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

Xiao Chen, Chaoran Li, Derui Wang, Sheng Wen, Jun Zhang, Surya Nepal, Yang Xiang, and Kui Ren

***Abstract*—Machine learning based solutions have been successfully employed for automatic detection of malware on Android. However, machine learning models lack robustness to adversarial examples, which are crafted by adding carefully chosen perturbations to the normal inputs. So far, the adversarial examples can only deceive detectors that rely on syntactic features (*e.g.*, requested permissions, API calls, *etc.*), and the perturbations can only be implemented by simply modifying application's 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.**

In this paper, we introduce a new attacking method 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 that can successfully deceive the machine learning detectors. We develop an automated tool to generate the adversarial examples without human intervention. 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 demonstrate our attack on two state-of-the-art Android malware detection schemes, MaMaDroid and Drebin. Our results show that the malware detection rates decreased from 96% to 0% in MaMaDroid, and from 97% to 0% in Drebin, with just a small number of codes to be inserted into the APK.

***Index Terms*—android malware detection, adversarial machine learning.**

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 [23], 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 [32]. 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 [13], machine learning based solutions have been increasingly adopted by anti-malware companies [24] 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 [25], [6]. Such adversarial examples are crafted by adding carefully designed perturbations to the legitimate inputs that force machine learning models to output false predictions [16], [25], [30].

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 [39]; 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 repacked/rebuilt 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. [8] 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. [17] 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 the success of adversarial examples against traditional machine learning detectors such as Random Forest (RF) and Support Vector Machine (SVM). Demontis et al. [9] 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. [7] also developed an ensemble learning

X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, and X. Yang are with the Faculty of Science, Engineering and Technology, Swinburne University of Technology, Hawthorn, VIC 3122, Australia. E-mail: xiaochen@swin.edu.au.

S. Nepal is with Data61, CSIRO, Australia.

K. Ren is with Department of Computer Science and Engineering, University at Buffalo, State University of New York, Buffalo, NY 14260, USA.

Manuscript received April 19, 2005; revised August 26, 2015.

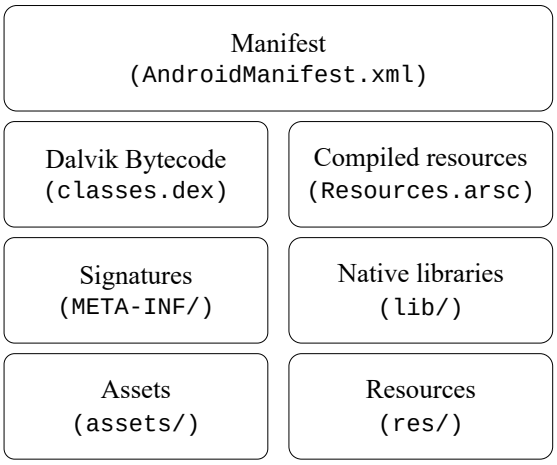


Fig. 1. File structure of APK. `AndroidManifest.xml` declares the essential information; `classes.dex` contains the Dalvik Bytecode; `resources.arsc` holds the compiled resources in binary format; META-INF, lib, assets, and res folders include the meta data, libraries, assets, and resources of the application,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

method against adversarial examples. Yang et al. [38] conducted new malware variants for malware detectors to test and strengthen their detection signatures/models. According to our research, all these ideas [9], [7], [38] 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 [33], [1], [2], [26]). 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 the manifest for the generation of adversarial examples. However, when the features are collected from the bytecode, it becomes very challenging to modify the bytecode 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 [7], [17], [8], [9], [38].

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 [22] 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 [7], [8], [9], [17] 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.*, [8], [17]) that simply apply perturbations in `AndroidManifest.xml`. Although it is easy to implement, they cannot target/protect recent Android malware detectors (*e.g.*, [10], [28]) 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], [31]. Our results show that, the malware detection rates decreased from 96% to 0% in MaMaDroid, and from 97% to 0% 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 target 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 `META-INF` is to store the signatures and certificates.

The state-of-the-art detectors usually use machine learning based classifiers to categorize applications as either malicious or benign [1], [2], [22], [26], [33]. Features employed by such classifiers are extracted from the APK archive by performing static analysis on the manifest and dexcode.

Manifest introduces the components of an application as well as its requested permissions. 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 vector of binary values, while each value indicates the presence/absence 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 Control-Flow-Graph (CFG) and Data-Dependency-Graph (DDG), contains rich semantic information and logical structure of the application. They are usually 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 [10]). Such features are proved to have strong discriminating power for identification of malware.

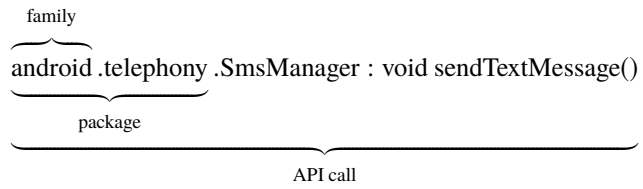
To evade being detected by machine learning based detectors, a malware sample has to be manipulated so that the extracted features for the learning systems look benign. Intuitively, the target files to be modified are those from which the features are extracted, *i.e.*, `AndroidManifest.xml` and/or `classes.dex`. 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 manifest file 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. Generally, machine learning based malware detection methods leverage two types of features: static and dynamic features. Static features are collected from disassembled APKs. Examples of such features include requested permissions, API call sequences, and control flow graphs. Dynamic features, on the other hand, are collected during the execution of the applications by monitoring their behavior and communication patterns. Since dynamic features are collected by feeding random inputs, it is more challenging to alter dynamic features than static features. Therefore, we target static features in this work, and leave the dynamic case for future work. Specifically, we target two typical solutions which have been widely analysed in this field, *i.e.*, MaMaDroid [22] and Drebin [1]. The semantic features that MaMaDroid uses are extracted from dexcode, and the syntactic string values which are adopted by Drebin are retrieved from both dexcode and manifest. 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:



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).

