6-2018

Advanced malware detection for android platform

Ke XU
Singapore Management University, kexu.2013@phdis.smu.edu.sg

ADVANCED MALWARE DETECTION FOR ANDROID PLATFORM

KE XU

SINGAPORE MANAGEMENT UNIVERSITY

2018

Advanced Malware Detection for Android Platform

by

Ke Xu

Submitted to School of Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Information Systems

Dissertation Committee:

Singapore Management University

2018

Advanced Malware Detection for Android Platform

Ke Xu

Abstract

In the first quarter of 2018, 75.66% of smartphones sales were devices running Android. Due to its popularity, cyber-criminals have increasingly targeted this ecosystem. Malware running on Android severely violates end users security and privacy, allowing many attacks such as defeating two factor authentication of mobile banking applications, capturing real-time voice calls and leaking sensitive information. In this dissertation, I describe the pieces of work that I have done to effectively detect malware on Android platform, i.e., ICC-based malware detection system (ICCDetector), multi-layer malware detection system (DeepRefiner), and self-evolving and scalable malware detection system (DroidEvolver) for Android platform.

Most existing malware detection methods are designed based on the resources required by malware. These methods capture the interactions between applications and Android system, but ignore the communications among components within or cross application boundaries. To address this challenge, we systemically analyze ICC patterns of benign applications and malware, and propose a new malware detection system, ICCDetector, which detects malware based on not required resources, but ICC patterns. Our experiments show that ICCDetector achieves an accuracy of 97.4%, roughly 10% higher than the baseline, with a lower false positive rate of 0.67%. In addition, the detected malware is further classified into five new malware categories according to their ICC characteristics, which clarifies the relationship between malware behaviors and ICC patterns.

As the complexities of mobile malicious behaviors vary significantly across malware, it is difficult to perform effective and efficient detection applying single classifier. In addition, both Android system and malware rapidly evolve over years. As a consequence, it is also challenging to practically detect malware relying on laborious human feature engineering and complicated feature extraction process. In this dissertation, we propose DeepRefiner, a novel detection system which identifies malware both effectively and efficiently. DeepRefiner includes multiple detection layers to distinguish malware complexities in the detection process, and performs refined detection for stealthy and sophisticated malware according to comprehensive bytecode semantics. We evaluate the detection performance of DeepRefiner with 62,915 malware and 47,525 benign applications, showing that DeepRefiner effectively detects malware with an accuracy of 97.74% with a false positive rate of 2.54%. We compare DeepRefiner with a state-of-the-art single classifier-based detection system, StormDroid, and ten widely used signature-based anti-virus scanners. The experimental results show that DeepRefiner significantly outperforms both StormDroid and anti-virus scanners.

Given the frequent changes in the Android framework and the continuous evolution of malware, it is challenging to detect malware over time in a both effective and scalable way. To address this challenge, we propose DroidEvolver, an Android malware detection system that can automatically and continually update itself during malware detection without any human involvement. While most existing malware detection systems can be updated by retraining on new applications with true labels, DroidEvolver requires neither retraining nor true labels to update itself, mainly due to the insight that DroidEvolver makes necessary and lightweight update using online learning techniques with evolving feature set and pseudo labels. We evaluate the detection accuracy of DroidEvolver on a dataset of 33,294 benign applications and 34,722 malicious applications developed over a period of six years. Using applications dated in 2011 as the initial training set, DroidEvolver achieves high detection accuracy (95.28%), which only declines by 1.68% on average per year over the next five years. Compared with the state-of-the-art over-time malware detection system MAMADROID, the accuracy and scalability of DroidEvolver is consistenly and significantly higher than the baseline.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my gratitude to my supervisors, collaborators, dissertation committee members, friends, university staffs, and family members, for their kindly help in many ways smoothing the progress of my PhD.

First and foremost, I thank my supervisors, Prof. Robert H. Deng and Prof. Yingjiu Li, for their consistently support and encouragement during my PhD training. Next, I thank my dissertation committee members Prof. Debin Gao and Dr. Tieyan Li, for taking time reviewing my thesis, providing valuable suggestions, and attending my defense. I thank my collaborators from worldwide institutions, for their help in my research. They are Kai Chen from Chinese Academy of Sciences and Jason Hong from Carnegie Mellon University. I acknowledge the friendship and support from my group members in the security group of SMU and my friends Siqi Ma, Yuan Tian and Xiaoxiao Tang. I am also grateful to the following university staffs: Pei Huan Seow, Yar Ling Yeo and Chew Hong Ong, for their unfailing support and assistance.

