COMPUTERS & SECURITY 73 (2018) 326–344

Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

# Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach

*Sen Chen* [a,b], *Minhui Xue* [c,d], *Lingling Fan* [a,b], *Shuang Hao* [e], *Lihua Xu* [a,*], *Haojin Zhu* [d], *Bo Li* [f]

[a] *East China Normal University, Shanghai, China*
[b] *Nanyang Technological University, Singapore*
[c] *New York University Shanghai, Shanghai, China*
[d] *Shanghai Jiao Tong University, Shanghai, China*
[e] *University of Texas at Dallas, USA*
[f] *University of California, Berkeley, USA*

ARTICLE INFO

*Article history:*
Received 17 April 2017
Received in revised form 31 October 2017
Accepted 12 November 2017
Available online

*Keywords:*
Malware detection
Adversarial machine learning
Poisoning attacks
Manipulation
KuafuDet

ABSTRACT

The evolution of mobile malware poses a serious threat to smartphone security. Today, sophisticated attackers can adapt by maximally sabotaging machine-learning classifiers via polluting training data, rendering most recent machine learning-based malware detection tools (such as Drebin, DroidAPIMiner, and MaMaDroid) ineffective. In this paper, we explore the feasibility of constructing crafted malware samples; examine how machine-learning classifiers can be misled under three different threat models; then conclude that injecting carefully crafted data into training data can significantly reduce detection accuracy. To tackle the problem, we propose KuafuDet, a two-phase learning enhancing approach that learns mobile malware by adversarial detection. KuafuDet includes an offline training phase that selects and extracts features from the training set, and an online detection phase that utilizes the classifier trained by the first phase. To further address the adversarial environment, these two phases are intertwined through a self-adaptive learning scheme, wherein an automated camouflage detector is introduced to filter the suspicious false negatives and feed them back into the training phase. We finally show that KuafuDet can significantly reduce false negatives and boost the detection accuracy by at least 15%. Experiments on more than 250,000 mobile applications demonstrate that KuafuDet is scalable and can be highly effective as a standalone system.

© 2017 Elsevier Ltd. All rights reserved.

We would like to thank Pwnzen Infotech Inc. for providing us with a copy of mobile malware to conduct the study, especially the Pwnzen Infotech Inc. co-founder Zhushou Tang for exchanging helpful industry experience. This work was supported in part by the National Natural Science Foundation of China, under Grants 61502170, 61272444, 61411146001, U1401253, and U1405251, in part by the Science and Technology Commission of Shanghai Municipality under Grant 13ZR1413000.

* *Corresponding author.*
*E-mail address:* lhxu@cs.ecnu.edu.cn (L. Xu).

https://doi.org/10.1016/j.cose.2017.11.007
0167-4048/© 2017 Elsevier Ltd. All rights reserved.

## 1. Introduction

Since last decade, the software development has been witnessed to have a massive shift toward mobile applications. With the growth of mobile applications and their users, the security and privacy concerns are increasingly becoming the focus of great concern to various stakeholders. For instance, more and more users store personal data in their mobile devices, even carrying out financial transactions such as online banking and shopping from their smartphones. Some of these data can be very sensitive. Consequently, hackers can have substantial financial gain from such sensitive data and thus find mobile devices to be lucrative targets.

It is not surprising that the demand for tools of automatically analyzing and detecting malicious applications has also grown. Most of the researchers' and practitioners' efforts in this area target the Android platform, the largest share of the mobile market. There has been a plethora of research in malware detection for Android. Static and dynamic analyses are two generic techniques primarily implemented by two approaches: signature-based (Schlegel et al., 2011; Zhou et al., 2013; Zhou and Jiang, 2012; Zhou et al., 2012) and behavior-based (Graziano et al., 2015; Rasthofer et al., 2016; Tam et al., 2015; Wu et al., 2014; Yan and Yin, 2012). Information flow analysis-based approach (Arzt et al., 2014; Enck et al., 2014; Gordon et al., 2015; Li et al., 2015; Wong and Lie, 2016) is also proposed to detect Android malware. We note that machine learning is one of the most promising techniques in detecting mobile malware (Aafer et al., 2013; Arp et al., 2014; Avdiienko et al., 2015; Chen et al., 2016a; Dash et al., 2016; Fan et al., 2016; Feizollah et al., 2017; Idrees et al., 2017; Meng et al., 2016; Rasthofer et al., 2014; Yang et al., 2014, 2015; Zhang et al., 2014). However, machine learning approaches also have a weakness: they are susceptible to adversarial countermeasures by attackers aware of their use. First, through reverse-engineering, attackers may become aware of classifiers and their parameters used to evade detection. Second, more sophisticated attackers can actively tamper with the classifiers by injecting well-crafted data into training data. Therefore, with Android's policy of open-source kernel, malware writers can gain an in-depth understanding of the mobile platform, hence intentionally alter the training set to reduce or eliminate its detection efficacy.

To our knowledge, most up-to-date works using machine learning mainly focused on detection accuracy and assumed that feature extraction is considered in an ideal environment (Chen et al., 2016c; Mariconti et al., 2017). No evasion techniques were conducted in the feature space when using machine learning-based detection approach. In this paper, we consider a threat model within a specific class of attacks, named *poisoning attack*, in which the attacker is assumed to control a subset of samples or inject additional seeds at will in order to mislead the learning algorithm. For example, in malware detection, a sophisticated attacker may have a good command of the whole training set and deliberately inject poisoning patterns to compromise the performance of the classifiers, which become more prone to misclassify malicious applications as benign ones. *This threat model conceptually underlies adversarial machine learning: it involves gradients of the function f represented by the learned model (e.g., SVM, logistic regression, K-Nearest Neighbor) in order to evaluate it on an input x. Attackers can then fully automatically either identify individual input features that are perturbed to achieve misclassification* (Papernot et al., 2016).

To test the ramifications of these causative attacks, we develop an adversarial model with three types of attackers according to different aggressiveness of attacks to simulate the real-world attacks. To do this, we adopt a customized adversarial crafting algorithm, characterized by the aggressiveness of attacks, to generate the crafted camouflage samples. The abstraction of crafting steps is somehow restricted in three ways. (i) To preserve the functionality of the modified application, we only add or remove features; (ii) we add a restricted number of features. For simplicity, we therefore decide to add entries to the *AndroidManifest.xml* and *Smali* files; (iii) since obscuring semantic features is much more challenging than confusing syntax features, we only use syntax features to craft samples. In spite of these restrictions in crafting, we achieve a significantly high misclassification rate on malicious applications when using 564 original non-robust features. To further perform a longitudinal comparison, we also apply our poisoning attack to Drebin (Arp et al., 2014), DroidAPIMiner (Aafer et al., 2013), and MaMaDroid (Mariconti et al., 2017), the three most recent machine-learning detection systems in academia. We thus validate that the threat model and the poisoning attack are indeed viable in malware detection. We conjecture that almost all the state-of-the-art machine-learning malware detection systems are suffering from the poisoning attack we exhibited in the paper.

To handle these adversarial attacks, we propose KuafuDet, a learning enhancing defense system with adversarial detection that includes an offline training phase that selects and extracts contributing features from the training set for preprocessing, and an online detection phase that utilizes the classifier trained by the first phase. Comparing to existing work, these two phases act together, through a self-adaptive learning scheme, as an iterative adversarial detection process. Additionally, we introduce the *camouflage detection* for verifying false alarms to protect against poisoning attacks. By using similarity analysis, the camouflage detection is applied to iteratively detect against malicious data distortion. In concrete, we train 16,000 Android application samples that are equally distributed, which are downloaded from Contagio Mobile Website,[1] Pwnzen Infotech Inc. and Drebin (Arp et al., 2014). All 195 robust features are extracted using static analysis on the given application, pruned by information gain. We further evaluate the results on 4000 applications. Our best detection classifier achieves up to 96% accuracy without adversarial environment, and by at least 15% accuracy when coping with the most powerful attackers, along with both low false negatives. Furthermore, we conduct an empirical evaluation on our test set and select 1000 malware as samples out of the set of 10,400 malicious samples and scan them using KuafuDet and other industrial malware detecting tools, such as Kaspersky and McAfee encapsulated in VirusTotal.[2] The coverage of KuafuDet significantly outperforms these top-of-the-line antivirus systems. Finally, we perform the entire process of KuafuDet,

[1] http://contagiominidump.blogspot.hk/.
[2] https://www.virustotal.com/.

using real-time streaming, on a server with 16 GB memory, quad-core i7-4800MQ at 3.6 GHz, and 1 TB hard drives and show that KuafuDet is scalable and efficient.

In this paper, we make the following key contributions that are fourfold.

1. We propose that poisoning attacks can be exhibited by three types of attackers in the real world, ranging from weak, strong, and sophisticated degrees. We hold evidence that the real-world mobile malware dataset is able to truly reflect three types of attackers we defined.
2. We adopt a customized adversarial crafting algorithm, semantically characterized by the aggressiveness of attacks, to generate the crafted camouflage samples using syntax features to largely simulate the real-world attacks.
3. We show that our poisoning attack is able to mislead DroidAPIMiner (Aafer et al., 2013), Drebin (Arp et al., 2014), and MaMaDroid (Mariconti et al., 2017), the three most recent machine-learning detection systems in academia.
4. We propose a two-phase iterative adversarial-based detection, termed KuafuDet, wherein similarity-based filtering is used to identify the false negatives that are the camouflaged malicious applications, further reinforcing the resilience of the malware detection system.

Our experiments show that attackers can also poison features while preserving maliciousness, and our experiments verify that the resulting fake variants with poisoned features impaired discriminative classifiers and succeed in lowering the detection score in a test environment. Other main findings are as follows:

- We observe that different feature categories have different impacts on crafted camouflage samples. The effect rate of API call leads to greater perturbation than permission.
- We emulate the feature extraction for all types of features that Drebin used and find that Drebin-used feature extraction is substantially more computationally complex and does not necessarily boost the accuracy.
- We find that in the data-imbalanced (benign-malicious ratio) environment, the accuracy of KuafuDet gradually degrades as we put in more benign applications, but the accuracy still remains relatively high.
- We find that similarity-based filtering analysis and relabeling have an excellent performance to handle against adversarial attacks.

To the best of our knowledge, this is the first paper to accommodate a newly designed two-phase adversarial machine learning mechanism into mobile malware detection to limit the possibility of mimicry and poisoning attacks, and further propose a learning enhancing system through adversarial detection of Android malware.

The rest of the paper is organized as follows. Section 2 defines the research problem. Section 3 presents the motivations and challenges. Section 4 provides the system overview followed by the implementation shown in Section 5. Section 6 summarizes experimental evaluation. Section 7 discusses limitations. Section 8 surveys related work. Finally, Section 9 concludes the paper.

### 1.1. *A note on ethics*

In this paper, we are very aware of the potential impact on malicious apps disclosure or exploited by other malicious third parties. In particular, in order to illustrate this methodology, the collection of mobile malware used and crafted was strictly followed by the Privacy Policy of the Pwnzen Infotech Inc., and conformed to the non-disclosure agreement (NDA) of the Pwnzen Infotech Inc. Furthermore, to facilitate research on mobile malware detection, we make the malicious Android applications (except those from Pwnzen Infotech Inc.) used in the paper publicly available to other researchers under http://nsec.sjtu.edu.cn/kuafuDet/kuafuDet.html, but no attempt was made to provide data from Pwnzen Infotech Inc. for people outside of our research group because of intellectual property. Only Pwnzen Infotech Inc. authorized employees, using internal computer systems from Pwnzen Infotech Inc., can have access to the dataset. Finally, we informed the team of the Pwnzen Infotech Inc. of the potential newly discovered malicious apps in order to help Pwnzen Infotech Inc. improve the quality of its products and services. We believe this study performs an important public service, as it shows that even state-of-the-art antivirus systems are somehow futile. Our ultimate goal is to inform developers and users of such potential poisoning attack, so that more comprehensive countermeasures can be taken in the future.