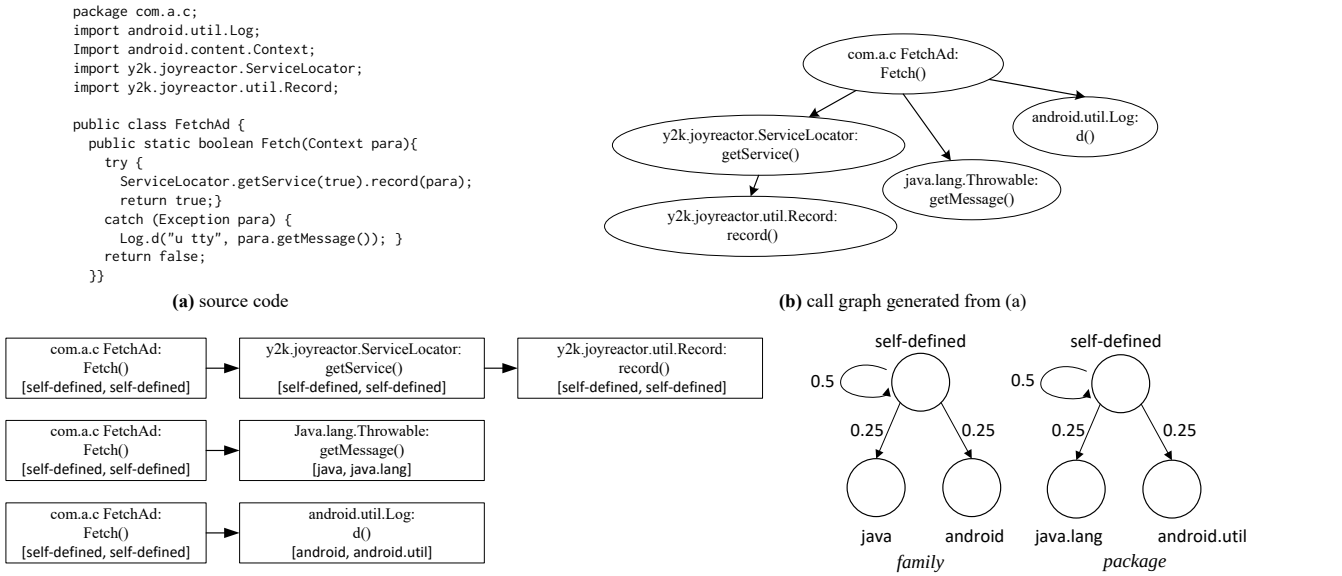
MaMaDroid recognises nine families and 338 packages from official Android documentation. Packages defined by application developer and obfuscated with identifier mangling, are abstracted as `self-defined` and `obfuscated`, respectively. Overall, there are 340 possible packages and 11 families.

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 the manifest file 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.

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



(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)

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

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

After the features being retrieved, Drebin learns a linear SVM classifier 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.

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.

Knowing the feature set, as a baseline assumption for attacking learning systems, has been widely adopted in similar works in this field [5], [8], [17], [20]. Therefore, in this paper, we consider the following four situations in our attack: 1) *Scenario F*: the adversary only knows the feature set; 2) *Scenario FB*: The adversary knows the feature set only, and can query the target detector as a black box; 3) *Scenario FT*: The adversary knows both the feature set and training set, but cannot query the target detector; and 4) *Scenario FTB*: The adversary knows both the feature set and the training set, and can also query the target system as a black box. Note that in the scenarios that allows querying the target system as a black-box (*i.e.*, scenario FTB and FB), the only information that the adversary can get is the predicted label from the black-box oracle when given an input. Also note that in the scenarios of having access to the training set (*i.e.*, scenario FTB and FT), the adversary can only have a copy of the training 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 adversarial image examples, 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 refine plain C&W and JSMA algorithms 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, denoted as X.

1) Refined 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 API calls 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 output of current model and the ground-truth. To address the aforementioned considerations, we apply two constraints in the optimisation. First, each feature after perturbation should be between 0 and 1. Second, the l_1 norm of the features in each family/package group should be equal to 1. The objective function is optimised with AdaGrad [11]. The feature values are iteratively updated until being 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 a sample 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:

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

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.

Deleting codes may jeopardise the functionality of the malware, therefore we 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.

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

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:

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

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, represented as θ, in each iteration. The iteration terminates when the sample is misclassified by the *pilot classifier*, or the maximum change number is reached.

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 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 any callers to any callees with static analysis, we therefore 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. The entry point activity can be located by searching `"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 above 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 consequently 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 filters out the API calls that are not in a standard Android SDK when processing API abstraction (which is not implemented in MaMaDroid).

