-based computation method by fully considering the operators’ computation characteristics. We can obtain the attribute values through the DL framework-independent information. For example, when computing the attribute values of a convolution operator, the kernel size can be obtained through the output shape of the convolution operator’s input parameter nodes.

Part of the rules used to compute the representative attribute values is summarized in Table 1. The needed information is the information used to compute the attribute values. The needed information can be collected from the corresponding operator in the in-App model. The input_shape denotes the shape of the corresponding operator’s input tensor. The output_shape denotes the shape of the corresponding operator’s output tensor. The weight_shape denotes the output_shape of the corresponding operator’s input parameter node.

**Model Code Generation.** ASTM can finally generate the model code with the constructed computational graph. The generated model code is the same as that used to develop a DL model and load the trained model weights. The model code has three parts, i.e., operator initialization, flow construction, and weight loading. The operator initialization code is to utilize the operator type and attributes to create operator instances. The flow construction code is to utilize the graph structure and operator instances to construct the model’s data flow. The weights loading code is to assign the trained parameters to the operator instances.

Without loss of generality, we choose the widely-used Keras as the DL framework to reconstruct the BP-enabled model. Algorithm 2 shows how ASTM generates the Keras model code. The algorithm takes as input the computational graph constructed before. First, ASTM adds all initialization code for every operator node (line 1–4). Then, it performs the topological sort on the computational graph to determine the execution order of the operators (line 5). Next, it generates the forward-pass code according to the operator execution order (line 6–7). ASTM adds a fragment of template code to construct the DL model instance (line 8). Finally, we add the code that loads the weights of the reconstructed model and assigns them to the corresponding operator node. (line 9–10).

---

**Algorithm 2: Model Code Generation.**

| Data: The computational graph: |
|--------------------------------|
| $\text{graph} = [\text{node}_1, \text{node}_2, \ldots, \text{node}_n]$ |
| Result: code |
| code $=$ []  
| for node in $\text{graph}$ do |
| if node.nodeType $== 0$ then |
| code $\cdot$ append(genInitCode(node)) |
| sortedNodeList = topologySort(graph) |
| for node in sortedNodeList do |
| code $\cdot$ append(genForwardCode(node)) |
| code $\cdot$ append(modelBuildingCode()) |
| for node in $\text{graph}$ do |
| code $\cdot$ append(genWeightInitCode(node)) |

### 6 EVALUATION

We implement the ASTM where the code reconstruction consists of 6.3K lines of Java code, and the model reconstruction consists of 1.9K lines of Python code.

In this section, we first perform a large-scale study on all found iApps. We then perform code and model reconstruction on 100 in-App models using the top two frameworks, i.e., TFLite and TensorFlow. The selected models perform vision-related tasks, which is the most widely used kind of task in iApps [51]. Next, we use ASTM to perform the robust assessment of the reconstructed results to show the effectiveness of our ASTM. Finally, we perform three representative physical adversarial attacks to demonstrate the ASTM’s security meaning.

#### 6.1 iApp Statistics

There are 25k APKs downloaded from five App markets, i.e., APKPure [9], 360 App Store [1], Baidu App Store [3], Xiaomi App Store [5], and Anzhi Market [2], in June 2021. We downloaded about 5k the most popular Apps on each market. After removing the duplicated Apps according to the iApps’ MD5, there are about 15k Apps left.

We find 3,064 iApps from the downloaded APKs. And the number of in-App models is 3,845. Some of these models are repeated in different iApps because they are open source or from the same intelligent SDK, e.g., Volcengine(from ByteDance) [14] or SenseTime [42]. After removing duplicate models, we find 800 unique in-App models. We count these in-App models according to the DL framework in Table 2.

We first divide these unique models into two categories according to the DL framework. One kind of framework is the open-source DL framework, e.g., TFLite. The other kind of framework is developed for private or licensed use by commercial companies. Among all the in-App models using open-source frameworks, the models using tflite and tensorflow frameworks account for 67.5% of the models’ total number.