## 2. Problem definition: Adversarial machine learning

We denote a sample set by $\{(x_i, y_i) \in (\mathcal{X}, \mathcal{Y})\}_{i=1}^{n}$, where $x_i$ is the $i$th malware sample vector of which each component exhibits the selected feature; if $x_i$ has the $j$th component, then $x_{ij} = 1$; otherwise $x_{ij} = 0$. $y_i \in \{0, 1\}$, $n$ is the total number of malware samples, and $\mathcal{X} \subseteq \{0,1\}^m$ is an $m$-dimensional feature space. In this paper, we consider binary classifiers with only two output classes where the attacker crafts malware dataset to evade detection and hence achieves his goals. The attacker tries to move away malware dataset $y_i = 1$ in any direction by adding a non-zero displacement feature vector $\delta_i$ to $x_i|_{y_i=1}$. For example, attackers may add good attributes to mobile malware to evade binary classifiers. We note that attackers will not be able to modify legitimate benign applications since an honest author has no interest in having his benign application classified as malware. Hence, crafting an adversarial sample $x^*$, misclassified by the function $f$ (where $f : x \to y = f(x)$), from a benign sample $x$ can be formalized as the following problem (Szegedy et al., 2014):

$$x^* = x + \delta_x \quad \text{s.t.} \quad f(x + \delta_x) \neq f(x), \tag{1}$$

where $\delta_x = x$ is the minimal perturbation yielding misclassification and $f$ can be the corresponding softmax function.

The goal of adversarial sample crafting in malware detection is to mislead the detection system, causing the classification for a particular application to change according to the attackers wishes. In this paper, we only focus on the

*poisoning attack* that results in malicious behavior being misclassified as benign (false negatives), because Inter-Component Communication (ICC) analysis is used to reduce false positives (Li et al., 2015; Octeau et al., 2016). We also assume the attacker has full access to the classifier used, and can inject as many variants' features as possible at will to the given classifier. For this reason, following Eq. (1), we further denote $x_{ij}^{max}(=1)$ and $x_{ij}^{min}(=0)$ as the maximum and the minimum values that the $j$th feature of the $i$th sample can take. Then a poisoning attack can be characterized in the following:

$$\begin{aligned} C_f\left(x_{ij}^{min}-x_{ij}\right) \le \delta_{ij} \le C_f\left(x_{ij}^{max}-x_{ij}\right), \quad \forall j \in [1,m], \\ = C_f\left(0-x_{ij}\right) \le \delta_{ij} \le C_f\left(1-x_{ij}\right), \quad \forall j \in [1,m], \end{aligned} \tag{2}$$

where $C_f \in [0, 1]$ controls the aggressiveness of attacks. $C_f = 0$ indicates no attacks, while $C_f = 1$ indicates the most aggressive attacks. To test the ramifications of causative attacks and clearly elaborate the challenges, we develop an adversarial model with three types of attackers with the corresponding $C_f$ values.

**Weak attacker ($C_f = 0.33$).** Our weak attacker is not aware of the statistical properties of the training features or labels at all. This attacker simply fakes additional labels with random binary features to poison the training dataset.

**Strong attacker ($C_f = 0.67$).** Our strong attacker is aware of the features we use for training and can have access to our ground-truth dataset (which comes from public sources). This attacker can manipulate partial features in the training data. However, this attacker is resource constrained and cannot manipulate any mobile application statistics which would require more time. The strong attacker crafts features by randomly selecting publicly available Android malware and then faking additional labels, so that the partial training labels can become nearly identical.

**Sophisticated attacker ($C_f = 1$).** Our strongest attacker, named sophisticated attacker, has full knowledge of our training feature set. Additionally, this attacker has sufficient time and economic resources to create arbitrary mobile application statistics. Therefore, the sophisticated attacker can fully manipulate almost all training features, which creates scenarios where relatively benign mobile applications and real-world malicious mobile applications appear to have nearly identical attributes at the training phase.

To do this, we adopt the adversarial crafting algorithm (Papernot et al., 2016) based on the Jacobian matrix

$$J_f = \frac{\partial f(\mathcal{X})}{\partial \mathcal{X}} = \left[\frac{\partial f_i(x)}{\partial x_j}\right]_{i\in\{0,1\}, j\in[1,m]}$$

where $f_0(x)$ outputs $x$ is benign and $f_1(x)$ outputs $x$ is malicious, with $f_0(x) + f_1(x) = 1$. To craft an adversarial sample, we use 73 benign features and 102 malicious features detailed in Section 5.1. We then take mainly two steps. In the first step, we compute the gradient of $f$ with respect to $x$ to estimate the direction in which a perturbation in $x$ would change $f$'s output. In the second step, we choose a perturbation $\delta$ of $x$ with maximal positive gradient into our target class $\mathcal{Y}_{y_i=0|y_i=1}$ denoted $y'$, and we customize the adversarial crafting algorithm (Papernot et al., 2016) according to our adversarial model with three types of attackers ($C_f$), to indicate the probability (Eq. (2)) of adding a specific feature. After computing the gradient, we iteratively choose a target feature of which the gradient is the largest for our target class and then update its value in $x$ to obtain our new input vector. We then re-update the gradient and repeat this process until either (i) we reach the bounded allowed changes (loop bound) or (ii) we successfully achieve a misclassification.

We didn't attempt to formalize the malware detection as an optimization problem, saying to maximize accuracy at the lowest resource cost with minimal adversarial perturbations, because simply using optimization may not be aligned with how we semantically understand the human malicious behaviors and motivations. For future research, we wish to generate an exact mapping rule between machine-crafted mobile malware and read-to-use malicious apps in the wild, and ultimately wish to provide a foundation for developing sustainably-secure anti-malware systems in the face of dynamic cyber-maneuvers.

## 3. Motivations and challenges

In this section, to highlight our contributions, we motivate our malware detection model by incorporating adversarial environment witnessed in the real world and then review several challenges of our work.

### 3.1. *Evolutionary chain*

Fig. 1 describes an evolutionary chain of mobile malware advancements that has been observed along the timeline: ranging from the seed explosion to the recent adversarial attack. Earlier menaces of malware are mainly pertinent to compromised SMS related functions. With every new technology comes abuse, the Android market is no exception. Since 2014, malware samples can be easily exploited by Android vulnerabilities, which heavily drives an arms race for the adversarial detection of mobile malware. As the nature of the attack is shifting from small-

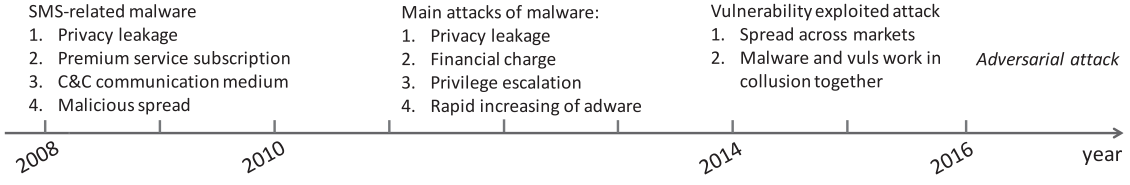


**Fig. 1 – A timeline of Android malware.**

```
1  <uses-permission android:name="android.permission.MANAGE_ACCOUNTS"/>
2  <uses-permission android:name="android.permission.AUTHENTICATE_ACCOUNTS"/>
3  <uses-permission android:name="android.permission.VIBRATE"/>
4  <uses-permission android:name="android.permission.CAMERA"/>
5  <uses-permission android:name="android.hardware.camera"/>
6  <uses-permission android:name="android.hardware.camera.autofocus"/>
7  <uses-permission android:name="android.permission.GET_TASKS"/>
```

**Fig. 2 – A case of permissions overprivileged.**

scale and low-tech toward large-scale and skilled ones, the additional efforts will have to be directed at taking targeted strategies to detect stubborn malware.

### 3.2. *Adversarial samples*

In previous work (Chen et al., 2016a), a number of malware samples in the dataset are misclassified into benign ones. We zoom in these misclassified samples and witnessed several real-world cases that reflect adversarial attacks from this dataset. Each observation corresponds to the three types of attackers defined above.

**Weak attacker.** Embedding a good portion of benign code into a malicious app (*e.g.*, manifest attributes and non-logical code in java code). Felt et al. (2011) show that more than half of Android applications are overprivileged, such as misusing *AndroidManifest.xml* configuration. We carefully analyze the false negative samples using mapping relations between permissions and API calls, and find that some declared permissions remain unused at the development stage. We can see from Fig. 2 that part of the code in the *AndroidManifest* file of *Polaris Office* is misclassified as benign. Although #1 to #7 permissions are declared, they have not been used at all since the app does not demonstrate corresponding behaviors. These permissions extracted as features for training classifiers weakly mislead the classification outcomes. As shown in Fig. 3, we delete redundant code (#1–#7) that is irrespective of the logical behavior of the sample. After we repeat the classification process, surprisingly the sample is classified as malicious.

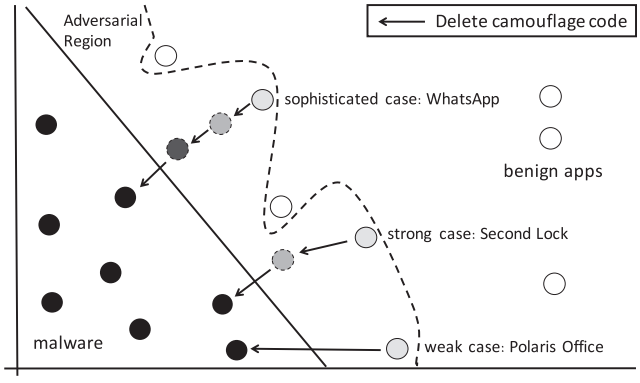


**Fig. 3 – The process from benign to malicious: the black dots refer to malicious apps, the white dots refer to benign apps, and the gray dots are originally malware but misclassified as benign due to the camouflage code injected in the adversarial region. The gray dots turn darker with the deletion of camouflage code and finally turn all black, exposing its malicious nature. The arrows direct toward the transformation process.**

**Strong attacker.** Hiding a good portion of malicious code into other formats in the application package. Some alternative malicious applications use techniques such as dynamic loading techniques to hide a good portion of malicious code into other formats. Usually these malicious code blocks include sensitive API calls. API calls used by most of machine learning classifiers will dramatically lead to the misclassification. For example, *SecondLock* behaves as same as "egdata" does. Some malicious code blocks or executable files are hidden in the files of other formats, such as `jar`, so, `jpg`, and `data`. These files contain sensitive API calls, such as `DownloadManager.enqueue` and `DownloadManager.query`, which are hidden by malware. The "assets" folder of *SecondLock* contains a png format file, which is not a standard `png` file and can be dynamically loaded. It can prevent the application from updating or automatically downloading other malicious applications. As shown in Fig. 3, we show two steps to correct the classification results. In step one, we remove five unused permissions, the result of classification moves toward the hyperplane, though the final result remains benign. In step two, we add some sensitive API calls that are hidden in the `png` file (*e.g.*, `DownloadManager.enqueue` and `DownloadManager.query`). This particular sample is finally classified as malicious.

