DeMistify: Identifying On-device Machine Learning Models Stealing and Reuse Vulnerabilities in Mobile Apps

Pengcheng Ren[1], Chaoshun Zuo[2], Xiaofeng Liu[1], Wenrui Diao[1]
Qingchuan Zhao[3(✉)], and Shanqing Guo[1(✉)]
[1]School of Cyber Science and Technology, Shandong University
{rpc,xiaofengliu}@mail.sdu.edu.cn, diaowenrui@link.cuhk.edu.hk, guoshanqing@sdu.edu.cn
[2]Ohio State University, zuo.118@osu.edu
[3]City University of Hong Kong, qizhao@cityu.edu.hk

ABSTRACT

Mobile apps have become popular for providing artificial intelligence (*AI*) services via on-device machine learning (*ML*) techniques. Unlike accomplishing these AI services on remote servers traditionally, these on-device techniques process sensitive information required by AI services locally, which can mitigate the severe concerns of the sensitive data collection on the remote side. However, these on-device techniques have to push the core of ML expertise (*e.g.*, models) to smartphones locally, which are still subject to similar vulnerabilities on the remote clouds and servers, especially when facing the model stealing attack. To defend against these attacks, developers have taken various protective measures. Unfortunately, we have found that these protections are still insufficient, and on-device ML models in mobile apps could be extracted and reused without limitation. To better demonstrate its inadequate protection and the feasibility of this attack, this paper presents DeMistify, which statically locates ML models within an app, slices relevant execution components, and finally generates scripts automatically to instrument mobile apps to successfully steal and reuse target ML models freely. To evaluate DeMistify and demonstrate its applicability, we apply it on 1,511 top mobile apps using on-device ML expertise for several ML services based on their install numbers from Google Play and DeMistify can successfully execute 1250 of them (82.73%). In addition, an in-depth study is conducted to understand the on-device ML ecosystem in the mobile application.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Android App; Machine Learning; On-device Model Reuse; Program Analysis

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

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3623325

ACM Reference Format:
Pengcheng Ren, Chaoshun Zuo, Xiaofeng Liu, Wenrui Diao, Qingchuan Zhao, and Shanqing Guo. 2024. DeMistify: Identifying On-device Machine Learning Models Stealing and Reuse Vulnerabilities in Mobile Apps. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3623325

1 INTRODUCTION

Recently, we have witnessed an ongoing practice in mobile apps to run machine learning (*ML*) models directly on users' smartphones instead of relying on a remote server or the cloud to deliver ML services, including selfie retouching, biomedical screen locking, speech recognition, *etc.* Primary advantages of this practice may include better privacy since there is no sensitive data going outside the smartphone [48], faster performance because it has no dependency on the quality of network connection, and larger usage scenarios where there is even no requirement on the network's connectivity.

Unfortunately, this inevitable integration of ML models and mobile apps exposes these on-device models at risk of leakage because their execution environment (*e.g.*, mobile operating systems and other apps running on the same smartphone) is rarely trustworthy. However, previous server-side model-stealing attacks (*e.g.*, training a surrogate model on the server-side leveraging reverse engineering relevant remote APIs [11, 23, 39–41, 49, 51, 56]) and the corresponding defense mechanisms (*e.g.*, [7, 16, 26, 27, 37]) can hardly apply in this new attack scenario.

Moreover, although there are a few recent works that attempt to shed light on this new attack scenario, they rely on heavy manual effort to extract these models from a limited number of apps [8], which use particular on-device ML frameworks (*i.e.*, TFLite [29]), automatically extract on-device models that are implemented specifically [48] (*e.g.*, model in plaintext at rest or being decrypted after being loaded into the memory), or evaluate the robustness of these models against adversary attacks. For example, [12] utilizes standard API to extract model information to train a surrogate model, [22] [21] takes a similar TensorFlow Hub fine-tuned model as an alternative. Even in these specific implementations, these approaches also be hindered because their strict assumptions are hard to scale. Specifically, they assume that *(i)* models in the plaintext are always interpretable, while a large number of apps use custom format making their plaintext infeasible to be analyzed, or *(ii)* the Java codes of ML-related services are not obfuscated and uses the standard APIs, which could not be held since most

apps adopt identifier renaming to obfuscate and encapsulate such APIs, or *(iii)* ML-related services could always be accessible and triggered manually in the user app, which may be impossible to be achieved; for example, an app requires login via facial recognition, while registration and face capture procedures are taken elsewhere.

Although defense solutions [20, 33, 47, 58] have been proposed and partially deployed, they still fall short in countering model reuse attacks, which have not yet been considered and expose such key intellectual property at risk. For software vendors, model reuse causes property issues, and for developers, adversarial attacks caused by model reusing expose users to security risks. For example, attackers can identify applications with reuse attack vulnerabilities through large-scale analysis, deploy these apps in the cloud, encapsulate an API for free use to cause financial loss, or infinite query the reused scripts, and construct adversarial examples to have a critical failure (*e.g.*, fingerprinting bypass). These reuse attacks have not been taken seriously. Although developers have adopted a variety of protections and believe it is security enough, such as model encryption and code obfuscation to prevent models from being stolen and exploited, in fact, these are not robust to reuse attacks. Considering the rapid growth of mobile apps deploying on-device models, *it is urgent to enable a comprehensive understanding of the inadequate protection against ML model reuse attacks.*

Though manually identifying and verifying such risks in individual apps are helpful; however, such tedious efforts are impractical to scale in practice due to the millions of apps available on the market. Therefore, in this paper, we propose DeMistify, an automated tool to demonstrate the feasibility of on-**De**vice **M**achi**n**e learning model **st**ealing and ident**ify** the corresponding model reusing vulnerabilities in mobile apps. At a high level, we aim to leverage obfuscation-resilient program slicing expertise to overcome the aforementioned limitation in related works. Specifically, DeMistify consists of three components: *(i) Model Locating* takes an app as its input, decompiles and statically analyzes its decompiled resources to detect the presence of on-device ML models and native libraries that process them. Next, according to the detected native libraries, *(ii) Model Execution Slicing* extracts all components in the minimum necessary that are related to the execution of those models by identifying the boundary functions at the Java level. *(iii) Model Reusing* takes testing cases as input, automatically generates scripts to reuse related ML services by dynamically instrumenting corresponding mobile apps, and finally returns the associated results for verification.

To show the vulnerability of the model reusing, DeMistify is evaluated with 1,511 top mobile apps using on-device ML expertise for several ML services based on their install numbers from Google Play as of September 2021 and successfully rerun 1250 of them (82.73%). In addition, an in-depth study is conducted to better understand the corresponding ecosystem of these on-device ML services. In particular, it reveals the current protection status of ML models and analyzes the robustness of their protection, such as 821 ML apps adopt customized formats, but 88.43% can be exploited by reuse attacks. Note that, DeMistify overcomes restrictions on the test attempts (*e.g.*, **query limitations** on servers), and that 90.3% of exploits contain sufficient low-dimensional information, which makes it promising in security applications including training dataset recovery and vulnerability identification.

Contribution. This paper contains the following contributions:

- **Novel Problem**: This paper aims to address a novel research problem, *i.e.*, how to successfully steal on-device ML models and correctly reuse them with relevant functional modules in a given mobile app with accurate results. This problem has not been well studied and becomes urgent to be addressed due to the significant increase of this on-device service in apps.
- **New Tool**: This paper presents a new tool, *i.e.*, DeMistify[1], that automatically steals on-device models in mobile apps and reuses associated services by slicing the minimum necessary components. It proposes a suite of automated programming analysis techniques to statically slice the necessary boundary functions and dynamically instrument mobile apps to reuse ML services alongside their models.
- **Empirical Evaluation**: DeMistify has evaluated with 1,511 top mobile apps using on-device ML services based on their install numbers from Google Play as of September 2021. It has successfully instrumented 1,511 apps with testing inputs and succeeded in 1250 apps obtaining meaningful results. In addition, an in-depth study on the ecosystem of on-device ML services in the Android reveals novel insights.

2 BACKGROUND

2.1 On-device ML Practice in Mobile Apps

With the widely used of on-device ML frameworks and SDKs, the development of mobile apps offering related services has become more efficient. Whether utilizing commercial or customized solutions, these on-device services rely heavily on the models. From a model processing perspective, the implementation of on-device ML can be generalized into three distinct procedures, namely: *(i)* model pre-loading, *(ii)* model loading, and *(iii)* model post-loading.