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 slightly decreased the evasion rate from 99% to 93% in our family mode 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 target 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 online[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 benign applications collected by PlayDrone [31] in 2014 (denoted by *oldbenign* in [22]). 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 [22]). The original experiments reported in [22] 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 modes, 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 target detector is unknown to the adversary (*i.e.*, Scenario F and FB), we collected a set of malware samples 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 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 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 train a substitute model to approximate MaMaDroid by using AdaGrad. Accordingly, a multi-layer perceptron (MLP) model is employed. The model contains 2 fully connected hidden layers, each has 128 nodes. Each training batch contains 256 samples and the substitute model is trained for 100 epochs. In addition, we introduce dropout after each hidden layer to prevent overfitting problem in the experiments. We set the dropout rate to 0.5. Note that MaMaDroid trained with the original dataset is used as benchmark for evaluation, and we only require black-box access to the pilot classifier (refer the definition to Section IV-A).

D. Experiment Results

In [22], MaMaDroid's performance was examined on three different machine learning classifiers. They are RF, SVM, and K-Nearest Neighbour (KNN). To be consistent with the experiments in [22], we also evaluate our proposed method on these classifiers, respectively. In addition, to investigate the robustness of Deep Neural Networks (DNN) in malware detection, we leverage the features of MaMaDroid to train a DNN-based detector. Specifically, our DNN-based detector consists of five hidden layers, each with 128, 64, 64, 64, 64 neurons, respectively. The F-measures in family and package modes are 0.92 and 0.95, respectively, which is comparable

TABLE II
ATTACK SCENARIOS

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

[1]https://bitbucket.org/gianluca_students/mamadroid_code
[2]https://virusshare.com
[3]https://apkpure.com

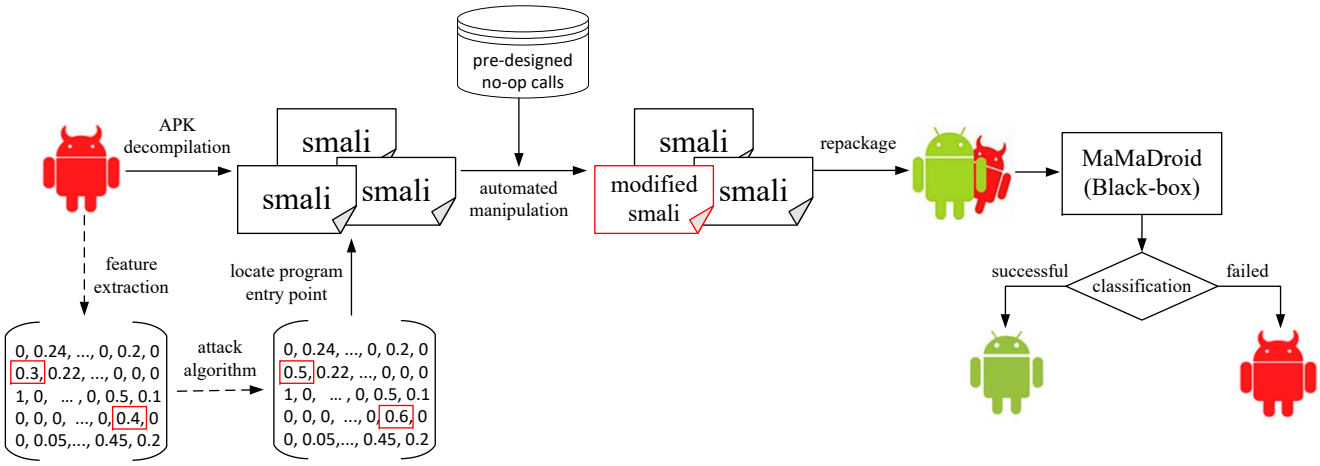


Fig. 3. The attack process: the dashed lines show the process of our attack algorithm, and the solid lines illustrate our APK manipulation procedure.

to the state-of-the-art. The proposed attack methods are also evaluated on the DNN-based detector.

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. **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 our attack are presented in Fig. 4-7. Specifically, Fig. 4 and Fig. 5 present the attack results of family mode using two attack algorithms, while Fig. 6 and Fig. 7 demonstrate the results of package mode. We applied the attack on aforementioned four machine learning algorithms (sub-figures (a)-(d)), 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 (4). The evasion rate before attack is also reported and acted as a baseline. 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 (C&W) reaches 100% in RF, 100% in SVM, 83% in 3-NN, and 100% in DNN, with average 55, 2, 65, and 1 API calls added to each malware samples for these algorithms, respectively. Even when the adversary only knows the feature set (*i.e.*, scenario F), the evasion rates with JSMA reach 62%, 75%, 58%, and 91%, 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. While different level of knowledge obtained by adversary affects the evasion rate in both algorithms, the impact on each factor is different. As demonstrated in Fig. 5 , 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 (refer to Fig. 4) 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 is more fine-grained. 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). The results of the experiment 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, 17 API calls need to be added in each application in the family mode, while this number increased to 257 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 15 times of number of calls need to be inserted for a successful evasion.