**Sophisticated attacker.** Embedding benign logic code in java code and dynamic code loading with reflection. A few malicious applications add benign logic code in source code. Benign logic code can be executed without any effects on malicious behaviors, which is used to obscure the feature extraction process to clone benign applications. This is similar to "testing code." By embedding benign logic code, the sophisticated attacker can add any code blocks or any combination of various techniques to mislead the machine-learning classifiers, making the classifiers less robust. Specifically, after the construction of Activity transition relations for *WhatsApp*, a repackaged malware from the third party rather than the official version, we find there exist embedded activities, standing alone with some methods, such as neither being affected by any other activities nor shown up in the system logic. Fig. 4 shows a code segment of *WhatsApp* with an embedded activity, `getMemoryLimited()` method initiates a system call to `getMemoryClass()`, `getLargeMemoryClass()`, etc. However, these system calls extracted as features for training cannot reflect system logic of the sample, which seriously misleads the classification outcomes. We therefore remove such embedded benign logic code and retrain the classifiers. As shown in Fig. 3, once we delete such code step by step, the malware sample is exposed. Fig. 3 also exemplifies that in the adversarial environment, the attack process is changeable and dynamic.

```
1 public class CalledActivity extends AppCompatActivity {
2   public void execute() {
3     // Call the embedded method
4     getMemoryLimited();
5     // Call the malicious method
6     getMaliciousMethod();
7   }
8   // Embedding benign logic code
9   public void getMemoryLimited() {
10    ActivityManager activityManager =
11    (ActivityManager)context.getSystemService(Context.ACTIVITY_SERVICE);
12    System.out.println(activityManager.getMemoryClass());
13    System.out.println(activityManager.getLargeMemoryClass());
14    System.out.println(activityManager.getRuntime().maxMemory()/(1024*1024);
15  }
16 }
```

**Fig. 4 – A case of Activity embedded with benign API calls.**

The alternative approach that one sophisticated attacker may take is through dynamic code loading. Dynamic code loading usually utilizes reflection mechanism to modify the runtime behavior of applications. It provides ability for sophisticated attackers to add malicious behaviors (malicious features) without having to change the original application, hence mislead the machine learning classifiers. For example, as depicted in Fig. 5, sophisticated attackers can incorporate two malicious methods in the class `MaliciousMethodsInDex` (lines #1–#6) – `sendDeviceInfo` and `sendCredential`, and instantiate them via DexClassLoader at runtime. Since the malicious codes are loaded at runtime, and not part of the application source, it is a challenge to classify them as malware through machine-learning classifiers.

### 3.3. *Challenges*

**Class imbalance.** We aim to train a classifier that produces binary predictions: each mobile application is classified as either benign or malicious. If there are significantly more malicious applications in one class than in the other class, this biases the output of supervised machine-learning algorithms. Prior research treats it simply by using 49 different malware families (Zhou and Jiang, 2012). In consequence, our dataset exhibits a modest class imbalance. We first define 217 Android malware families and then classify them into eight categories, such as Expense, Fraud, Payment, Privacy, Remote, Rogue, Spread, and System (see Table 1), to present a systematic characterization of existing Android malware. The eight malware categories represent almost all coverage of existing Android malicious behaviors. We observe that it is common that many malware families belong to multiple categories in parallel. This phenomenon indicates that malicious family is not limited to a single malicious behavior. As shown in Table 1, Privacy and Fraud occupy the highest proportions among all. Therefore, we are able to find the reasonable distribution of our real-world malware during our data acquisition. As the ratio of malware to benign apps in the real world is highly imbalanced, this class imbalance usually represents a significant challenge for reducing false negatives.

**Quality of ground-truth dataset.** Prior work on malware dataset focused on validating their approaches by using a small out-dated dataset. These predictors can be used as ground truth for training high-performance classifiers. In contrast, there is no comprehensive dataset of malware that is available in the real world. We employ as ground truth the set of malware from five different platforms. In particular, the set obtained from Pwnzen Infotech Inc. is the most recent malware in the real world. However, we acknowledge that this dataset does not cover all platforms uniformly.

## 4. System overview

In this section, we provide a high-level overview for our system design.

```
1 class MaliciousMethodsInDex {
2   // send device info by Socket connection
3   void sendDeviceInfo (String data) { … }
4   // send crendential by Http connection
5   void sendCredential (String data) { … }
6  }
7
8 DexClassLoader loader = new DexClassLoader( … );
9 Class class = loader.loadClass("MaliciousMethodsInDex");
10 Object object = class.newInstance();
11 Method method = class.getDeclaredMethod("sendCredential");
12 method.invoke();
```

**Fig. 5 – A loading example of using reflection (demonstrated as dynamic pseudo code).**

**Table 1 – Malware category.**

| Malware category | Percentage |
|---|---|
| Privacy | 47.1% |
| Fraud | 33.9% |
| Rogue | 20.1% |
| Spread | 20.1% |
| System | 11.1% |
| Remote | 10.6% |
| Expense | 10.1% |
| Payment | 6.3% |

Note: We take a look at malware category with miscellaneous datasets. We find that many Android malware families belong to multiple malware categories, thereby the sum of percentages is greater than 100%.

### 4.1. *Key ideas*

In putting our approach in practice for massive-scale detection, we aim to achieve two important goals: one is to identify and filter the suspicious false negatives (*i.e.*, the malicious applications that are camouflaged), the other is to achieve accuracy and scalability at the same time. To achieve these two goals, we propose two key techniques: similarity-based filtering and two-phase iterative adversarial detection, as shown below.

(i) **An automated similarity-based approach to filter suspicious false negatives.** In general, attackers have two characteristics: first, attackers may acquire dual intrinsic attributes of applications, thus we assume the suspicious ones are the ones with both strong reflection on malicious features and benign features; second, in our threat model, attackers may have certain level of knowledge of training set, thus we decouple the similarity metrics from machine-learning classifiers. Based on these two characteristics, if the trained classifier could incorporate the similarities across the applications in the training set to lead to a further fine-grained detection, the learning system would be periodically enhanced with these newly discovered malware and suspicious false negatives.

(ii) **A two-phase iterative adversarial detection approach to achieve accuracy and scalability.** False negatives can be reduced based on our understanding of an attacker's threat level. Thus self-adaptive learning (SAL) scheme, where the suspicious false negatives, as well as the identified malicious applications, are fed back to the training process for a desired trade-off between accuracy and scalability.

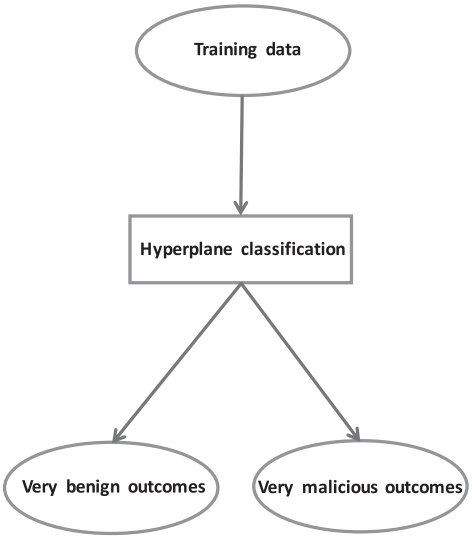


**Fig. 7 – Choosing very benign and malicious outcomes.**

### 4.2. *Overall architecture of KuafuDet*

The overall architecture of KuafuDet is shown in Fig. 6, which is comprised of two intertwined phases. In the **Training Model** phase, KuafuDet extracts features from labeled applications based on our combined set of contributing features, trains classification model offline, and updates classifiers at a certain interval of time; In the **Online Detection** phase, KuafuDet classifies large sets of online Android applications (from multiple online Android markets) into different categories, benign and malicious; Meanwhile, KuafuDet, through a **Self-adaptive Learning** scheme, discovers new information from both the identified malware and the filtered suspicious false negatives from **Camouflage Detector**, and incorporates them into **Training** to stabilize the detection accuracy (Fig. 7).

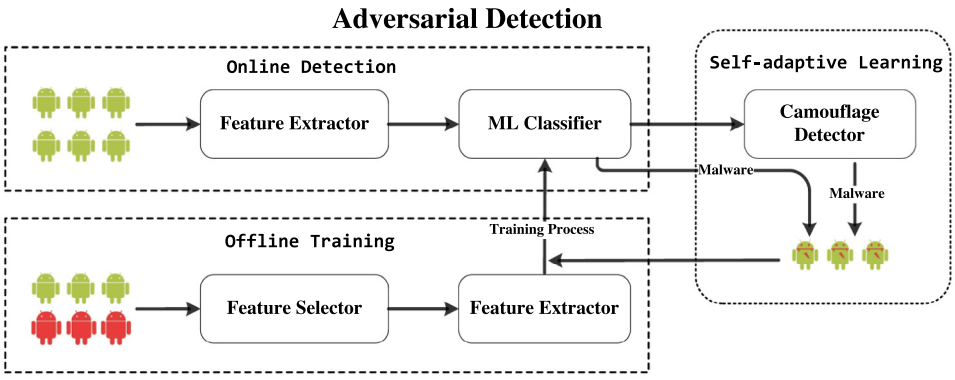


**Fig. 6 – The KuafuDet framework through adversarial detection.**

**Table 2 – Features.**

| Syntax features | | | Semantic features |
|---|---|---|---|
| Permission | Intent and hardware | API Call | Sequence |
| READ_PHONE_STATE | INTENT.ACTION.DELETE | URL.openConnection | (chmod 777, Runtime, getRuntime, exec) |
| WRITE_SMS | INTENT.ACTION.GET_CONTENT | TelephonyManager.getDeviceId | (getDeviceId, URL, openConnection) |
| INSTALL_PACKAGES | HARDWARE.TOUCHSCREEN | PackageManager.checkPermission | (DownloadManager, Uri, Request, enqueue) |
| … … | … … | … … | … … |

## 5. System design

In retrospect, the quality of discriminative classifier is the key to the accuracy of malware detection. On one hand, when the trained classifier is trained once and used for all time, it is not able to correspond to the new malware. On the other hand, aggressive attackers may obfuscate their representations in terms of contributing features to impair discriminative classifiers. Thus it might lead to high false negatives that the malicious applications evade detection. In order to perform accurate and scalable adversarial detection, our proposed adversarial detection approach contains two phases, training and detection, intertwined by the self-adaptive learning (SAL) scheme. In particular, we conduct our similarity-based analysis in Camouflage Detector to filter the suspicious false negatives.

The implementation of KuafuDet involves the following steps:

1. In the feature selection stage, we decompile APKs to generate *Smali* code via Apktool,[3] we extract 195 out of 564 features using manual pruning along with information gain validated.
2. In the training stage, we use different machine-learning classifiers, such as Support Vector Machine (SVM), Random Forest (RF), and *K*-Nearest Neighbor (KNN), based on 195 dimensional features we selected.
3. In the camouflage detection stage, we perform similarity-based filtering to identify the false negatives that are the camouflaged malicious applications.

### 5.1. *Feature selector*

The features considered in this study are classified into two categories: syntax features ($\mathcal{S}^{\{PERM,INT,H,API\}}$) and semantic features ($\mathcal{S}'^{\{Sequence\}}$).

#### 5.1.1. Syntax features

Through closely examining more than 250,000 applications from various sources (breakdowns shown in Section 6.1), we notice that the malicious applications tend to have drastically different permissions, intents, hardwares and API calls, which support the assumption that malicious applications are distinguishable from benign ones. To facilitate reading, we show a coarse-grained description of syntax features used in this paper.