(I) Model Pre-loading. The first step in implementing on-device ML services for mobile apps involves locating and verifying the presence of local models. If the models are not found, the apps may need to trigger additional mechanisms to download the relevant models. This is because while some apps choose to retrieve models from servers during their initial setup, the majority of mobile apps carry the models internally. Furthermore, to safeguard against theft, some models are encrypted, while others remain in plaintext. Notably, plaintext models at rest may be stored in custom formats that are often undocumented. Additionally, since several SDK providers require license validation, mobile apps utilizing these SDKs must also undergo the validation process before loading models into memory.

(II) Model Loading. Once the on-device models are prepared and the necessary execution environmental requirements are met, mobile apps proceed to load the models into memory. Since models are not always stored in plaintext, apps employ two strategies during the loading procedure. One is to load the encrypted file, while the other involves decrypting the models before loading them. To ensure both security and performance, this loading procedure is often implemented using native code in the Android platform.

(III) Model Post-loading. After loading into memory, apps may need to decrypt the model if it hasn't been decrypted during the loading procedure. Afterward, the app can proceed to interpret

[1] DeMistify is available at https://github.com/MGYN/DeMistify.

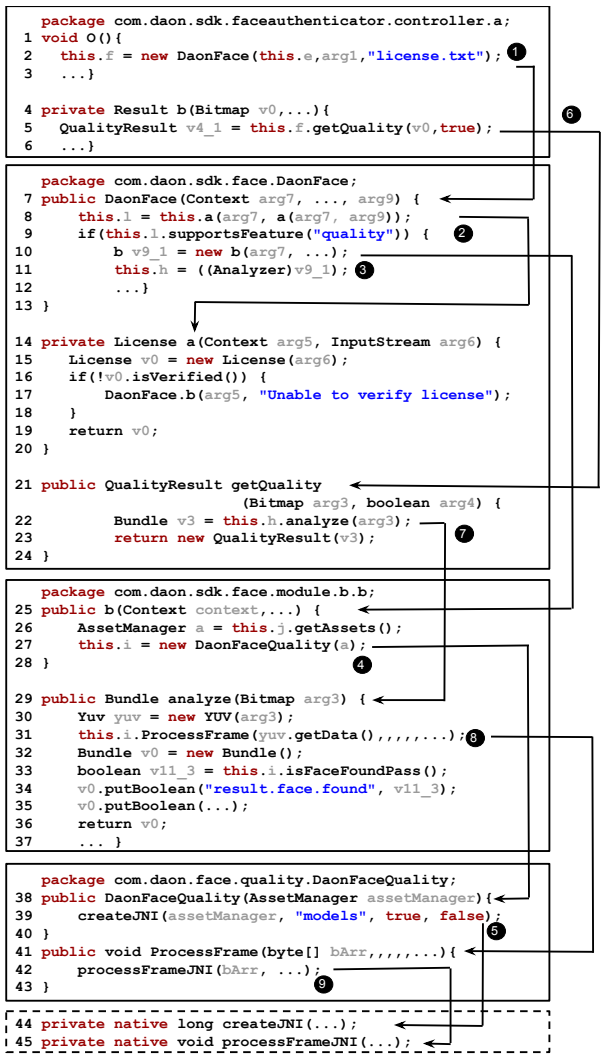

Figure 1: Facial Recognition Login in CIMR DIALCOM.

these models and utilize them to perform relevant tasks on provided inputs. Similarly, the core process of model inference usually takes place in native code, while the input validation and acceptance are typically handled at the Java level in Android.

3 OVERVIEW

3.1 A Running Example

This running example demonstrates how an app implements on-device ML services with local models. In particular, this example is obtained from a finance app, CIMR DIALCOM, which has more than 100,000 installs and implements an on-device ML service allowing users to log into the system via facial recognition.

Specifically, as shown in Figure 1, this app first initializes the instance in step ❶ and checks the validity of its license in step ❷, which is loaded from the file `license.txt` at line 2. If the license has been verified, the second procedure of this app starting at line 10 is to load the associated model. There are three steps, *i.e.*, initializing two different instances of two different classes in step ❸ and ❹, and invoking a native method via JNI to load the model into the memory in step ❺. Next, it invokes the method `getQuality` at line 5 to start the on-device ML service allowing users to log into the system via facial recognition beginning from step ❻. In particular, the method `getQuality` passes the captured face image storing in the `bitmap` format to the method `processFrame` at line 31 via step ❼, and then `processFrame` passes the face image to the designate native method `processFrameJNI` at line 42 through step ❽ and ❾. Finally, the facial recognition result is obtained by invoking the method `isFaceFoundPass` at line 33 which finishes the facial recognition login mechanism.

3.2 Challenges and Insights

To steal the on-device ML models and reuse associated services in a given app automatically, there are three main research challenges:

C1-How to Bypass Restrictions on On-device ML Services. From the running example in step ❷, we can recognize that on-device ML models and services contain restrictions to prevent theft and unauthorized usage. Consequently, the first challenge is to bypass these restrictions. While it may initially seem simple by employing the same license file as the one utilized in mobile apps, this approach may prove ineffective and difficult to scale. Thus, the development of a universal method to bypass all potential restrictions associated with the steal of on-device ML models is challenging.

C2-How to Slice On-device ML Services Accurately. Once the restrictions on reusing on-device ML services have been successfully bypassed, the next challenge is how accurately recognize and slice the components executing these services. The main difficulty in this task is determining the minimum necessary set of components. This is crucial because a larger set of components often necessitates additional configurations and preparations for reusing. These include providing correct parameter values, triggering a specific execution state, and facilitating data exchanges between different components. For instance, consider the function at line 26, which involves a complex process of parsing results. As such, defining the minimum necessary components is far from trivial, and as far as our knowledge extends, no attempts have been made in the existing literature to address this challenge.

C3-How to Reuse On-device ML Services Correctly. Furthermore, even if the minimum execution components for reusing on-device services have been identified, another challenge arises when attempting to reuse these services without proper documentation. In the running example, the initial function at line 2 and the analyze function at line 5 are considered essential. However, automatically determining the execution order and restoring their parameters (*e.g.* "this.e" and "arg1") present significant challenges. Without detailed information on component usage, including the parameter value types and the execution order of related APIs, achieving correct results when reusing these services becomes extremely difficult. Considering the potential utilization of custom data structures among mobile apps, developing a generic approach to effectively reuse on-device ML services is indeed a non-trivial task.

To address these challenges, we have the following insights:

Pengcheng Ren[1], Chaoshun Zuo[2], Xiaofeng Liu[1], Wenrui Diao[1]
Qingchuan Zhao[3](✉), and Shanqing Guo[1](✉)

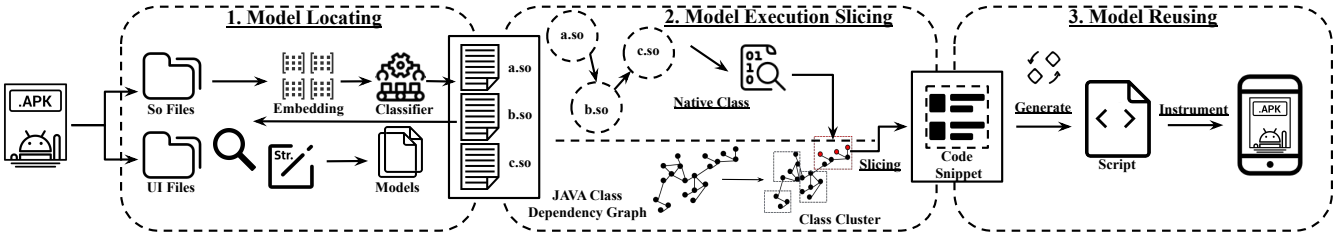

Figure 2: Overview of DeMistify**.**

S1-Bypassing Restrictions and Slicing On-device ML Service via Boundary Function Recognition: Considering the difficulties associated with bypassing restrictions and tracing call logic in on-device ML services, we propose a novel and generic approach to identify boundary functions through slicing. These boundary functions serve as the direct interfaces to invoke a target service and enable the reuse of their own code with minimal effort. Fortunately, boundary functions not only assist in accurately slicing these services but also assist in bypassing license verification and other usage restrictions. We have observed that these restrictions are typically implemented as part of the boundary functions. (*e.g.*, in the running example, line 2 serves as a boundary function that can bypass the restriction). An important insight we have regarding on-device services is that they are often implemented in native libraries to raise the barrier against attacks (*e.g.*, reverse engineering). These native libraries often expose limited interfaces (*i.e.*, JNI) that are formatted with outstanding features [13] (*e.g.*, starting with `Java_`) for communication between the code at the Java level. Leveraging this limited number of JNIs, we propose to recognize the target boundary functions by backward tracing.