Of all models using private frameworks, the models utilizing frameworks provided by Volcengine and SenseTime account for 74.8%. We find that although the number of these models is large, these models’ functionalities are similar. About 80% of the models’ functionalities are related to the feature detection of face and body, e.g., face landmark detection, pose detection, and face detection.
Table 2: Statistics on on-device DL inference frameworks. The last column counts the unique models using the corresponding framework.

| Category       | Framework/SDK                      | Count |
|----------------|------------------------------------|-------|
| Open-source framework (372) | TensorFlow [7] | 128   |
| Other SDK (428)      | TFLite [6]                     | 125   |
|                  | NCNN [47]                        | 42    |
|                  | Caffe [27]                       | 35    |
|                  | MNN [28]                         | 32    |
|                  | TNN [48]                         | 7     |
|                  | ONNX [4]                         | 3     |
|                  | Voleengine(ByteDance) [14]       | 167   |
|                  | SenseTime [42]                   | 153   |
|                  | Kwai [30]                        | 22    |
|                  | MindSpore(Huawei) [25]           | 13    |
|                  | Huya [26]                        | 7     |
|                  | Meishe [39]                      | 4     |
|                  | Unknown                          | 62    |

We find that the reason for a large number of such models is that different iApps use different versions of SDKs provided by these companies, and the in-App models of the same functionalities will also be updated with the SDK.

We also count the model sizes of different DL frameworks. We find the median and mean size of in-App models are 396 KB and 1,270 KB, respectively. And about 90% of the models are less than 3 MB in size.

6.2 Code Reconstruction Evaluation

| Model Task       | Count | Mean | Median |
|------------------|-------|------|--------|
| Style Transformation | 54 | 1059.9 | 530.8 |
| Classification    | 11   | 19484.5 | 8719.2 |
| Object Detection  | 11   | 8063.2 | 6757.2 |
| Super Resolution  | 8    | 2309.4 | 1974   |
| Semantic Segmentation | 7 | 6796.1 | 2714.9 |
| OCR               | 3    | 4435.0 | 2134.8 |
| Text Detection    | 3    | 4905.3 | 2711.3 |
| Pose Estimation   | 1    | 2304.6 | 2304.6 |
| Depth Estimation  | 1    | 252.2  | 252.2  |
| Face Comparison   | 1    | 89157.7 | 89157.7 |

Table 3: Statistics on the selected 100 unique DL models by task. The second column counts the amount of models for each task. The third and forth column shows the mean and median of the model size (KB).

We perform a study on the sliced processing code and find that (1) IO processing mainly focuses on data type conversion, e.g., image to an array, data size adjustment, data cropping, data normalization, etc. 58% of the iApps resize the input image to less than 512×512px.

(2) Almost all iApps use a group of loop statements to process the image by pixel. (3) The mean and median LoC of the sliced code is 1,971 and 274. 60% of the reconstructed codes’ LoC is less than 308.

6.2.2 Representative Cases. We perform three case studies to show the effectiveness of the code reconstruction.

Case 1. Bundle ID: uk.tensorzoom. This DL model is used to carry out image super-resolution. The post-processing code is responsible for converting the iApp’s inference result to an image (Figure 5). The inference result is a float array, and each float number represents an int number (line 5) to represent a pixel with the ARGB channels, where each channel is represented by 8 bits, and the value ranges from 0 to 255. To prevent numerical overflow when converting the results, the post-processing code performs a min-max normalization (line 12~19) on the inference results.

Case 2. Bundle ID: ru.photostrana.mobile. This DL model is used to carry out face segmentation. The pre-processing code is responsible for converting the input image to a float array (Figure 6). This procedure involves separating the RGB channels of a pixel (line 12~13), normalizing the RGB values (line 14~15), and putting
them into the array in a specific order (line 16). We find that most
iApps store the input in the order of height x width x channel, and
this iApp stores the input in the order of channel x width x height.
If the preset channel order cannot be followed, the inference result
will be wrong.

**Case 3. Bundle ID: com.blink.academy.nomo.** This DL model is
used to carry out a 21-class semantic segmentation. We present the
post-processing code in Figure 7. As shown in line 13–14, the iApp
only visualizes a certain class (the 15th class presents the person).
Other segmentation results are discarded by the post-processing
code.