- Permission ($\mathcal{S}^{\{PERM\}}$): Each APK has an AndroidManifest file in its root directory, which is an essential profile including information about the application. Android OS must process this profile before it runs any installation. The profile file declares which permissions the application must have in order to access protected parts of the API and interact with other applications. It also declares the permissions that others are required to have in order to interact with the application's components.
- Intent ($\mathcal{S}^{\{INT\}}$): Communication between different components is mainly through intent, which can be regarded as the "medium" where information about massive asynchronous data exchange and calls to different components is shared between different components and applications.
- Hardware ($\mathcal{S}^{\{H\}}$): Features about requesting access to specific hardware of the smartphone should be declared in the manifest file, such as NFC and GPS, since the combination of such hardware modules may have harmful impact on the phone.
- API Call ($\mathcal{S}^{\{API\}}$): Android API calls monitoring, based on the reverse engineering, can monitor those API calls, such as sending SMS, accessing user location, and device ID. The Android platform provides a framework API that applications can be used to interact with the underlying Android OS. The framework API consists of a core set of packages and classes. Most applications use a fairly large number of API calls.

Here, we use statistical metrics-driven manual pruning (Chen et al., 2016a) with information gain to cross-check the feature selection. Although information gain facilitates the automatic feature selection, it ignores the class information and distribution of the features. When these features are used to detect malware, the performance would drop down dramatically. For example, `READ_INPUT_STATE` (resp. `ACTION.SET_WALLPAPER`) corresponds to Permission (resp. Intent) exhibits a high information gain than many others but it relates to only a small subset of malware. Such highly specific features are undesirable for classification. In summary, we generate 175 types of syntax features.

$$\sum \overbrace{\underbrace{\#\bigcap \mathcal{S}^{\{PERM\}}}_{61} + \underbrace{\#\bigcap \mathcal{S}^{\{INT\}}}_{12} + \underbrace{\#\bigcap \mathcal{S}^{\{H\}}}_{5} + \underbrace{\#\bigcap \mathcal{S}^{\{API\}}}_{97}}^{\mathcal{S}^{\{PERM,INT,H,API\}}}.$$

#### 5.1.2. Semantic features

The semantic feature ($\mathcal{S}'^{\{Sequence\}}$) represents malicious behaviors that occur sequentially, which are extracted via static analysis. For instance, the sequence (`DownloadManager`, `Uri`,

[3] http://ibotpeaches.github.io/Apktool/.

**Table 3 – Comparison of consequential features (using DREBIN dataset).**

| Detection tool within features used | Accuracy |
|---|---|
| 195-dimensional features used in KUAFUDET | 96.55% |
| 564-dimensional features used in KUAFUDET | 95.80% |
| 5000 features used in DREBIN | 94.05% |
| 500,000 features used in DREBIN | 93.90% |

`Request`, `enqueue`) in Table 2 indicates that a download request requires to follow a certain order: construct a request object and transfer the URL of the file to *enqueue* method to finish the download process. These sequences can characterize some interesting malicious behaviors that cannot be captured by the syntax features and can reflect the malicious behaviors more explicitly for a large number of apps, with the purpose of training classifiers. We de facto take several sensitive behaviors into consideration, such as "Send SMS," "Request for chmod," "Uninstall application," "Get location," "Get wifi info," and "Start http connection." We then extract the sequence of key strings that reflect interesting malicious behaviors. For example, requesting for chmod is described as the sequence of "chmod 777," Runtime, getRuntime, and exec, we define 20 types of semantic features for detecting malware. By generating Android malware, these semantic features can be extended.

$$\sum \frac{\#\bigcap \mathcal{S}'^{\{Sequence\}}}{20}.$$

To characterize each of the applications using static analysis, we generate a final set of 195 out of 564 types of features, as partially shown in Table 2. In summary, we use 195-dimensional feature vectors for the study (breakdowns shown in Table A.11).

#### 5.1.3. Justifying feature selection

In DREBIN, the feature set contains thousands of arbitrary strings that appear in the manifest file or in the disassembled code of the app chosen by developers. In particular, DREBIN extracts eight types of features: hardware components, requested permissions, app components, filtered intents, restricted API calls, used permissions, suspicious API calls, network addresses. These massive features chosen from this dataset act as noises, misleading the classifier. We readily emulate the feature extraction for all types of features that DREBIN used since DREBIN authors do provide the feature vectors of their own dataset for evaluation by other researchers. As shown in Table 3, we use DREBIN dataset (Arp et al., 2014), massive DREBIN-used features, and simulate his algorithm. We conclude that extracting DREBIN-used features was a substantially more computationally complex process than our feature selection due to the sheer number of features extracted. In fact, these features do not necessarily boost the accuracy. Our approach also validates that 195 types of features used in KUAFUDET are proper and will not trigger the classical curse of dimensionality.

### 5.2. *Machine learning classifiers*

With these 195-dimensional features that result from our feature selector, we utilize a number of popular algorithms widely used in security contexts, including Support Vector Machine (SVM), Random Forest (RF), and *K*-Nearest Neighbor (KNN). We leverage existing implementations of these algorithms in WEKA (Hall et al., 2009). In particular, SVMs seek to determine the maximum margin hyperplane to separate the classes of malicious and benign applications. When a hyperplane cannot perfectly separate the binary class samples based on the features we fed in, we then tune the parameters such as regularization penalty and non-negative slack variables. We also perform multiple rounds of stratified random sampling due to the data imbalance as stated in Section 3.3. Random Forest (RF) and *K*-Nearest Neighbor (KNN) are also tuned in an analogous manner.

### 5.3. *Camouflage detector*

To further discover camouflage in malware, we manually pick a fair number of applications from the farthest very benign outcomes and very malicious outcomes from the classification hyperplane, respectively. In particular, these three machine-learning algorithms use the corresponding *distance* to classification hyperplane. And those hand-picked applications are the most benign and most malicious predictions and would be updated along with training set updating. We assume these applications have not been poisoned by any malicious third parties. We then measure the similarity between the training set and the selected most benign applications, and vice versa the selected most malicious applications after classification. By further tuning the similarity threshold, we relabel the camouflage malware of the training set as malicious samples to make the classifier robust. Moreover, based on similarity analysis, we are able to identify the camouflage malware from false negatives. Our similarity-based approach is based on extracted robust features and those non-bypassed samples that are farthest from the hyperplane.

#### 5.3.1. Measuring similarity

We use Jaccard index, Jaccard-weight similarity, and Cosine similarity to measure the similarity of applications. The similarity indices are characterized by two vectors *A* and *B*, where *A* represents the feature vectors of the applications in the training set and *B* represents the feature vectors of hand-picked (the most benign or the most malicious) applications. The similarity indices used in this paper are the following:

**Jaccard index**: The Jaccard index, denoted by $J(\cdot)$, is defined as the size of the intersection divided by the size of the union of the sample sets *A* and *B*:

$$J(A,B)=\frac{|A\cap B|}{|A\cup B|}=\frac{|A\cap B|}{|A|+|B|-|A\cap B|}.$$

Jaccard index is not accurate enough because it does not reflect the actual differences of frequencies. For example, an API is used 10 times and 100 times respectively in two applications, but the Jaccard distance simply treats them equally.

**Jaccard-weight similarity**: The Jaccard-weight similarity is defined as follows, which is computed by two steps.

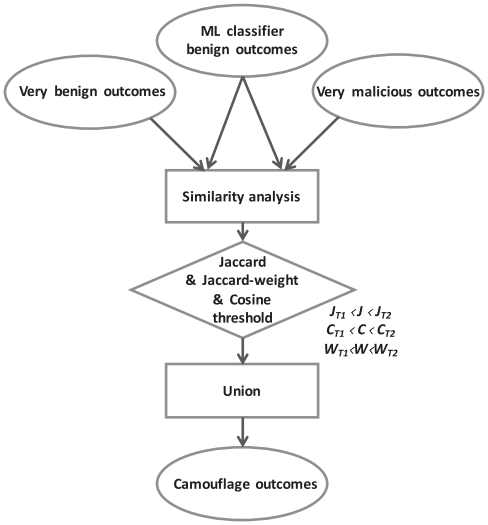


**Fig. 8 – Similarity analysis.**

Step 1. The weight of each component of the feature vector $J_f$ is defined as the percentage that the number of apps which exhibit that feature over the total number of apps.
Step 2. For any two app $a \in A$ and $b \in B$, we consider if both the $k$th component of feature vectors are non-zero, that is $a^k = b^k = 1$, where $a^k$ and $b^k$ denote the $k$th component of feature vectors of apps $a$ and $b$, respectively. We collapse the Jaccard-weight similarity $W(a, b)$ as follows:

$$W(a,b) = \frac{\sum_{k=1}^{n} J_f^k \ (a^k = b^k = 1)}{\sum_{k=1}^{n} J_f^k}$$

where $n = 175$ is the dimension of features and $(\cdot)$ is the indicator function.

**Cosine similarity**: The cosine similarity, denoted by $\cos(\theta)$, is defined using a dot product of the two vectors $A$ and $B$ divided by the product of their magnitudes as:

$$\cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}.$$

We use Jaccard index ($J$), Jaccard-weight similarity ($W$), and Cosine similarity ($C$) to measure the similarity of applications. If the similarity between two applications exceeds a certain threshold, the application will be selected as a malware candidate and fed back to the training process for further fine-grained detection. We want to select as many malware candidates as possible for periodically retraining the classifiers. To be specific, a low threshold likely leads to high false negatives, while a high threshold leads to low false negatives. As shown in Fig. 8, during our experiments, we empirically choose parameters $J_{T_1} < J < J_{T_2}$, $W_{T_1} < W < W_{T_2}$, and $C_{T_1} < C < C_{T_2}$ as corresponding thresholds and then take the union of three outcomes for picking the camouflage malware. From an attacker's perspective, in order to evade the detection, the fraction of two sets $A$ and $B$ must be below a given threshold $0 < p < 1$ for Jaccard index: $|A \cap B| \le p \times |A|$ and $|A \cap B| \le p \times |B|$. An optimal attack strategy is to schedule a group of accounts according to the set of such action sets $A$ or $B$ that has the maximum cardinality so as to minimize the probability that two accounts are caught. But finding $A$ with the maximum cardinality remains an open problem in intersection set theory (Brunk, 2009), which poses a limitation to the attacker.

## 6. Experimental evaluation

We evaluate KuafuDet with applications downloaded from different popular third-party Android markets, as well as in real industrial environments such as Pwnzen Infotech Inc. The goals are to evaluate our system in aspects of: (i) the robustness of our detection under three attacks; (ii) the capabilities of accurately detecting malicious applications; (iii) the efficiency and scalability of real-time analysis, and adaptability to new Android malware; and (iv) the capabilities of detecting coverage.

### 6.1. *Experimental dataset*

As mentioned earlier, most studies lack a large number of data samples. We fulfill the need by presenting the first large collection of 252,900 Android application samples, including 10,400 malicious samples, which covers the majority of existing to recent ones, as shown in Table 4. Specifically, these 252,900 APK files we collected consist of 242,500 benign applications that are downloaded from Google Play Store, and the other 10,400 malicious APK files where 1260 have been validated in Zhou and Jiang (2012) and the remaining are downloaded from Contagio Mobile Website (340 APKs), Pwnzen Infotech Inc. (4500 APKs) and Arp et al. (2014) (4300 APKs). Our malicious applications include all varieties of the threats for Android, such as phishing, trojans, spyware, and root exploits. In the following, we randomly select various portions of benign apps and malicious apps (various ratios of # benign *vs* # malicious) for different experimental goals. Specifically, we select 1000 malware as samples out of the set of 10,400 malicious samples

**Table 4 – Datasets for adversarial detection of Android malware.**

| Source | | Universal | Data-driven Analysis | Training | Test | Comparison |
|---|---|---|---|---|---|---|
| Benign | | **242,500** | 10,000 | 8000 | 2400 | 0 |
| Malicious | Pwnzen Infotech Inc. | 4500 | 4500 | 3500 | 1000 | 600 |
| | Zhou and Jiang (2012) | 1260 | 1000 | 1000 | 260 | 150 |
| | Arp et al. (2014) | 4300 | 4200 | 3200 | 700 | 150 |
| | Contagio | 340 | 300 | 300 | 40 | 100 |
| Apps | | **252,900** | 20,000 | 16,000 | 4000 | 1000 |

and scan them using KuafuDet and other industrial malware detecting tools. For comparison, the 1000 samples contain both benchmarks before 2014 and the most recent datasets, more than half of which are the most recent malware.