S2-Reusing ML Services via Dynamic Instrumentation. Once the boundary functions have been identified, our objective is to invoke them in their original order and provide the appropriate input parameters for effective reuse, thereby minimizing the risk of failure. To accomplish this, we propose leveraging dynamic instrumentation to trigger the boundary functions at the appropriate time, using our task-controlled parameter values. Furthermore, to determine the execution order of these boundary functions and parameter values, we propose to rely on the inter-procedural call graph to uncover the boundary function execution order. Additionally, we employ static value set analysis to resolve the parameter values.

3.3 Threat Model

The consequences of model leakage are quite severe. First, training a model consumes a lot of resources, and leaking a model can result in a loss of intellectual property. Second, a leaked model makes it easy for malicious actors to find adversarial inputs to bypass or confuse ML systems that use the model, leading to a critical failure.

Attack Scenario. In this work, we assume that the attackers aim to (i) steal and reuse the on-device ML models or (ii) attack the associated AI tasks. In scenario *(i)*, we assume that the attackers can deploy the apps and reused scripts in the cloud, and encapsulate an API for free use, and in scenario *(ii)*, we assume that the attackers can infinite query the reused scripts, construct adversarial examples and send it to the victim apps without no conditions required.

Attack Capability. To achieve the above attack scenario, we assume that the attackers can access and reverse the victim app from markets and install them on a smartphone locally. In addition, reverse engineering tools (*e.g.*, Soot [44] and Apktool [3]) have been well developed and publicly available.

3.4 Scope and Assumptions

DeMistify is implemented as an automated tool to steal on-device ML models and reuse associated services in Android apps. As such, apps on other platforms (*e.g.*, iOS) are out of scope. In addition, DeMistify focuses on mobile apps that leverage native libraries to use relevant models to perform on-device ML services. Therefore, apps that entirely implement these tasks at the Java code level are not covered in this study. Moreover, even though DeMistify is resilient to common obfuscations (*i.e.*, identifier renaming), its current version assumes that apps have not been heavily hardened that cannot be unpacked by common tools (*e.g.*, Soot [44] and Apktool [3]).

4 METHODOLOGY

This section presents the design details of DeMistify. As shown in Figure 2, DeMistify consists of three primary components. The first component is Model Locating (§4.1) which locates native libraries that are able to process these models and detects the presence of local ML models. The located native libraries are then taken as inputs to the second component, *i.e.*, Model Execution Slicing (§4.2), which aims to obtain program slicing to the minimum necessary for the execution of particular on-device ML services. Finally, as the third component, Model Reusing (§4.3) attempts to make use of the slices to conduct their original tasks given legitimate inputs.

4.1 Model Locating

As the first component of DeMistify, Model Locating is designed for two purposes, *i.e.*, *(i)* locating native libraries that are in charge of conducting on-device services via processing ML models and *(ii)* detecting the presence of these models for these services.

Although previous studies [48, 54] have demonstrated the effectiveness of using a predefined dictionary for keyword matching to detect on-device apps in native libraries (*e.g.*, `.so` files), it faces challenges in scalability and may fail to detect new cases. To address these limitations and propose a more reliable solution to locating on-device ML services, inspired by native libraries often containing sufficient information (*e.g.*, prompt, log, and error output) as hints

for development purposes, we suggest training an advanced representation learning classifier that embeds these library descriptions, effectively capturing the semantics. By identifying the semantics of this information, we can effectively locate these services.

Ground-truth Dataset Construction. We need to build a ground truth dataset for Model Locating to characterize apps performing ML services with native libraries and models. However, to the best of our knowledge, there are no existing public datasets of ML apps and so files. Therefore, we use the qualitative open-coding techniques [46] to determine whether an app sample is an ML app or not. More specifically, two mobile security researchers with 5 years of experience were asked to verify the apps following the guide as elaborated below. First, the annotators decompile the app to view the native functions which are likely ML services. If there is one, the annotators notice the invocation of "System.loadLibrary("xx")" to find out which `.so` file is the native function to be defined. Next, the annotators search the descriptions from the `.so` file to find the descriptions that indicate the ML services. Then, the annotators view the resource files to find model files based on the suffix and file path. Finally, the annotators annotate each app as ML or not using two rules: *(i)* if the app contains AI-task native function and description and *(ii)* if the app contains suspicious model files. To eliminate bias, the annotators annotated independently and then checked the consistency of the results. We present five apps annotated by the annotator at Table 1.

Training the Classifier. We begin by annotating native library descriptions that can be used to train the model. For this, we selected 1000 apps with top installs for annotation. Specifically, we made the above annotations on these apps and obtained a total of 54 ML apps. We filtered them and obtained 18 typical ML apps containing different ML libraries and services in several categories, and 40 non-ML apps were randomly selected for descriptions annotated. Then we extracted these libraries and preprocessed their descriptions, which led to 101, 387 descriptions for training. After that, we labeled the description of all non-ML libraries as false and used it to filter the ML-related libraries, manually annotated the rest, and finally get 3, 171 descriptions as true. Then we adopt the essential idea in the Word2vec [36] and IotSPOTTER [25], represent each word by embedding it as a numerical vector and train a classifier with `BiLSTM` [6] by splitting the dataset into a training set, testing set, and validation set with the ratio of 8:1:1. After that, this classifier can give the confidence level of each description. To minimize the false positives, we consider the library which has more than 10 descriptions with a confidence level greater than 0.8 as our interest. In addition, we also adopt the string matching ideas adopted by [54] to parse the file paths of apps complementary to detect models for better coverage and identification success rate.

4.2 Model Execution Slicing

Once the target on-device ML models and their corresponding native libraries have been located, DeMistify aims to identify specific code snippets for executing the target service. At a high level, Model Execution Slicing first identifies the Java Native Interfaces (*JNIs*) within the native libraries, next, leverages the identified JNIs to recognize and slice particular clusters of Java classes that perform target services, and finally, pinpoints the most relevant boundary functions which can be reused to perform the task with minimal effort.

Identifying JNIs From Native Libraries. Since core operations related to using local models are in native libraries [48, 54], which are invoked by methods at the Java level via JNI, it will significantly reduce the search space of functions at the Java level if identifying these JNIs first and relying on them to find the functions of our interests. To this end, Model Execution Slicing analyzes the *.so* files related to ML model usages and searches for the JNIs, where their names start with `Java_`, from all exported functions in a *.so* file. Furthermore, since a native library may have dependencies on other libraries, and these dependencies are declared in the dynamic section with a reserved keyword of `needed library`, Model Execution Slicing follows these dependencies to exhaustively identify all JNIs to avoid the potential of missing any cases.

Recognizing Java Classes for On-device ML. After identifying JNIs from native libraries, Model Reusing can utilize them to identify the minimum set of Java classes that contain methods for invoking these JNIs. To achieve this, Model Reusing first constructs a weighted Java class dependency graph (*JCDG*), next applies the graph community detection algorithm to cluster these Java classes, recognizes the clusters of Java classes that are of our interest and slices these classes hierarchically to get the potential least effort.

Building the Weighted Java Class Dependency Graph. Taking inspiration from [31] who introduced the concepts of density and attractiveness for weighted networks, Model Execution Slicing constructs the weighted JCDG which is a directed graph and describes the operational dependencies among Java classes, with each node representing a Java class and each directed edge indicating the number of method calls from the caller class to the callee class. Specifically, the weight of a path is calculated based on the observation that developers often implement a functionality through several subtasks, and the codes to accomplish these tasks are often called close to each other. In other words, classes with more call relationships are much closer than those with less, which is a good representation of the relationship between classes. Besides, since we focus on class dependencies, it is not enough to compute method dispatches with the static type, we need to solve object-oriented programming (*OOP*) language features (*e.g.*, polymorphism) to obtain more precise dependencies, as such, we built and optimized the *context-sensitive pointer analysis* idea in FlowDroid [4] to get the runtime type in function calls. Therefore, the weight of the directed path (*i.e.*, W_{path}) from caller class A (*i.e.*, C_A) to callee class B (*i.e.*, C_B) is:

$$W_{path}(C_A, C_B) = \sum_{i=1}^{n \in methods(A)} W_{path}(M_A^i, C_B) \quad (1)$$

$$W_{path}(M_A^i, C_B) = \sum_{j=1}^{n \in methods(B)} W_{path}(M_A^i, M_B^j) \quad (2)$$

In particular, the algorithm of class closeness detection is shown in algorithm 1 where N is the set of nodes, E denotes the set of edges, and wE stands for the set of weights of edges. First, N, E, and wE are initialized at line 2, and then it iterates every method in the input set of system classes C_s and puts the class name ($cSrc$) of

Table 1: Confirm Results of 5 Mobile Apps in Ground Truth.