### 6.3 Robustness Assessment

#### 6.3.1 Testing Methods

We utilize the PGD (Project Gradient Descent) attack [38] to test the in-App model’s robustness. PGD is an iterative attack method that can search for a subtle perturbation to fool the DL model. Here, we formulate the PGD attack. We denote

\[
x^{(t+1)}_{adv} = \text{Proj}_{x} + \alpha \text{sgn}(\nabla_{x} \mathcal{L}(x^{(t)}_{adv}, y))
\]

where \(\mathcal{L}\) is the loss function, \(C(\cdot)\) represents the model output of \(x^{(t)}_{adv}\), and \(\text{sgn}(\cdot)\) is the sign function. \(\text{Proj}\) is a projection function which can project the input to the hypersphere \(x + S\).

Assume the original input is \(x\), we can represent the final adversarial example as:

\[
x^*_{adv} = x + \epsilon
\]

Where \(\epsilon\) is the attack budget which means the manipulation limit
for the input. We use MSE (Mean Square Error) as the loss function
to generate the adversarial examples for different tasks.

We collect the dataset for each in-App model. Each dataset consists of
20 images. Some datasets are the subset of the open source datasets,
e.g., COCO [35], VOC [18], and CelebA [37], and the other datasets are
prepared by ourselves through the image search engine. We release sample datasets on the anonymous website
https://github.com/anonymous4896/public_data.

#### 6.3.2 Metrics

We measure the in-App models’ robustness through the following metrics for different kinds of tasks. The larger the metric value is, the higher the attack success rate is, and the worse the robustness of the in-App model. We propose four metrics according to the way of counting failure cases of different kinds of tasks.

**Type-1 Task.** The first type of task performs classification, OCR,
and face comparison. The metric of type-1 task is

\[
\frac{1}{N} \sum_{i=1}^{N} \mathbb{1}[y_i \neq y'_i]
\]

It measures the percentage of failure cases. \(N\) is the number of the
testing samples. The \(y'_i\) denotes the inferred label of \(x_i\), i.e., \(y'_i = f(x_i)\). The \(y'_i\) denotes the \(x_i + \epsilon\)’s inferred label, i.e., \(y'_i = f(x_i + \epsilon)\).

In our context, if the value of the metric is larger than 0.6, we say
the model is detected with robustness issues. Note that 0.6 represents
that 60% of testing inputs are misclassified at least.

**Type-2 Task.** The second type of task performs object detection
test. Assume the original bounding boxes are \(B = (b_1, b_2, ..., b_n)\) of all
testing samples, we calculate

\[
\frac{1}{n} \sum_{i=1}^{n} \mathbb{1}[c(b_i) \neq c(b'_i)]
\]

as the metric of type-2 task. The \(c(b)\) denotes the classification results of the bounding box \(b\). The \(b\) denotes the bounding box detected on the original input, and the \(b'\) denotes the bounding box detected on the attacked results. Note that the value of \(b_i\) may not be the same as that of \(b'_i\).

In our context, if the value of the metric is larger than 0.6, we say
the model is detected with robustness issues. Note that 0.6 represents
that 60% of bounding boxes are misclassified at least.

**Type-3 Task.** The third type of task performs semantic segmentation,
depth estimation, and pose estimation. For an image of size

\[
\text{Figure 6: Part of the reconstructed code of Case 2.}
\]

\[
\text{Figure 7: Part of the reconstructed code of Case 3.}
\]
Figure 8: The robustness testing results of 100 unique in-App models by performing PGD attacks. We report the results by the type of tasks. The $\epsilon$ is the attack budget. Testing iteration number represents the iteration times of the attack. The model count denotes the number of models detected to have robustness issues.

$\begin{align*}
\text{Type-1 Task (Total: 15)} & : \\
\epsilon = 8 & : 5, \\
\epsilon = 12 & : 10, \\
\epsilon = 16 & : 4, \\
\epsilon = 20 & : 2, \\
\epsilon = 24 & : 0.
\end{align*}$

$\begin{align*}
\text{Type-2 Task (Total: 14)} & : \\
\epsilon = 8 & : 5, \\
\epsilon = 12 & : 10, \\
\epsilon = 16 & : 4, \\
\epsilon = 20 & : 2, \\
\epsilon = 24 & : 0.
\end{align*}$