Finally, we measure the efficiency and scalability of KuafuDet performance, and perform the entire process of KuafuDet, using real-time streaming, on a server with 16 GB memory, quad-core i7-4800MQ at 3.6 GHz, and 1 TB hard drives.

### 6.2. *Experimental results*

For a meaningful comparison, we list the results that are used to train on different classifiers with respect to the aspects of false negative (denoted as FN) and accuracy. FN rate refers to all malicious instances that are classified as benign applications. Accuracy simply measures that the classifier makes the correct prediction. Because we use our classifier as a tool for prioritizing the response to Android malware disclosures, we focus on improving the accuracy and reducing the false negative.

#### 6.2.1. *Evaluation on attacks against the detection*

Our collected dataset (16,000 samples as a training set and 4000 samples as a test set, shown in Table 4) serves as a benchmark for evaluating robustness of Android malware detection systems. As mentioned, DroidAPIMiner, Drebin, and MaMaDroid are the three most recent Android malware detection systems. Since Support Vector Machine (SVM) is the only jointly-used algorithm by three detection systems, to conduct a fair comparison, we adopt SVM to investigate the misclassification rate of the three detection systems together with ours (when the adversarial detection mechanism is not included) under poisoning attacks. We first show that by poisoning their training set, it is possible to mislead their classifiers, along with ours; we then analyze the robustness of our discriminative classifiers against the three distinct attack strategies.

(i) **Misclassification of Machine Learning Detection Systems.**

To perform a longitudinal study, we first apply our poisoning attack to DroidAPIMiner, Drebin, MaMaDroid, and KuafuDet without adversarial detection (Without AD). Specifically, *Without AD* means that the camouflage detector component is disabled. We mimic sophisticated attacks to investigate how ineffective these systems perform under our poisoning attack. We assume to have control over a subset of samples and automatically generate the crafted camouflage samples as follows: (i) We can only add or remove features. We must preserve the utility of the modified application, which we achieve by only adding features from benign set, and only those that do not interfere with the functionality of the application. (ii) We can add a restricted number of features. We thus validate that adversarial attack is indeed viable in security critical domains.

More specifically, we customize the adversarial crafting algorithm (Papernot et al., 2016) according to our adversarial model, to indicate the probability of adding or removing a specific feature. Therefore, for machine-learning mechanisms that are based on syntax features, such as DroidAPIMiner and Drebin, we can directly apply our customized adversarial crafting algorithm; for machine-learning mechanisms that mainly consider semantic and behavioral features, such as MaMaDroid, which relies on application behavior using Markov chain modeling, we instead target on crafting its feature in terms of call sequences. Specifically, we first extract a set of call sequences that are only frequently used by benign samples, and then add them to the malicious samples, to mimic our sophisticated attack.

As shown in Table 5, we obtain 80.05%, 75.20%, and 68.95% misclassification rates (*i.e.*, FN) on DroidAPIMiner, Drebin, and MaMaDroid, respectively. We show that our poisoning attack on SVM is successfully validated through three machine-learning tools.

We here take a detailed discussion on the misclassification rate of the systems:

- Our sophisticated attackers are able to mislead the machine learning detection systems.
- MaMaDroid relies on transitional call sequences, rather than single API calls, to train its classifier, merely inserting syntax features as we did for poisoning other systems is not considered as a successful attack by our crafting algorithm. We thereby manipulate its feature space through call sequences.
- MaMaDroid achieves lower misclassification rates than Drebin and DroidAPIMiner in the sophisticated attacks, sacrificing more computational time cost over each application due to call graph construction and feature extraction. Furthermore, MaMaDroid requires a sizable amount of memory when performing classification because of its large feature sets and the extraction of call graph.
- KuafuDet (Without AD) also can be attacked through sophisticated attacks and the misclassification rate (62.60%) indicates that it still suffers from adversarial samples.
- DroidAPIMiner, Drebin, and MaMaDroid can be thwarted if we embed native code (as a strong attacker defined in our threat model) and dynamic code loading with reflection (as a sophisticated attack), because malicious code is loaded or determined at runtime. The attackers can pollute training data using a large-scale crafted samples through these techniques.

In conclusion, the state-of-the-art machine learning-based malware detection systems are possible to be misled by the poisoning attacks we exhibit in the paper. By their nature, classifiers based on syntax features are more vulnerable than the ones based on semantic features. On the other hand, semantic features extraction does require more computational costs than syntax features. Hence, statistical robust features,

**Table 5 – Misclassification rate comparison of adversarial detection.**

| Detection Tool | DroidAPIMiner | Drebin | MaMaDroid | KuafuDet (Without AD) |
|---|---|---|---|---|
| Misclassification rate (FN) | 80.05% | 75.20% | 68.95% | 62.60% |

**Table 6 – The performance of adversarial detection.**

| Conventional Detection | | | SVM | | | RF | | | KNN |
|---|---|---|---|---|---|---|---|---|---|
| FN | | | 4.90% | | | 2.50% | | | 3.40% |
| Accuracy | | | 94.95% | | | 96.35% | | | 95.80% |
| Attacker | Weak | Strong | Sophisticated | Weak | Strong | Sophisticated | Weak | Strong | Sophisticated |
| Without AD | | SVM | | | RF | | | KNN | |
| FN | 8.60% | 49.80% | 62.60% | 5.60% | 41.80% | 55.90% | 5.90% | 26.20% | 45.40% |
| Accuracy | 93.10% | 72.50% | **64.30%** | 94.80% | 76.40% | **67.85%** | 94.55% | 84.40% | **72.00%** |
| Within AD | | SVM | | | RF | | | KNN | |
| FN | 5.80% | 10.00% | 12.60% | 4.70% | 11.20% | 14.50% | 4.10% | 9.20% | 11.60% |
| Accuracy | 94.50% | 92.40% | **89.30%** | 95.65% | 92.00% | **88.85%** | 95.45% | 92.90 | **90.45%** |

pruned by information gain, are adopted in KuafuDet to accommodate scalable, generic, and large-scale malware detection. In addition, KuafuDet provides specific mechanisms to countermeasure the dynamic loading and native code embedding poisonings. KuafuDet parses the native code and dynamic code from package, and then extracts the corresponding features to keep the robustness of classifiers.

In summary, we conjecture that almost all the state-of-the-art machine learning-based malware detection systems are suffering from the poisoning attack we exhibited in the paper.

(ii) **Robustness of KuafuDet.**

Here, we analyze the robustness of our discriminative classifiers when encountering three distinct attack strategies. The first attack strategy is to launch a causative attack without any knowledge of the training data or ground truth. This *weak attacker* in principle amounts to injecting noise into the system. The second attack strategy corresponds to the *strong attacker*, who only manipulates partial features in the training set. The third, the most aggressive attacker we consider is the *sophisticated attacker*. This attacker can fully manipulate almost all training features to launch a sophisticated attack, which creates scenarios where relatively benign mobile applications and real-world malicious mobile applications appear to have nearly identical attributes at the training stage.

As shown in Table 6, the weak attacker is not able to force the accuracy of our malware detection below 90%. This suggests that discriminative classifiers can be relatively robust to this type of random noise-based attack. When dealing with the strong attacker, performance degrades to approximately 90% accuracy. The sophisticated attacker can cause the accuracy to drop to approximately 65% by incorporating thousands of training set (green bar in Fig. 9). The sophisticated attacker represents a practical upper bound for the accuracy loss that a realistic attacker can inflict on our detection system. We see that injecting carefully crafted data into training data can significantly reduce detection accuracy. However, with the help of adversarial detection, holistic performance upgrades by at least 15% accuracy with respect to each listed classifier. Hence, performance of our adversarial detection remains above baseline levels listed in Meng et al. (2016) even for our strongest attackers due to the use of similarity-based filtering to increase classifier robustness. Analysis on false negatives has an analogous interpretation (see Fig. 10). As shown in Table 6, the algorithm KNN outperforms other algorithms under adversarial environment because of its higher resistance to random classification noise, which is aligned with the conclusion drawn from the recent research (Wang et al., 2017).

Fig. 11 displays scatter-plots of system accuracy ($(TP + TN)/(TP + TN + FP + FN)$) as a function of the size of randomly crafted datasets, where TP and TN denote the number of samples correctly classified as malicious and benign, respectively. FP and FN indicate the number of samples mistakenly identified as malicious and benign, respectively. To ease presentation, the plots are fitted by Loess curves with 95% confidence interval bands that depict the upper and lower confidence bounds for all points within the range of data, making it especially useful for comparing groups for which no theoretical models exist. As can be seen from Fig. 11, as the dataset size grows, the accuracy drops slightly. This is somewhat expected as more potential evasion takes place. Furthermore, as discussed earlier, KuafuDet offers significantly better robustness of detection accuracy, regardless of either the size of the dataset or the type of classifier applied. Analysis on the robustness of recall has an analogous interpretation.

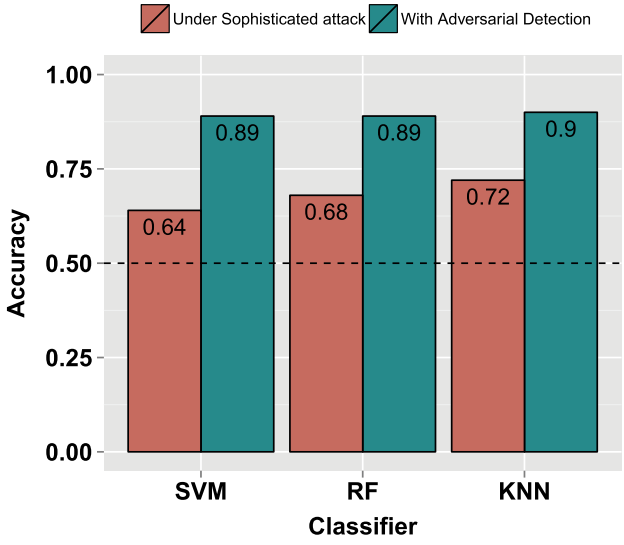


**Fig. 9 – Detection accuracy.**

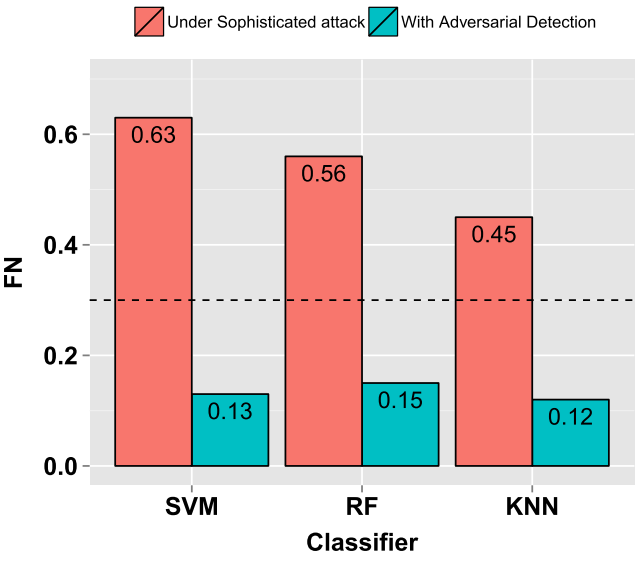


**Fig. 10 – Detection false negatives.**

#### 6.2.2. *Evaluation on accuracy*

We hereby compare our work with the previous work with respect to accuracy. We opt to apply a large portion of our dataset used in KuafuDet, which intergrates the dataset owned by Drebin (Arp et al., 2014) and StormDroid (Chen et al., 2016a), in order to evaluate accuracy performance. Because the dataset used in DroidAPIMiner and MaMaDroid are not publicly available, to be fair, we do not apply the dataset that is not even used in DroidAPIMiner and MaMaDroid *per se* to them to have an asymmetric advantage over it. Hence, Drebin and StormDroid are only considered. As shown in Table 7, our accuracy rate (96.35%) completely outperforms the accuracy rate in StormDroid (93.80%) (Chen et al., 2016a) and Drebin (93.90%) (Arp et al., 2014), let alone use our combined dataset.