Last but not least, I am grateful to my parents, Ruqing Xu and Guiyun Li, for their generous care and believes in me. They make my life brighter and much easier:-)

List of Publications

Conference Papers

K. Xu, Y. Li, R. H. Deng, and K. Chen. DeepRefiner: Multi-layer Android Malware Detection System Applying Deep Neural Networks. In *Proceedings of the 3rd IEEE European Symposium on Security and Privacy*, United Kingdom, 2018.

Journal Papers

K. Xu, Y. Li, and R. H. Deng. ICCDetector: ICC-Based Malware Detection on Android. *IEEE Transactions on Information Forensics and Security*, 11(6), 2016, 1252-1264.

Chapter 1

Introduction

1.1 Overview of Android Malware Detection

As one of the most popular platforms for mobile devices, Android provides a wealth of functionalities to its users. Unfortunately, mobile devices running Android are increasingly targeted by attackers and infected with malware. In contrast to other platforms, Android allows for installing applications from unverified sources, such as third-party markets, which makes bundling and distributing malware easy for attackers. Malware running on Android severely violates end users security and privacy, allowing many attacks such as defeating two factor authentication of mobile banking applications, capturing real-time voice calls and leaking sensitive information. More importantly, malware evolves and distributes especially rapidly. Mobile malware attacks increased more than three times between 2015 and 2016 [64]. Furthermore, new techniques proposed in the academia have been quickly adopted by malware authors [86]. As a consequence, malware detection methods are needed for stopping the proliferation of malware in Android markets and mobile devices.

Detecting malware on mobile devices presents additional challenges compared to desktop/laptop computers. The limited battery life of mobile devices makes it infeasible to use traditional approaches requiring constant scanning and complicated computation. Therefore, most mobile malware detection methods rely on pre-defined features and build detection models to detect known malware or zero-day malware from unknown applications. This strategy, however, has several drawbacks.

First of all, most existing detection methods (e.g., Kirin [40] and DroidMat [117]) are designed to detect malware according to required resources, such as permission, API calls, and system calls. These methods, however, ignore the communications among components [7] within or cross application boundaries. As a consequence, these methods perform poorly when identifying many typical malware which requires few or no suspicious resources.

In addition, most malware detection methods rely on single classifiers to detect as many malware as possible in one round. Although these methods achieve acceptable detection performance, they require complicated and time-consuming analysis of all applications without distinguishing application complexities in the detection process. For example, even for applications that can easily classified, DroidMiner [123] builds behavior graphs through relatively expensive analysis and further mines modalities from call graphs. Furthermore, single-classifier based detection systems may not provide reliable prediction results for certain highly sophisticated applications at classification time [104].

Furthermore, the effectiveness and practicality of most learning-based malware detection methods are constrained by laborious human feature engineering and complicated feature extraction which are conducted in static analysis and/or dynamic analysis such as those performed in Soot [107], FlowDroid [11], Epicc [82] and TaintDroid [39]. As Android framework evolves over years, so do benign applications and malware. As a result, it becomes increasingly difficult for domain experts to identify the features for malware detection and assimilate the fast evolving knowledge about Android system and malware detection. If the human engineered features cannot catch up with the evolution of malware, malware detection methods may be evaded.

Moreover, although bytecode semantics are useful for detecting stealthy and advanced malware, it is difficult to comprehensively capture bytecode semantics by only looking at the presence or absence of certain API calls or permissions without considering bytecode contexts as in most existing works (e.g., Drebin [10], DroidMat [117] and StormDroid [26]). While some recent works (e.g., DroidMiner [123] and DroidSIFT [127]) construct API call graphs to capture bytecode semantics at application level, they fail to capture bytecode semantics at method level.

Last but not least, most learning-based detection systems that are built through training on older malware often make poor and ambiguous decisions when faced with modern malware (commonly known as *concept drift*). As a result, most existing detection systems (e.g., DroidAPIMiner [2], Drebin [10], and ICCDetector [120]) require periodically retrain the model. However, if the model is retrained too frequently, there will be little novelty in the information obtained from new applications. On the other hand, the retraining process requires manual labeling of all the processed applications, which is constrained by available resources.

In this dissertation, we systematically investigate malicious behaviors and patterns from different perspectives and propose effective detection systems with the assistance of state-of-the-art technologies. We first target on Inter-Component Communication (ICC) patterns and design ICCDetector, which detects malware based on not required resources but ICC patterns. Next, we propose DeepRefiner, a novel system to detect malware both efficiently and effectively. DeepRefiner applies deep neural networks to automatically learn detection features from applications and removes complex feature extraction from the detection process. Last but not least, we propose DroidEvolver to detect malware over time in a both effective and scalable way. With the ability of self-evolving, DroidEvolver automatically determine when to update and how to update itself without expert's instructions and without requiring true labels of new applications. The details of these works are presented as follows.