Package Name	Native Functions	So name	So Description	Model Files	ML app?
com.calliweb.lamaisonbleue	detectFacesJni(..) trackSingleFaceJni(..)	libface_detector_v2_jni.so	left_eye_closed Model data or file location is required to init FaceDetector	contours.tfl blazeface.tfl	✓
com.kepchat.androidv4	TrackingFace(..)	libnama.so	face_confirmation_softmax_threshold	fa20160614.model	✓
com.vysionapps.face28	fuTrackFace(..)	libface28.so	facet category, unable to create facet for	face_makeup.bundle	✓
com.momo.mobile.shoppingv2.android	receiveDetections(..)	libmlkitcommonpipeline.so	coarse_classification_result_for_ocr	mobile_object_localizer_anchors.pb	✓
consumer.danone.mum	createModelWithBuffer(..)	libtensorflowlite_jni.so	Convolution does not support more than 1 runtime tensor	diaper_detection.lite	✓

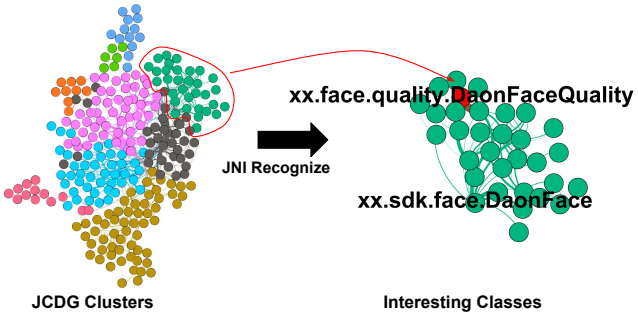

Figure 3: An Example of Recognize Java Classes.

each method if it calls an arbitrary method belonging to a class in the class set C to the N set at lines 3 – 6. Next, it continues to iterate every method in a *cSrc* to find all callee methods *cDes*, obtains the class name *cDesN* of these callees, and creates an associated edge at lines 7 – 11. Then, it updates the weight of the edge from *cSrc* to *cDesN* at lines 12 – 13. Finally, it passes the JCDG with nodes to have the graph community detection at line 14.

Slicing Classes via Graph Community Detection. Finally, Model Execution Slicing applies the popular graph community detection algorithm [17] on the weighted JCDG to cluster Java classes and detected the functional modular containing JNIs. This is because it is used to detect the community in a graph where each community contains several nodes that are more densely connected internally than externally. As shown in Figure 3, this algorithm divides the JCDG derived from the app in §3.1 into several clusters. By identifying the JNI class name of "xxx.DaonFaceQuality", we can isolate the green cluster as our interesting. As such, the classes are the minimum necessary components responsible for target services and will be used to identify boundary functions. In addition, to ensure the validity of clustering results (*e.g.*, the current classes contain false positives that can lead to wrong boundary functions), Model Execution Slicing adopts extra hierarchical slicing to generate multiple possible outcomes according to the upper and lower bounds obtained by the clusters. Although this may not represent the minimum effort, this can ensure that the results are correct as much as possible. Specifically, it starts at the upper bound and deletes the outermost class at a time to generate candidate classes, and repeats on the current result until only JNIs are left. To this end, we can obtain several subclusters, representing different necessary components to achieve model reuse.

Pinpointing Boundary Functions. After identifying the Java classes responsible for delivering ML services with the least effort,

Algorithm 1: On-device DL/ML Classes Cluster

Input: C_s:system classes;
Output: C:cluster of JCDG;
1 `GetSDKClass(`C_s`)` **begin**
2 $N, E, wE \leftarrow \emptyset$
3 **for** $cSrc \in C_s$ **do**
4 $cSrcN \leftarrow$`PkgName(`$cSrc$`)`
5 $N \leftarrow N \cup cSrcN$
6 **for** $f_m \in$ `GetMethods(`$cSrc$`)` **do**
7 **for** $f_c \in$ `GetCalls(`f_m`)` **do**
8 $cDes \leftarrow$ `GetClass(`f_c`)`
9 $cDesN \leftarrow$`PkgName(`$cDes$`)`
10 $edge \leftarrow$`GenEdge(`$cSrcN, cDesN$`)`
11 $E \leftarrow E \cup edge$
12 $wE[edge] \leftarrow wE[edge] + 1$
13 $jcdg \leftarrow$`Graph(`N, E, wE`)`
14 **return** `CommunityDetection(`$N, jcdg, 0$`)`

the final step for Model Execution Slicing is to pinpoint the boundary functions which represent the highest-level functions that can be invoked to perform the ML service without unnecessary side effects. These boundary functions are designed to minimize the required resources as input and trigger subsequent procedures without additional configuration or data preparation. Crucially, these functions cannot be invoked by any function within the classes in a cluster, otherwise, it would indicate the presence of another high-level function that is even closer to the boundary of the cluster. In addition, to avoid dead code, Model Execution Slicing applies another rule that a boundary function must be invoked by at least one function that completely belongs to a different cluster. After that, the functions of "DaonFace" and "getQuality" in the running example will be considered as boundary functions to perform the next step.

4.3 Model Reusing

After identifying the boundary functions, DeMistify proceeds to perform two subtasks to facilitate the reuse of the corresponding on-device services: *(i)* generating reuse scripts and *(ii)* successfully reusing target services via dynamic instrumentation.

Generating Reuse Scripts. As the first step of Model Reusing, two problems need to be solved. First, it would be insufficient if only identify all boundary functions without specific orders; otherwise, it is possible that the target ML task would not be triggered properly. Second, these functions cannot be executed without giving the correct parameters. Therefore, Model Reusing determines the execution orders by traversing the inter-procedural control-flow graph

(*ICFG*) and reconstructs the parameters by backward tracking the inter-procedural data-flow graph (*IDFG*). Considering that these two steps need precise analysis to deal with OOP language features. To this end, we also use the aforementioned *context-sensitive pointer analysis* technique to construct the graphs.

Orders Determined by Traversing The ICFG. The Android operating system provides a large set of callbacks for mobile apps to use to accomplish a variety type of asynchronous tasks in the background. Given this multi-threading nature in Android apps, Model Reusing first utilize FlowDroid [4] to generate the main entrance (*e.g.*, dummy function), then Model Reusing determines the execution order of boundary functions by traversing the ICFG. Particularly, starting from the main entrance, Model Reusing simulates the execution of function blocks and tracks each statement, and maintains object type operation, such as new, assignment, and return, when encountering a function call, it will track the real function call based on the current runtime object type, and determine the boundary function orders by the visiting order.

Reconstructing parameters by tracing backward. After determining the execution order of these boundary functions, Model Reusing is required to provide sufficient and accurate values to the parameters of all boundary functions. In particular, Model Reusing reconstructs the values to these parameters by traversing in the backward direction of the inter-procedural data-flow graph (*IDFG*). This parameter reconstruction procedure is performed with the following systematic strategies to optimize the whole framework.

(I) Constant Values. Values to some parameters could be constant values, such as hardcoded strings. For these parameters, the corresponding strategy is to resolve such values even in the case that they are generated via complicated computations, *e.g.*, splitting, concatenation, and *etc.*

(II) Variable Values. In cases of variable values, static-based solutions to resolve values may not be able to produce concrete values. Therefore, based on the declared types of parameters in on-device ML boundary functions, Model Reusing addresses these variable values with the following two strategies.

- *Pre-defined Termination Types*: Some variable values are generated outside the app and serve as input(*e.g.*, the data structure of `bitmap` is to accept the data of an image captured by the camera). As such, we first divide the inputs into four types according to the ML service [12], namely image, voice, video, and text. These inputs are always collected through system API calls, so we consulted the developer documentation one by one according to the input type, there are 11 APIs (listed in Table 2) in total to handle these tasks. Therefore, instead of intending to resolve the actual values, Model Reusing would like to uncover the pre-defined types of the data structures if they are designed to accept values from the outside.
- *Return Values of Other Methods*: Some other variable values that are generated within the app may also be difficult to resolve if they are the return values of other methods. Since these methods may not be included in the clustered Java classes obtained in §4.2, Model Reusing also has to execute such methods with appropriate parameter values, which makes this procedure an iterative one. For example, boundary function A requires a parameter, but this parameter is the return value of non-boundary function B. The simplest way is to call the B function to get this return value, but if B needs to pass parameters, Model Reusing also needs to process these parameters. To optimize this iterative process to avoid legacy static analysis problems (*e.g.*, path explosion), Model Reusing only repeats the process within a loop once and terminates at methods that entirely generate values depending on the outside resources (*e.g.*, network response) having no direct relationship to target tasks.