4) Evaluation results by manipulation strategy: As presented in Section IV-B, two strategies can be applied to the proposed APK manipulation method. In simple manipulation

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2019.2932228, IEEE Transactions on Information Forensics and Security

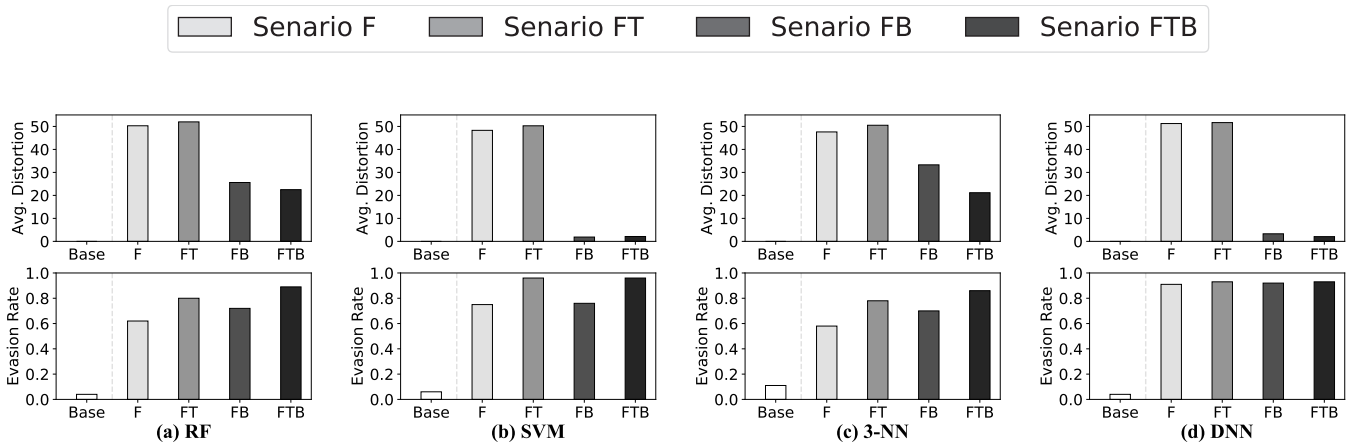


Fig. 4. The evasion rate and average distortion of adversarial examples generated by *JSMA* in the *family* mode

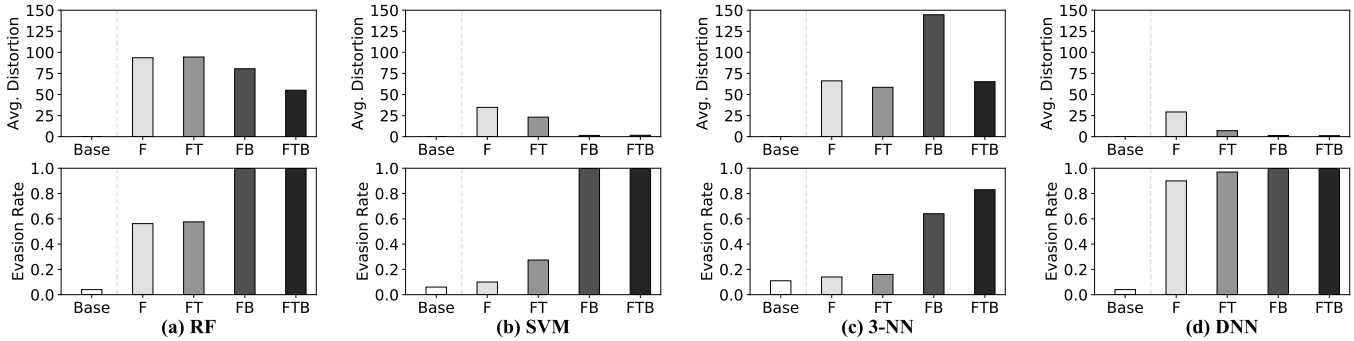


Fig. 5. The evasion rate and average distortion of adversarial examples generated by *C&W* in the *family* mode.

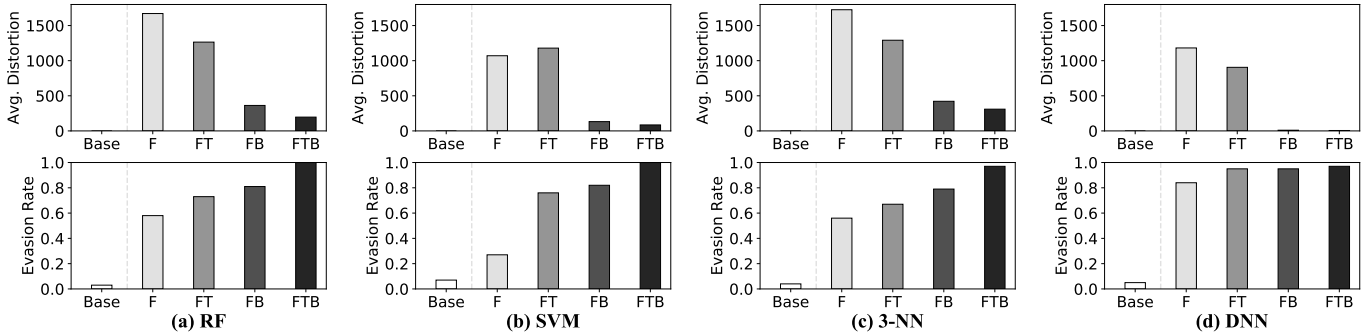


Fig. 6. The evasion rate and average distortion of adversarial examples generated by *JSMA* in the *package* mode.

