SHADOWDROID: Practical Black-box Attack against ML-based Android Malware Detection

Jin Zhang*†, Chennan Zhang*†, Xiangyu Liu‡, Yuncheng Wang§, Wenrui Diao*†(✉), and Shanqing Guo*†
*School of Cyber Science and Technology, Shandong University
{zhangjinzfy, zcn}@mail.sdu.edu.cn, {diaowenrui, guoshanqing}@sdu.edu.cn
†Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University
‡Alibaba Inc., eason.lxy@alibaba-inc.com
§Taishan College, Shandong University, wyunc@mail.sdu.edu.cn

2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS) | 978-1-6654-0878-3/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICPADS53394.2021.00084

***Abstract*—Machine learning (ML) techniques have been widely deployed in the field of Android malware detection. On the other hand, ML-based malware detection also faces the threat of adversarial attacks. Recently, some research has demonstrated the possibility of such attacks under the settings of white-box or grey-box. However, a more practical threat model – black-box adversarial attack has not been well validated and evaluated.**

In this paper, we bridge this research gap and propose a black-box adversarial attack approach, SHADOWDROID, against ML-based Android malware detection. On a high level, SHADOWDROID tries to construct a substitute model of the target malware detection system. Utilizing this substitute model, we can identify and modify the key features of a malicious app to generate an adversarial sample. During the experiment, we evaluated the effectiveness of SHADOWDROID against nine ML-based Android malware detection frameworks. It achieved successful malware evading on five platforms. Based on these results, we also discuss how to design a robust malware detection system to prevent adversarial attacks.

***Index Terms*—Adversarial attack, Android malware detection.**

I. INTRODUCTION

Android is the most popular mobile platform, and its success primarily benefits from the massive apps which provide rich functionalities to end-users. On the other hand, the coming of malicious apps has become a realistic security threat. The security community has proposed lots of approaches to improve the accuracy and efficiency of Android malware detection. In recent years, machine learning (*ML* for short) techniques have been applied to this field, and lots of approaches have been designed, such as Drebin [22], MaMaDroid [38], and DroidEvolver [50].

In practice, the issue of adversarial attacks can affect the deployments of ML-based Android malware detection solutions. The attackers try to construct adversarial examples (APKs) to evade the detection. The previous research [32], [26], [51] demonstrated the adversarial attacks under the settings of white-box or grey-box, which requires the pre-knowledge of the victim detection platforms. However, such an adversary model is not practical in the real world. In most cases, the attackers cannot obtain the victim detection platform's detailed internal technical design details, such as the adopted detection features and classification algorithms. One common scenario is that online malware detection platforms only allow users to upload APK files and check the final detection result as a black box. Therefore, there is a research question that has not been well answered, that is, *whether the attackers can evade ML-based Android malware detection without pre-knowledge*.

Our work. In this work, we consider a black-box setting, in which ***the attacker cannot know the technical design of ML-based Android malware detection platforms***. Under such a setting, we propose a practical black-box adversarial attack scheme, called SHADOWDROID, to evade the detection. The high-level idea is dynamically constructing a substitute model which is similar to the target detection platform. Based on this substitute model, to a malicious app, we identify its key features affecting the classification results and modify them, like generating an adversarial example. Note that, during the manipulation process, the app's original functionalities should not be affected. Finally, we submit the modified malicious app to the target detection platform, and the expected result is that this app is classified as benign.

To demonstrate the effectiveness of SHADOWDROID, we evaluated it on nine ML-based Android malware detection frameworks (seven for open-source versions and two for re-implementation by ourselves) with diverse technical implementations, including Drebin [22], Maldozer [35], MaMaDroid [38], DroidEvolver [50], Adagio [31], Opcode-CNN [39], CSBD [19], API-CNN [40], and Code2Pic [53]. SHADOWDROID succeeded on five platforms and failed on four ones. Considering our black-box threat model, such a success rate is reasonable. We also explored the fundamental reasons behind the results. In addition, based on the experiment results, we discuss how to design a robust malware detection system to defend adversarial example attacks, including the robust features, classification algorithms, and dataset.

Contributions. Here, we summarize the main contributions:

- **Black-box attack.** We propose SHADOWDROID, a black-box adversarial attack approach against ML-based Android malware detection. The high-level idea is to construct a substitute model, identify the key features of a malicious APK file, and generate an adversary example to evade detection.
- **Evaluations in the wild.** We carried out comprehensive experiments on nine ML-based Android malware

2690-5965/21/$31.00 ©2021 IEEE
DOI 10.1109/ICPADS53394.2021.00084

platforms to demonstrate the effectiveness of our attack. SHADOWDROID succeeded on five platforms and failed on four ones. The reasons for success and failure are discussed in depth.

Roadmap. The rest of this paper is organized as follows. Section II gives the necessary background about Android malware detection and adversarial attack. Section III introduces the high-level idea of our attack and the threat model used in this paper. The detailed attack scheme is illustrated in Section IV. The experiment design and the corresponding evaluation result are given in Section V. We discussed some limitations of this research in Section VI. The related work is reviewed in Section VII, and Section VIII concludes this work.

