Android HIV: A Study of Repackaging Malware for Evading Machine-Learning Detection

Xiao Chen [1*], Chaoran Li[2], Derui Wang[2], Sheng Wen[2], Jun Zhang[2], Surya Nepal[3], Yang Xiang[2] and Kui Ren[3]

[1]School of Information Technology, Deakin University
[2]Faculty of Science, Engineering & Technology, Swinburne University of Technology
[3]Data 61, CSIRO
[4]Department of Computer Science and Engineering, University at Buffalo, State University of New York

***Abstract*—Machine learning based solutions have been successfully employed for automatic detection of malware in Android applications. However, as is known, machine learning models lack robustness to adversarial examples, which are crafted by adding minor, yet carefully chosen, perturbations to the normal inputs. So far, the adversarial examples can only deceive Android malware detectors that rely on syntactic features (*e.g.*, requested permissions, specific API calls, *etc.*), and the perturbations can only be implemented by simply modifying Android manifest. While recent Android malware detectors rely more on semantic features from Dalvik bytecode rather than manifest, existing attacking/defending methods are no longer effective due to the rising challenge in adding perturbations to Dalvik bytecode without affecting their original functionality.**

In this paper, we introduce a new highly-effective attack that generates adversarial examples of Android malware and evades being detected by the current models. To this end, we propose a method of applying optimal perturbations onto Android APK using a substitute model (*i.e.*, a Deep Neural Network). Based on the transferability concept, the perturbations that successfully deceive the substitute model are likely to deceive the original models as well (*e.g.*, Support Vector Machine in Drebin or Random Forest in MaMaDroid). We develop an automated tool to generate the adversarial examples without human intervention to apply the attacks. In contrast to existing works, the adversarial examples crafted by our method can also deceive recent machine learning based detectors that rely on semantic features such as control-flow-graph. The perturbations can also be implemented directly onto APK's Dalvik bytecode rather than Android manifest to evade from recent detectors. We evaluated the proposed manipulation methods for adversarial examples by using the same datasets that Drebin and MaMadroid (5879 malware examples) used. Our results show that, the malware detection rates decreased from 96% to 1% in MaMaDroid, and from 97% to 1% in Drebin, with just a small distortion generated by our adversarial examples manipulation method.

I. Introduction

With the growth of mobile applications and their users, security has increasingly become a great concern for various stakeholders. According to McAfee's report [30], the number of mobile malware samples has increased to 22 millions in third quarter of 2017. Symantec further reported that in Android platform, one in every five mobile applications is actually malware [43]. Hence, it is not surprising that the demand for automated tools for detecting and analysing mobile malware has also risen. Most of the researchers and practitioners in this area target Android platform, which dominants the mobile OS market. To date, there has been a growing body of research in malware detection for Android. Among all the proposed methods [16], machine learning based solutions have been increasingly adopted by anti-malware companies [32] due to their anti-obfuscation nature and their capability of detecting malware variants as well as zero-day samples. Despite the benefits of machine learning based detectors, it has been revealed that such detectors are vulnerable to adversarial examples [34], [8]. Such adversarial examples are crafted by adding carefully designed perturbations to the legitimate inputs, that force machine learning models to output false predictions [19], [33], [41].

Analogously, adversarial examples for machine learning based detection are very much like the HIV which progressively disables human beings' immune system. We chose malware detection over Android platform to assess the feasibility of using adversarial examples as a core security problem. In contrast to the same issue in other areas such as image classification, the span of acceptable perturbations is greatly reduced: an image is represented by pixel values in the feature space and the adversary can modify the feature vector arbitrarily, as long as the modified image is visually indistinguishable [50]; however, in the context of crafting adversarial examples for Android malware, a successful case must comply with the following restrictions which are much more challenging than the image classification problem: 1) the perturbation must not jeopardise malware's original functions, and 2) the perturbation to the feature space can be practically implemented in the Android PacKage (APK), meaning that the perturbation can be realised in the program code of an unpacked malware and can also be packed/built into an APK.

So far, there are already a few attempts on crafting/defending adversarial examples against machine learning based malware detection for Android platform. However, the validity of these works is usually questionable due to their impracticality. For example, Chen et al. [10] proposed to inject crafted adversarial examples into the training dataset so as to reduce detection accuracy. This method is impractical because it is not easy for attackers to gain access to the training dataset in most use cases. Grosse et al. [20] explored the feasibility of crafting adversarial examples in Android platform, but their malware detecting classifier was limited to Deep Neural Network (DNN) only. They could not guarantee

*Contact E-mail: x.chen@deakin.edu.au

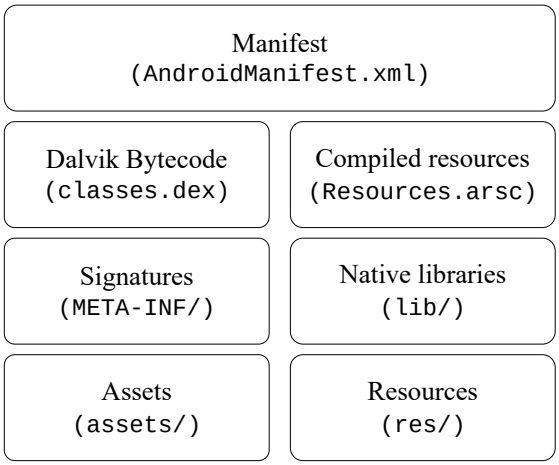


Fig. 1. File structure of APK. `AndroidManifest.xml` declares the essential information; `classes.dex` contains the Dalvik Bytecode; `resources.arsc` holds the compiled resources in binary format; META-INF, lib, assets, and res folders include the meta data, libraries, assets, and resources of the application, respectively.

the success of adversarial examples against traditional machine learning detectors such as Random Forest (RF) and Support Vector Machine (SVM). Demontis et al. [11] proposed a theoretically-sound learning algorithm to train linear classifiers with more evenly-distributed feature weights. This allows one to improve system security without significantly affecting computational efficiency. Chen et al. [9] also developed an ensemble learning method against adversarial examples. Yang et al. [49] conducted new malware variants for malware detectors to test and strengthen their detection signatures/models. According to our research, all these ideas [11], [9], [49] can only be applied to the malware detectors that adopt syntactic features (*e.g.*, permissions requested in the manifest or specific APIs in the source code [44], [1], [3], [36]). However, almost all recent machine learning based detection methods rely more on the semantic features collected from Dalvik bytecode (*i.e.*, `classes.dex`). This disables existing methods of crafting/defending adversarial examples in Android platform. Moreover, it is usually simple for existing methods to modify `AndroidManifest.xml` for the generation of adversarial examples. However, when the features are collected from classes.dex, it becomes very challenging to modify the classes.dex file without changing the original functionality due to their programmatic complexity. Therefore, existing works are not of much value in providing proactive solutions to the ever-evolving adversarial examples in terms of Android malware variants. [9], [20], [10], [11], [49].

In this paper, we propose and study a highly-effective attack that generates adversarial malware examples in Android platform, which can evade being detected by current machine learning based detectors. In the real world, defenders and attackers are always engaged in a never-ending war. To increase the robustness of Android malware detectors against malware variants, we need to be proactive and take potential adversarial scenarios into account while designing malware detectors to achieve creating such a proactive design. The work in this paper envisions an advanced method to craft Android malware adversarial examples. The results can be used for Android malware detectors to identify malware variants with the manipulated features. For the convenience of description, we selected two typical Android malware detectors, MaMaDroid [29] and Drebin [1]. Each of these two selects semantic or syntactic features to model malware behaviours.

We summarise the key contributions of this paper from different angles of view as follows:

- Technically, we propose an innovative method of crafting adversarial examples on recent machine learning based detectors for Android malware (*e.g.*, `Drebin` and `MaMaDroid`). They mainly collected features (either syntactic or semantic ones) from Dalvik bytecode to capture behaviors of Android malware. This contribution is distinguishable from the existing works [9], [20], [10], [11], [49] because can only target/protect the detectors relying on syntactic features.
- Practically, we designed an automated tool to apply the method to the real-world malware samples. The tool calculates the perturbations, modifies source files, and rebuilds the modified APK. This is a key contribution as the developed tool adds the perturbations directly to APK's `classes.dex`. This is in contrast to the existing works (*e.g.*, [9], [11]) that simply apply perturbations in `AndroidManifest.xml`. Although it is easy to implement, they cannot target/protect recent Android malware detectors (*e.g.*, [12], [38]) which do not extract features from Manifest.
- We evaluated the proposed manipulation methods of adversarial examples by using the same datasets that Drebin and MaMaDroid (5879 malware samples) used [1], [42]. Our results show that, the malware detection rates decreased from 96% to 1% in MaMaDroid, and from 97% to 1% in Drebin, with just a small distortion generated by our adversarial example manipulation method.

The rest of the paper is organised as follows. Section II gives an introduction to Android application packaging which forms the basis for adding perturbations. Section III presents the details of two typical targeted Android malware detectors as well as the attack scenarios. Section IV and V show how to craft adversarial examples against MamaDroid and Drebin, respectively, followed by discussions on open issues in Section VI. Related work comes in Section VII, and finally, Section VIII concludes the paper.

II. Android Application Package

Android applications are packaged and distributed in the form of APK files. The APK file is a `jar`-like archive that packs the application's dexcode (`.dex` files), resources, assets, and manifest file. The structure of an APK is shown in Fig.1. In particular, `AndroidManifest.xml` is designed for the meta-data such as permissions requested, definitions of components like Activities, Services, Broadcast Receivers and Content Providers. `Classes.dex` is used to store the Dalvik bytecode to be executed on the Android Runtime environment. `Res` folder contains graphics, string resources, user interface

layouts, *etc*. `Assets` folder includes non-compiled files and `METAINF` is to store the signatures and certificates.

The state-of-the-art detectors usually use machine learning based classifiers to categorise suspicious applications to be either malicious or benign ones [1], [29], [36], [3], [44]. Features employed by such classifiers are extracted from the APK archive by performing static analysis on the manifest and dexcode files. Manifest introduces the components of an application, as well as its requested permissions, activities, services, broadcast receivers, and hardware features, *etc*. Such information is presented in a binary XML format inside `AndroidManifest.xml`.