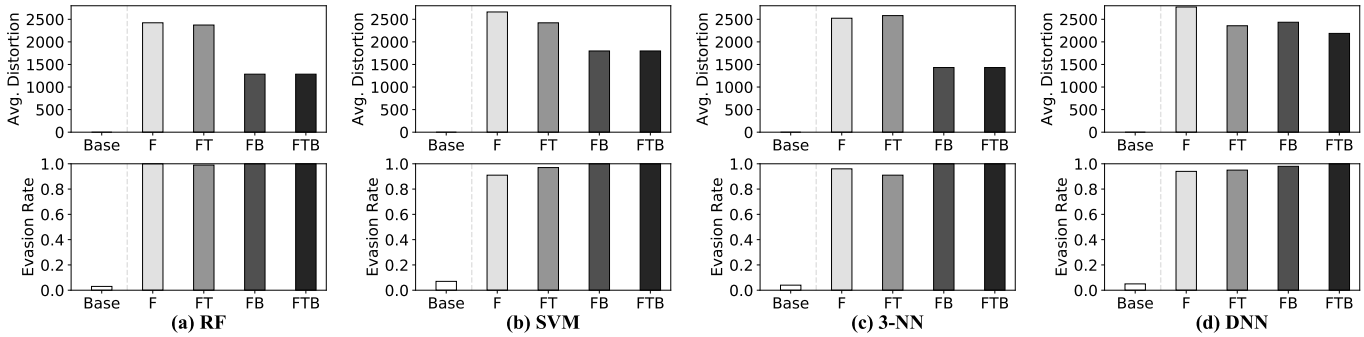


Fig. 7. The evasion rate and average distortion of adversarial examples generated by *C&W* in the *package* mode.

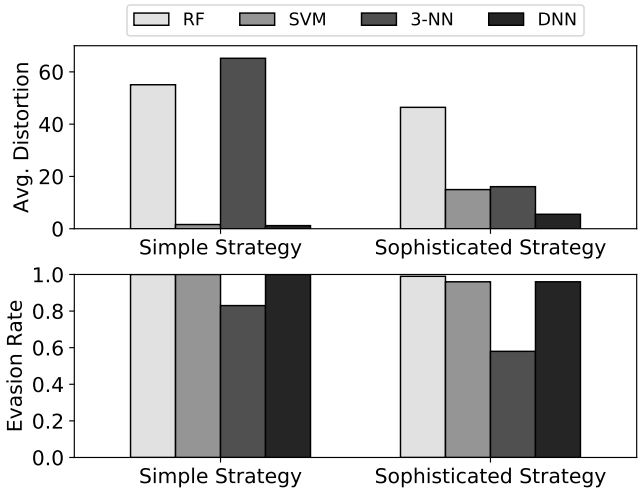


Fig. 8. Comparison of applying simple manipulation strategy and sophisticated manipulation strategy in the family mode by C&W

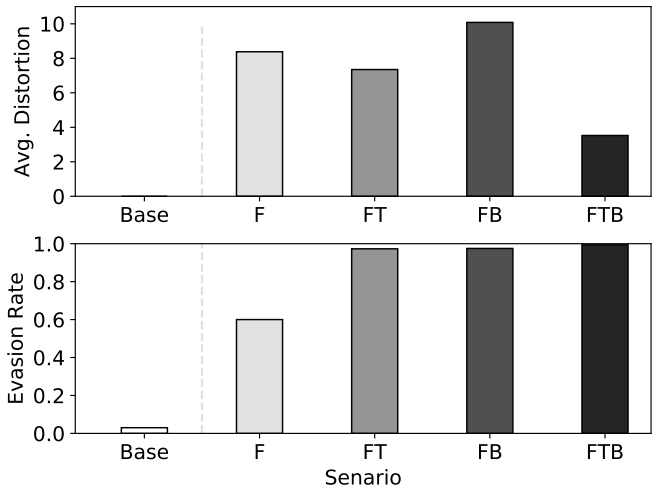


Fig. 9. The average distortion and evasion rate of adversarial example generated by *JSMA* on Drebin.

strategy, API calls originated from any caller can be inserted into the smali code; while in sophisticated manipulation strategy, only API calls originated from `android`, `google`, `self-defined` and `obfuscated` can be added. Thus, we examine the feasibility of the sophisticated manipulation strategy, by restricting that only the values of the calls originated from aforementioned families can be modified in the feature space. Fig. 8 presents the evasion rates and the corresponding average distortions of applying simple manipulation strategy and sophisticated manipulation strategy in scenario FTB, respectively. In the simple manipulation strategy, the evasion rate are 100%, 100%, 83%, and 100%, in RF, SVM, 3-NN, and DNN, respectively, with 55, 2, 65, and 9 API calls in average to be added; while in the sophisticated manipulation strategy, the evasion rate slightly decreased to 99%, 96%, 58%, and 100%, respectively. The number of API calls to be injected in the sophisticated manipulation strategy are in average 46, 14, 16, and 9, respectively. The results demonstrate that sophisticated manipulation strategy can also achieve a high evasion rate with only a small number of API calls to be injected into the APK.