$\begin{align*}
\text{Type-3 Task (Total: 9)} & : \\
\epsilon = 8 & : 5, \\
\epsilon = 12 & : 10, \\
\epsilon = 16 & : 4, \\
\epsilon = 20 & : 2, \\
\epsilon = 24 & : 0.
\end{align*}$

$\begin{align*}
\text{Type-4 Task (Total: 62)} & : \\
\epsilon = 8 & : 5, \\
\epsilon = 12 & : 10, \\
\epsilon = 16 & : 4, \\
\epsilon = 20 & : 2, \\
\epsilon = 24 & : 0.
\end{align*}$

$\begin{align*}
\text{M} \times N, \text{the attack effect is measured with}
\begin{align*}
\frac{1}{MN} \sum_{i,j} [s_{ij} \neq s'_{ij}]
\end{align*}
\end{align*}$

The $p_{ij}$ and $p'_{ij}$ denote the $i$-th row and $j$-th column pixel of original image and the perturbed image, respectively. The $s_{ij}$ is the semantic label of pixel $p_{ij}$, and the $s'_{ij}$ is the semantic label of the adversarial pixel $p'_{ij}$.

In our context, if the value of the metric is larger than 0.6, we say the model is detected with robustness issues. Note that 0.6 represents that 60% of pixels are wrongly labeled at least.

**Type-4 Task.** The fourth kind of task performs style transformation and super resolution. The structural similarity index (SSIM) measures the perceived quality of the attack results compared to the original results. We use

$1 - \text{SSIM}(f(x), f(x + \epsilon))$

as the metric to measure the decrease of the SSIM value. The $f(\cdot)$ denotes the task. $f(x)$ and $f(x + \epsilon)$ are the outputs of a type-4 task, when it takes as input the $x$ and $x + \epsilon$.

In our context, if the value of the metric is larger than 0.6, we say the model is detected with robustness issues. Note that 0.6 represents that attack makes the SSIM value drop by 0.6 at least.

6.3.3 Testing Results. For each type of task, we experiment with five $\epsilon$ (in Equation 2) values and two iteration numbers. Figure 8 shows the testing results with different experimental settings. We count the models has robustness issues by tasks.

When performing the testing through the PDG configured with $\epsilon=8$ and iteration number=10, robustness issues are detected in 56% of models (56 out of 100). As the value of $\epsilon$ increases from 8 to 16, the number of models that are detected with robustness issues increases from 56 to 78. When we double the testing iterations (from 10 iterations to 20 iterations), we find the number of models detected with robustness issues is stable, increasing by about 14.2%. We show some representative attack results in Figure 9.

6.4 Detected Physical Attacks against iApps

We use ASTM to detect the security issues of three iApps by performing physical adversarial attacks.

**Case 1. Bundle ID: com.seefoodtechnologies.nothotdog.**
Knife (attacked by searched tablecloth)  Knife  Hotdog

The confidence of recognizing it as hotdog is 0.97
The confidence of recognizing it as hotdog is 0.01
The confidence of recognizing it as hotdog is 1.00

Figure 10: The physical attack results of case 1.

This iApp can recognize whether there is a hot dog in the environment. We perform a physical attack on the in-App model. We implement that if a tablecloth with a special pattern is placed on the table, the iApp will recognize the knife on the table as a hot dog. When visually impaired people use the iApp to identify hot dogs, they face serious safety issues by mistakenly holding a knife. We use ASTM to reconstruct the BP-enabled DL model and processing code of the hot dog detection module. Then we take a photo of the knife, segment the knife out, and generate 50 images of the knife of different sizes and rotation angles. Then we search for a specific grid background by using project gradient descent so that when we put the knife images on the grid background, the composed image is recognized as a hot dog. The loss function used to compute the gradient is the L1 norm between the hot dog’s label and the recognition results of the composed image. We perform 100 rounds of the project gradient descent.

The attack results are shown in Figure 10. We show the recognition results before and after putting the knife on the searched grid tablecloth. The searched grid tablecloth makes the confidence in viewing a knife as a hot dog from 0.006 to 0.97. As a reference, the confidence of a real hot dog is 0.99.

Case 2. Bundle ID: com.sogou.map.android.maps.

This iApp can be used for road navigation. By using the ASTM, we find the lane detection module in the iApp will fail when there is a paper tape in a special location on the road. The failure of lane detection can lead to serious safety problems, such as vehicles suddenly stopping or driving out of lanes.