Contents presented in the manifest are informative, implying the intentions and behaviours of an application. For instance, requesting `android.permission.SEND_SMS` and `android.permission.READ_CONTACTS` permissions indicate that the application may send text messages to your contacts. Features retrieved from the manifest are usually constructed as a group of binary value vectors, each of which indicates the presence of a certain element in the manifest. Dexcode, or Dalvik Bytecode, is the operational code on Android platform. All the Java source codes are compiled and assembled into a single Dalvik Executable (`classes.dex`). Features extracted from `classes.dex`, such as CFG and DDG, contains rich semantic information and logical structure of the application. Such features are proved to have strong discriminating power for identification of malwares. Features extracted from the dexcode are presented in two forms: 1) the raw sequence of API calls, and 2) the statistic information retrieved from the call graph (*e.g.*, similarity scores between two graphs [12]).

To evade being detected by machine learning based solutions, a malware sample has to be manipulated so that the extracted features for the learning systems look benign. Intuitively, the target files to be modified are those from which the features are extracted, *i.e.*, `AndroidManifest.xml` and/or `classes.dex`. While both of these files are in binary format and are not readable by human, decompiling tools such as apktool are used to convert them into a readable format. Specifically, the binary XML can be transformed into plain-text XML, and the Dalvik bytecode can be disassembled to smali files, which are more human-friendly as intermediate presentations of bytecode. The processed `AndroidManifest.xml` and smali files can be edited and reassembled to an APK.

III. Targeted Systems and Attack Scenarios

We propose a framework to craft adversarial examples that can evade machine learning based detection. We target two typical solutions which have been widely analysed in this field, *i.e.*, MaMaDroid [29] and Drebin [1]. The semantic features that MaMaDroid uses are extracted from classes.dex, and the syntactic string values which are adopted by Drebin are retrieved from both `AndroidManifest.xml` and `classes.dex`. We provide an overview of MaMaDroid and Drebin below.

A. MaMaDroid

MaMaDroid extracts features from the CFG of an application. It uses the sequence of abstracted API calls rather than the frequency or presence of certain APIs, aiming at capturing the behavioural model of the mobile application. MaMaDroid operates in two modes, namely `family` mode and `package` mode. API calls will be abstracted to either family level or package level according to their mode. For instance, the API call sendTextMessage() is abstracted as:

$$\overbrace{\underbrace{\overbrace{\text{android}}^{\text{family}}.\underbrace{\text{telephony}}_{\text{package}}.\text{SmsManager} : \text{void sendTextMessage}()}_{\text{API call}}}$$

Family mode is more lightweight, while package mode is more fine-grained. We demonstrate the results of attacking both.

MaMaDroid firstly extracts the CFG from each application, and obtains the sequences of API calls. Then, the API calls are abstracted using either of the above-mentioned modes. Finally, MaMaDroid constructs a Markov chain, with the transition probabilities between each family or package, used as the feature vector to train a machine learning classifier. Fig. 2 illustrates the feature extraction process in MaMaDroid. Sub-graph (a) is a code snippet that has been decompiled from a malicious application; sub-graph (b) shows the call graph extracted from the source code; sub-graph (c) is the abstracted call graph generated from (b); and finally, sub-graph (d) presents the Markov chain generated based on (c).

MaMaDroid recognises nine families and 338 packages from official Android documentation. Packages, which are defined by application developer and obfuscated with identifier mangling, are abstracted as `self-defined` and `obfuscated`, respectively. Overall, there are 340 possible packages and 11 families.

Given the extracted features, MaMaDroid leverages RF, KNN, and SVM to train the malware detector and test the performance on several datasets (which were collected over different time periods). RF outperforms the other two classifiers, with its F-measure reaching 0.98 and 0.99 in the family and package modes, respectively.

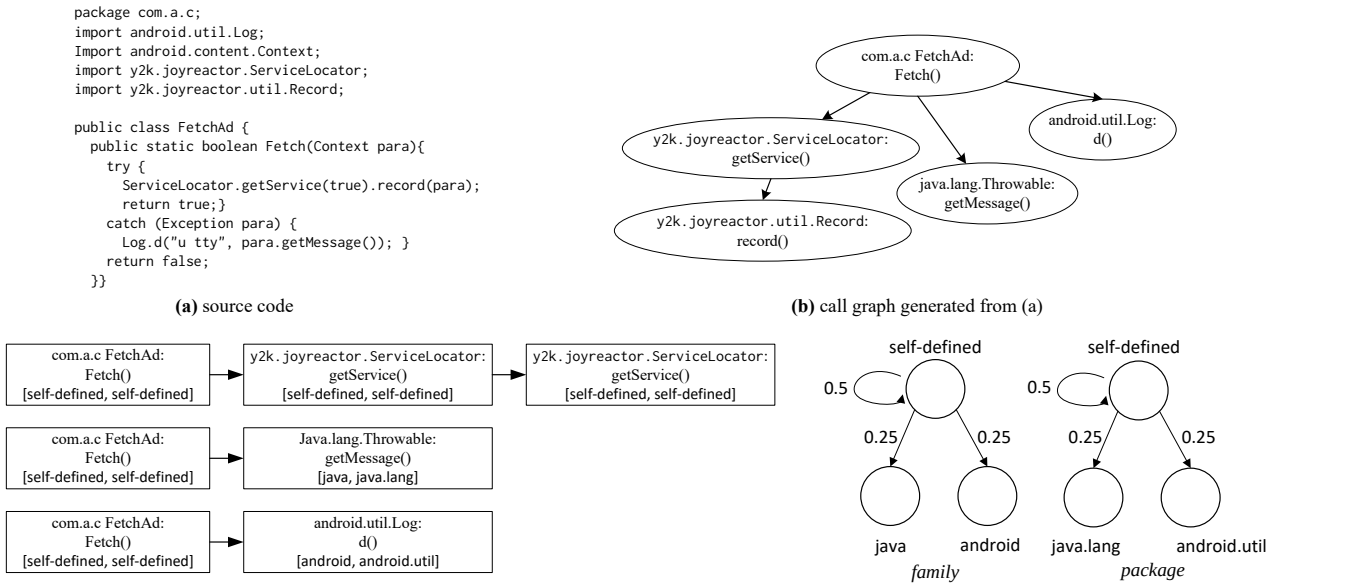
B. Drebin

Drebin is an on-device lightweight Android malware detector. Drebin extracts features from both the manifest and the disassembled dexcode through a linear sweep over `AndroidManifest.xml` and the disassembled smali files of the application. The features such as permissions, activities, and API calls are presented as strings. Eight sets of features are retrieved, as listed in Table I.

TABLE I. Overview of Drebin feature set

Drebin feature sets	
manifest	S_1 Hardware components
	S_2 Requested permissions
	S_3 App components
	S_4 Filtered intents
dexcode	S_5 Restricted API calls
	S_6 Used permissions
	S_7 Suspicious API calls
	S_8 Network addresses

The extracted features are put into a multidimensional vector (S) to create a $|S|$-D space, in which we can have 0



(a) source code

(b) call graph generated from (a)

(c) call sequence extracted from (b), with corresponding family/package abstraction in sqaure brackets

(d) Markov chains generated from (c), in family and package mode.

Fig. 2. Process of feature extraction in MaMaDroid, from (a) to (d)

or 1 value along each dimension, indicating the presence or absence of the corresponding feature. The following shows an example of the feature vector $\varphi(x)$ of a malicious application that sends premium SMS messages and thus requests certain permissions and hardware components.

$$\varphi(x) \mapsto \begin{pmatrix} \cdots \\ 0 \\ 1 \\ \cdots \\ 1 \\ 0 \\ \cdots \end{pmatrix} \begin{matrix} \cdots \\ \texttt{permission.SEND\_SMS} \\ \texttt{permission.RECORD\_AUDIO} \\ \cdots \\ \texttt{hardware.camera} \\ \texttt{hardware.telephony} \\ \cdots \end{matrix}$$

After the features being retrieved, Drebin learns a linear SVM classfier to discriminate between benign and malicious applications. The classification performance on Drebin was evaluated on a dataset consisting 5,560 malware samples and 123,453 benign applications, which are collected between August 2010 and October 2012. The *recall* on malware class has 94% with a false-positive rate of 1%.

C. Attack Scenarios

The knowledge of the target system obtained by the adversary may vary in different situations. This includes the feature set, the training set, the classification algorithm as well as the parameters. We argue that in the real world, it is not likely for the adversary to have full knowledge of the classification algorithm used in the target detector. However, the adversary can probe the detector through feeding desired inputs and getting the corresponding outputs.

In this paper, we consider the following four situations in our attack: 1) *Scenario F*: the adversary only knows the feature set; 2) *Scenario FT*: The adversary knows both the feature set and training set, but does not have the black-box access to the target detector; 3) *Scenario FB*: The adversary knows the feature set only, and has access to the target system as a black box; and 4) *Scenario FTB*: The adversary knows both the feature set and the training set, and also has access to the target system as a black box. Knowing the feature set, as a base assumption for attacking learning systems, has been widely adopted in similar works in this field [25], [6]. Note that in the FTB and FT scenarios, the adversary can only have the feature set, but he/she cannot inject new samples or modify the existing ones in the training set.

IV. Attack on MaMaDroid

A. Attack Algorithm

We introduce an evasion attack on MaMaDroid in this section. The purpose is to make a piece of malware evasive with minimal API call injections into its original smali code. We assume that we only have black-box access to the target (MaMaDroid) detector. In other words, we can get output from MaMaDroid by feeding input, but we do not know how it processes internally. There are two considerations for the features used in MaMaDroid. First, because the features are actually the state transition probabilities of the call graph, the probabilities of the transitions departing from the same node in the call graph will increment up to 1. Second, the feature value should be bounded between 0 and 1. We will address these considerations in our algorithms.

We employ two adversarial example crafting algorithms that have been widely adopted to generate evasive malware examples. To study a more effective way of attacking, we craft adversarial example by either optimising an adversarial objective function (*i.e.*, refer as $C\&W$), or perturbing influential features based on the indicative forward derivatives (*i.e.*, refer as $JSMA$). C&W and JSMA are originally developed for crafting image example, which has continuous pixel values

as the features. In our case, we are going to calculate the perturbation based on the number of API calls, which are discrete. Therefore, we need to modify plain C&W and JSMA to cater our needs. We construct a neural network F as a substitute to launch the attack. In the malware detection case, F is a binary classifier which has a 2D output. Let the input features of the original malware form an n dimensional vector like X.