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:

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

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 [17], 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 (`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
```

C. Experiments & Evaluations

We present our attack performance on Drebin by reporting the evasion rate and the average distortion in different real world scenarios. Dataset described in Section IV-C is used in the experiments.

Fig. 9 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 output from Drebin oracle), 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 III 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 III
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. Comparison with Existing works

We compare our attack method with another two works in evading machine learning based malware detection. Chen et al. [8] proposed to poison the training dataset to mislead machine learning detectors. Grosse et al. [17] proposed a white-box attack against deep learning based malware detection models. Both of these works require the access to the feature set, the training set and the machine learning model, therefore we compare our results of scenario FTB with them, which requires the same knowledge of the target model. However, the comparison works make additional assumptions. [8] further assumes that the adversary is capable of injecting tainted samples into the training set, and [17] only considers the situation that the adversary knows the detailed structure and parameters of the targeting machine learning model (*i.e.*, white-box). These assumptions are more restrictive that are unlikely to happen in real world scenarios.

Table IV presents the evasion rate of our methods and the methods proposed in [8] and [17]. The results show that our methods outperform the compared methods in terms of evasion rate.

TABLE IV
COMPARISON WITH EXISTING WORKS (EVASION RATE)

	MaMaDroid				Drebin
	RF	SVM	3-NN	DNN	
Our JSMA Attack	**89%**	**96%**	**86%**	**93%**	**99.4%**
Our C&W Attack	**100%**	**100%**	**83%**	**100%**	—
Chen et al. [8]	—	68.95%	—	—	75.2%
Grosse et al. [17]	—	—	—	—	69.35%

B. Applicability of Our Attack

A great number of machine learning based Android malware detection techniques have been proposed in the past few years. The main differences and key contributions of these techniques are the features they extracted to profile the malware samples. As we could not demonstrate the effectiveness of the proposed method on every machine learning detector, we therefore selected two typical detectors, Drebin and MaMaDroid that use syntactic and semantic features, respectively. In prior works, they have been selected as baseline methods to evaluate the performance of adversarial attacks, e.g., [8], [17].

Our proposed attack framework can be applied in most machine learning based detectors, which extract features from either manifest or bytecode of an application. We just need to refine the attack algorithm according to the constraint and inter-dependency of the features used in the target detector, just as what we have done for attacking MaMaDroid and Drebin.

C. Artifacts in Our Attack

It could be argued that adding a certain number of dummy calls or no-op APIs, such as logging output and reading files, introduces artifacts into the APK, which may make the application look suspicious. To investigate whether our attack will introduce such side effect to the original APK, we observe the prevalence of such no-op API calls (*e.g.* android.util.log) in the applications in the wild. According to our observation on the experiment dataset, we find that it is normal for an application, either benign or malicious, to have a certain number of no-op API calls. For example, 17.9% benign applications and 16.3% malware samples in the dataset have more than 100 `android.util.log()` calls in their source code. The percentages further increased to 28.7% and 40% for the applications to have more than 50 `android.util.log()` calls, in benign and malicious applications, respectively. There is no specific indication whether malware samples or benign applications tend to have more such calls than the other. Note that the average number of calls we inserted to craft adversarial examples is only 17 for family mode (refer to Fig. 4), and 257 for package mode (refer to Fig. 6). Therefore, the injected code will not bring strong indication that suggests an application to be benign or malicious.

D. Defending Methods

1) Adversarial training method: The idea of adversarial training is to recursively feed crafted adversarial examples into the training dataset to strengthen the robustness of the machine learning model. We evaluate the effectiveness of adversarial training method on the proposed attack. Fig. 10 shows the F-measure of benign and malicious applications with the proposed adversarial training method, with varying percentage of adversarial malware samples added in the training set. Note that the malware test set contains the equal number of original and adversarial malware samples. The F-measure of benign and malicious samples increased from 69% to 80% and from 64% to 82.5% by adding 1% of adversarial examples into the training dataset, respectively. Their F-measure further increased to 83% and 87%, respectively, when adding more adversarial examples into the training dataset. Although this defending method is simple and effective, it strongly relies on the a priori knowledge of the attack algorithm.

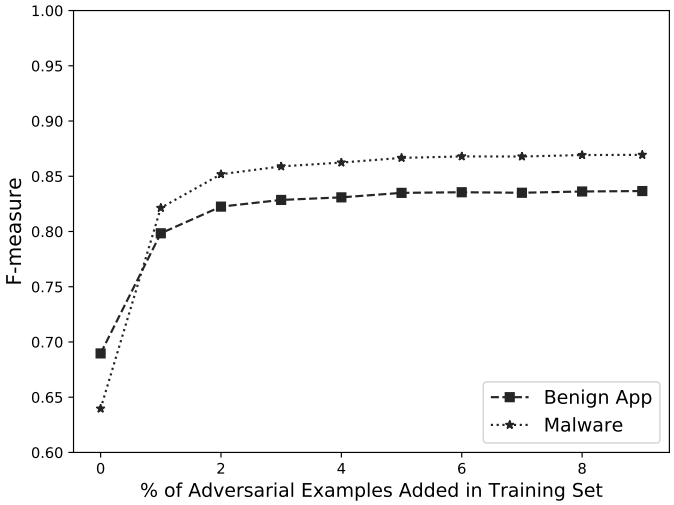


Fig. 10. F-measure of benign and malicious applications with the proposed adversarial training defending method on MaMaDroid, with varying percentage of adversarial malware samples added in the training set.

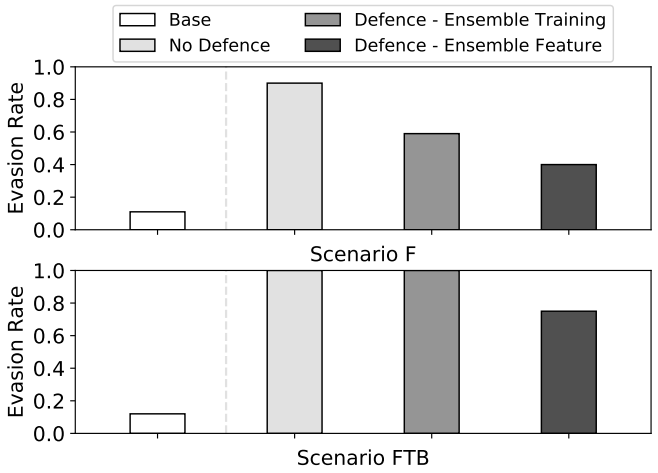