As evidenced by Table 7, we achieve the highest accuracy because of the feature selection and similarity-based filtering.

#### 6.2.3. *Robustness of imbalanced data*

Our experiments aim to evaluate the robustness of KuafuDet when the ratio goes imbalanced in the adversarial environment.

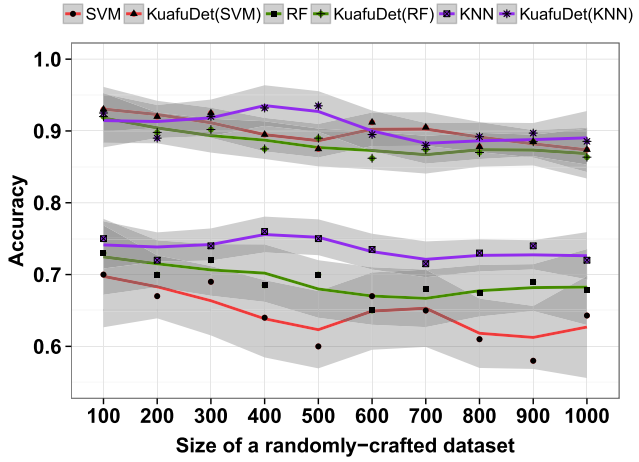


**Fig. 11 – The robustness of KuafuDet (accuracy).**

**Table 7 – Accuracy comparison.**

| Detection Tool | Accuracy | # Malware |
|---|---|---|
| KuafuDet | **96.35%** | 10,400 |
| Drebin | 93.90% | 5,560 |
| StormDroid | 93.80% | 3,620 |

**Table 8 – Results for different malware to benign apps ratios.**

| Ratio | 1:1 | 1:5 | 1:10 | 1:20 | 1:50 |
|---|---|---|---|---|---|
| Accuracy | 96.40% | 96.15% | 95.80% | 94.60% | 93.75% |

**Table 9 – Time cost of KuafuDet in units of seconds.**

| # APKs | Total time | AVG time/APK |
|---|---|---|
| 200 | 518 | 2.59 |
| 400 | 1066 | 2.67 |
| 600 | 1578 | 2.63 |
| 800 | 2097 | 2.62 |
| 1000 | 2778 | 2.78 |

To evaluate the robustness of KuafuDet in the data-imbalanced environment, we first apply 4000 malicious and 4000 benign apps (*i.e.*, 1:1 ratio) for training our classifiers, and gradually add benign ones to achieve different ratios up to 1:50. To be specific, we conduct experiments with ratios including 1:1, 1:5, 1:10, 1:20, and 1:50. We use 10-fold cross-validation in our experiments. As shown in Table 8, the accuracy degrades as we approach real-world ratio of malware and benign apps, but the accuracy still remains above 90%. In the self-adaptive scheme, we, nevertheless, show the capability to tackle the problem of the imbalanced data ratio.

#### 6.2.4. *Evaluation on time cost, scalability, and adaptability*

To support a high-performance malware detection, KuafuDet is designed to run on top of an open-source distributed stream-processing engine *Storm*.[4] KuafuDet supports a large-scale detection of a data stream by a set of worker units that connect to each other, forming a topology: A submitted application is first disassembled to extract its features; then, the metrics-driven pruning and information gain analysis are run, and two-phase iterative adversarial detection is finally activated. Each operation here is delegated to a worker unit on the topology and all the data associated with the application are in a single stream. Running on top of the *Storm* stream processor, KuafuDet is tested on the platform of Pwnzen Infotech Inc. We show that the size of the test group is unaffected with the efficiency of our tool. As shown in Table 9, average detection time per application is less than 3 seconds, which is indeed capable of scaling up to the massive datasets.

#### 6.2.5. *Evaluation on coverage*

To circumvent the over-fitting issue and to better understand the coverage of KuafuDet, we randomly sample 1000 mali-

[4] http://storm.apache.org/.

**Table 10 – Coverage comparison.**

| Detection Tool | Percentage |
|---|---|
| KuafuDet | **96.20%** |
| ESET-NOD32 | 79.50% |
| McAfee | 75.50% |
| Ikarus | 72.50% |
| Kaspersky | 72.10% |
| Avira | 69.30% |
| VIPER | 67.50% |
| Qihoo-360 | 62.30% |
| Symantec | 40.40% |

cious applications from 217 Android malware families from our dataset to cover almost all the existing Android malicious behaviors. We scan them using KuafuDet and other well-known industrial malware detection tools, such as Kaspersky and McAfee encapsulated in VirusTotal. Although the coverage of KuafuDet, with the combined top features, is 96.20%, better than what can be achieved by any individual scanner, including such top-of-the-line antivirus systems as ESET-NOD32 (79.50%), McAfee (75.50%), Ikarus (72.50%), Kaspersky (72.10%), and Avira (69.30%), industrial tools deal with millions of applications, many of which are zero-day. We argue that comparing with industrial tools is to understand the different emphases in academia and industry. The breakdowns of the coverage study is presented in Table 10.

Since KuafuDet decouples the similarity-based filtering from machine-learning classifiers, it enables us to periodically enhance the learning system. Moreover, KuafuDet also considers an attacker threat dimension, which makes the whole system design completely adaptable to new malware.

## 7. Discussion

Our study is limited in five ways as discussed in the following.

(i) **The granularity of classifiers.** The hyperplane between benign and malicious can be blurred and subjective, which depends on specific security requirements and uses cases to determine whether a pattern is really benign or malicious. For example, if individual users root their own devices and use the game hacking applications, game developers are very likely to treat them as malicious because they bypass the in-app purchase. In practice, this kind of apps is defined as "grayware" that has no clear distinctive difference between the benign and the malicious. "Grayware" is now becoming a great threat to mobile devices since attackers achieve more profit in this way. KuafuDet is a generic and coarse-grained architecture for classifying applications with high accuracy. As for such grayware, KuafuDet can be tuned for a specific detection.

(ii) **The limitation of decompilation technologies.** We extract features from the *manifest* and *Smali* files that are successfully decompiled. However, in our experiments, we find that a few APK files cannot be decompiled successfully. As for this situation, we change the decompiling tool in our experiments. For future study, we will explore the possibility to use reinforcement techniques to prevent the APKs from reverse-engineering. This would increase the difficulty of unpacking the original APK for attackers.

(iii) **The scarcity of empirical samples.** Although we find some case studies reflecting the different attacker levels for machine learning classifiers, we still lack a huge number of samples to scrutinize. We note attackers might be able to deliberately force the benign and malicious access patterns to co-occur in one log, such as triggering the benign pattern first and then launching the attack, though we have not observed in the wild. This perhaps requires to dilute the poisoned logs and possibly requires human analysts to contribute external knowledge. We hope in future study that using Game Theory is beneficial to our interpretation of the attackers psychology so as to take targeted strategy to detect stubborn malware.

(iv) **The limitation of selected features.** Although the 195 features are representative and extracted by using statistical metrics-driven manual pruning with information gain in our experiments, new malicious behaviors might disturb the feature space, making the system less effective.

(v) **The limitation of similarity-based approach.** (1) Attackers can somehow approximate our similarity-based approach by inferring the similarity thresholds used. However, it is actually difficult to infer our selected samples that are used to calculate the thresholds. Furthermore, the thresholds will change as selected samples update over time, for which attackers will take great efforts to exploit. (2) KuafuDet, through a self-adaptive learning scheme, discovers new information from both the identified malware and the filtered suspicious false negatives from camouflage detector. We acknowledge that this process would cause false positives. For example, SMS-related applications are benign applications, but they also have sensitive behaviors that are typical in Android malware.

## 8. Related work

Contemporary machine learning-based techniques typically model the detection problem as a binary classification problem. Together with system analysis techniques, the malicious behaviors can be studied and employed to increase their detection performance, especially for mobile applications in the wild.

### 8.1. *Machine learning-based detection*

Arp et al. (2014) built the Drebin system, which works with a massive feature set extracted from the manifest file and the app's source code and trains an SVM classifier for malware detection. Although Drebin has accommodated thousands of features with an impressive performance results, it suffers two challenges: first, the malware is out-dated and well recorded

in malware detection tools; second, a comprehensive coverage of different attacking and evasion techniques is missing.

DroidAPIMiner (Aafer et al., 2013) mainly extracts the top 169 API calls, which are used more frequently in the malware than in the benign set, package level information, as well as some dangerous parameter information as features to analyze a large corpus of Android malware. Because of the evolution of both malware and the Android API, it requires constant retraining on most common calls.

Most recently, Chen et al. (2016c) suggest the use of semantic features of mobile apps to retain classifier value over time, building on the intuition that certain semantic attributes of mobile malware are invariant. Experiments verify that the incorporation of semantic features can significantly improve the performance of Android malware classification. Deo et al. (2016) propose to assess the quality of binary classification by using probabilistic predictors. Although they both consider retraining, adversarial environment is missing.

MUDFLOW (Avdiienko et al., 2015) argues that the pattern of sensitive information flows in malware is statistically different from those in benign apps. From an application, MUDFLOW uses static analysis to extract the flow paths, and these flow paths are then mapped to a feature vector used in a specific classifier. DroidSIFT (Zhang et al., 2014) is unique in designing features in terms of the generation of API dependency graphs $G$ for each app, and the construction of the feature vector of the app. The features represent the similarity of the graphs $G$ corresponding to the database of graphs of known benign apps and malware apps. Finally, the feature vectors are used in anomaly detection. However, the dataset for detection is not large enough, yielding low effectiveness and confidence in classification. DroidMat (Wu et al., 2012) uses a static feature-based mechanism, which considers static information (*e.g.*, permission, intent messages, API calls) for detecting Android malware. It uses $K$-means algorithm to enhance the malware modeling capability and KNN algorithm to classify apps as benign or malicious. However, it does not extract semantic features for training and does not take adversarial environment into consideration. Shabtai et al. (2012) presented a host-based malware detection system that continuously monitors mobile devices to detect malicious data using a supervised machine-learning anomaly detection technique. It focuses on host-based malware, while our approach focuses on mobile malware detection in adversarial environment. Most recently, MaMaDroid (Mariconti et al., 2017) is built to maintain resilience to API changes, but it requires a large amount of memory when performing classification and a substantial amount of time per app.