1) Modified C&W: C&W crafts adversarial malware with tunable attack confidence while optimising the distortion on the original malwale features. We modify C&W to search for an adversarial malware sample through optimising an objective function with the following constraints:

$$\begin{aligned} &\min_\delta ||\delta||_2^2 + c \cdot f(X+\delta) \\ &s.t.\ X+\delta \in [0,1]^n, \\ &and\ ||X_g + \delta_g||_1 = 1, g \in 1...k. \end{aligned} \tag{1}$$

Here, δ is the perturbation to be optimised and c is a constant to balance the two terms in the objective function. We use line-search to determine the value of c. The first term in the objective function minimises the l_2 distortion on the original features, which means the change on the MaMaDroid feature should be small enough to limit the amount of code we insert into the Smali code. The second term is a specially designed adversarial loss function f. Suppose t is the ground-truth class of the current malware example X. Our goal is to make X be incorrectly classified into the other classes (in our case, the benign class). Thus, f takes the following format:

$$f(X) = max(Z(X)_t - max\{Z(X)_i : i \neq t\}, -\kappa) \tag{2}$$

in which $Z(X)$ is the pre-softmax output from the substitute F, κ is a hyper-parameter which can adjust the attack confidence and f will maximise the loss between the current model output and the correct output. In the case of MaMadroid, the features used in malware detection are the transition probabilities. Therefore, we have two constraints in the optimisation. First, each feature after perturbation should be between 0 and 1. Second, the features can be divided into k groups where each group (like g) contains the features belonging to the same Android family. Therefore, the l_1 norm of the features in each group should be equal to 1. The objective function is optimised using AdaGrad [13]. The feature values are iteratively updated until the sample is misclassified. We use either the substitute model (in scenario F and FT), or the MaMaDroid oracle (in scenario FB and FTB), which we refer as the *pilot classifier*, to determine whether an example is misclassified.

Since the current feature X is a set of probabilities, to make the perturbation viable during the code injection into the original smali code, we change the optimisation variable from δ_i on X (the perturbation on the probabilities) to ω on A (the perturbation on the number of API calls). For the perturbation on the i-th feature in group g, we have:

$$\delta_i^g = \frac{a_i^g + \omega_i^g}{a^g + \omega^g} - \frac{a_i^g}{a^g}. \tag{3}$$

Algorithm 1 C&W based Attack Method

$\boldsymbol{F}$ is the substitute, $\boldsymbol{X}$ is the input example, $\boldsymbol{t}$ is the ground truth label, $\boldsymbol{X^*}$ is the corresponding adversarial example, $\boldsymbol{a}$ is the vector of the API call numbers for each API call, ω is the change made to API calls, $\boldsymbol{Y}$ is the output label from the substitute given an input example, $\boldsymbol{c}$ is a constant balancing the distortion and the adversarial loss, $\boldsymbol{C}$ is the upper bound of $\boldsymbol{c}$ in line-search, $\boldsymbol{\kappa}$ is a hyper-parameter controlling the strength of attack. $\boldsymbol{\gamma}$ is the number of the maximal allowed gradient update iteration, and α is the step length in gradient descent.

Input: $\boldsymbol{F}$, $\boldsymbol{X}$, $\boldsymbol{a}$, $\boldsymbol{\gamma}$, $\boldsymbol{t}$, $\boldsymbol{\kappa}$, $\boldsymbol{c}$, $\boldsymbol{C}$, α.

$\boldsymbol{X^*} \leftarrow \boldsymbol{X}$
$max\_iter \leftarrow \boldsymbol{\gamma}$
$\boldsymbol{Objective_{adv}} \leftarrow \|\omega\|_2^2 + c \cdot f(\frac{a_i^g+\omega_i^g}{a^g+\omega^g} \mid \boldsymbol{F}, \boldsymbol{\kappa})$
while $\boldsymbol{c} < \boldsymbol{C}$ & $\boldsymbol{Y} = \boldsymbol{t}$ **do**
  while iter $<$ max_iter **do**
    $Compute\ gradients\ \nabla_\omega \boldsymbol{Objective_{adv}}(\boldsymbol{X^*})$
    $\omega \leftarrow \omega + clip(\alpha \cdot \nabla_\omega \boldsymbol{Objective_{adv}}(\boldsymbol{X^*}))$
    $\boldsymbol{X^*} \leftarrow \boldsymbol{X^{*g}} = \frac{a_i^g+\omega_i^g}{a^g+\omega^g}$
    $iter++$
  end while
  $\boldsymbol{c} \leftarrow \boldsymbol{c} * 10$
end while
return $\boldsymbol{X^*}$

wherein $\omega^g = \sum_i \omega_i^g$ and a_i^g is the number of API calls indicated by the i-th feature in the g-th group. We change the optimiser from δ to ω. Accordingly, we change the first term of the adversarial objective function to $||\omega||_2^2$, in order to minimise the total number of code injections.

Since we can only inject code to make adversarial examples, we apply a ReLu function (*i.e.*, $ReLu(\omega) = max(0, \omega)$) to clip ω to non-negative values after each iteration. As the result, the first constraint (*i.e.*, $\frac{a_i^g+\omega_i^g}{a^g+\omega^g} \in [0,1]$) is automatically satisfied. To satisfy the second constraint (the sum of the feature values in the same group being 1), we normalise $\sum_i \frac{a_i^g+\omega_i^g}{a^g+\omega^g}$ for each group after each gradient descent round. The detailed algorithm is in Algorithm 1.

2) Modified $JSMA$: JSMA finds adversarial examples using the forward derivatives of the classifier. JSMA iteratively perturbs important features to determine the Jacobian matrix based on the model input and output features. The method first calculates the Jacobian matrix between the input features X and the outputs from F. In the case of MaMaDroid, we want to find the Jacobian between the API call numbers A and the outputs from F, given the relationship between API call numbers and the probabilities (*i.e.*, the input features). The Jacobian can be calculated as follows:

$$J_F(A) = [\frac{\partial F(X)}{\partial X}\frac{\partial X}{\partial A}] = [\frac{\partial F_j(X)}{\partial x_i}\frac{\partial x_i}{\partial a_i}]_{i \in 1...n, j \in 0,1} \tag{4}$$

wherein i is the index of the input feature and j is the index of the output classes (in our case it is binary). x_i is the i-th feature, a_i is the corresponding i-th API call, and F_jX is the output of the substitute at the j-th class. Suppose t is the ground truth label. To make an adversarial example, $F_t(X)$ should decrease while the outputs of other classes $F_j(X), j \neq t$ are increased.

Algorithm 2 JSMA based Attack Method

$\boldsymbol{A}$ is the vector of the API call numbers for each API call, $\boldsymbol{N}$ is the vector of the sums of every API calls group, $\boldsymbol{X}$ is the input example, $\boldsymbol{X^*}$ is the corresponding adversarial example, $\boldsymbol{\gamma}$ is the number of the maximal allowed iteration, $\boldsymbol{\theta}$ is the change made to API calls, $\boldsymbol{F}$ is the substitute, $\boldsymbol{F^*}$ is the adversarial network, $\boldsymbol{Y^*}$ is the adversarial network output

Input: $\boldsymbol{A}$, $\boldsymbol{N}$, $\boldsymbol{X}$, $\boldsymbol{\gamma}$, $\boldsymbol{\theta}$,$\boldsymbol{Y^*}$

$\boldsymbol{X} \leftarrow \frac{\boldsymbol{A}}{\boldsymbol{N}}$
$\boldsymbol{X^*} \leftarrow \boldsymbol{X}$
$// \ Search\ Domain\ is\ all\ features$
$\Gamma \leftarrow \{1...|\boldsymbol{X}|\}$
$max\_iter \leftarrow \gamma$
$// \ Groundtruth\ class$
$t \leftarrow argmax\ \boldsymbol{F^*}(\boldsymbol{X^*})$
$// \ Current\ class$
$c \leftarrow argmax\ \boldsymbol{F^*}(\boldsymbol{X^*})$
while $c \neq t$ & $iter < max\_iter$ & $\Gamma \neq \emptyset$ **do**
    $Compute\ forward\ derivative\ \nabla \boldsymbol{F}(\boldsymbol{X^*})$
    $\boldsymbol{x_1},\ \boldsymbol{x_2} \leftarrow saliency\_map(\nabla \boldsymbol{F}(\boldsymbol{X^*}),\ \Gamma,\ \boldsymbol{Y^*})$
    $Translate\ \boldsymbol{X^*}\ to\ specific\ numbers\ of\ calls\ \boldsymbol{O}$
    $Modify\ \boldsymbol{O}\ at\ \boldsymbol{x_1}\ position\ in\ \boldsymbol{X^*}\ \ by\ adding\ \boldsymbol{\theta}$
    $Recalculate\ \boldsymbol{X^*}\ using\ \boldsymbol{O}$
    $Repeat\ from\ line\ \mathbf{12}\ to\ \mathbf{14}\ replacing\ \boldsymbol{x_1}\ with\ \boldsymbol{x_2}$
    $Remove\ \boldsymbol{x_1}\ \&\ \boldsymbol{x_2}\ from\ \Gamma$
    $c \leftarrow argmax\boldsymbol{F^*}(\boldsymbol{X^*})$
    $iter++$
end while
return $\boldsymbol{X}$

Based on the calculated Jacobian, we can construct a saliency map $S(A,t)$ to direct the perturbation. The value for feature i in the saliency map can be computed as:

$$S(A,t)[i] = \begin{cases} 0, if J_{it}(A) > 0\ or\ \sum_{j \neq t} J_{ij}(A) < 0, \\ |J_{it}(A)|(\sum_{j \neq t} J_{ij}(A)), otherwise. \end{cases} \quad (5)$$

According to the saliency map, we pick one API call (i) that has the highest $S(A,t)[i]$ value to perturb during each iteration. The maximum amount of allowed changes is restricted to γ. The number of the selected API call will be increased by a small amount, like θ, in each iteration. The iteration terminates when the sample is misclassified by the *pilot classifier*, or the maximum change number is reached. The detailed algorithm is introduced in Algorithm 2.