II. BACKGROUND

In this section, we introduce the necessary background of Android malware detection and adversarial attacks.

A. Android Malware Detection

Traditional malware detection methods can be divided into static analysis and dynamic analysis. The static analysis mainly achieves the goal by analyzing the specific contents of target apps, such as control flow graphs and requested permissions. On the other hand, dynamic analysis monitors the behaviors of apps at runtime to detect malicious features.

In recent years, machine learning has been used in Android malware detection. It is combined with static analysis or dynamic analysis, ML-based malware detection methods contain the following three steps:

1) **Feature Extraction.** The unique features of an APK file need to be extracted for further training. In general, only static features are considered due to the efficiency concern, like API calls [22], [38], [40], control flow graph [19], [31], and permissions [22], [30]. The selected features represent specific behaviors of apps, which significantly influences the accuracy of the prediction model.
2) **Feature Encoding.** The extracted information requires further being encoded into vectors before being fed into machine learning models. Standard encoding methods include one-hot encoding [40], text encoding [54], and global integer mapping [41].
3) **Model Training.** The encoded features extracted from benign and malicious apps constitute benign and malicious samples in the training dataset, respectively. They are being fed into a specific machine learning model for training. In this process, a suitable algorithm needs to be selected. Support vector machine (SVM) [27], proven powerful in classification tasks, has been used for Android application classification in many works [19], [22], [31]. With deep learning getting popular nowadays, a significant number of researchers began to use methods of deep learning, such as CNN [35], [39], [40], RNN [41], to detect Android malware.

Most of these ML-based malware detection methods achieved fairly good malware classification accuracy, say more than 90% [22], [38], [50].

B. Adversarial Attacks

Although machine learning techniques performed well in multiple fields, including Android malware detection, it suffers from the threat of adversarial attacks, as first indicated by Szegedy et al. [47]. An ML model can be simply defined as a function $y = F(x)$, which maps the input x to the corresponding output y. An attacker can craft a well-designed example x', and the differences between x' and x are very small. However, the y' outputted by the function will be widely divergent to y. Crafting the adversarial example x' from the original sample x can be formalized as below: [32]:

$$x' = x + \delta_x = x + min \|z\| \text{ s.t. } F(x+z) \neq F(x)$$

where δ_x is the minimal perturbation z contributing to model's mis-classification, according to a norm $\|\bullet\|$ which is suitable for the input domain.

In the field of Android malware detection, the input x of $F(x)$ is usually an APK file, and the output y is the corresponding label (benign or malicious). The goal of the attacker is to turn the output of a malware detection system from malicious to benign. The attacker needs to find the minimal perturbation to deceive the machine learning model without changing malware's original functions. Researchers have figured out some methods so far. For instance, Chen et al. [25] proposed to reduce detection accuracy by injecting crafted adversarial examples into training data. Chen et al. [26] used the Carlini method, Wagner Attack (C&W) [24], and Jacobian-based Saliency Map Attack (JSMA) [43] to craft adversarial examples to evade Android malware detection systems. However, most of the previous research is under the settings of white-box or grey-box, which means that the attacker has the entire or part of the knowledge of target detection systems. *However, in most cases, the details of target systems can not be easily accessed in the real world, and the assumption of black-box is more practical.*

III. THREAT MODEL AND METHODOLOGY

In this section, we give our threat model and the high-level attack approach.

A. Threat Model

As mentioned in Section II-B, most of the previous adversarial attack approaches require the knowledge of the target models, such as training datasets, detection features, and detection algorithm. However, in most cases, the attacker cannot obtain such knowledge. For example, online malware detection services only allow the users to upload samples and get the final results. Therefore, a black-box threat model is more practical.

In our threat model, ***the attacker has no prior knowledge of the target model***. She only can input APK (Android Package) samples to this detection system and check the result, say

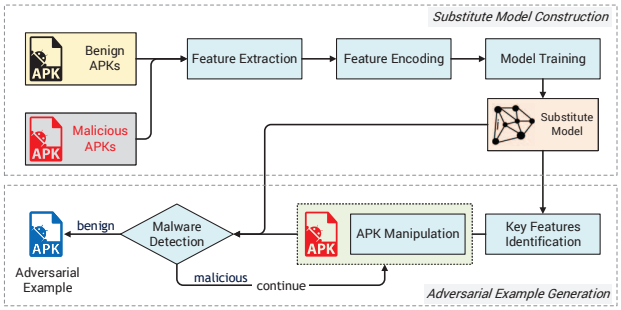

Fig. 1: Attack approach overview.

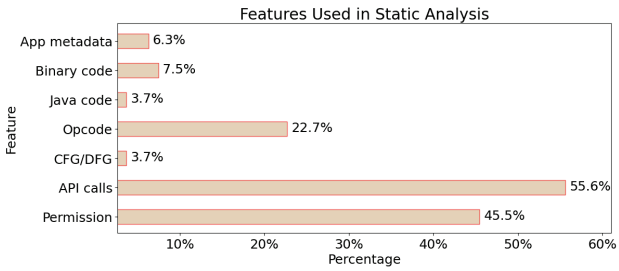

Fig. 2: Percentage of features usage.

benign or malicious. To an APK sample that should have been labeled as malicious, a successful attack is to modify this sample and make it evade the detection, say being incorrectly labeled as benign. Also, during this process, the functionalities of the modified sample are not changed.