### 8.2. *Evasion techniques*

Currently, the issues of understanding machine-learning security in adversarial settings mainly focus on spam email detection (Biggio et al., 2014; Brückner et al., 2012; Debarr et al., 2013; Wang et al., 2014; Zhang et al., 2016). Recently, many statistical adversarial models are proposed to construct effective adversarial samples, such as using deep neutral networks (Grosse et al., 2016; Li and Li, 2016; McDaniel et al., 2016; Papernot et al., 2016; Shen et al., 2016). As seen from a generic perspective, Wang et al. (2016) utilized the notation of topological spaces and oracles to explain why an adversarial sample can bypass a classifier, and they generated a sufficient and necessary condition to determine the robustness of classifiers under adversarial environment. Goodfellow et al. (2014) explained and generated adversarial samples for adversarial training to reduce test error. However, all of these studies did not focus on specific causation leading to evasion in the mobile malware context. They also did not show the feasibility how these adversarial samples work in the wild.

For conventional malware evasion, one straightforward evasion technique is to repackage a benign app with small snippets of malicious code added to several classes. Moreover, attackers could also use reflection, dynamic code loading, or native code (Poeplau et al., 2014). Such attempts to escape detection are likely to be deemed suspicious. Among them, DroidChameleon (Rastogi et al., 2014) integrates three types of transformation techniques and generates obfuscated mobile malware. Mystique (Meng et al., 2016) develops a framework to automatically generate malware covering four attack features and two evasion features to obfuscate the generated malware. For the general defense, Cao and Yang (2015) presented a proof-of-concept machine unlearning prototype that can rapidly forget data to regain privacy, security, and usability. The current paper is an extension of a poster (Chen et al., 2016b).

In summary, previous works either conduct evasion techniques without considering the feature space or only use machine learning-based approaches. With our experiments and real-world case studies, it is obvious that attacks can also poison features while preserving maliciousness, and our experiments verified that the resulting fake variants with poisoned features impaired discriminative classifiers and succeeded in lowering the detection score in a test environment. To the best of our knowledge, this is one of the first papers to accommodate adversarial machine learning into mobile malware detection. We are also the first paper to show the possibility to defend against adversarial attacks on mobile malware, to the greatest extent.

## 9. Conclusion

We reviewed several challenges for the malware detection problem. We showed how the conventional machine learning classifiers can fail against determined attackers. Based on these insights, we designed and evaluated three types of attackers targeting the training phases to poison our detection. Through simulation, we presented practical bounds for the accuracy loss to each attacker. To address this threat, we therefore proposed our detection system, KuafuDet, and showed it significantly reduces false negatives and boosts the detection accuracy by at least 15%.

We argue that it is essential to inform researchers considering how attackers will adapt to the conventional detection, as well as to inform developers working on the next-generation malware detection systems. We conjecture that the arms race will be over only when the effectiveness of early detection will sufficiently increase the cost of infection.

## Appendix A. Syntax and Semantic Features

**Table A.11 – 175 syntax features and 20 semantic features for training classifiers.**

| PERMISSION | SET_WALLPAPER | TelephonyManager.getSubscriberId | LocationManager.addNmeaListener |
|---|---|---|---|
| ACCESS_COARSE_LOCATION | SET_WALLPAPER_HINTS | TelephonyManager.getVoiceMailNumber | LocationManager.addProximityAlert |
| ACCESS_FINE_LOCATION | STATUS_BAR | TelephonyManager.hasIccCard | LocationManager.addTestProvider |
| ACCESS_LOCATION_EXTRA_COMMANDS | SYSTEM_ALERT_WINDOW | TelephonyManager.isNetworkRoaming | LocationManager.clearTestProviderLocation |
| ACCESS_NETWORK_STATE | UPDATE_DEVICE_STATS | SmsManager.divideMessage | LocationManager.getBestProvider |
| ACCESS_WIFI_STATE | USE_CREDENTIALS | SmsManager.getDefault | LocationManager.getGpsStatus |
| AUTHENTICATE_ACCOUNTS | VIBRATE | SmsManager.sendDataMessage | LocationManager.getLastKnownLocation |
| BATTERY_STATS | WAKE_LOCK | SmsManager.sendMultipartTextMessage | LocationManager.requestLocationUpdates |
| BLUETOOTH | WRITE_APN_SETTINGS | SmsManager.sendTextMessage | LocationManager.sendExtraCommand |
| BROADCAST_SMS | WRITE_SETTINGS | HttpURLConnection.disconnect | WifiManager.addNetwork |
| BROADCAST_STICKY | WRITE_SMS | HttpURLConnection.getContentEncoding | WifiManager.calculateSignalLevel |
| CALL_PHONE | WRITE_EXTERNAL_STORAGE | HttpURLConnection.getPermission | WifiManager.createWifiLock |
| CAMERA | **INTENT** | HttpURLConnection.getRequestMethod | WifiManager.disconnect |
| CHANGE_COMPONENT_ENABLED_STATE | action.DELETE | HttpURLConnection.getResponseCode | WifiManager.enableNetwork |
| CHANGE_CONFIGURATION | action.GET_CONTENT | HttpURLConnection.getResponseMessage | WifiManager.getConfiguredNetworks |
| CHANGE_NETWORK_STATE | action.MAIN | HttpURLConnection.usingProxy | WifiManager.getConnectionInfo |
| CHANGE_WIFI_MULTICAST_STATE | action.PICK | ContentResolver.bulkInsert | WifiManager.getDhcpInfo |
| CHANGE_WIFI_STATE | action.SEND | ContentResolver.getType | WifiManager.getScanResults |
| CLEAR_APP_CACHE | action.SET_WALLPAPER | ContentResolver.openAssetFileDescriptor | WifiManager.getWifiState |
| CONTROL_LOCATION_UPDATES | action.VIEW | ContentResolver.query | WifiManager.isWifiEnabled |
| DELETE_CACHE_FILES | category.BROWSABLE | ContentResolver.registerContentObserver | WifiManager.removeNetwork |
| DELETE_PACKAGES | category.DEFAULT | ContentResolver.update | WifiManager.saveConfiguration |
| DEVICE_POWER | category.HOME | ContentResolver.delete | WifiManager.setWifiEnabled |
| DISABLE_KEYGUARD | category.INFO | Runtime.getRuntime | NotificationManager.cancel |
| EXPAND_STATUS_BAR | category.LAUNCHER | Runtime.exec | NotificationManager.notify |
| FLASHLIGHT | **HARDWARE** | Runtime.addShutdownHook | PackageManager.checkPermission |
| GET_PACKAGE_SIZE | camera | Runtime.maxMemory | PowerManager.isInteractive |
| GET_TASKS | camera.autofocus | URLConnection.addRequestProperty | PowerManager.isScreenOn |
| INSTALL_PACKAGES | sensor.accelerometer | URLConnection.connect | PowerManager.newWakeLock |
| INTERNET | telephony | URLConnection.getConnectTimeout | **SEMANTIC** |
| KILL_BACKGROUND_PROCESSES | touchscreen | URLConnection.getContent | *Install application* |
| MODIFY_PHONE_STATE | **API CALL** | URLConnection.getContentType | *Uninstall application* |
| MOUNT_UNMOUNT_FILESYSTEMS | URL.openConnection | URLConnection.getDefaultUseCaches | *Get installed packages* |
| NFC | URL.openStream | URLConnection.getPermission | *Monitor URI* |
| PERSISTENT_ACTIVITY | URL.getContent | URLConnection.getURL | *Download file* |
| PROCESS_OUTGOING_CALLS | TelephonyManager.getCallState | URLConnection.setConnectTimeout | *Get location* |
| READ_CALL_LOG | TelephonyManager.getCellLocation | URLConnection.setReadTimeout | *Read SD card* |
| READ_CONTACTS | TelephonyManager.getDeviceId | ActivityManager.getLargeMemoryClass | *Write SD card* |
| READ_EXTERNAL_STORAGE | TelephonyManager.getDeviceSoftwareVersion | ActivityManager.getRunningAppProcesses | *Request for chmod* |
| READ_LOGS | TelephonyManager.getNeighboringCellInfo | ActivityManager.isLowRamDevice | *Start http connection* |
| READ_PHONE_STATE | TelephonyManager.getNetworkCountryIso | ActivityManager.killBackgroundProcesses | *Send Sms* |
| READ_PROFILE | TelephonyManager.getNetworkOperator | ActivityManager.restartPackage | *Receive Sms* |
| READ_SMS | TelephonyManager.getNetworkOperatorName | BluetoothAdapter.cancelDiscovery | *Delete Sms* |
| RECEIVE_BOOT_COMPLETED | TelephonyManager.getNetworkType | BluetoothAdapter.getAddress | *Intercept Sms receiver* |
| RECEIVE_MMS | TelephonyManager.getPhoneType | BluetoothAdapter.getBondedDevices | *Get wifi info* |
| RECEIVE_SMS | TelephonyManager.getSimCountryIso | BluetoothAdapter.getRemoteDevice | *Get Logs* |
| RECEIVE_WAP_PUSH | TelephonyManager.getSimOperator | BluetoothSocket.connect | *Get Class loader* |
| RECORD_AUDIO | TelephonyManager.getSimOperatorName | DownloadManager.enqueue | *Get contacts* |
| RESTART_PACKAGES | TelephonyManager.getSimSerialNumber | DownloadManager.query | *Get account* |
| SEND_SMS | TelephonyManager.getSimState | LocationManager.addGpsStatusListener | *Get phone type/Sim serial number/device id/subscriber id/IMSI* |

## REFERENCES


Aafer Y, Du W, Yin H. DroidAPIMiner: mining API-level features for robust malware detection in Android. In: Security and privacy in communication networks. Springer; 2013. p. 86–103.

Arp D, Spreitzenbarth M, Hubner M, Gascon H, Rieck K. DREBIN: effective and explainable detection of Android malware in your pocket. In: Proceedings of the annual symposium on Network and Distributed System Security (NDSS). 2014.

Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, et al. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: ACM SIGPLAN notices, vol. 49. ACM; 2014. p. 259–69.

Avdiienko V, Kuznetsov K, Gorla A, Zeller A, Arzt S, Rasthofer S, et al. Mining apps for abnormal usage of sensitive data. In: 2015 IEEE/ACM 37th IEEE international conference on software engineering, vol. 1. IEEE; 2015. p. 426–36.

Biggio B, Fumera G, Roli F. Security evaluation of pattern classifiers under attack. IEEE Trans Knowl Data Eng 2014;26(4):984–96.

Brunk F. 2009. Intersection problems in combinatorics [Ph.D. thesis]. University of St Andrews.

Brückner M, Kanzow C, Scheffer T. Static prediction games for adversarial learning problems. J Mach Learn Res 2012;13(Sep):2617–54.

Cao Y, Yang J. Towards making systems forget with machine unlearning. In: Security and privacy (SP), 2015 IEEE symposium on. IEEE; 2015. p. 463–80.

Chen S, Xue M, Tang Z, Xu L, Zhu H. StormDroid: a streaminglized machine learning-based system for detecting Android malware. In: Proceedings of the 11th ACM on Asia conference on computer and communications security. ACM; 2016a. p. 377–88.

Chen S, Xue M, Xu L. Poster: towards adversarial detection of mobile malware. In: Proceedings of the 22nd annual international conference on mobile computing and networking. ACM; 2016b. p. 415–16.

Chen W, Aspinall D, Gordon AD, Sutton C, Muttik I. More semantics more robust: improving android malware classifiers. In: Proceedings of the 9th ACM conference on security & privacy in wireless and mobile networks. ACM; 2016c. p. 147–58.

Dash SK, Suarez-Tangil G, Khan S, Tam K, Ahmadi M, Kinder J, et al. DroidScribe: classifying Android malware based on runtime behavior. Mobile Security Technologies (MoST 2016) 2016;7148:1–12.