B. APK Manipulation

In our study, the development of the APK file modification method was guided by the following design goals: 1) the modified APK will keep its original functionality; and 2) the modification will not involve additional human efforts, *i.e.*, it can be applied automatically via running scripts.

As introduced in section III, the feature vector that MaMaDroid uses to classify applications are the transition probabilities between **states** (either families or packages). Intuitively, the modification approach we apply is to add a certain number of API calls from specific callers to callees into the code to change feature values in the feature space. Since we can obtain the total number of calls that go from a specific caller to a specific callee by static analysis, we can calculate how much the feature values will be affected by adding a single call.

The APK manipulation process is designed with two strategies, namely simple manipulation strategy and sophisticated manipulation strategy. The following explains their details and limitations, respectively.

Simple manipulation strategy was motivated by the process that MaMaDroid extracts and calculates its feature values. MaMaDroid extracts all API calls from `classes.dex`, and abstracts them as either their families or packages merely based on their root domain in the package names. For instance, The self-defined class `"MyClass"` in a self-defined package like `android.os.mypack`, and the system class `"StorageManager"` in the system package `android.os.storage`, will both be abstracted as `android` family or `android.os` package. By adding such self-defined classes, we are able to mislead the abstraction of API calls in MaMaDroid.

According to the above observation, we design some code blocks that can include an arbitrary number of calls from any caller to any callee. The java source code shown below is an example of adding two `android to android` calls. Arbitrary number of calls can be added by simply invoking `callee()` multiple times in the `caller()`.

```
package android.os.mypack

public class Myclass {
   public static void callee() {}
   public static void caller() {
      callee();
      callee();}}
```

Our approach proceeds by injecting the required self-defined classes into the source of the target APK, and invoking the corresponding `caller` methods in the `onCreate()` method of its entry point activity class (by locating `"android.intent.action.MAIN"` in the manifest). Since source code cannot be perfectly reverse-engineered to Java, we perform the code insertion on the smali code. As mentioned in Section II, the modified smali codes can be rebuilt to make an APK again. The following listing presents the smali code of the added Java source code (with constructor methods omitted).

```
.class public Landroid/os/mypack/Myclass;
.source "Myclass.java"

.method public static callee()V
   .locals 0
   return-void
.end method

.method public static caller()V
   .locals 0
   .line 6
   invoke-static {},
       Landroid/os/mypack/Myclass;->callee()V
   return-void
.end method
```

The described modification process can add an arbitrary number of calls from any callers to any callees, by simply runing an automated script. It also ensures that the process will not affect the functionality of the original application. However, it modifies the CFG that MaMaDroid extracted from the APK, and additionally modifies its feature values.

Simple manipulation takes advantage of the design flaw in the feature abstraction process in MaMaDroid, thus can possibly be defended by implementing white-list filter (which is not implemented in MaMaDroid), the details of this defence method and how to counter this defence is discussed in Section VI.

Sophisticated manipulation strategy is designed to bypass the white-list filter, in which system provided non-functional API calls are inserted into the smali code. For instance, invoking a `Log.d()` method in the `onCreate()` method of the entry `activity` class (*e.g.*, `com.my.project.MainActivity`), will result in adding one *self-defined to android* call in the family mode, or one *self-defined to android.util* call in the package mode. Since the calls that we inserted are in the *activity* class of the project, it is abstracted to `self-defined` or `obfuscated` according to the abstraction rule of MaMaDroid. Therefore, with sophisticated manipulation, calls only originated from self-defined or obfuscated family/package can be inserted. Such limitation decreased the evasion rate from 99% to 59% in our experiment. An example of added smali code for a `log.d()` method is presented as follows.

```
const-string p0, ""
const-string p1, ""

.line 13
invoke-static {p0, p1},
    Landroid/util/Log;->d(Ljava/lang/String;
    Ljava/lang/String;)I
```

We developed a script to automatically perform the code insertion process. We firstly prepared the above described *no-op* code blocks from each caller to each callee. These code block are independent to the application, thus can be repeatedly used in the attack. The number of calls to be inserted from specific callers to callees were calculated by our attack algorithms described in Section IV-A. Then, we used regular expression to locate the `onCreate()` method in the smali code of the entry point *activity* class, and add any necessary code blocks to the end of the `onCreate()` method. Fig. 3 demonstrates the attack process, in which the dashed lines show the process of our attack algorithm, and the solid lines illustrate our APK manipulation procedure.

C. Experiment Settings

The experiments to be presented in the following two subsections evaluate the effectiveness of crafted adversarial examples. More specifically, we are going to answer the following two questions: 1) can the modified malware sample effectively evade from the targeted detector? and 2) can the modification be easily applied to the original APK? For the convenience of experiments, we built MaMaDroid based on the source code that the authors published in their online repository[1].

1) Dataset: To evaluate the performance of the crafted adversarial examples, we use the same datasets that have been used in MaMaDroid. First, the set of benign applications consists of 5,879 clean applications collected by PlayDrone [42] in 2014 (denoted by *oldbenign* in [29]). The set of malware includes 5,560 samples that were initially used in Drebin [1] and collected between 2010 and 2012 (denoted by *drebin* in [29]). The original experiments reported in [29] also tested several combinations of other old and new datasets collected over years to evaluate the robustness of their approach. Using only one set of data does not affect our research target, *i.e.*, to craft adversarial example that can fool and evade the malware detector. The classification results on the chosen datasets are promising, of which the F-measures reach 0.88 and 0.96, in the family and package mode, respectively. Our work is to generate malware samples for evading the detection, therefore, our test set obtains only malware samples. We carefully prepare the test set by manually checking that every sample can be installed and launched on an Android smart phone. We randomly select 1,000 qualified malware samples to form the test set, leaving the rest of the malware samples, together with the benign application samples to be the training set.

As discussed in Section III-C, to simulate the scenarios where the original training dataset of the targeted detector is unknown to us (Scenario F and FB), we collected a set of malware and another set of benign applications from VirusShare [2] and APKPure [3], respectively. VirusShare dataset consists of 24,317 malware samples collected between May 2013 to March 2014, while APKPure dataset consists of 10,000 applications we crawled from its website on January 2018. The applications from APKPure are submitted to VirusTotal to examine their benignity. We discard the samples that are reported by at least one anti-virus engine as malicious. Finally, the APKPure dataset contains 9,664 application. We randomly selected 4,560 malware samples and 5,879 benign applications from VirusShare and APKPure datasets, respectively, to form the surrogat dataset (to eliminate the influence caused by different number of training samples in the original and surrogate datasets). In the FT and FTB scenarios, we use the `original` dataset to train the target detector, as well as our attack algorithm; while in the F and FB scenarios, we use the `original` dataset to train the target detector and the `surrogate` dataset to train the attack algorithm.

2) Experiment Work Flow: Given a malicious APK as the input, we firstly decompiled it with apktool, and constructed its feature vector. The attack algorithm then optimised the perturbations to be added to the feature vector, *i.e.*, the number of calls added from each caller to callee. Then, corresponding pre-designed code blocks were inserted into the smali files, which were then recompiled into a new APK. The manipulated APK was submitted to MaMaDroid oracle to get the classification result. The attack was declared successful if the modified APK was labelled as benign. This process makes sure that our attack method not only changes the feature vector, but also effectively modifies the APK. We additionally verified that all

[1]https://bitbucket.org/gianluca_students/mamadroid_code

[2]https://virusshare.com

[3]https://apkpure.com

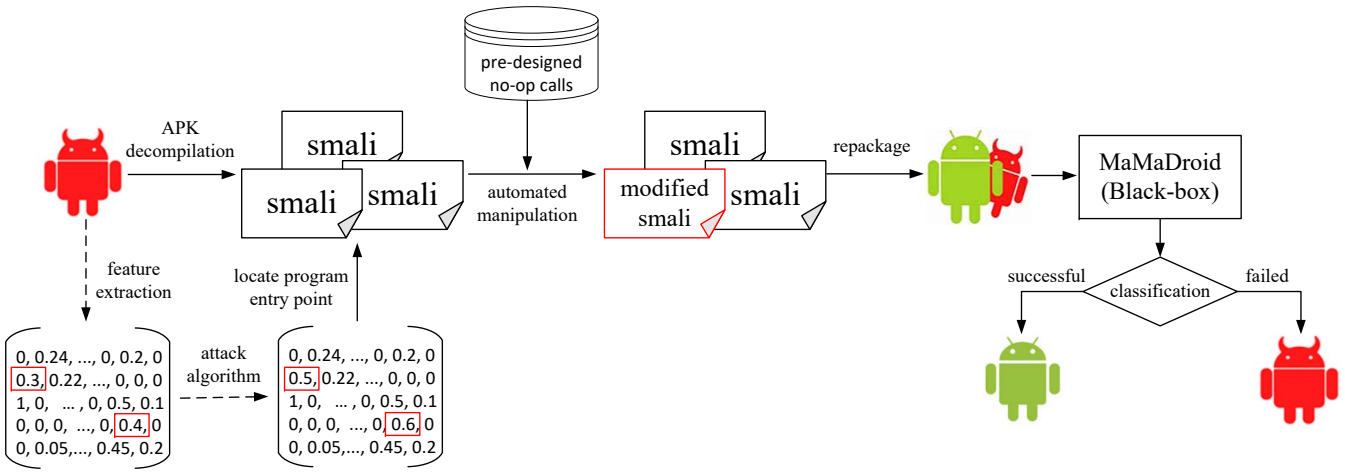


Fig. 3. The attack process: the dashed lines show the process of our attack algorithm, and the solid lines illustrate our APK manipulation procedure.

the modified APKs can be successfully installed and launched on an Android smartphone. It was difficult to verify whether the functionality was affected or not. However, we presume that since the calls we added were non-functional, they will not have changed the functionality of original APK.

As we have explained before, we will run experiments in four deliberate scenarios (refer to Section III-C). The details of the settings for each scenario are listed in Table II. In the experiments, we will train a substitute model to approximate MaMaDroid by using AdaGrad. Accordingly, a multi-layer perceptron (MLP) model will be employed, and the model architecture is shown in Table III. Each training batch will contain 256 samples and the substitute model will be trained for 100 epochs. In addition, we will introduce dropout to prevent overfitting problem in the experiments. Note that MaMaDroid trained with the original dataset will be used as benchmark for evaluation, and we only require black-box access to the pilot classifier (refer the definition to Section IV-A1).