B. Methodology

Based on our threat model, we do not know the features and detection algorithms used by the target model. Therefore, first, we try to construct a substitute model of the target model under the attack. Then, for malware, we continue to modify its features. If the substitute model misclassifies it, the adversarial example is generated. Finally, we submit the adversarial example to the target model, based on the transferability property [42], adversarial examples generated based on the substitute model may evade the target model, if the adversarial example doesn't evade the detection, we will continue to modify its features until it evades the detection or reaches the maximum number of modifications.

IV. Approach

In this section, we illustrate our attack approach in detail. As shown in Figure 1 , ShadowDroid contains two main stages – *substitute model training* and *adversarial sample generation*. The whole attack process is automated.

- **Substitute Model Construction** In this stage, we need to train a substitute model of the target model. The main challenges are how to select appropriate APK features and encoding methods for model construction. Also, a classification algorithm is needed. Finally, this substitute model should be adaptable for different malware detection systems.
- **Adversarial Example Generation** Based on the offline substitute model, we construct adversarial samples through modifying the features of malicious APKs. In consideration of attack efficiency, the modifications should not be random. Therefore, we need to determine the features to modify and not affect the core (malicious) functionalities.

A. Substitute Model Construction

After selecting an attack target (an ML-based Android malware detection system), we need to construct a substitute model which has (nearly) the same malware classification performance as the target model. Ideally, this substitute model should be suitable for any selected model. Therefore, we must carefully determine which detection features and classification algorithms should be used to build the substitute model.

Feature Selection. Many features could be used for Android malware detection, like permission, API calls, app components, and control flow graphs (CFG). To our model, the used features should meet the following criteria.

- Since our substitute model is still a malware detection system, the selected features should distinguish malicious apps from benign ones.
- Our goal is to modify the features used by our local model to make the target model mis-classify the malware, so the features used by our substitute model must be correlated with the features used by the attacked models.

According to the recent systematic literature review research of ML-based Android malware detection [37], permissions and API calls are the most frequently used static classification features among 79 detection models. As shown in Figure 2, around half of the models use permissions (45.5%) and API calls (55.6%). Also, API calls are associated with other features, including opcode, CFG (Control Flow Graph) / DFG (Data Flow Graph), Java code, and binary code. For example, adding extra API calls may also change the CFG. On the other aspect, it is not easy to modify some other features without changing the functionalities, like opcode, CFG, and DFG. Therefore, *we select permissions and API calls as the classification features*.

We ever consider another feature selection strategy, that is, using all possible classification features. However, after preliminary exploration, we found that the distributions of different features have significant differences. For example, the vector dimensions of opcode are far more than other ones. During this situation, most feature modification operations will be opcode modifications. Therefore, we adopt the former strategy, say select the most common features.

Feature Encoding. Next, we need to encode API calls and permissions as vectors to facilitate classification algorithm reading. First, we need to obtain the API calls set and permission set. (1) For API calls, we select some particular API calls that can better reflect apps' behaviors. The set mainly consists of two kinds of API calls: restricted API

TABLE I: Classification algorithm assessment results.

Algorithm	Accuracy	Recall
SVM	99.7%	99.8%
Random forest	96.1%	96.4%
Decision tree	97.2%	99.2%
Logistic regression	96.7%	97.2%
DNN	95.2%	95.2%

calls (protected by permissions) and suspicious API calls (frequently used by malware). We obtained the corresponding lists through the tools provided by PScout [23], [7] and an unofficial implementation [13] of Drebin implementation [22]. (2) For permissions, we use the official permission list of Android [1].

After obtaining the set S containing the selected API calls and all permissions, we can encode apps' features to a unified format. The status of an app using the API calls and permissions in S can reflect its behaviors, say the feature existence. Therefore, to an app x, we turn it into a vector $\varphi(x)$.

$$\varphi(x) \to \{0,1\}^{|S|}$$

To the i-th dimension:

$$\varphi_i(x) = \begin{cases} 1, if\ S_i \in x \\ 0, otherwise \end{cases} \tag{1}$$

It means that if app x contains the i-th feature, the value of its i-th dimension is set to 1, otherwise set to 0.

Model Training. In the final step, we need to select a classification algorithm to train our model, suitable for our selected detection features, and distinguish malware from benign software. Following the evaluation measures of general Android malware detection systems, we experimented with evaluating five ML algorithms: SVM (Support Vector Machines), random forest, decision tree, logistic regression, and DNN (Deep Neural Network). During the experiment, in short, we used 1000 malicious APK files and 1000 benign ones as the training set, then used the mentioned five algorithms for training, and used a set containing 1000 APK files for testing.

The experiment results are listed in Table I. We can see that SVM achieves better than other algorithms in both accuracy and recall rates. Also, SVM is suitable for high-dimensional sparse data [36], as our selected features. Therefore, SVM is used as the classification algorithm in our substitute model.