We first use ASTM to reconstruct the BP-enabled DL model and processing code of the lane detection module. Then we choose a road and take 20 photos of the road with different camera poses. Next, we search the length and position of the paper tape, which can fail the lane detection module by using project gradient descent. The loss function used to compute the gradient is the L1 norm between the ground truth and the attack target. The attack target is to make the paper tape recognized as a lane while making the actual lane location undetectable. We perform 50 rounds of the paper tape’s length and position searches.

We show the security issues detect by ASTM in Figure 11. We visualize the lane detection results on the image. The first row shows the original view and three attacks. The second row shows the corresponding attack results. The paper tape can change the lane detection results. These attacks cause the lane to be missed, or the lane’s type to be incorrectly identified and brings serious security risks.

Case 3. Bundle ID: com.mensajerosurbanos.app.mensajero.

This iApp can identify receipts. By using ASTM, we find that if the receipt utilizes a special background, it can evade the detection without affecting the user’s reading. The iApp’s competitor can cooperate with the company that prints the receipt to print the receipt with a specific background. They can degrade the receipt detection performance and cause the iApp to lose users.

We use ASTM to reconstruct the BP-enabled DL model and processing code of the receipt recognition module. We then search a background used to print the receipt, which can fool the receipt recognition. In order to reduce the impact of the receipt background on the readability of the receipt content, we use crossed dotted lines to form the background. Then we search for the number of dashed lines, the rotation angle, and the dashed line grayscale using the projected gradient descent. The loss function used to compute the gradient is the L1 norm between the receipt’s label and the recognition results of the receipt with the searched background. We perform 50 rounds of the project gradient descent.

The attack results are shown in Figure 12. We show the receipt recognition results before and after adding the searched background. The searched background makes the confidence in the receipt recognition drop from 0.99 to 0.096. As a reference, the confidence of a receipt with random background is 0.99.

7 CONCLUSION

This work proposes ASTM to enable auto testing of iApps for App markets by two novel reconstruction techniques. The experimental results show that the ASTM can successfully reconstruct runnable IO processing code and BP-enabled DL models from commercial iApps. We perform a large-scale robustness assessment on the in-App models with the ASTM’s help. ASTM also detects three representative real-world attacks against iApps. We believe ASTM can be further used to enable finding new attack surfaces lying in the coupling between code and DL models.

REFERENCES

[1] [n.d.]. 360 App Store. https://ext.se.360.cn/.
[2] [n.d.]. Anzhi Market. http://www.anzhi.com/.
[3] [n.d.]. Baidu App Store. https://shouji.baidu.com/.
[4] [n.d.]. ONNX. https://onnx.ai/.
[5] [n.d.]. Xiaomi App Store. https://app.mi.com/.
[6] 2021. TensorFlow Lite. https://www.tensorflow.org/lite
[7] Martin 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, William Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.
[8] Mario Almeida, Stefanos Laskaridis, Abladein Mehrotra, Lukasz Dudziak, Ilias Leontiadis, and Nicholas D Lane. 2021. Smart at what cost? Characterising Mobile Deep Neural Networks in the wild. In Proceedings of the 21st ACM International Conference on Measurement Conference.
[9] APKPure. [n.d.]. Download APK free online downloader | APKPure.com. https://apkpure.com/.
[10] Tanzirul Azim, Arash Alavi, Iulian Neamtiu, and Rajiv Gupta. 2019. Dynamic slicing for android. In International Conference on Software Engineering (ICSE).
Figure 11: The physical attack results of case 2.

Figure 12: The physical attack results of case 3.

[11] Anthony Berthelier, Thierry Chateau, Stefan Duffner, Christophe Garcia, and Christophe Blanc. 2021. Deep model compression and architecture optimization for embedded systems: A survey. Journal of Signal Processing Systems (2021).

[12] Ketan Bhardwaj, Matt Saunders, Nikita Jansea, and Ada Gavrilovska. 2019. Serving mobile apps: A slice at a time. In Proceedings of the Fourteenth EuroSys Conference (EuroSys).

[13] 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).

[14] Bytedance. [n.d.]. Volcengine. https://www.volcengine.com/.