Debarr D, Sun H, Wechsler H. Adversarial spam detection using the randomized hough transform-support vector machine. In: Machine Learning and Applications (ICMLA), 2013 12th international conference on, vol. 1. IEEE; 2013. p. 299–304.

Deo A, Dash SK, Suarez-Tangil G, Vovk V, Cavallaro L. Prescience: probabilistic guidance on the retraining conundrum for malware detection. In: Proceedings of the 2016 ACM workshop on artificial intelligence and security. ACM; 2016. p. 71–82.

Enck W, Gilbert P, Han S, Tendulkar V, Chun B-G, Cox LP, et al. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Transactions on Computer Systems (TOCS) 2014;32(2):5.

Fan L, Xue M, Chen S, Xu L, Zhu H. Poster: accuracy vs. time cost: detecting android malware through pareto ensemble pruning. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. ACM; 2016. p. 1748–50.

Feizollah A, Anuar NB, Salleh R, Suarez-Tangil G, Furnell S. Androdialysis: analysis of android intent effectiveness in malware detection. Computers & Security 2017;65:121–34.

Felt AP, Chin E, Hanna S, Song D, Wagner D. Android permissions demystified. In: Proceedings of the 18th ACM conference on computer and communications security. ACM; 2011. p. 627–38.

Goodfellow IJ, Shlens J, Szegedy C. 2014. Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572.

Gordon MI, Kim D, Perkins JH, Gilham L, Nguyen N, Rinard MC. Information flow analysis of Android applications in DroidSafe. In: Proceedings of the annual symposium on Network and Distributed System Security (NDSS). 2015.

Graziano M, Canali D, Bilge L, Lanzi A, Balzarotti D. Needles in a haystack: mining information from public dynamic analysis sandboxes for malware intelligence. In: 24th USENIX Security Symposium (USENIX Security 15). 2015. p. 1057–72.

Grosse K, Papernot N, Manoharan P, Backes M, McDaniel P. 2016. Adversarial perturbations against deep neural networks for malware classification. arXiv preprint arXiv:1606.04435.

Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH. The WEKA data mining software: an update. ACM SIGKDD Explorations Newsletter 2009;11:10–18.

Idrees F, Rajarajan M, Conti M, Chen T, Rahulamathavan Y. PIndroid: a novel android malware detection system using ensemble learning methods. Computers & Security. 2017;doi:10.1016/j.cose.2017.03.011.

Li L, Bartel A, Bissyandé TF, Klein J, Le Traon Y, Arzt S, et al. IccTA: detecting inter-component privacy leaks in Android apps. In: Proceedings of the 37th international conference on software engineering, vol. 1. IEEE Press; 2015. p. 280–91.

Li X, Li F. 2016. Adversarial examples detection in deep networks with convolutional filter statistics. arXiv preprint arXiv:1612.07767.

Mariconti E, Onwuzurike L, Andriotis P, De Cristofaro E, Ross G, Stringhini G. MAMADROID: detecting Android malware by building Markov chains of behavioral models. In: Proceedings of the annual symposium on Network and Distributed System Security (NDSS). 2017.

McDaniel P, Papernot N, Celik ZB. Machine learning in adversarial settings. IEEE Security & Privacy 2016;14(3):68–72.

Meng G, Xue Y, Mahinthan C, Narayanan A, Liu Y, Zhang J, et al. Mystique: evolving Android malware for auditing anti-malware tools. In: Proceedings of the 11th ACM on Asia conference on computer and communications security. ACM; 2016. p. 365–76.

Octeau D, Jha S, Dering M, McDaniel P, Bartel A, Li L, et al. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In: ACM SIGPLAN notices. vol. 51. ACM; 2016. p. 469–84.

Papernot N, McDaniel P, Jha S, Fredrikson M, Celik ZB, Swami A. The limitations of deep learning in adversarial settings. In: Security and Privacy (EuroS&P), 2016 IEEE European symposium on. IEEE; 2016. p. 372–87.

Poeplau S, Fratantonio Y, Bianchi A, Kruegel C, Vigna G. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In: NDSS, vol. 14. 2014. p. 23–6.

Rasthofer S, Arzt S, Bodden E. A machine-learning approach for classifying and categorizing Android sources and sinks. In: Proceedings of the annual symposium on Network and Distributed System Security (NDSS). 2014.

Rasthofer S, Arzt S, Miltenberger M, Bodden E. Harvesting runtime values in Android applications that feature anti-analysis techniques. In: Proceedings of the annual symposium on Network and Distributed System Security (NDSS). 2016.

Rastogi V, Chen Y, Jiang X. Catch me if you can: evaluating Android anti-malware against transformation attacks. IEEE Transactions on Information Forensics and Security 2014;9(1):99–108.
Schlegel R, Zhang K, Zhou X, Intwala M, Kapadia A, Wang X. Soundcomber: a stealthy and context-aware sound Trojan for smartphones. In: Proceedings of the annual symposium on Network and Distributed System Security (NDSS), vol. 11. 2011. p. 17–33.
Shabtai A, Kanonov U, Elovici Y, Glezer C, Weiss Y. Andromaly: a behavioral malware detection framework for Android devices. Journal of Intelligent Information Systems 2012;38(1):161–90.
Shen S, Tople S, Saxena P. AUROR: defending against poisoning attacks in collaborative deep learning systems. In: Proceedings of the 32nd annual conference on computer security applications. ACM; 2016. p. 508–19.
Szegedy C, Zaremba W, Sutskever I, Bruna J, Erhan D, Goodfellow I, et al. Intriguing properties of neural networks. In: Proceedings of the 2014 international conference on learning representations. Computational and biological learning society. 2014.
Tam K, Khan SJ, Fattori A, Cavallaro L. CopperDroid: automatic reconstruction of Android malware behaviors. In: Proceedings of the annual symposium on Network and Distributed System Security (NDSS). 2015.
Wang B, Gao J, Qi Y. 2016. A theoretical framework for robustness of (deep) classifiers under adversarial noise. arXiv preprint arXiv:1612.00334.
Wang F, Liu W, Chawla S. On sparse feature attacks in adversarial learning. In: Data Mining (ICDM), 2014 IEEE international conference on. IEEE; 2014. p. 1013–18.
Wang Y, Jha S, Chaudhuri K. 2017. Analyzing the robustness of nearest neighbors to adversarial examples. arXiv preprint arXiv:1706.03922.
Wong MY, Lie D. IntelliDroid: a targeted input generator for the dynamic analysis of Android malware. In: Proceedings of the annual symposium on Network and Distributed System Security (NDSS). 2016.
Wu C, Zhou Y, Patel K, Liang Z, Jiang X. AirBag: boosting smartphone resistance to malware infection. In: Proceedings of the annual symposium on Network and Distributed System Security (NDSS). 2014.
Wu D-J, Mao C-H, Wei T-E, Lee H-M, Wu K-P. DroidMat: Android malware detection through manifest and API calls tracing. In: Information security (Asia JCIS), 2012 seventh Asia joint conference on. IEEE; 2012. p. 62–9.
Yan LK, Yin H. DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In: Presented as part of the 21st USENIX security symposium (USENIX Security 12). 2012. p. 569–84.
Yang C, Xu Z, Gu G, Yegneswaran V, Porras P. DroidMiner: automated mining and characterization of fine-grained malicious behaviors in Android applications. In: European symposium on research in computer security. Springer; 2014. p. 163–82.
Yang W, Xiao X, Andow B, Li S, Xie T, Enck W. AppContext: differentiating malicious and benign mobile app behaviors using context. In: Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE international conference on, vol. 1. IEEE; 2015. p. 303–13.
Zhang F, Chan PP, Biggio B, Yeung DS, Roli F. Adversarial feature selection against evasion attacks. IEEE Transactions on Cybernetics 2016;46(3):766–77.
Zhang M, Duan Y, Yin H, Zhao Z. Semantics-aware Android malware classification using weighted contextual API dependency graphs. In: Proceedings of the 2014 ACM SIGSAC conference on computer and communications security. ACM; 2014. p. 1105–16.
Zhou W, Zhou Y, Grace M, Jiang X, Zou S. Fast, scalable detection of piggybacked mobile applications. In: Proceedings of the third ACM conference on data and application security and privacy. ACM; 2013. p. 185–96.
Zhou Y, Jiang X. Dissecting Android malware: characterization and evolution. In: Security and privacy (SP), 2012 IEEE symposium on. IEEE; 2012. p. 95–109.
Zhou Y, Wang Z, Zhou W, Jiang X. Hey, you, get off of my market: detecting malicious apps in official and alternative Android markets. In: NDSS, vol. 25. 2012. p. 50–2.

**Sen Chen** is pursuing his Ph.D. degree at the School of Computer Science and Software Engineering of East China Normal University, focusing primarily on areas of smartphone security, Android malware, vulnerability and program analysis. He has received the MobiCom 2016 Travel Grant Award. He is currently serving as a visiting scholar in Cyber Security Lab at Nanyang Technological University. He is currently advised by Professor Lihua Xu (ECNU) and Yang Liu (NTU).

**Minhui Xue** is pursuing his Ph.D. degree at the School of Computer Science and Software Engineering of East China Normal University. He is also serving as a visiting scholar at the Courant Institute of Mathematical Sciences and Tandon School of Engineering at New York University, as well as a research assistant at New York University Shanghai, advised by Professor Keith W. Ross (NYU). Previously, he received a Bachelor of Science degree in the field of fundamental mathematics from East China Normal University in July 2013. His current research interests are in data-driven analysis of online social networks and privacy.

**Lingling Fan** is pursuing her Ph.D. degree at the School of Computer Science and Software Engineering of East China Normal University, focusing on software testing, model checking, and Android application analysis. She is interested in software testing and analysis, aiming at bug revelation and bug localization, and malware detection. She received a Bachelor of Science degree in the field of computer science and technology from ECNU, as an excellent graduate student of Shanghai. She received an Excellent Student award from ECNU. She is currently advised by Professor Lihua Xu (ECNU) and Yang Liu (NTU).

**Shuang Hao** is an assistant professor at the Department of Computer Science at the University of Texas at Dallas. He is broadly interested in all aspects of network and system security. His work follows a measurement- and data-driven approach to characterize and detect critical security issues in large-scale systems. His current research focuses on anomaly detection, underground economics, DNS analysis, web and mobile security. He obtained his Ph.D. in Computer Science at the Georgia Institute of Technology, and he did a Postdoctorate at the University of California, Santa Barbara.

**Lihua Xu** is an associate professor at School of Computer Science and Software Engineering, East China Normal University. She received her Ph.D. and M.Sc. degree from University of California at Irvine. Her current research focuses on software engineering, automated software analysis and testing, and mobile security.

**Haojin Zhu** is currently a professor at Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. He received his B.Sc. degree (2002) from Wuhan University (China), his M.Sc.(2005) degree from Shanghai Jiao Tong University (China), both in computer science and the Ph.D. in Electrical and Computer Engineering from the University of Waterloo (Canada), in 2009. His current research interests include network security and data privacy. He serves as the associate/guest editor of IEEE Internet of Things

Journal, IEEE Wireless Communications, IEEE Network, and Peer-to-Peer Networking and Applications.

**Bo Li** is currently a Postdoctoral Fellow at the EECS department at UC Berkeley University, working with Prof. Dawn Song. She will join the EECS department at University of Illinois at Urbana – Champaign as an assistant professor in June 2018. Her research focuses on both theoretical and practical aspects of machine learning, security, privacy, game theory, social networks, and adversarial deep learning.