Under a formal representation, we use the training set:

$$\{(\varphi(x_1), y_1), (\varphi(x_2), y_2), ..., (\varphi(x_n), y_n)\}, y_i \in \{-1, 1\}$$

where -1 means benign and 1 means malicious. The basic idea is to find a hyperplane $\omega \in R^{|S|}$ in the sample space to distinguish different types of samples. To app x, its classification result can be expressed by the following formula:

$$f(x) = \begin{cases} 1\ (malicious), if\ (\omega, \varphi(x)) > 1 \\ -1\ (benign), otherwise \end{cases} \tag{2}$$

Note that the training set comes from the classification results of the target model. For example, we input 2,000 APK files to the target model, 850 APK files are labeled as benign and 1,150 labeled as malicious. Our substitute model will use the 2,000 APK files with the corresponding labels as the training set.

B. Adversarial Examples Generation

In this subsection, we introduce the method of generating adversarial examples. Since we have no knowledge about the target model, we can only modify the features of the malicious app based on the information of our substitute model to achieve the evading purpose. We first find the features that significantly impact the classification results and then modify these features to generate adversarial examples.

Key Feature Identification. This step is based on the JSMA (Jacobian-based Saliency Map Attack) approach [43], which can analyze the impact of input disturbance on the output result to find the corresponding adversarial disturbance. It was first used in the image classification field and can find the pixels that significantly impact the classification result. Then it iteratively modifies these pixels until the images are successfully identified as the target type. Its basic idea is to establish a mapping relationship between feature i and target output type t:

$$S(i,t) = \begin{cases} 0, if\ \dfrac{\partial f_t(x)}{\partial x_i} < 0\ or\ \sum\limits_{j \neq t} \dfrac{\partial f_j(x)}{\partial x_i} > 0 \\ |\dfrac{\partial f_t(x)}{\partial x_i}| \sum\limits_{j \neq t} \dfrac{\partial f_j(x)}{\partial x_i}, otherwise \end{cases} \tag{3}$$

in which, $S(i,t)$ represents the impact of changing the i-th feature on the classification into the t-th category. *The larger the value of $S(i,t)$, the greater the effect of the i-th feature.*

Our goal is to make malicious apps misclassified as benign ones. In our model, we use SVM as the classification model. After being trained with the training set, we can get its hyperplane ω. Therefore, for malware x, our goal can be expressed as follows:

$$(\omega, \varphi(x)) > 1 \to (\omega, \varphi(x)) < -1$$

Since SVM is not a differentiable function, we cannot calculate the value of $\frac{\partial f_j(x)}{\partial x_i}$ in Equation 3. However, to our goal, we only need to select the features i that $\omega_i < 0$, and their impact on target category is proportional to $|\omega_i|$, so $S(i,t)$ can be expressed by the following equation:

$$S(i,t) = \begin{cases} |\omega_i|, if\ \omega_i < 0 \\ 0, otherwise \end{cases} \tag{4}$$

Then we calculate $S(i,t)$ for each feature i. According to the results, we can obtain the corresponding key feature list L (by $S(i,t)$ in descending order).

APK Manipulation. After we obtain the key feature list L, the next step is to modify the (malicious) app accordingly. The modifying process is illustrated in Figure 3. Since we use API calls and permissions as features, we need to modify the app's DEX code and manifest file. Therefore we first use apktool[3]

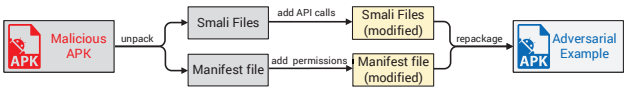

Fig. 3: APK file manipulation process.

to unpack and decompile the app, then modify features. There are two requirements for this modification process. The first is that our modification cannot break the original functions of the APP. The second is that to prevent the modified APP from being overly bloated, the number of modifications should be as small as possible.

(1) To API calls, the goal of our modification is to change the value of some dimensions from 0 to 1. If we only added it once, this modification may effectively attack the model based on the existence of the API call, but has little impact on the model that uses API call sequence as features. To solve this problem, for a single API call, we will add it multiple times. For example, the process of adding the API `getDeviceId()` is listed below:

```
.class public Ladd/redundantAPI;
.super Ljava/lang/Object;
.source "redundantAPI.java"
.method public add_method_1()V
    .locals 1
    const/4 v0, 0x0
    .local v0, "keyEvent":Landroid/view/
    KeyEvent;
    invoke-virtual {v0}, Landroid/view/
    KeyEvent;->getDeviceId()I;
    return-void
.end method
    ...
.method public add_method_n()V
    .locals 1
    const/4 v0, 0x0
    .local v0, "keyEvent":Landroid/view/
    KeyEvent;
    invoke-virtual {v0}, Landroid/view/
    KeyEvent;->getDeviceId()I;
    return-void
.end method
```

Here we create a new class `redundantAPI`. In this class, we define n methods and add API `getDeviceID()` into each method. To make the modified app not too redundant, n should be related to the size of the app and should not be set as small as possible, like 1% of the total API amount of the app.