Table 2: Targeted Types for Termination.

Type	Class	API
Input	Bitmap	createBitmap(int[], int, int,...)
	BitmapFactory	decodeFile(String)
	Media	getBitmap(ContentResolver, Uri)
	AudioRecord	read(short[],int,int)
	AudioRecord	read(byte[],int,int)
	AudioRecord	read(java.nio.ByteBuffer,int)
	PictureCallback	onPictureTaken(byte[], Camera)
	PreviewCallback	onPreviewFrame(byte[], Camera)
	EditText	getText()
	EditText	getEditableText()
	Editable	toString()

```
var path = "pic-1.jpg";
// convert to bitmap
var file = Java.use('java.io.File').$new(path);
var FS = Java.use('java.io.FileInputStream').$new(file);
var bitmap = Java.use('android.graphics.BitmapFactory').
     decodeStream(FS);
. . .
// boundary function #1
var s33124763 = Java.use('com.google.android.gms.vision.
     face.FaceDetector$Builder').build();
s33124763.isOperational()
// boundary function #2
var s696379067 = Java.use('com.google.android.gms.vision.
     Frame$Builder').$new();
s696379067.setBitmap(bitmap);  //  input
var s1845700457 = s33124763.detect(s696379067);
. . .
```

Listing 1: Reusing Script Generated by Demistify

Model Reuse via Dynamic Instrumentation. Upon generating the model reuse scripts, the final step for Model Reusing is to reuse the target on-device services and verify the success. To gain a clearer understanding of how the reuse scripts are executed, we present the key components of a script generated by DeMistify in Listing 1. In this script, the **path** specifies the default image configured for each test trial, which is utilized for reusing the on-device model. Additionally, every boundary function is appropriately prepared for triggering. Subsequently, DeMistify automatically installs the relevant application, restarts it, injects the script using Frida [15], and collects the results without requiring any additional configuration.

5 EVALUATION

In this section, we evaluate DeMistify and answer the following research questions based on experimental results.

- **RQ1:** How does DeMistify perform on a large scale?
- **RQ2:** How effective is DeMistify? We address this question by considering three sub-questions:

- **RQ2.a:** How effective are the components of DeMistify?
- **RQ2.b:** What is the reuse ability of DeMistify?
- **RQ2.c:** Is DeMistify highly resilient to model protection and code obfuscation?

5.1 Implementation and Experiment Setup

We implemented DeMistify with more than 700 lines of Python code and around 6500 lines of Java code. The implementation of the Model Locating depends on a few libraries: we use apktool [3] to collect the libraries from apps, strings command to extract descriptions, Word2vec [36] to train an embedding, and `BiLSTM` [6] to train the classifier. In the Model Execution Slicing, it leverages the readelf command to extract all exported functions in every `.so` file and then depends on Soot [44] to build the JCDG and cluster the classes. For the Model Reusing, it depends on Soot [44], FlowDroid [4], and Frida [15] to build ICFG, resolving parameters, dynamically instrument code reuse scripts, monitor the completion of model reuse code execution, and return the consequent results.

Evaluation Environment. The experiments are conducted on two servers. One is equipped with an Intel Xeon Gold 5215 CPU with 512GB memory running Ubuntu 18.04. This server crawls apps from Google Play and analyzes them with DeMistify. The other has an Intel i7-9700 3.00 GHz CPU with 16GB memory running Windows 10. It is connected to one Pixel 2 smartphone running Android 11 to evaluate the effectiveness of the generated reusing scripts.

5.2 RQ1: Scalability of DeMistify

This section presents the accuracy verification, the dataset collection, and the distribution analysis of ML SDKs and frameworks.

Identifing Accuracy and Precision of ML Apps. To establish the ground truth for our evaluation, we initially executed ModelXRay [48] based on the install numbers from our dataset until we got 152 apps labeled as ML apps and 152 non-ML apps. Then, we manually verified all 304 apps and adjusted the labels as our ground truth based on (§4.1), resulting in 138 apps as ML and 166 apps as non-ML. We evaluated [48] and DeMistify on these apps, and the results showed that [48] achieved an accuracy of 89.5% with a false negative (FN) rate of 6.5% and a false positive (FP) rate of 13.8%. We conducted further analysis on the FP and FN cases of [48]. In FN, we found that these apps use unpopular ML frameworks whose keywords are not in the string dictionary, besides, in FP, there are some strings that are matched incorrectly (*e.g.*, PatternNode is a match for rnn) and some libraries only contains individual strings but does not provide ML services. In contrast, DeMistify overcame these limitations and achieved an accuracy of 98.4% with only 1.4% FN and 1.8% FP. In terms of precision, [48] achieved 84.9% precision, while DeMistify achieved 97.8%, this relatively high accuracy and precision shows DeMistify is suitable for large-scale study.

Efficiency of DeMistify**.** We evaluate the runtime overhead of [48] and DeMistify on ML app identification with 50 parallel processes. On average, it takes 8.42 seconds for DeMistify to process an app in our dataset, comparable to [48] (i.e., on average, 6.96 seconds to process an app). We also evaluate the runtime of DeMistify on Model Reusing, it only takes 63.27 seconds and 1.26G memory on average for DeMistify to generate a reuse script. The reason it is so efficient is because, we only use FlowDroid [4] to generate entry points, bypassing a lot of its time-consuming operations. This proves that DeMistify can be used to analyze large-scale apps.

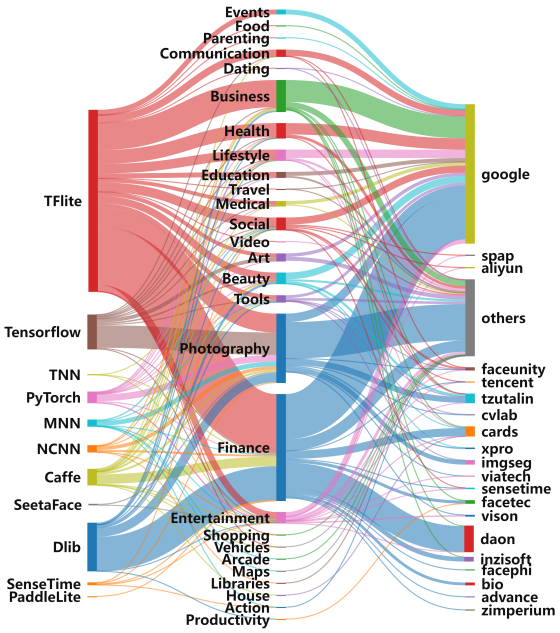

Figure 4: Distribution of Framework and SDK over Category.

Dataset Collection. The dataset used in this study comprises a total of 427,471 top free Android apps sourced from Google Play across 27 categories based on their install numbers as of September 2021. Furthermore, out of the nearly 420-thousand apps, we applied semantic analysis in §4.1 and supplemented it with the algorithm proposed in a previous study [48] which relied on keyword matching for on-device ML app and model recognition, resulting in the identification of 2,207 apps that provide on-device ML services. Subsequently, we manually inspected, installed, and ran these identified apps, filtering out those with garbled code, crashes, and no relevant service identification. Ultimately, we verified and included 1511 apps in our test dataset. It's worth noting that throughout the entire process, only one person was involved. On average, this person spent at least 7 minutes per app, with approximately 5 minutes for viewing and 2 minutes for installation and execution.

Generality of DeMistify **on different ML SDK and Framework.** To assess the generality of DeMistify, we examined the distribution of 1,511 apps. By summarizing the package names as SDKs and inferring the library descriptions as frameworks, we identified a total of 11 known frameworks and 118 SDKs. Among them, 20 SDKs were implemented in more than three apps. We also plotted them in Figure 4 to visualize the distribution of these SDKs and frameworks across different categories. It is obvious that DeMistify is robust in large-scale detection rather than focusing on specific frameworks or SDKs and can conduct comprehensive analysis across various frameworks and SDKs.

Table 3: Overall Effective Evaluation Results.

Item	Value
# DL/ML Apps Tested	1,511
# Models	15,435
# JNI	3,630
# JCDG	1,511
# Nodes	21,664,793
# Edges	87,495,777
# Clusters	1,981,850
# Java Classes	39,456
# Boundary Functions	14,483
# Dead Code	3,301
# Active Functions	11,182
# Lifecycle Callback	1,964
Ave. Funcs in Callback	5.69
# Traced Parameters	19,051
# Constants	1,208
# Termination	2,539
# Return Value	15,304
Ave. Depths	2.51
# Reuse Succeed Apps	1250
# Success Rate	82.73%

5.3 RQ2: Effectiveness Evaluation of DeMistify

5.3.1 ***RQ2.a: How effective are the components of*** DeMistify***?***

This section presents the effectiveness evaluation of DeMistify in Table 3, and the details of each component are presented as follows.