D. Experiment Results

In [29], MaMaDroid's performance was examined on four different machine learning classifiers. They are RF, 1-Nearest Neighbour (1-NN), 3-Nearest Neighbour (3-NN), and SVM. To be consistent with the experiments in [29], we also evaluate our proposed method on the four classifiers, respectively.

TABLE II. ATTACK SCENARIOS

Scenario	Pilot Classifier	Training Set
F	Substitute	Surrogate
FT	Substitute	Original
FB	MaMaDroid	Surrogate
FTB	MaMaDroid	Original

TABLE III. SUBSTITUTE ARCHITECTURE

Layer	Substitute
Input	Feature dimension
Dense	128
Dropout	0.5
Dense	128
Dropout	0.5
Dense	2

The effectiveness of the crafted adversarial examples is evaluated in terms of evasion rate and distortion. **Evasion rate** is defined as the ratio of malware samples that are misclassified as benign, to the total number of malware samples in the testing set. **Average distortion** is defined as the number of API calls added to the smali code for each malware sample.

1) Overall results: The overall results of attack performance is presented in Fig. 4-7, where Fig. 4 and Fig. 5 are the results in the family mode using JSMA and C&W, respectively. Similarly, Fig. 6 and Fig. 7 are the results in the package mode using JSMA and C&W, respectively. In our experiment, we applied the attack on four machine learning algorithms (subfigures (a)-(d)) proposed in [29], under four real world scenarios (x-axes) as discussed in Section III-C. Simple manipulation strategy is applied in this experiment, while Sophisticated manipulation strategy is evaluated in Subsection (5). The evasion rate before attack is also reported and acted as a baseline of our attack. The evasion rate as well as the average distortion for each sample is reported. The results indicate that the proposed attack methods effectively evaded MaMaDroid in most of the real world scenarios. For instance, the evasion rate on RF increased from 4% (before attack) to 56%-99% (after attack) in the family mode, and from 3% to 58%-99% in the package mode, depending on the scenario and attack algorithm. It is worth to note that in scenario FTB, where adversary gains most knowledge of MaMaDroid, the evasion rate (JSMA) reaches 96% in SVM, 89% in RF, 86% in 3-NN, and 85% in 1-NN, with average distortion added to each malware samples being 2, 23, 21, 19, for the four algorithms, respectively. Even when the adversary only knows the feature set (scenario F), the evasion rates with JSMA reach 75%, 62%, 58%, and 61%, in the above mentioned algorithms, respectively.

2) Evaluation results by scenarios: An important observation is the improvement of attack effectiveness with the increase of adversary's knowledge of the target system. Table

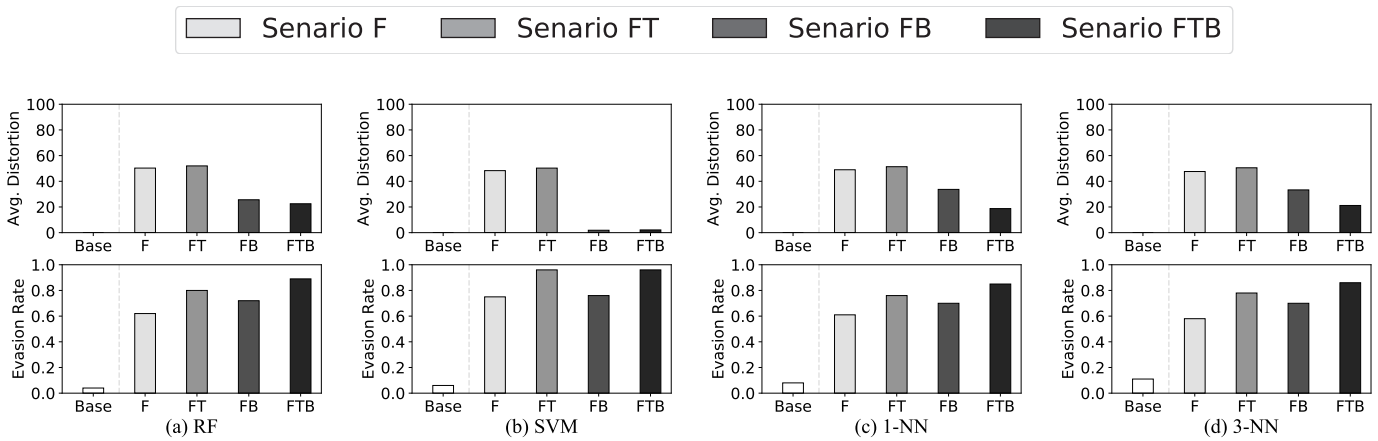


Fig. 4. The evasion rate and average distortion of adversarial examples generated by *JSMA* in the *family* mode

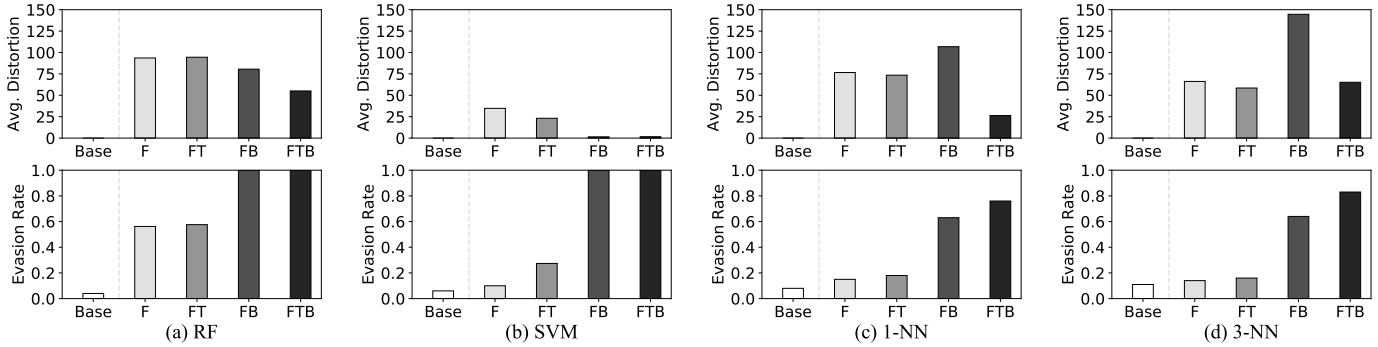


Fig. 5. The evasion rate and average distortion of adversarial examples generated by *C&W* in the *family* mode.

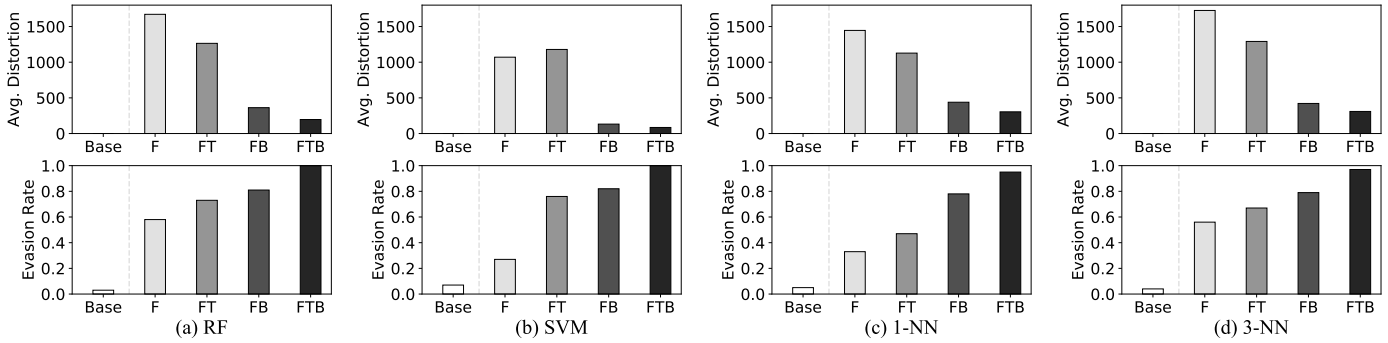


Fig. 6. The evasion rate and average distortion of adversarial examples generated by *JSMA* in the *package* mode.

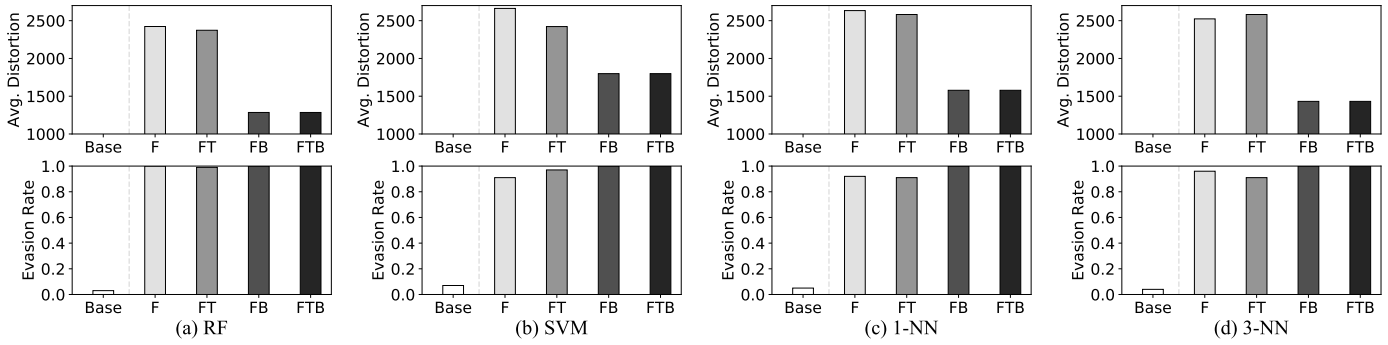


Fig. 7. The evasion rate and average distortion of adversarial examples generated by *C&W* in the *package* mode.