(2) To permissions, we will modify the `AndroidManifest.xml` file. For specific permission, we only need to add it to this file. For example, if we want to add `android.permission.CALL_PHONE`, the code snippet will be inserted into the `AndroidManifest.xml` file.

```
    <uses-permission android:name="android.
    permission.CALL_PHONE"/>
```

After the modifications, we use apktool and apksigner [2] to repackage the APK file and generate a potential adversarial example. If this example could pass the detection of the target model, the adversarial example of malicious x has been generated. Otherwise, the modifications will be conducted again until achieving the evading or reaching the modification times threshold.

Note that, *during the app manipulation, we only add redundant API calls and permissions to the app*, which are never invoked. Therefore, the app functionalities are not changed.

V. Experiments and Evaluations

We implemented a prototype of SHADOWDROID and carried out various assessment experiments. Here we introduce our experiment setup and discuss the results.

A. Dataset and Malware Detection Platforms

APK Training Dataset. For training the substitute model, we used 500 malware and 500 benign apps. The malware set came from Androzoo [20], while the benign apps came from five popular app platforms, including Baidu Mobile Assistant [5], AppChina [4], PP Assistant [6], and Tencent App Store [18].

Detection Platforms. Since SHADOWDROID is designed as a general black-box attack approach, we need as many as public Android malware detection models as the attack targets. After literature review and searching GitHub, unfortunately, we found most malware detection models proposed in research papers did not open-source their code. Finally, in total, 9 models were used in our evaluation (7 for open-source versions and 2 for re-implementation by ourselves), as listed in Table II. These models use various features and classification algorithms, we implemented them and tested their accuracy, as summarized in Table II.

Among these models, though they may use similar types of features, the implementations could be different. For example, DroidEvolver [50] uses API calls as detection feature, but it use all Android API calls as the mapping space. Adagio [31] uses call graph as the feature and embeds it into a large vector space. The approach proposed by Nix et al. [40] uses call sequence as the feature and CNN as the classification algorithm, so we call it API-CNN for short. Also, we fixed its implementation bug of input sequence processing. Similarly, Opcode-CNN is short for the approach of McLaughlin et al. [39], and Code2Pic for the approach of Yen et al. [53]. Note that, the unofficial implementation of Drebin on GitHub [13] is inconsistent with the approach described in the Drebin paper [22]. Therefore, we re-implemented Drebin. Also, we re-implemented Maldozer [35].

B. Experiment Setup

For each malware detection framework, we used 1000 malicious apps for evading testing. After generating an adversarial sample, we submitted it to the framework and checked the result. If it is still labeled as malicious, SHADOWDROID will conduct another round of modifications until reaching the

TABLE II: Malware detection models used in the experiment.

Models	Implementation	Main Features	Classification Algorithm	Success Rate	Modifications Rounds	Avg Added APIs	Avg Added Permissions
Drebin [22]	Re-implementation [12]	8 features†	SVM	100%	9.5	776.4	5.86
Maldozer [35]	Re-implementation [15]	Call sequence	CNN	100%	7.8	407.4	5.03
API-CNN [40]	Unofficial open-source [9]	Call sequence	CNN	100%	2.5	232.5	1.24
Code2Pic [53]	Official open-source [10]	Java code	CNN	100%	2.5	210.4	1.15
DroidEvolver [50]	Official open-source [14]	API calls	Online learning	47.2%	47.1	6752.4	14.6
MaMaDroid [38]	Official open-source [16]	Call graph	Random forest	0	-	-	-
Adagio [31]	Official open-source [8]	Call graph	SVM	0	-	-	-
Opcode-CNN [39]	Official open-source [17]	Opcode sequence	CNN	0	-	-	-
CSBD [19]	Unofficial open-source [11]	CFG	Random forest	0	-	-	-

† Hardware features, requested & used permissions, app components, filtered Intents, restricted API calls, suspicious API calls, and network addresses.

threshold of 100 rounds. If a malicious app is classified as benign, the attack is successful.

C. Evaluations

Overall Results. The results are listed in Table II. SHADOWDROID can achieve a 100% success rate on Drebin, Maldozer, API-CNN, and Code2Pic. Also, the average modification rounds are all less than 10. Most app modification operations are adding APIs, from adding 210.4 APIs to 778.4 APIs. For DroidEvolver, 472/1000 malicious apps (47.2%) can be misclassified. To a successful attack, the average modification round is up to 47.1, adding 6752.4 APIs and 14.6 permissions. SHADOWDROID failed on four frameworks (MaMaDroid, Adagio, Opcode-CNN, and CSBD), and we discuss the causes below.

Result Analysis. To the successful cases, Drebin uses eight features such as API calls, permissions, manifest components, etc. These features are encoded as vectors according to the existence of these features, and the vector dimension is up to 217,766. Therefore, in our successful attack cases, the APK manipulation needs around 9.5 rounds. On the other hand, both Maldozer and API-CNN used API call sequences as a feature, but the difference is that API-CNN uses pseudo-dynamic analysis to get the possible API call sequences, while Maldozer cascades the API calls in all classes. Our API call injection operations have significant impacts on dimensional vectors, say fewer modification rounds.