Effectiveness of Model Locating. The objective of Model Locating is to identify the presence of target on-device ML services and locate their native implementations. In the evaluation of 1,511 apps, this component has successfully recognized 3,630 JNI classes that are closely associated with the target on-device services. Furthermore, it has identified a total of 15,435 models from all the apps, averaging approximately 10 models per app.

Effectiveness of Model Execution Slicing. After recognizing and locating on-device ML services in native implementations, Model Execution Slicing focuses on identifying and slicing the minimum necessary implementations at the Java level for reuse. In this process, Model Execution Slicing generates a total of 1,511 JCDGs, with each JCDG generated for a specific mobile app. These JCDGs consist of 21,664,793 nodes (representing unique Java classes within each app), 87,495,777 edges, and 1,981,850 clusters. Additionally, Model Execution Slicing identifies 39,456 classes that are directly involved in performing on-device ML services. These classes encompass both individual development classes and commercial SDKs. Among these classes, 14,483 boundary functions have been pinpointed. To analyze these boundary functions further, using the methods proposed in §4.2, DeMistify has identified 3,301 boundary functions that are never invoked and are considered dead code. Among the remaining boundary functions, 11,182 active boundary functions are categorized into 1,964 lifecycle callbacks (on average 5.69 functions in each callback) which are used to recognize their execution orders.

Effectiveness of Model Reusing. In Model Reusing, the value parameters for the detected boundary functions are resolved, and reusing scripts are generated accordingly. Specifically, Model Reusing successfully resolves 19,051 parameters for 11,182 active boundary functions. It addresses 1,208 constant values, identifies 2,539 predefined termination parameter types, and resolves 15,304 values that are returned by arbitrary methods. On average, each return value requires the component to trace back 2.51 steps in depth.

To investigate the accuracy of these results, we randomly selected 10 apps, manually reused them, and compared the statistical results with DeMistify. Specifically, Model Locating reports 86 models with one false positive and zero false negative. As for the recognized JNIs, Model Execution Slicing reports 21 JNIs with three false positives, due to they are utility classes declared in ML native libraries. When considering the active boundary function, Model Execution Slicing collects 44 active boundary functions with nine false positives and one false negative. However, these false positives are some configuration functions, which did not have any impact on the results. As for the false negative, it was identified as dead code, which was caused by the limitations of pointer analysis on reflection.

5.3.2 ***RQ2.b: What is the reuse ability of*** DeMistify***?***

After manually verifying the success of reusing on-device ML services, which return the expected results, DeMistify successfully reuses 1250 out of 1,511 apps in the test dataset, achieving a success rate of 82.73%. Additionally, we measured the informativeness of the results and found that 90.3% of the reusable apps contain rich quantitative and multidimensional information that can be exploited by adversary attacks. The remaining apps are mainly related to background editing, which directly returns the editing results (*e.g.*, removing the background). This is because DeMistify slices the functional modules of ML service and retains the most original data of inference results which will be further processed by the app. In particular, Table 4 shows the detail of our model reusing results of the top 10 reusable mobile apps as examples to show how we determine if the reuse was successful. The table includes the package name of the app, the reused service, the associated input, the original detection result, and the result obtained in our reusing evaluation, which is classified as either correct or wrong. If the original and reused results match, we consider the model reuse to be successful. Even small partial differences in the results would lead to the conclusion that reusing has not been achieved. As expected, all on-device ML models in these apps were successfully reused, regardless of their different functionalities.

Regarding the failed apps, we conducted a manual analysis of the top 50 apps and summarized the failure reasons. At a high level, the primary cause of failure can be attributed to limitations in the tools used. Specifically, six apps failed due to an incomplete call-graph generated by FlowDroid [4], eight apps encountered limitations in Frida [15], where seven apps contained garbled code that couldn't be resolved, five execution timeouts, and one instrumentation failure. Furthermore, there were five cases where failure resulted from the absence of local models despite the presence of native libraries. Additionally, two apps failed because we couldn't provide appropriate inputs, electrocardiography, and road condition. Lastly, out of the remaining apps, seven apps had issues with correctly resolving parameters, six apps had incorrect class clustering, and ten apps failed to produce valid outputs. Among these, five apps failed to generate any output when used, while the other five apps belonged to the same SDK provider but utilized a unique implementation that significantly differed from the others.

5.3.3 ***RQ2.c: Is*** DeMistify ***highly resilient to model protection and code obfuscation?***

Table 4: Reuse results of Top 10 Reusable Mobile Apps Providing Different On-device ML Services.

Package Name	Reuse Service	Input	Subtask	Original detection	Reuse detection	Correct?
com.commsource.beautyplus	Selfie editing	Selfie image	FaceRect FaceCount	(71.0, 81.0, 193.0, 203.0) 1	(71.0, 81.0, 193.0, 203.0) 1	✓ ✓
com.videoplayer.livetalk	Video chat	Facial image or frame	FaceRect IsSmiling IsEyesOpen Landmarks	(37.23, 49.01) + 181.52 0.007 0.99 (95.01, 120.73)	(37.23, 49.01) + 181.52 0.007 0.99 (95.01, 120.73)	✓ ✓ ✓ ✓
makeup.selfiecamera.beautycam.beautyplus	Selfie editing	Selfie image	FaceRect Landmarks	$(L:44, T:77, R:193, B:226, Label:face)$ $(Point(46,125)...Point(124,185))$	$(L:44, T:77, R:193, B:226, Label:face)$ $(Point(46,125)...Point(124,185))$	✓ ✓
com.camera.mi9	Face enhancement	Facial image or frame	FaceRect IsSmiling IsEyesOpen Landmarks	(39.86, 52.85) + 173.38 0.007 0.99 (97.77, 121.75)	(39.86, 52.85) + 173.38 0.007 0.99 (97.77, 121.75)	✓ ✓ ✓ ✓
com.faceenhancer.android	Selfie editing	Selfie image	FaceRect Landmarks	$(L:64, T:92, R:188, B:216, Label:face)$ $(Point(53,120)...Point(124,197))$	$(L:64, T:92, R:188, B:216, Label:face)$ $(Point(53,120)...Point(124,197))$	✓ ✓
com.face.gender.changer.faceapp	Transform gender	Facial image	FaceRect Landmarks	$(L:32, T:52, R:212, B:231, Label:face)$ $(Point(41,119)...Point(129,192))$	$(L:32, T:52, R:212, B:231, Label:face)$ $(Point(41,119)...Point(129,192))$	✓ ✓
com.ophotovideoapps.facenhancer	Selfie editing	Selfie image	FaceRect Landmarks	$(L:32, T:52, R:212, B:231, Label:face)$ $(Point(71,120)...Point(111,199))$	$(L:32, T:52, R:212, B:231, Label:face)$ $(Point(71,120)...Point(111,199))$	✓ ✓
com.playdigital.modo	Document scan	Document image	DocRect	$(RectF(1.5184212, 3.2016373, 318.8564, 218.48717))$	$([0]docs(81.5\%)RectF(1.5184212, 3.2016373, 318.8564, 218.48717))$ $([1]docs(18.2\%)RectF(21.968193, 76.56427, 319.31763, 209.98932))$	✓
et.song.newVSUFO.wifi.app.air.h264	Gesture Recognize	Frame	Control	take photo video record	take photo video record	✓ ✓
com.smithandsons.cutpaste.photoeditor	Backgroud replace	Any image	Foreground Backgroud Masks	foreground image grayscale image $byte[](...0, 0, 0, 1, 1, 1, 1, 1, 1...0, 0, 1...)$	foreground image grayscale image $byte[](...0, 0, 0, 1, 1, 1, 1, 1, 1...0, 0, 1...)$	✓ ✓ ✓

On-device ML Models Protection. In terms of model protection, we collected 15,435 models and evaluated them with different protection types *i.e.*, plaintext, encrypted with entropy higher than 7.99 [48], and custom format which cannot be parsed by common frameworks [12]. The evaluation results indicate that 61% of the models are in plaintext, while the remaining 39% of the models adopt protection measures. Among the protected models, 37% are in a custom format, and 2% are encrypted. Additionally, we assessed the rate of app-level adoption of model protection. Approximately 26.54% of the apps employ plaintext models, 5.62% of the apps utilize encrypted models, and the remaining 67.84% of the apps contain custom format models. Note that, DeMistify is task-oriented rather than model-oriented. Therefore, we evaluated the success rate on apps where multiple on-device models perform the same task collectively. As shown in Table 5, 978 of under protection can still be successfully reused using DeMistify. This is because DeMistify focuses on the functional modules rather than the specific model itself, making it highly resilient to model protection measures.