Fig. 11. Evasion rate of applying ensemble learning defence mechanism on MaMaDroid family mode in two scenarios (FTB and F). **Base**: the evasion rate before attack as a baseline; **No Defence**: attack the model without implementing the defence; **Defence - Ensemble Training**: attack the model with implementing ensemble learning method as defence, in which each of 10 classifier is trained with 1/10 training samples; **Defence - Ensemble Feature**: attack the model with implementing ensemble learning method as defence, in which each of 10 classifier is trained with 1/10 features.

2) Ensemble learning method: Ensemble of classifiers is one of the effective defences for black-box adversarial example. Instead of training one classifier using the full feature set and all training samples, a number of sub-classifiers are trained with either a subset of features, or a subset of training samples. The final classification result is then made based on the decision of different sub-classifiers with a specific rule, such as majority vote. We demonstrate the effectiveness of ensemble learning defence method against our attack. Two scenarios are considered: scenario F and scenario FTB. In scenario F, the attacker cannot query the target model, therefore he/she is not aware of the existence of the defence mechanism. In scenario FTB, the attacker can query the target model with the defence mechanism implemented, and get its prediction result.

Fig. 11 presents the results of applying ensemble learning method to defend our C&W attack on MaMaDroid (family mode), in aforementioned two scenarios, respectively. Two ensemble strategies are implemented in both scenarios, in which each of 10 classifiers is trained with either 1/10 training samples, or 1/10 of features. Evasion rates with and without defence are reported. The results suggest that the ensemble learning method is effective in defending the attack when the attacker has least knowledge of the target model (i.e., scenario F), where the evasion rate decreased from 90% to 40-59% in different ensemble strategies. However, when the attacker is capable to query the target model (e.g., the model is implemented as an online service), the defence method cannot effectively defend the proposed attack.

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. [20] proposed an attack against PDFrate, 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. [5], 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. [27] 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. [17] 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. [8] 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. [18] demonstrated a generative adversarial network-based (GAN) approach to craft adversarial examples of malware.

In additions, the works [17], [8], [18] 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. 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 [14], [29], [19]. 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. [12] proposed DAPASA, an approach to detect Android piggybacked applications through sensitive subgraph analysis. Xu et al. [35] leveraged the inter-component communication patterns to detect Android malware. Yang et al. [36] 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 [21]. Du et al. adopted community structures of weighted function call graphs to detect Android malware. Zhang et al. [40] proposed a semantic-based approach to classify Android malware via dependency graphs. Gascon et al. [15] developed a method based on efficient embeddings of function call graphs with an explicit feature map. Furthermore, Yang et al. [37] 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 [7], [17], [8], [9], [38] 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 [4], [3], [34]. 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 0% 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 0%. 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 and effectiveness of an adversarial training and an ensemble learning defending method. In the next stage, we plan to continue the in depth analysis of various defence mechanisms. 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] Z. Aung and W. Zaw. Permission-based android malware detection. *International Journal of Scientific & Technology Research*, 2(3):228–234, 2013.

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

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

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

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

[20] 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.

[21] 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.

[22] 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.

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

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

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

[35] 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.

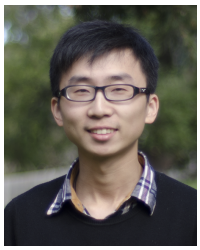
[36] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *European symposium on research in computer security*, pages 163–182. Springer, 2014.

[37] 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.

[38] 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.

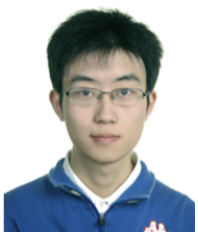
[39] X. Yuan, P. He, Q. Zhu, R. R. Bhat, and X. Li. Adversarial examples: Attacks and defenses for deep learning. *CoRR*, abs/1712.07107, 2017.

[40] 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.



Xiao Chen received his B.Eng. degree from Hangzhou Dianzi University, China (2010); M.IT degree from University of Melbourne, Australia (2012); M.Sc. by research degree from Deakin University, Australia (2014); and Ph.D. degree from Swinburne University of Technology, Australia (2019). His research interests include mobile security, network traffic analysis, and adversarial machine learning.

Chaoran Li received the Bachelor of Information Technology degree from Deakin University Australia in 2018. He is currently working towards the Ph.D. degree at Swinburne University of Technology. His research interests include machine learning, especially in adversarial deep learning.