Code2Pic uses Java codes as its feature. It first converts the DEX files into Java codes, then calculates the TF-IDF values of these Java codes. Finally, it generates a picture based on these values and uses a convolutional neural network (CNN) for training and classification. Following our attack approach, we injected lots of Smali codes when adding API calls. Therefore, it would also significantly affect its TF-IDF values, and the images generated by the values were also affected.

DroidEvolver uses five online learning models and votes to get the final classification result. In some cases, our adversarial examples only succeeded in one or two models, not affecting the final result. Therefore, SHADOWDROID cannot achieve a 100% attack successful rate.

Among those failure models, MaMaDroid uses the API call graph as its feature. It takes the call relationship between the packages or families the API belonged to as the input. Also, we set MaMaDroid in family mode. MaMadroid uses Soot [49] to create function call graphs from all entry points, and the default entry points of Soot analysis are the four major components of Android (Activity, Service, Content Provider, and Broadcast Receiver). However, our APK manipulation does not add Android components, not affecting MaMaDroid.

Adagio also uses the call graph as its feature. Unlike MaMadroid, Adagio uses Androguard to extract the call graph. However, our added APIs only impact some sub-structures of the call graph, contributing little to this model's decision. An effective method to evade Adagio might be deleting the significant nodes or edges in the call graph, which may affect the app functions.

Opcode-CNN uses opcode sequences as its feature. Our addition of API calls would only increase the invoke and return-void opcodes. Therefore, to the whole app, the opcodes we added had little impact.

CSBD used the CFG signature as its feature. It first extracts the CFG from an APK file and then calculates the signatures of the CFG's basic blocks (according to blocks' opcodes). Finally, it selects some signatures and generates a vector based on the existence of these signatures. While attacking, we added API calls to the self-defined method `add_method_i()`, which would increase many basic blocks of the control flow graph, but their signatures still kept the same. Since we can only affect the one-dimensional value, our manipulations had little impact on its vector value.

In summary, mainly the features used by the detection platform determine the results of our attack. For the successful attack cases, their features are highly correlated with the features used by our substitute model. For the failure cases, our modification operations have little impact on the features.

VI. DESIGN A ROBUST MALWARE DETECTION SYSTEM

For designing a robust malware detection system, we need to consider three aspects – feature, classification algorithm, and training dataset.

(1) *Robust Features.* When making an adversarial sample, we only add redundant code to avoid affecting the malware's functionalities. Therefore, the robust features should be able to resist redundant codes. According to our experimental results, for API-CNN (API call sequence), Maldozer (API call sequence), and Code2Pic (Java code), SHADOWDROID

achieved a 100% success rate with a few modification rounds. The main reason is that the redundant codes we insert can significantly change the Smali codes and API call sequences. On the contrary, the features requiring precise modifications to evade have strong defense capabilities. For example, our attacks on Adagio and MaMadroid failed, and they all use call graphs as the feature. For evading such a graph structure, we must accurately add nodes or edges, which is challenging under the black-box attack setting. Therefore, we can use robust features (e.g., call graph) which can resist redundant codes injections.

(2) *Robust Classification Algorithm.* Inspired by DroidEvolver, the combination of multiple algorithms may be a promising solution. Ensemble learning is a multi-classifier classification method. It obtains the final prediction result by voting for each classifier. If we want to break it, we must successfully attack most of the classifiers, which can improve the model's robustness. In our experiments, DroidEvolver and Drebin both use high-dimensional sparse vectors as features, but SHADOWDROID cannot achieve the 100% success rate on DroidEvolver. The main reason is the deployment of ensemble learning. Therefore, we recommend using ensemble learning (or similar solutions) instead of a single algorithm.

(3) *Robust Training Dataset.* We can use the adversarial training method [48] to learn adversarial examples, which can enhance the robustness of the model.

VII. Related Works

This section reviews the related work on the attacks against the ML-based Android malware detection systems.

Adversarial attack on Android malware detection is a hot research topic. However, most of these works adopt the white-box or grey-box threat model. Chen et al. [26] expanded C&W and JSMA which are originally used to craft image examples to generate adversarial examples for evading two Android malware detection model: Drebin [22] and MaMaDroid [38]. These two different models put forward different APK manipulation strategies based on the features chosen by these two classifiers. Some malware detection models use the whole graph structure as features [31], aiming at maintaining the apps' original functionality. In order to evade these models, Xu et al. [51] proposed MANIS using approaches of the n-strongest nodes and the gradient sign method to craft adversarial examples without changing any node in the original app's graph. In contrast, Grosse et al. [32] optimised method proposed in [43] to handle binary features. Demontis et al. [29] implement obfuscation or string encryption to camouflage specific parts of the app. While Rastogi et al. [45] leveraged reflection to hide some edges in the app's call graph. It is worth mentioning that most of these researches are at least under the scenario of knowing the features used by models. In comparison, our attack is under the setting of knowing nothing of the internal knowledge (e.g., features, encoding methods).