[15] Pin-Yu Chen, Yash Sharma, Huan Zhang, Jinfeng Yi, and Cho-Jui Hsieh. 2018. Ead: elastic-net attacks to deep neural networks via adversarial examples. In Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 32.

[16] Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Min Wu, and Xingping Yi. 2020. A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability. Computer Science Review (2020).

[17] Yujin Huang, Han Hu, and Chunyang Chen. 2021. Robustness of on-device models: Adversarial attack to deep learning models on android apps. In 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) IEEE.

[18] Huawei. [n.d.]. MindSpore. https://www.mindsponge.cn/.

[19] HUYA. [n.d.]. huya. https://www.huya.com/.

[20] Yansong Gao, Bao Gia Doan, Zhi Zhang, Siqi Ma, Jiliang Zhang, Anmin Fu, Surya Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, and Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. 2010. Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and In Cetus Users and Compiler Infrastructure Workshop (CETUS).

[21] Xiaowei Huang, Daniel Kroening, Wenjie Ruan, James Sharp, Youcheng Sun, Anthony Berthelier, Thierry Chateau, Stefan Duffner, Christophe Garcia, and Christophe Blanc. 2021. Deep model compression and architecture optimization for embedded systems: A survey. Journal of Signal Processing Systems (2021).

[22] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. 2021. Knowledge distillation: A survey. International Journal of Computer Vision (2021).

[23] Xiaowei Huang, Daniel Kroening, Wenjie Ruan, James Sharp, Youcheng Sun, Enese Thamo, Min Wu, and Xingping Yi. 2020. A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability. Computer Science Review (2020).

[24] Yu Liu, Cheng Chen, Ru Zhang, Tingting Qin, Xiang Ji, Haoxiang Lin, and Mao Yang. 2020. Enhancing the interoperability between deep learning frameworks by model conversion. In Proceedings of the 20th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM.

[25] Huawei. [n.d.]. MindSpore. https://www.mindsponge.cn/.

[26] Uday Kamath, John Liu, and James Whitaker. 2019. Deep learning for NLP and speech recognition. Vol. 84. Springer.

[27] Kuaishou. [n.d.]. Kwai, Fantastic Social Video Network. https://www.kwai.com/.

[28] Alexey Kurakin, Ian Goodfellow, Samy Bengio, et al. 2016. Adversarial examples explained. In Proceedings of the 2016 International Conference on Machine Learning (ICML):

[29] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In In Cetus Users and Compiler Infrastructure Workshop (CETUS).

[30] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. 2021. Pruning and quantization for deep neural network acceleration: A survey. Neurcomputation (2021).

[31] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Xian Zhang, and Hyoungshick Kim. 2020. Backdoor attacks and countermeasures on mobile apps: A slice at a time. In Proceedings of the 2020 ACM Conference on Computer and Communications Security (CCS)

[32] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In In Cetus Users and Compiler Infrastructure Workshop (CETUS).

[33] 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. arXiv preprint arXiv:1408.5093 (2014).

[34] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Pin-Yu Chen, Yash Sharma, Huan Zhang, Jinfeng Yi, and Cho-Jui Hsieh. 2018. Ead: elastic-net attacks to deep neural networks via adversarial examples. In Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 32.

[35] Anthony Berthelier, Thierry Chateau, Stefan Duffner, Christophe Garcia, and Christophe Blanc. 2021. Deep model compression and architecture optimization for embedded systems: A survey. Journal of Signal Processing Systems (2021).

[36] Yu Liu, Cheng Chen, Ru Zhang, Tingting Qin, Xiang Ji, Haoxiang Lin, and Mao Yang. 2020. Enhancing the interoperability between deep learning frameworks by model conversion. In Proceedings of the 20th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM.

[37] Meishe. [n.d.]. Meicam. https://www.meishesdk.com/.

[38] Ziyi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. 2014. Microsoft coco: Common objects in context. In European Conference on Computer Vision (ECCV).

[39] Yu Liu, Cheng Chen, Ru Zhang, Tingting Qin, Xiang Ji, Haoxiang Lin, and Mao Yang. 2020. Enhancing the interoperability between deep learning frameworks by model conversion. In Proceedings of the 20th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM.