TABLE IV. EVASION RATE OF $JSMA$ AND $C\&W$

		C&W				JSMA			
Algorithm	Baseline	Scenario F	Scenario FT	Scenario FB	Scenario FTB	Scenario F	Scenario FT	Scenario FB	Scenario FTB
RF	0.04	0.56	0.58	0.99	0.99	0.62	0.8	0.72	0.89
SVM	0.06	0.1	0.27	0.99	0.99	0.75	0.96	0.76	0.96
1-NN	0.08	0.15	0.18	0.63	0.76	0.61	0.76	0.7	0.85
3-NN	0.11	0.14	0.16	0.64	0.83	0.58	0.78	0.7	0.86

IV lists the evasion rate of utilising JSMA and C&W in the family mode. While different level of knowledge obtained by adversary affects the evasion rate in both algorithms, the impact on each factor is different. In particular, in the scenarios which black-box access to MaMaDroid oracle is acquired (*i.e.*, FB and FTB), the evasion rate of C&W in all four algorithms are significantly higher than the evasion rate in the scenarios which black-box access is not granted (*i.e.*, F and FT). In the meanwhile, the possession of training set (F *versus* FT, FB *versus* FTB) has little impact on the evasion rate. However, the evasion rate in JSMA are to the contrast. The possession of training set influenced the evasion rate significantly, while the access to black-box model is less important.

3) Evaluation results by operation modes: As introduced in Section III, MaMaDroid runs in either the family mode or the package mode. Family mode is more lightweight, while package mode abstracts the API calls to a more fine-grained level. The original classification performance in the package mode is slightly better than that in the family mode, with the original (baseline) evasion rate falls in the range of 1%-6% on various algorithms (compared with 4%-11% in the family mode). Our experimental results indicate that the attack is more effective in the package mode than in the family mode, in terms of evasion rate. For instance, when attacking using JSMA, the evasion rate in the package mode with RF reaches 100% in scenario FTB (Fig. 6(a)), while it is 89% in the family mode in the same scenario (Fig. 4(a)). However, the average distortion of the adversarial example in the package mode is significantly higher than in the family mode. In average, 35 calls need to be added in each application in the family mode, while this number increased to 814 in the package mode. The results disclose that while using more fine-grained features slightly enhance the classification accuracy, it's resistance to our attack is significantly higher than using highly abstracted features (*i.e.*, family mode), considering that more than 23 times of number of calls need to be added for a successful evasion.

4) Evaluation results by parameters.: Fig.8(a) presented the upper bound of API calls added by JSMA for SVM, RF, 1-NN, and 3-NN. It can be found that it requires less API call insertion for attacking SVM. The possible reason is that the SVM can be well approximated by the neural network substitute. Therefore, by only inserting a few API calls, a malware can be misclassified by SVM. Adversarial example for 1-NN, 3-NN, and RF generally requires more API calls. The possible reason is that the decision boundaries of these algorithm were not well approximated by the substitute neural network. The actual API call modifications under different upper bounds are summarised in Fig.8(b). Optimising an adversarial example for SVM requires few actual modifications to achieve the optimal solution. While optimising adversarial examples towards other classifiers tends to be more dynamic. More features within the allowed range to achieve the optimum.

5) Evaluation results by manipulation strategy.: As presented in Section IV-B, two strategies can apply to the proposed APK manipulation method. In simple manipulation strategy, calls originated from any caller can be inserted into the smali code; while in sophisticated manipulation strategy, only API calls originated from `self-defined` or `obfuscated` family/package can be added. Thus, we examine the feasibility of the sophisticated manipulation strategy, by restricting that only the values of the calls originated from self-defined/obfuscated can be modified in the feature space. We craft both low-confidence and high-confidence adversarial examples by setting the hyper-parameter κ in the algorithm as 0 and 100, respectively. Fig. 9 presents the evasion rates and the corresponding average distortions. In the low-confidence attack, the evasion rate are 22%, 18%, 29%, and 28%, in RF, SVM, 1-NN, and 3-NN, respectively, with 50, 30, 24, and 25 calls in average to be added. In the high-confidence attack, 59%, 64%, 53%, 45% of the malware samples evade the detection of the above mentioned classifiers, respectively. The distortion of such attack is also high, with in average 2021, 281, 671, 824 API calls to be added to each malware sample. However, we argue that since the process of code insertion is automated, adding more number of API calls is a trivial task.

V. ATTACK ON DREBIN

A. Attack Algorithm

We adopt the Jacobian-based attack to craft an adversarial example for Drebin, since the features of Drebin are binary. $JSMA$ perturbs a feature from 0 to 1, in each iteration. Regarding the Jacobian for Drebin, we calculate it based on the following formula:

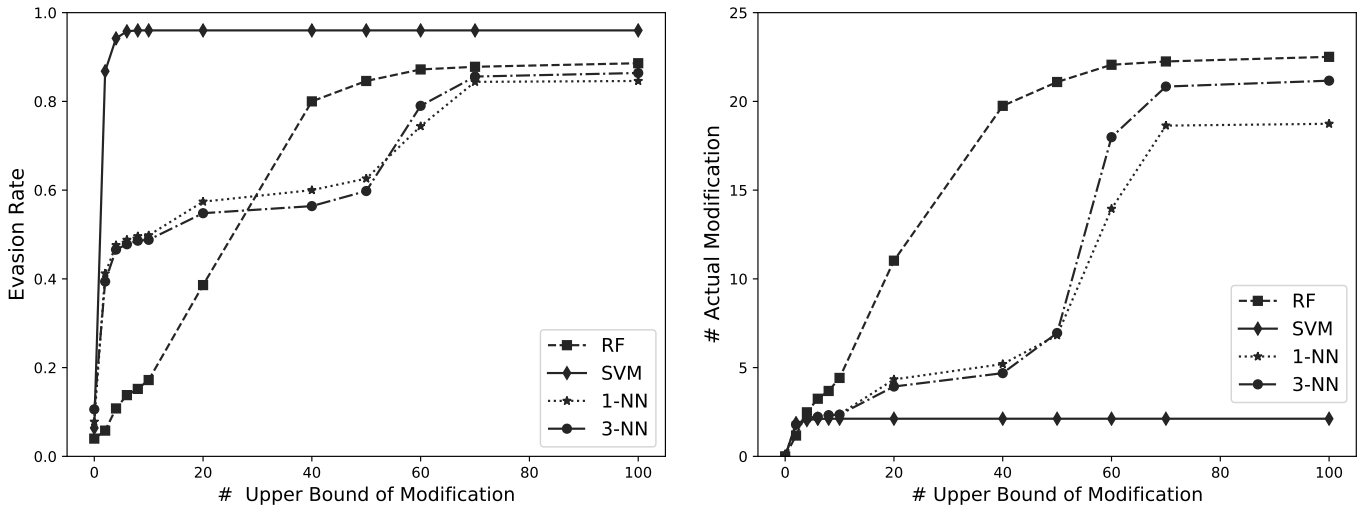
$$J_F(X) = [\frac{\partial F(X)}{\partial X}] = [\frac{\partial F_j(X)}{\partial x_i}]_{i \in 1...n, j \in 0,1} \qquad (6)$$

wherein X is the binary feature vector for Drebin and i is the classification result (*i.e.*, malware if $i = 1$). Based on the Jacobian matrix, we select the most influential feature to perturb in each iteration. In other words, we perturb the i-th feature for which $i = arg\ max_{i \in 1...n, x_i = 0} F_0(x_i)$. We change the selected one feature from 0 to 1 in each iteration, until the example is misclassified, or we reach the maximum amount of allowed change (*i.e.*, γ).

B. APK Manipulation

Drebin extracts features from both `manifest` and `dexcode`. Different from previous work that only modifies the features in manifest [20], we analyse the capability of modifying the features obtained from the `dexcode`.

As explained in Section III-B, Drebin retrieves features by applying a linear scan on related source files



(a) evasion rate (b) actual modification

Fig. 8. Different upper bound of modification in the family mode by JSMA; (a) for evasion rate, and (b) for actual modification

(`AndroidManifest.xml` and smali files), which only searches for the presence of particular strings (*e.g.*, name of API calls), rather than examining whether the calls are executed. Therefore, our strategy is to add code containing the required features but never being invoked or executed. The listing below presents an example of adding a "`suspicious API: getSystemService()`" feature to the smali code.

```
.method private addSuspiciousApiFeature()V
   .locals 1
   const-string v0, "phone"
   .line 17
   invoke-virtual {p0, v0},
      La/test/com/myapp/MainActivity;->
      getSystemService(Ljava/lang/String;)
      Ljava/lang/Object;
   move-result-object v0
   check-cast v0,
       Landroid/telephony/TelephonyManager;
   return-void
.end method
```

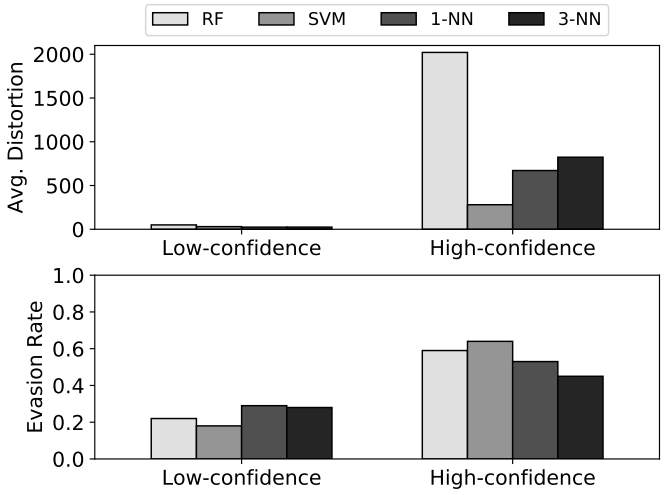


Fig. 9. Results of applying sophisticated manipulation strategy in the family mode by C&W, in Low-confidence (κ=0) and High-confidence (κ=100)

C. Experiments & Evaluations

We present our attack performance on Drebin by reporting the evasion rate and the average distortion in different real world scenarios. The **evasion rate** is defined as the ratio of malware samples that are misclassified as benign, to the total number of malware samples in the testing set. **Average distortion** is defined as the average number of code blocks added to the `classes.dex` for each malware sample. Dataset described in Section IV-C is used in the experiments.