Yang et al. [52] also proposed a black-box attack method. Their method is to alter the features in bytecode that are not important to malware but could influence malware detectors' classification. However, their approach would modify the apps' original functionalities. For example, according to their experimental results, most of the variants generated by the confusion attack cannot run. On the contrary, our approach does not change the apps' original functionalities by only adding redundant components.

On other platforms, adversarial attacks also bring practical security threats. Anderson et al. [21] focused on static Windows PE malware evasion. They proposed a black-box attack based on reinforcement learning (RL). Hu et al. [33] proposed a generative adversarial network (GAN) based algorithm named MalGAN to generate adversarial malware examples. Rosenberg et al. [46] generated adversarial examples by modifying the malware's API call sequences and non-sequential features (printable strings). Demetrio et al. [28] successfully evaded the detection of MalConv [44] by modifying the header of PE files. Hu et al. [34] attacked the RNN-based malware detection platform using substitute models.

VIII. Conclusion

In this work, we propose a black-box adversarial attack scheme against ML-based Android malware detection. This scheme focuses on crafting adversarial examples under the setting of knowing nothing of target models. The experimental results have proved that our methods can attack some models successfully. Further, our approach does not influence apps' original functions, making it practical in the real world. We believe defenders should attach great importance to adversarial examples crafting techniques towards ML-based Android malware detection systems.

Acknowledgements

We thank the anonymous reviewers for their insightful comments. This work was partially supported by Shandong Provincial Natural Science Foundation, China (Grant No. ZR201911070257) and Qilu Young Scholar Program of Shandong University.

References

[1] "Android Permission List," https://developer.android.google.cn/reference/android/Manifest.permission.
[2] "apksigner," https://developer.android.google.cn/studio/command-line/apksigner.
[3] "Apktool," http://ibotpeaches.github.io/Apktool/.
[4] "AppChina," http://www.appchina.com/.
[5] "Baidu Mobile Assistant," https://shouji.baidu.com.
[6] "PP Assistant," https://www.25pp.com.
[7] "PScout," https://github.com/dlgroupuoft/PScout.
[8] "Source Code of Adagio," https://github.com/hgasco-dagio.
[9] "Source Code of API-CNN," https://github.com/vikram-mm/Android-Malware-Detection.
[10] "Source Code of Code2Pic," https://github.com/chu840121/yys.
[11] "Source Code of CSBD," https://github.com/MLDroid/csbd.
[12] "Source Code of Drebin (re-implemented version)," https://github.com/zhangjin19960527/Drebin.
[13] "Source Code of Drebin (unofficial version)," https://github.com/MLDroid/drebin.
[14] "Source Code of DroidEvolver," https://github.com/DroidEvolver/DroidEvolver.
[15] "Source Code of Maldozer (re-implemented version)," https://github.com/zhangjin19960527/Maldozer.
[16] "Source Code of MaMadroid," https://github.com/IanWE/mamadroid.

[17] "Source Code of Opcode-CNN," https://github.com/niallmcl/Deep-Android-Malware-Detection.

[18] "Tencent App Store," https://sj.qq.com.

[19] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, R. State, and Y. L. Traon, "Empirical Assessment of Machine Learning-Based Malware Detectors for Android - Measuring the Gap between In-the-Lab and In-the-Wild Validation Scenarios," *Empirical Software Engineering*, vol. 21, no. 1, pp. 183–211, 2016.

[20] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon, "AndroZoo: Collecting Millions of Android Apps for the Research Community," in *Proceedings of the 13th International Conference on Mining Software Repositories (MSR), Austin, TX, USA, May 14-22, 2016*, 2016.

[21] H. S. Anderson, A. Kharkar, B. Filar, and P. Roth, "Evading Machine Learning Malware Detection," *Black Hat*, 2017.

[22] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 23-26, 2014*, 2014.

[23] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android Permission Specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS), Raleigh, NC, USA, October 16-18, 2012*, 2012.

[24] N. Carlini and D. A. Wagner, "Towards Evaluating the Robustness of Neural Networks," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy (Oakland), San Jose, CA, USA, May 22-26, 2017*, 2017.

[25] S. Chen, M. Xue, L. Fan, S. Hao, L. Xu, H. Zhu, and B. Li, "Automated Poisoning Attacks and Defenses in Malware Detection Systems: An Adversarial Machine Learning Approach," *Computers & Security*, vol. 73, pp. 326–344, 2018.

[26] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, and K. Ren, "Android HIV: A Study of Repackaging Malware for Evading Machine-Learning Detection," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 987–1001, 2020.

[27] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000.

[28] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando, "Explaining Vulnerabilities of Deep Learning to Adversarial Malware Binaries," in *Proceedings of the Third Italian Conference on Cyber Security (ITASEC), Pisa, Italy, February 13-15, 2019*, 2019.

[29] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli, "Yes, Machine Learning Can Be More Secure! A Case Study on Android Malware Detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 4, pp. 711–724, 2019.