[40] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. 2015. Deep Learning Face Recognition. Vol. 84. Springer.

[41] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In In Cetus Users and Compiler Infrastructure Workshop (CETUS).

[42] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. 2021. Pruning and quantization for deep neural network acceleration: A survey. Neurcomputation (2021).

[43] 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. arXiv preprint arXiv:1408.5093 (2014).

[44] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Pin-Yu Chen, Yash Sharma, Huan Zhang, Jinfeng Yi, and Cho-Jui Hsieh. 2018. Ead: elastic-net attacks to deep neural networks via adversarial examples. In Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 32.

[45] Anthony Berthelier, Thierry Chateau, Stefan Duffner, Christophe Garcia, and Christophe Blanc. 2021. Deep model compression and architecture optimization for embedded systems: A survey. Journal of Signal Processing Systems (2021).

[46] Yu Liu, Cheng Chen, Ru Zhang, Tingting Qin, Xiang Ji, Haoxiang Lin, and Mao Yang. 2020. Enhancing the interoperability between deep learning frameworks by model conversion. In Proceedings of the 20th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM.

[47] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. 2015. Deep Learning Face Recognition. Vol. 84. Springer.

[48] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. 2021. Pruning and quantization for deep neural network acceleration: A survey. Neurcomputation (2021).

[49] 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. arXiv preprint arXiv:1408.5093 (2014).

[50] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Pin-Yu Chen, Yash Sharma, Huan Zhang, Jinfeng Yi, and Cho-Jui Hsieh. 2018. Ead: elastic-net attacks to deep neural networks via adversarial examples. In Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 32.
[42] SenseTime. [n.d.]. SenseTime. https://www.sensetime.com/.
[43] Shashi Pal Singh, Aja Kumar, Hemant Darbari, Lenali Singh, Anshika Rastogi, and Shikha Jain. 2017. Machine translation using deep learning: An overview. In 2017 international conference on computer, communications and electronics (compelx) IEEE, 162–167.
[44] Manu Sridharan, Stephen J Fink, and Rastislav Bodik. 2007. Thin slicing. In Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. 112–122.
[45] Zhichuang Sun, Ruimin Sun, Long Lu, and Alan Mislove. 2021. Mind your weight (s): A large-scale study on insufficient machine learning model protection in mobile apps. In 30th USENIX Security Symposium (USENIX Security).
[46] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. arXiv preprint arXiv:1312.6199 (2013).
[47] Tencent. [n.d.]. ncnn. https://github.com/Tencent/ncnn.
[48] Tencent. [n.d.]. TNN. https://github.com/Tencent/TNN.
[49] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, and Efthy-chios Protopapadakis. 2018. Deep learning for computer vision: A brief review. Computational intelligence and neuroscience 2018 (2018).
[50] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao. 2019. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In 2019 IEEE Symposium on Security and Privacy (SP).
[51] Mengwei Xu, Jiawei Liu, Yuanqiang Liu, Felix Xiaoouzhu Lin, Yunxin Liu, and Xu-anzhc Liu. 2019. A first look at deep learning apps on smartphones. In Proceedings of the Web Conference.
[52] Lei Xue, Hao Zhou, Xiaopu Luo, Le Yu, Dinghao Wu, Yajin Zhou, and Xiaobo Ma. 2020. Packrgrind: An adaptive unpacking system for android apps. IEEE Transactions on Software Engineering (TSE) (2020).
[53] Lei Xue, Hao Zhou, Xiaopu Luo, Yajin Zhou, Yang Shi, Guo Fei Gu, Fengwei Zhang, and Man Ho Au. 2021. Happer: Unpacking Android apps via a hardware-assisted approach. In 2021 IEEE Symposium on Security and Privacy (S&P).
[54] Qiyang Zhang, Xiang Li, Xiangying Che, Xiao Ma, Ao Zhou, Mengwei Xu, Shang-guang Wang, Yun Ma, and Xuanzhe Liu. 2022. A Comprehensive Benchmark of Deep Learning Libraries on Mobile Devices. arXiv preprint arXiv:2202.06512 (2022).
[55] Quan-shi Zhang and Song-Chun Zhu. 2018. Visual interpretability for deep learning: a survey. Frontiers of Information Technology & Electronic Engineering (2018).