Fig. 10 reports the result of our proposed attack. In scenario FTB, where the adversary gets most knowledge of Drebin (*i.e.*, the feature set, the training set, and black-box access), 99% of malware samples in the testing set are misclassified after the attack, with average 3.5 features to be added in each sample. While in scenario F, where the adversary obtains least knowledge of Drebin (*i.e.*, only the feature set), 60% adversarial malware examples can evade from detection.

Table V presents the average number of features inserted into each malware sample, from which we observe that the most added features are in the sets of *restricted API calls* and *suspicious API calls*.

TABLE V. NUMBER OF FEATURES ADDED IN EACH SET

source file	feature sets	avg. number added
dexcode	S_5 Restricted API calls	2.17
	S_6 Used permissions	0.1
	S_7 Suspicious API calls	1.21
	S_8 Network addresses	0.02

VI. DISCUSSION

A. Why We Are Successful

A critical challenge in crafting adversarial malware examples is how to map the perturbed feature space into the

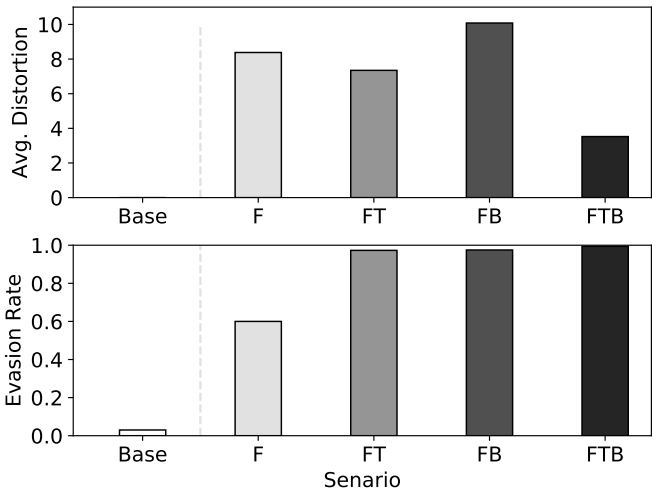


Fig. 10. The average distortion and evasion rate of adversarial example generated by *JSMA* on Drebin.

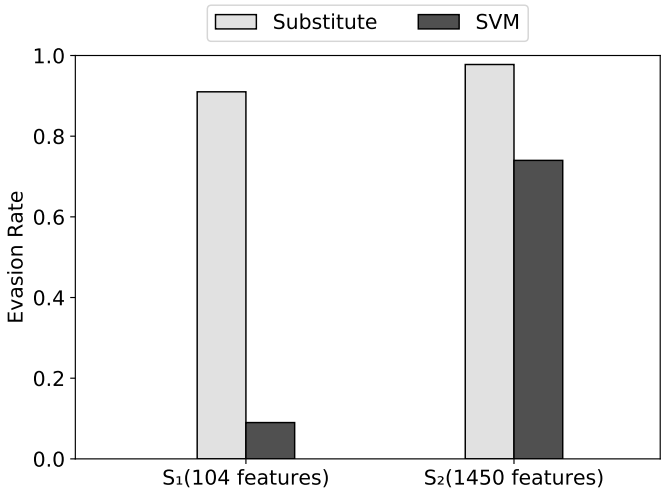


Fig. 11. Empirical study of transferability on different number of modifiable features, where S_1 includes the system-provided permissions, and S_2 includes both system-provided and user defined permissions.

problem space. In other words, the challenge is to modify APK in a way that the desired features are reflected while the code functionality remains the same. Adding a specific API call into an application's call graph without affecting its original functionality is a non-trivial task. In this paper, instead of explicitly matching the API calls, MaMaDroid makes use of their abstractions to realise the feature reflection. While using the abstracted API may be more resilient to API changes and the size of feature set stays manageable, the above challenge can be solved.

We summarise the reasons as follows. First, as described in Section IV-B, both of our proposed strategies can successfully apply the perturbed features into the application's smali code, and further recompile the manipulated code into an APK. Second, similar treatments can be applied on Drebin, as described in Section V-B. This is one of the key reasons that lead the proposed method to success. In addition, there are also some other aspects. For example, by taking the advantage of transferability of adversarial examples on various machine learning models, we train a substitute model to approximate the target detector. The optimal perturbations on the feature space can then be calculated. We have carried out and presented more empirical studies as shown in Section VI-B.

B. Transferability

The proposed attack framework is inspired by the transferability of adversarial examples among different machine learning models. In previous works, it has been demonstrated that adversarial examples crafted for one model may also be misclassified by another model [40][27]. We further investigate the limitation of tranferability by varying the number of features that are allowed to be modified in the attack.

We conducted the experiments under the assumption that the adversary can only modify a subset of the feature set S. However, we limited the adversary to two subsets denoted as S_1 and S_2. In our settings, S_1 consisted of 104 features indicating the presence of system-provided permissions in the manifest (*e.g.*, *android.permission.SEND_SMS)*, while S_2 had 1,450 features, which included system permissions as well as custom permissions defined by the application developer (*e.g.*, com.a.b.permission.DEADLY_ACTIVITY). S was used to train the original Drebin system and the substitute model, while only S_1 and S_2 were allowed to be modified, of course separately in the tests.

Fig. 11 compares the two cases of modifying S_1 and S_2 when crafting the adversarial example. While the evasion rate on the substitute model was high in both cases (91% in S_1 and 97.7% in S_2), the transferabilities to SVM were quite different. More specifically, the evasion rate slightly decreased from 97.7% to 74% when S_2 was modified, indicating that a large portion of the adversarial examples generated on the substitute model were also misclassified by Drebin. However, the evasion rate reduced dramatically from 91% to 9% if only S_1 was touched, showing that only a small portion of the adversarial examples were effective on Drebin. This observation shows that the number of modifiable features has a significant impact on the transferability between models.

C. Side Effect of Our Attack

It could be argued that adding such number of dummy calls or no-op APIs may make the application look suspicious. To investigate whether our attack will introduce 'side effect' to the original APK, we applied our attack algorithm on the benign applications, where the target of the attack could be changed to misclassify the benign applications as malware. According to the experiment results, around 20% of the applications were misclassified as malware, while the remaining 80% were still recognised as benign applications after we have applied the perturbations on those benign ones.. This observation, to some extent, indicated that our attack is "target-oriented". In other words, the perturbations will have more impacts on the malware classification while have slight influence on misleading the classifiers with benign applications.

D. Defence Methods

1) White-list filtering: Filtering out the API calls that are not in a standard Android SDK when processing API abstraction can effectively defence most of the camouflage API calls we designed in simple manipulation strategy. However, we have several ways to counter this defence method. Firstly, it is difficult to filter out all camouflage APIs as well as including all legitimate ones. For instance, some of the legitimate packages (such as com.google.firebase) are not incorporated in Android SDK, whose source code therefore are included in the application's project, which can be edited by the adversary. Thus creating and invoking a non operational method in the package `com.google.firebase` will alter the call graph, but will not be filtered out by the white-list.

2) Ensemble learning method: Ensemble of classifiers is one of the effective defences for black-box adversarial example. A number of classifiers are trained with either a subset of features, or a subset of training samples. The final classification result is made based on the decision of different classifiers (*e.g.*, a majority voting). Therefore, to break ensemble defence, an adversarial example must works for most of the classifiers. However, in our $C\&W$ attack, we can adjust the attack confidence to let the adversarial example move further in the feature space, until it is moved across the decision boundaries of all the classifiers. Another way to bypass ensemble defence is to optimise an adversarial example over the expectation of multiple sub-classifiers. However, this way of bypassing ensemble defence is time consuming and computational-heavy.

VII. Related Works

A. Adversarial Attacks to Malware Detection

Recently, there are some research works that studied the security aspect of various machine learning based malware detectors. We give them a brief overview as follows:

Srndic et al. [25] proposed an attack against PDFRrate, an online malicious PDF file detection system. They modified the fields in PDF file that was not rendered by PDF readers. They are extracted as features to discriminate malicious files from benign ones. Similar work was done by Biggio et al. [6], who leveraged gradient descent attack to evade detection. Due to the relative simplicity of the PDF file structure, it is easy to alter the file without changing the original content. Rosenberg et al. [37] proposed a black-box attack against machine learning based malware detectors in Windows OS based on analysing API calls. The attack algorithm iteratively added no-op system calls (which are extracted from benign softwares) to the binary code. The proposed method could only be applied to the detection systems that embedded the call sequence into a feature vector. It could not work if the features are statistical information extracted from the call sequence, such as similarity score or probability. Grosse et al. [20] extended an existing adversarial example crafting algorithm to the Android domain. They trained a deep feed-forward neural network classifier with the feature set adopted in Drebin. It had a comparable detection performance with Drebin. Then, they launched a white-box attack on the DNN model. In our work, we further customised the algorithm they proposed, and demonstrated a successful black-box attack on the original Drebin system. Chen et al. [10] proposed a poisoning attack for Android malware detection systems through polluting the training set of the original detectors. However, to inject tainted samples into the training set is an arguable assumption in real world scenarios. Hu et al. [21] demonstrated a generative adversarial network-based (GAN) approach to craft adversarial examples of malware.

In additions, the works [20], [10], [21] used binary features to indicate the presence of a certain permission or API. The modification on these features usually cannot affect the functionality of the applications. For instance, the adversary can request a new permission in the manifest but will not implement it in the code. Most of recent works will adopt semantic features such as the ones extracted from the control flow graphs. They usually require more cautions to tamper with if we want the application functionality not to be affected.

B. Adversarial Example Crafting

Adversarial examples for DNN models were initially studied in computer vision systems. Szegedy et al. firstly used L-BFGS to craft adversarial images for image classifiers [40]. As a result, the fast gradient sign method (FGSM) was proposed to rapidly craft adversarial examples through taking only a single step in gradients updating. [19]. Basic iterative method (BIM) improved FGSM into something that iteratively updated the features by a fixed value based on the adversarial gradients [23]. Alternatively, $JSMA$ was proposed to craft adversarial examples using the forward derivatives rather than back propagating adversarial gradients [34]. Deepfool relied on local linearities of neural networks to find adversarial examples with minimal perturbations [31]. To craft high-confidence adversarial examples, $C\&W$ introduced a special objective function to tune the confidence value for the generated examples [8].