On-device ML Apps Protection. We would like to understand how the identifier renaming obfuscation affects the success rate, which can hide meaningful information and resist some adversary attacks that rely on plaintext string of standard API [12]. We use [38] to detect obfuscated apps, which depends on soot [44] to identify this obfuscation by analyzing whether the proportion of short class names reaches a threshold. Note that app obfuscation and model encryption are two different things. An obfuscated app can use an unencrypted model, and an app without using obfuscation can also use an encrypted model. As shown in Table 5, 88.43% of 821 obfuscated apps can be reused and 690 apps are at risk of standard API extraction with 75.94% can be reused. Because DeMistify only depends on modular slicing and doesn't rely on the string information during reusing, the confusion doesn't affect success rates, hence, DeMistify is highly resilient to code obfuscation.

Table 5: Statistics of Reusable.

Item	Value	Reusable
# Apk Tested	1511	1250 (82.73%)
# Non-Obfuscation	690	524 (75.94%)
# Obfuscation	821	726 (88.43%)
# Plaintext	401	274 (68.33%)
# Flatbuf	171	122 (71.35%)
# ProtoBuf	217	146 (67.28%)
# Text	13	4 (30.77%)
# Encrypt	85	38 (44.71%)
# Customized format	1025	940 (91.71%)

6 DISCUSSION

Case Study in Adversarial Attack. Previous research on black-box adversarial attacks has been focused on the cloud, they rely on the query to train surrogate models, and their effectiveness is largely limited by the number of queries. However, DeMistify overcomes this restriction, which means DeMistify can obtain a higher fidelity surrogate model with infinite queries. Therefore, We discuss the promising use cases of DeMistify. Specifically, we select a victim app (Ton) that uses a popular SDK provided by Google with more than 100, 000 installs. In particular, it takes a facial image as input and returns the information of smiling probability, eye open probability, and face landmark.

Then we leverage DeMistify to generate the reuse script and run the associated on-device ML services of smiling probability for all facial images in CelebA-HQ [32] and collect the corresponding results. Finally, we train a surrogate model with different sizes of training sets, eight sizes evenly increase from 3,000 to 24,000, and construct an adversarial attack with FGSM [18]. Figure 5 shows the attack success rate with different scales, and it shows the attack success rate presents an upward trend. It seems the decision boundary of the victim model and surrogate model is closer with the training set greater. In other words, without the limitation, DeMistify reveals a much higher security risk.

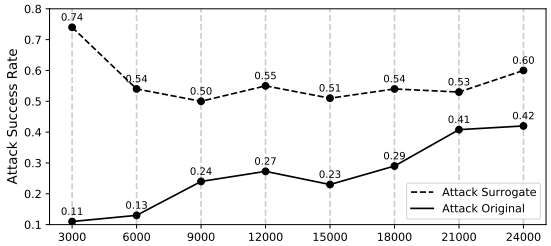

Figure 5: Success Rate Comparison Of Different Dataset.

Limitations. While we wish we could make DeMistify perfect, its current prototype still contains three limitations. First, DeMistify operates under the assumption that apps providing on-device ML services will statically carry their models. While this assumption holds true for the majority of cases, there are instances where an app is designed to retrieve models from its server during its first initialization. Second, as DeMistify is built on top of several open-source tools and frameworks, it inherits some of their limitations, such as incomplete graphs generated by FlowDroid [4], unrecognized special symbols in Soot [44] and Frida [15], as well as execution timeouts resulting from these tools. These limitations could render DeMistify incapable of reusing target services in certain apps. Third, since DeMistify relies on the generation of intermediate results (*e.g.*, JNIs, boundary functions), limited by pointer analysis techniques (*i.e.*, reflection-related problems), there could be some false positives and negatives, although in most cases there is no impact, it may still lead DeMistify to generate wrong reuse scripts in some cases.

Threats to Validity. The internal threats to validity are the implementations of our technique, the ground truth, and the bias in success rate. Our technique is simple and easy to implement and its core part is based on popular tools. Also, we will release our code for future studies. All the rules on ground truth and verification are present in this paper. For the comparison tool, we directly use the released version in [48]. The external threats to validity come from the selected datasets for our evaluation. Our collection of datasets is entirely dependent on the number of installs on Google Play based on their percentage in different categories and was downloaded the latest version before we started our study from androzoo.

7 RELATED WORK

ML Model Extraction and Analysis. Alongside the rapid progress of ML technique and evolution in mobile hardware, their consequent security problems have started to receive attentions from the academia [9, 24, 28]. However, the majority of these works focus on the models in the cloud including both directly predicting the property of models via relevant APIs [27, 49, 51] and stealing models from the side-channel, such as cache [19, 55], energy consumption [5, 53]. Regarding on-device models, only a few studies have been proposed, leaving this field largely unexploited. In particular, these works propose methods to identify deep learning apps via keywords in the path [48] and the file suffix [54], and solutions to distinguish encrypted model files from those in plaintext via entropy and also how to decrypt them when they are at rest or being loaded into the memory [48], or evaluate adversary attacks by tracing inference functions in plaintext [12] and using the highly similar fine-tuned model from TensorFlow Hub as an alternative [22] and train the surrogate model [21]. Different from these works, we focus on model reuse attacks, our work proposes a powerful toolkit that is generic to different frameworks and models, allowing analysts to analyze on-device ML in a given mobile application without any information.

ML Model Protection. Since ML models are always believed to be a core intellectual property, the field of research on how to protect them from being stolen and unauthorized usage has been devoted a tremendous amount of efforts. In particular, these works intend to protect models in the cloud. For example, some solutions leverage trust computation techniques including homomorphic encryption [7, 16, 37] to prevent attackers from directly monitoring sensitive data related to the model property which could be used for model reverse engineering. In addition, some other solutions intend to identify whether models in the cloud are in the process of being stolen by monitoring relevant API usages [26, 27]. Moreover, some works also focus on identifying unauthorized usage of ML models via water-marking techniques [2, 57]; however, such a technique may only work in a limited category of application. Our work is designed to facilitate these analysis on on-device ML practices.

Program Analysis in Mobile Apps. There have been many efforts to conduct program analysis in mobile apps for security purposes. At a high level, they are either static analysis [4, 10, 35, 50, 52, 59, 60] and dynamic analysis [1, 14, 30, 34, 42, 43, 45, 61]. Unlike these works that entirely focus on Java-level implementations, our work conduct analysis on code at both Java and native level.

8 CONCLUSION

This study introduces a novel methodology and develops an automated tool (*i.e.*, DeMistify) to detect model stealing and reuse attacks in mobile apps that utilize local ML models for task delivery. The proposed tool utilizes a combination of static and dynamic program analysis techniques to identify the minimum set of relevant APIs for AI tasks, generate appropriate input values, and dynamically verify the presence of model reuse vulnerabilities. Evaluation of the tool on 1,511 mobile apps containing local ML models demonstrates a reuse success rate of 82.73%. Additionally, the analysis uncovers key participants in the on-device DL/ML service ecosystem and highlights DeMistify's ability to uncover security threats.

ACKNOWLEDGMENTS

We sincerely thank all anonymous reviewers for their constructive feedback. This work was partly supported by CityU APRC grant 9610563, the Research Grants Council of Hong Kong (CityU 21219223, C1029-22G), National Natural Science Foundation of China under Grant No.62372268, Shandong Provincial Natural Science Foundation (No. ZR2020MF055, No.ZR2021LZH007, No.ZR2022LZH013 and No.ZR2020QF045), and Jinan City "20 New Universities" Funding Project (2021GXRC084). Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily of supported organizations.

REFERENCES

[1] 2017. UI/Application Exerciser Monkey. https://developer.android.com/tools/help/monkey.html.

[2] Yossi Adi, Carsten Baum, Moustapha Cisse, Benny Pinkas, and Joseph Keshet. 2018. Turning your weakness into a strength: Watermarking deep neural networks by backdooring. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 1615–1631.

[3] Apktool. [n. d.]. A tool for reverse engineering 3rd party, closed, binary Android apps. https://ibotpeaches.github.io/Apktool/.

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[5] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. 2018. CSI neural network: Using side-channels to recover your artificial neural network information. *arXiv preprint arXiv:1810.09076* (2018).