[30] R. Feng, S. Chen, X. Xie, L. Ma, G. Meng, Y. Liu, and S. Lin, "MobiDroid: A Performance-Sensitive Malware Detection System on Mobile Platform," in *Proceedings of the 24th International Conference on Engineering of Complex Computer Systems (ICECCS), Guangzhou, China, November 10-13, 2019*, 2019.

[31] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural Detection of Android Malware using Embedded Call Graphs," in *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security (AISec), Co-located with CCS 2013, Berlin, Germany, November 4, 2013*, 2013.

[32] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. D. McDaniel, "Adversarial Examples for Malware Detection," in *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II*, 2017.

[33] W. Hu and Y. Tan, "Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN," *CoRR*, vol. abs/1702.05983, 2017.

[34] ——, "Black-Box Attacks against RNN Based Malware Detection Algorithms," in *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*, 2018.

[35] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "MalDozer: Automatic Framework for Android Malware Detection using Deep Learning," *Digital Investigation*, vol. 24, pp. 48–59, 2018.

[36] V. Kecman, "Support vector machines–an introduction," in *Support Vector Machines: Theory and Applications*. Springer, 2005, pp. 1–47.

[37] Y. Liu, C. Tantithamthavorn, L. Li, and Y. Liu, "Deep Learning for Android Malware Defenses: a Systematic Literature Review," *Journal of the ACM*, vol. 37, 2021.

[38] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. J. Ross, and G. Stringhini, "MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models," in *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[39] N. McLaughlin, J. M. del Rincón, B. Kang, S. Y. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, and G. Ahn, "Deep Android Malware Detection," in *Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy (CODASPY), Scottsdale, AZ, USA, March 22-24, 2017*, 2017.

[40] R. Nix and J. Zhang, "Classification of android apps and malware using deep neural networks," in *Proceedings of the 2017 International Joint Conference on Neural Networks (IJCNN), Anchorage, AK, USA, May 14-19, 2017*, 2017.

[41] R. Oak, M. Du, D. Yan, H. C. Takawale, and I. Amit, "Malware Detection on Highly Imbalanced Data through Sequence Modeling," in *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security (AISec), London, UK, November 15, 2019*, 2019.

[42] N. Papernot, P. D. McDaniel, and I. J. Goodfellow, "Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples," *CoRR*, vol. abs/1605.07277, 2016.

[43] N. Papernot, P. D. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The Limitations of Deep Learning in Adversarial Settings," in *Proceedings of the 1st IEEE European Symposium on Security and Privacy (EuroS&P), Saarbrücken, Germany, March 21-24, 2016*, 2016.

[44] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, "Malware Detection by Eating a Whole EXE," in *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*, 2018.

[45] V. Rastogi, Y. Chen, and X. Jiang, "DroidChameleon: Evaluating Android Anti-malware against Transformation Attacks," in *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS), Hangzhou, China, May 08 - 10, 2013*, 2013.

[46] I. Rosenberg, A. Shabtai, Y. Elovici, and L. Rokach, "Query-Efficient Black-Box Attack Against Sequence-Based Malware Classifiers," in *The 36th Annual Computer Security Applications Conference (ACSAC), Virtual Event / Austin, TX, USA, 7-11 December, 2020*, 2020.

[47] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus, "Intriguing Properties of Neural Networks," in *Proceedings of the 2nd International Conference on Learning Representations (ICLR), Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[48] F. Tramèr, A. Kurakin, N. Papernot, I. J. Goodfellow, D. Boneh, and P. D. McDaniel, "Ensemble adversarial training: Attacks and defenses," in *Proceedings of the 6th International Conference on Learning Representations (ICLR), Vancouver, BC, Canada, April 30 - May 3, 2018*, 2018.

[49] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research (CASCON), November 8-11, 1999, Mississauga, Ontario, Canada*, 1999.

[50] K. Xu, Y. Li, R. H. Deng, K. Chen, and J. Xu, "DroidEvolver: Self-Evolving Android Malware Detection System," in *Proceedings of the 6th IEEE European Symposium on Security and Privacy (EuroS&P), Stockholm, Sweden, June 17-19, 2019*, 2019.

[51] P. Xu, B. Kolosnjaji, C. Eckert, and A. Zarras, "MANIS: Evading Malware Detection System on Graph Structure," in *Proceedings of the 35th ACM/SIGAPP Symposium on Applied Computing (SAC), online event, Brno, Czech Republic, March 30 - April 3, 2020*, 2020.

[52] W. Yang, D. Kong, T. Xie, and C. A. Gunter, "Malware Detection in Adversarial Settings: Exploiting Feature Evolutions and Confusions in Android Apps," in *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC), Orlando, FL, USA, December 4-8, 2017*, 2017.

[53] Y.-S. Yen and H.-M. Sun, "An Android Mutation Malware Detection Based on Deep Learning Using Visualization of Importance from Codes," *Microelectronics Reliability*, vol. 93, 2019.

[54] Y. Zhang, Y. Sui, S. Pan, Z. Zheng, B. Ning, I. W. Tsang, and W. Zhou, "Familial Clustering for Weakly-Labeled Android Malware Using Hybrid Representation Learning," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 3401–3414, 2020.