Adversarial examples of physical objects were also developed to fool object detectors [28], [7], [2]. The adversarial physical object was made through expectation over transformation (EoT) [2]. An adversarial example is optimised based on a set of transformation functions given a raw example. The constraint for optimisation is the expectation of the distortion over the examples generated with different transformation functions.

Apart from the adversarial examples in computer vision systems, adversarial examples have also been studied for sequential data (*e.g.*, text data). Papernot et al. adopted JSMA to perturb Amazon review and change sentimental analysis results [35]. Later on, two methods, namely ADDSENT and ADDANY, were proposed to craft adversarial examples for reading comprehension systems [22]. Quite recently, Hotflip appeared as a gradient-based method to perturb text at either character or word level. [14].

C. Android Malware Detection

Researchers have developed many Android malware detection methods in the last decade. So far, there are a few survey published in this field. Readers could refer to these surveys for typical methods [17], [39], [24]. In this subsection, we mainly focus on those which were published recently and used machine learning techniques as their core algorithms.

In this field, almost all recently proposed detectors relied on semantic features to model malware behaviours. For example,

Fan et al. [15] proposed DAPASA, an approach to detect Android piggybacked applications through sensitive subgraph analysis. Xu et al. [46] leveraged the inter-component communication patterns to detect Android malware. Yang et al. [47] developed DroidMiner to scan suspicious applications to determine when they contain malicious modalities. DroidMiner can also be used to diagnose the malware family. Similar idea has also been developed by Li et al. in the work [26]. Du et al. adopted community structures of weighted function call graphs to detect Android malware. Zhang et al. [51] proposed a semantic-based approach to classify Android malware via dependency graphs. Gascon et al. [18] developed a method based on efficient embeddings of function call graphs with an explicit feature map. Furthermore, Yang et al. [48] considered user-event-driven components and the related sequences of callbacks from the Android framework to the application code. They further developed a program representation to capture those callback sequences so as to differentiate Android malware from benign applications.

As explained in Section I, existing works [9], [20], [10], [11], [49] will not work properly when recent detectors relied more on semantic features. In this paper, we presented an advanced method of crafting adversarial examples by applying perturbations directly on the APK `classes.dex` file. The generated adversarial examples will also be effective on recent detectors that rely more on semantic features.

VIII. Conclusion and Future Work

Recent studies in adversarial machine learning and computer security have shown that, due to its weakness in battling against adversarial examples, machine learning could be a potential weak point of a security system [5], [4], [45]. This vulnerability may further result in the compromise of the overall security system. The underlying reason is that machine learning techniques are not originally designed to cope with intelligent and adaptive adversaries, who can manipulate input data to mislead the learning system.

The goal of this work has been, more specifically, to show that adversarial examples can be very effective to Android malware detectors. To this end, we first introduced a DNN based substitute model to calculate optimal perturbations that also comply with the APK feature interdependence. We next developed an automated tool to implement the perturbations onto the source files (*e.g.*, smali code) of a targeted malware sample. According to the evaluation results, the Android malware detection rates decreased from 96% to 1% in MaMaDroid (*i.e.*, a typical detector that uses semantic features). We also tested Drebin (*i.e.*, a typical detector that uses syntactic features but also collects some features from classes.dex). We found Drebin's detection rates decreased from 97% to 1%. To the best of our knowledge, our work is the first one to overcome the challenge of targeting recent Android malware detectors, which mainly collect semantic features from APK's '`classes.dex`' rather than syntactic features from '`AndroidManifest.xml`'.

Our future work will focus on two areas: defence mechanisms against such attacks and attack modifications to cope with such mechanisms. For this paper, we only present in Section VI-D a brief discussion about the feasibility of a white-list filtering and an ensemble defence method. We further explored the countermeasures to those potential defending schemes. In the next stage, we plan to continue the in depth analysis of various defence mechanisms. We can divide future work into two subgroups: 1) detection of adversarial examples, and 2) making the detectors resistant to adversarial attacks. We will also compare between the effectiveness of different substitute models' architectures.

References

[1] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[2] A. Athalye and I. Sutskever. Synthesizing robust adversarial examples. *arXiv preprint arXiv:1707.07397*, 2017.

[3] Z. Aung and W. Zaw. Permission-based android malware detection. *International Journal of Scientific & Technology Research*, 2(3):228–234, 2013.

[4] M. Barreno, B. Nelson, A. D. Joseph, and J. D. Tygar. The security of machine learning. *Machine Learning*, 81(2):121–148, Nov 2010.

[5] M. Barreno, B. Nelson, R. Sears, A. D. Joseph, and J. D. Tygar. Can machine learning be secure? In *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security*, ASIACCS '06, pages 16–25, 2006.

[6] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 387–402. Springer, 2013.

[7] T. B. Brown, D. Mané, A. Roy, M. Abadi, and J. Gilmer. Adversarial patch. *arXiv preprint arXiv:1712.09665*, 2017.

[8] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, may 2017.

[9] L. Chen, S. Hou, and Y. Ye. Securedroid: Enhancing security of machine learning-based detection against adversarial android malware attacks. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, pages 362–372, 2017.

[10] 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*, 73:326–344, 2018.

[11] 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*, 2017.

[12] Y. Du, J. Wang, and Q. Li. An android malware detection approach using community structures of weighted function call graphs. *IEEE Access*, 5:17478–17486, 2017.

[13] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[14] J. Ebrahimi, A. Rao, D. Lowd, and D. Dou. Hotflip: White-box adversarial examples for text classification. In *Proceedings of ACL*, 2018.

[15] M. Fan, J. Liu, W. Wang, H. Li, Z. Tian, and T. Liu. Dapasa: detecting android piggybacked apps through sensitive subgraph analysis. *IEEE Transactions on Information Forensics and Security*, 12(8):1772–1785, 2017.

[16] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan. Android security: A survey of issues, malware penetration, and defenses. *IEEE Communications Surveys Tutorials*, 17(2):998–1022, Secondquarter 2015.

[17] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan. Android security: a survey of issues, malware penetration, and defenses. *IEEE communications surveys & tutorials*, 17(2):998–1022, 2015.

[18] 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*, pages 45–54. ACM, 2013.

[19] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *Computer Science*, 2014.

[20] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel. Adversarial examples for malware detection. In *European Symposium on Research in Computer Security*, pages 62–79. Springer, 2017.

[21] W. Hu and Y. Tan. Generating adversarial malware examples for black-box attacks based on gan. *arXiv preprint arXiv:1702.05983*, 2017.

[22] R. Jia and P. Liang. Adversarial examples for evaluating reading comprehension systems. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2021–2031, 2017.

[23] A. Kurakin, I. Goodfellow, and S. Bengio. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533*, 2016.

[24] M. La Polla, F. Martinelli, and D. Sgandurra. A survey on security for mobile devices. *IEEE communications surveys & tutorials*, 15(1):446–471, 2013.

[25] P. Laskov et al. Practical evasion of a learning-based classifier: A case study. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 197–211. IEEE, 2014.

[26] Y. Li, T. Shen, X. Sun, X. Pan, and B. Mao. Detection, classification and characterization of android malware using api data dependency. In *International Conference on Security and Privacy in Communication Systems*, pages 23–40. Springer, 2015.

[27] Y. Liu, X. Chen, C. Liu, and D. Song. Delving into transferable adversarial examples and black-box attacks. *arXiv preprint arXiv:1611.02770*, 2016.

[28] J. Lu, H. Sibai, and E. Fabry. Adversarial examples that fool detectors. *arXiv preprint arXiv:1712.02494*, 2017.

[29] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. *arXiv preprint arXiv:1612.04433*, 2016.

[30] McAfee. Mcafee mobile threat report q1, 2018. Technical report, McAfee Labs, 2018.

[31] S. M. Moosavidezfooli, A. Fawzi, and P. Frossard. Deepfool: A simple and accurate method to fool deep neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2574–2582, 2016.

[32] N/A. Machine learning for malware detection. Technical report, Kaspersky, 2017.

[33] N. Papernot, P. Mcdaniel, S. Jha, M. Fredrikson, Z. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *Proceedings - 2016 IEEE European Symposium on Security and Privacy, EURO S and P 2016*, pages 372–387, 5 2016.

[34] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 372–387. IEEE, 2016.

[35] N. Papernot, P. McDaniel, A. Swami, and R. Harang. Crafting adversarial input sequences for recurrent neural networks. In *Military Communications Conference, MILCOM 2016-2016 IEEE*, pages 49–54. IEEE, 2016.

[36] N. Peiravian and X. Zhu. Machine learning for android malware detection using permission and api calls. In *Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on*, pages 300–305. IEEE, 2013.

[37] I. Rosenberg, A. Shabtai, L. Rokach, and Y. Elovici. Generic black-box end-to-end attack against rnns and other api calls based malware classifiers. *arXiv preprint arXiv:1707.05970*, 2017.

[38] F. Shen, J. Del Vecchio, A. Mohaisen, S. Y. Ko, and L. Ziarek. Android malware detection using complex-flows. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pages 2430–2437. IEEE, 2017.

[39] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and A. Ribagorda. Evolution, detection and analysis of malware for smart devices. *IEEE Communications Surveys & Tutorials*, 16(2):961–987, 2014.

[40] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *Computer Science*, 2013.

[41] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *CoRR*, abs/1312.6199, 2013.

[42] N. Viennot, E. Garcia, and J. Nieh. A measurement study of google play. In *ACM SIGMETRICS Performance Evaluation Review*, volume 42, pages 221–233. ACM, 2014.

[43] P. Wood. Internet security threat report. Technical report, Symantec, California, 2015.

[44] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62–69. IEEE, 2012.

[45] W. Wu. Adversarial sample generation: Making machine learning systems robust for security. Technical report, Trend MIcro, 2018.

[46] K. Xu, Y. Li, and R. H. Deng. Iccdetector: Icc-based malware detection on android. *IEEE Transactions on Information Forensics and Security*, 11(6):1252–1264, 2016.

[47] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *European symposium on research in computer security*, pages 163–182. Springer, 2014.

[48] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 89–99, May 2015.

[49] 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*, pages 288–302. ACM, 2017.

[50] X. Yuan, P. He, Q. Zhu, R. R. Bhat, and X. Li. Adversarial examples: Attacks and defenses for deep learning. *CoRR*, abs/1712.07107, 2017.

[51] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1116. ACM, 2014.