[6] Keras Bidirectional. [n. d.]. https://keras.io/api/layers/recurrent_layers/bidirectional/.

[7] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2017. Practical secure aggregation for privacy-preserving machine learning. In *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1175–1191.

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

[9] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 367–379.

[10] Hyunwoo Choi, Jeongmin Kim, Hyunwook Hong, Yongdae Kim, Jonghyup Lee, and Dongsu Han. 2015. Extractocol: Automatic Extraction of Application-level Protocol Behaviors for Android Applications. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. ACM, New York, NY, USA, 593–594. https://doi.org/10.1145/2785956.2790003

[11] Jacson Rodrigues Correia-Silva, Rodrigo F Berriel, Claudine Badue, Alberto F de Souza, and Thiago Oliveira-Santos. 2018. Copycat cnn: Stealing knowledge by persuading confession with random non-labeled data. In *2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[12] Zizhuang Deng, Kai Chen, Guozhu Meng, Xiaodong Zhang, Ke Xu, and Yao Cheng. 2022. Understanding Real-world Threats to Deep Learning Models in Android Apps. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*.

[13] JNI Doc. [n. d.]. https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/design.html#wp16696.

[14] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM TOCS* 32, 2 (2014), 5.

[15] Frida. [n. d.]. A world-class dynamic instrumentation framework. https://frida.re/.

[16] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International conference on machine learning*. PMLR, 201–210.

[17] Michelle Girvan and Mark EJ Newman. 2002. Community structure in social and biological networks. *Proceedings of the national academy of sciences* 99, 12 (2002), 7821–7826.

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

[19] Sanghyun Hong, Michael Davinroy, Yiğitcan Kaya, Stuart Nevans Locke, Ian Rackow, Kevin Kulda, Dana Dachman-Soled, and Tudor Dumitraş. 2018. Security analysis of deep neural networks operating in the presence of cache side-channel attacks. *arXiv preprint arXiv:1810.03487* (2018).

[20] Jiayi Hua, Yuanchun Li, and Haoyu Wang. 2021. MMGuard: Automatically Protecting On-Device Deep Learning Models in Android Apps. In *2021 IEEE Security and Privacy Workshops (SPW)*. IEEE, 71–77.

[21] 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.

[22] Yujin Huang, Han Hu, and Chunyang Chen. 2021. Robustness of on-device models: Adversarial attack to deep learning models on android apps. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 101–110.

[23] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. 2018. Chiron: Privacy-preserving machine learning as a service. *arXiv preprint arXiv:1803.05961* (2018).

[24] Andrey Ignatov, Radu Timofte, Andrei Kulik, Seungsoo Yang, Ke Wang, Felix Baum, Max Wu, Lirong Xu, and Luc Van Gool. 2019. Ai benchmark: All about deep learning on smartphones in 2019. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. IEEE, 3617–3635.

[25] Xin Jin, Sunil Manandhar, Kaushal Kafle, Zhiqiang Lin, and Adwait Nadkarni. 2022. Understanding IoT Security from a Market-Scale Perspective. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1615–1629.

[26] Mika Juuti, Sebastian Szyller, Samuel Marchal, and N Asokan. 2019. PRADA: protecting against DNN model stealing attacks. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 512–527.

[27] Manish Kesarwani, Bhaskar Mukhoty, Vijay Arya, and Sameep Mehta. 2018. Model extraction warning in mlaas paradigm. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 371–380.

[28] Juhyun Lee, Nikolay Chirkov, Ekaterina Ignasheva, Yury Pisarchyk, Mogan Shieh, Fabio Riccardi, Raman Sarokin, Andrei Kulik, and Matthias Grundmann. 2019. On-device neural net inference with mobile gpus. *arXiv preprint arXiv:1907.01989* (2019).

[29] TensorFlow Lite. [n. d.]. ML for Mobile and Edge Devices. https://www.tensorflow.org/lite.

[30] Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. 2014. DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Seattle, WA) *(NSDI'14)*. USENIX Association, Berkeley, CA, USA, 57–70. http://dl.acm.org/citation.cfm?id=2616448.2616455

[31] Ruifang Liu, Shan Feng, Ruisheng Shi, and Wenbin Guo. 2014. Weighted graph clustering for community detection of large social networks. *Procedia Computer Science* 31 (2014), 85–94.

[32] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. 2015. Deep Learning Face Attributes in the Wild. In *Proceedings of International Conference on Computer Vision (ICCV)*.

[33] Mace. [n. d.]. Convert a model to c++ code. https://mace.readthedocs.io/en/latest/micro-controllers/basic_usage.html#convert-a-model-to-c-code.

[34] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 224–234.

[35] Abner Mendoza and Guofei Gu. 2018. Mobile Application Web API Reconnaissance: Web-to-Mobile Inconsistencies and Vulnerabilities. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP'18)*.

[36] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[37] Payman Mohassel and Yupeng Zhang. 2017. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*. IEEE, 19–38.

[38] ObfDetector. [n. d.]. https://github.com/CirQ/ObfDetector.

[39] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. 2019. Knockoff nets: Stealing functionality of black-box models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4954–4963.

[40] Soham Pal, Yash Gupta, Aditya Shukla, Aditya Kanade, Shirish Shevade, and Vinod Ganapathy. 2019. A framework for the extraction of deep neural networks by leveraging public data. *arXiv preprint arXiv:1905.09165* (2019).

[41] 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.

[42] Vaibhav Rastogi, Yan Chen, and William Enck. 2013. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy* (San Antonio, Texas, USA) *(CODASPY '13)*. ACM, New York, NY, USA, 209–220. https://doi.org/10.1145/2435349.2435379

[43] Lenin Ravindranath, Suman Nath, Jitendra Padhye, and Hari Balakrishnan. 2014. Automatic and Scalable Fault Detection for Mobile Applications. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services* (Bretton Woods, New Hampshire, USA) *(MobiSys '14)*. ACM, New York, NY, USA, 190–203. https://doi.org/10.1145/2594368.2594377

[44] Soot. [n. d.]. A Java optimization framework. https://github.com/Sable/soot.

[45] David Sounthiraraj, Justin Sahs, Garrett Greenwood, Zhiqiang Lin, and Latifur Khan. 2014. SMV-Hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*. San Diego, CA.

[46] Anselm Strauss and Juliet Corbin. 1990. *Basics of qualitative research*. Sage publications.

[47] Zhichuang Sun, Ruimin Sun, Changming Liu, Amrita Roy Chowdhury, Long Lu, and Somesh Jha. 2020. Shadownet: A secure and efficient on-device model inference system for convolutional neural networks. *arXiv preprint arXiv:2011.05905* (2020).

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

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

[50] Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A measurement study of google play. In *ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '14, Austin, TX, USA - June 16 - 20, 2014*. 221–233. https://doi.org/10.1145/2591971.2592003

[51] Binghui Wang and Neil Zhenqiang Gong. 2018. Stealing hyperparameters in machine learning. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 36–52.

[52] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. 2014. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1329–1341.

[53] Lingxiao Wei, Bo Luo, Yu Li, Yannan Liu, and Qiang Xu. 2018. I know what you see: Power side-channel attack on convolutional neural network accelerators. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 393–406.

[54] 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.

[55] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. 2020. Cache telepathy: Leveraging shared resource attacks to learn {DNN} architectures. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2003–2020.

[56] Honggang Yu, Kaichen Yang, Teng Zhang, Yun-Yun Tsai, Tsung-Yi Ho, and Yier Jin. 2020. CloudLeak: Large-Scale Deep Learning Models Stealing Through Adversarial Examples.. In *NDSS*.

[57] Jialong Zhang, Zhongshu Gu, Jiyong Jang, Hui Wu, Marc Ph Stoecklin, Heqing Huang, and Ian Molloy. 2018. Protecting intellectual property of deep neural networks with watermarking. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. 159–172.

[58] Tong Zhou, Yukui Luo, Shaolei Ren, and Xiaolin Xu. 2023. NNSplitter: An Active Defense Solution to DNN Model via Automated Weight Obfuscation. *arXiv preprint arXiv:2305.00097* (2023).

[59] Chaoshun Zuo and Zhiqiang Lin. 2017. SmartGen: Exposing Server URLs of Mobile Apps With Selective Symbolic Execution. In *Proceedings of the 26th World Wide Web Conference (WWW'17)*. Perth, Australia.

[60] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. 2019. Why Does Your Data Leak? Uncovering the Data Leakage in Cloud From Mobile Apps. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*. San Francisco, CA.

[61] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2017. AuthScope: Towards Automatic Discovery of Vulnerable Authorizations in Online Services. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*. Dallas, TX.