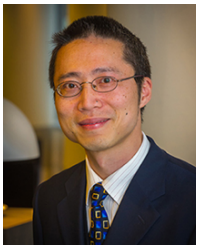


Derui (Derek) Wang received his bachelor's degree of Engineering from Huazhong University of Science and Technology (HUST), China (2011); M.Sc. by research degree from Deakin University, Australia (2016). Currently, he is a PhD student with Swinburne University of Technology and CSIRO's Data61, Australia. His research interests include adversarial machine learning, deep neural network, applied machine learning, decision making system, and complex network.

Sheng Wen received his Ph.D. degree from Deakin University, Australia, in October 2014. Currently he is a senior lecturer in Swinburne University of Technology. He has received over 3 million Australia Dollars funding from both academia and industries since 2014. He is also leading a medium-size research team in cybersecurity area. He has published more than 50 high-quality papers in the last six years in the fields of information security, epidemic modelling and source identification. His representative research outcomes have been mainly published on top journals, such as IEEE Transactions on Computers (TC), IEEE Transactions on Parallel and Distributed Systems (TPDS), IEEE Transactions on Dependable and Secure Computing (TDSC), IEEE Transactions on Information Forensics and Security (TIFS), and IEEE Communication Survey and Tutorials (CST). His research interests include social network analysis and system security.

Jun Zhang received the Ph.D. degree in computer science from the University of Wollongong, Wollongong, Australia, in 2011. He is an Associate Professor with the School of Software and Electrical Engineering and the Deputy Director of Swinburne Cybersecurity Lab, Swinburne University of Technology, Australia. He has published over 90 research papers in refereed international journals and conferences, such as IEEE Communication Survey and Tutorials, IEEE/ACM Transactions on Networking, IEEE Transactions on Image Processing, IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Information Forensics and Security, ACM Conference on Computer and Communications Security, and ACM Asia Conference on Computer and Communications Security. His research interests include cybersecurity and applied machine learning. He has been internationally recognized as an Active Researcher in cybersecurity, evidenced by his chairing (PC Chair, Workshop Chair, or Publicity Chair) of eight international conferences since 2013, and presenting of invited keynote addresses in two conferences and an invited lecture in IEEE SMC Victorian Chapter.

Surya Nepal is a Senior Principal Research Scientist at CSIRO Data61. He currently leads the distributed systems security group. His main research focus is on the development and implementation of technologies in the area of distributed systems (including cloud, IoT and edge computing) and social networks, with a specific focus on security, privacy, and trust. He has more than 200 peer-reviewed publications to his credit. He has co-edited three books including security, privacy, and trust in cloud systems by Springer, and co-invented 3 patents. He is a member of the editorial boards of IEEE Transactions on Service Computing, ACM Transactions on Internet Technology and Frontiers of Big Data- Security Privacy, and Trust. He is currently a theme leader of Cybersecurity Cooperative Research Centre (CRC), a national initiative in Australia. He holds a conjoint faculty position at UNSW and an honorary professor position at Macquarie University.



Yang Xiang received his Ph.D. in Computer Science from Deakin University, Australia. He is currently a full professor and the Dean of Digital Research & Innovation Capability Platform, Swinburne University of Technology, Australia. His research interests include cyber security, which covers network and system security, data analytics, distributed systems, and networking. In particular, he is currently leading his team developing active defense systems against large-scale distributed network attacks. He is the Chief Investigator of several projects in network and system security, funded by the Australian Research Council (ARC). He has published more than 200 research papers in many international journals and conferences. He served as the Associate Editor of IEEE Transactions on Dependable and Secure Computing, IEEE Internet of Things Journal, IEEE Transactions on Computers, IEEE Transactions on Parallel and Distributed Systems, and the Editor of Journal of Network and Computer Applications. He is the Coordinator, Asia for IEEE Computer Society Technical Committee on Distributed Processing (TCDP). He is a Senior Member of the IEEE.

Kui Ren received the Ph.D. degree from the Worcester Polytechnic Institute. He is SUNY Empire Innovation Professor of Computer Science and Engineering and the director of the Ubiquitous Security and Privacy Research Laboratory (UbiSeC) in the Department of Computer Science and Engineering, University at Buffalo, State University of New York. His current research interests include Data and Computation Outsourcing Security in the context of Cloud Computing, Wireless Systems Security in the context of Internet of Things, and Crowdsourcing-based Large-scale Data Acquisition. He has published frequently in peer-reviewed journal and conference papers. His research has been supported by multiple federal research grants including eight awards from US National Science Foundation and two grants from Department of Energy. He has been serving as the TPC co-chairs for many IEEE and ACM conferences. He serves/has served as associate editors for IEEE Transactions on Service Computing, IEEE Transactions on Dependable and Secure Computing, IEEE Transactions on Mobile Computing, IEEE Transactions on Information Forensics and Security, IEEE Wireless Communications, IEEE Internet of Things Journal, IEEE Transactions on Smart Grid, Oxford The Computer Journal, and Elsevier Pervasive and Mobile Computing. He is an IEEE Fellow and an ACM Distinguished Member.