1.2 ICC-based Malware Detection System

In this dissertation, we first investigate in design a detection system to detect malware according to their ICC patterns. Instead of being independent to each other, Android applications may communicate through the Inter-Component Communication (ICC) mechanism provided by Android, which is designed to reduce the developers' burden and promote functionality reuse [7]. Although ICC facilitates inter-application collaboration, it can be exploited by malware to obfuscate malicious behaviors and bypass existing detection methods.

As a pioneer to address such challenge, we systemically analyze ICC patterns of benign applications and malware, and propose ICCDetector to detect malware according to their ICC patterns. The ICC patterns of an application represent how it use the ICC mechanism, and can be extracted from applications' APK files. ICCDetector is trained with the ICC patterns extracted from known benign applications and malware, and relies on single classifier to identify malware from unknown applications. The experimental results demonstrate the effectiveness of ICCDetector with an accuracy of 97.4% and a false positive rate of 0.67%.

1.3 Multi-layer Malware Detection System

As malicious behaviors vary significantly across mobile malware, it is challenging to detect malware both efficiently and effectively. Also due to the continuous evolution of malicious behaviors, it is difficult to extract features by laborious human feature engineering and keep up with the speed of malware evolution. To solve these challenges, we propose DeepRefiner to identify malware both efficiently and effectively. The novel technique enabling effectiveness is the semantic-based deep learning. We use Long Short Term Memory on the semantic structure of Android bytecode, avoiding missing the details of method-level bytecode semantics. To achieve efficiency, we apply Multilayer Perceptron on the xml files based on the finding that most malware can be efficiently identified using information only from xml files.

We evaluate the detection performance of DeepRefiner with 62,915 malicious applications and 47,525 benign applications, showing that DeepRefiner effectively detects malware with an accuracy of 97.74% and a false positive rate of 2.54%. We compare DeepRefiner with a state-of-the-art single classifier-based detection system, StormDroid, and ten widely used signature-based anti-virus scanners. The experimental results show that DeepRefiner significantly outperforms StormDroid and anti-virus scanners. In addition, we evaluate the robustness of DeepRefiner against typical obfuscation techniques and adversarial samples. The experimental results demonstrate that DeepRefiner is robust in detecting obfuscated malicious applications.

1.4 Self-evolving and Scalable Malware Detection System

Given the frequent changes in the Android framework and the continuous evolution of malware, it becomes increasingly difficult to build detection systems that are trained with older applications (both benign and malicious) and make outstanding performance when faced with modern applications after operating for long periods of time. Rapid-aging detection system is a huge concern in both industry and academia. As in research area, most existing detection systems need to periodically retrain their detection models with labeled applications. However, the retraining process is time-consuming and requires manual labeling of all the processed applications, which is constrained by available resources.

While facing these challenges, we propose a novel malware detection system for Android that relies on the ability of self-evolving and real-time update to achieve both effective and scalable detection over time. The proposed system, DroidEvolver, automatically and efficiently updates itself to adapt to new changes discovered from both Android framework and new applications without requiring true labels of applications.

We evaluate the detection performance of DroidEvolver with 34,722 benign applications and 33,294 malicious applications, which have fairly balanced ratio between benign applications and malicious applications from 2011 to 2016. We first evaluate how well DroidEvolver performs by training and testing with applications developed in same time period. As shown in the result, DroidEvolver's accuracy varies from 95.19% to 96.18% in different time periods. We then evaluate the over time detection performance of DroidEvolver. When DroidEvolver is evaluated on testing sets that are newer than training sets by one to five years, the average accuracy of DroidEvolver is 90.52%, 85.95%, 84.00%, 85.2% and 86.87%, respectively. In addition, DroidEvolver declines by 1.68% in terms of detection accuracy per year on average over five years. Finally, we evaluate the robustness of DroidEvolver against typical obfuscation techniques, showing that DroidEvolver is robust in detecting obfuscated malware.

1.5 Organization

The reminder of this dissertation is organized as follows: Chapter 2 is a literature review which examines closely related research. Chapter 3 presents details of ICCDetector as a ICC-based malware detection syste. Chapter 4 describes Deep-Refiner as a multi-layer malware detection system applying deep neural network. Chapter 5 describes DroidEvolver as an effective and scalable detection system to identify malware both effectively and scalably. Finally, Chapter 6 summarizes the contributions of this dissertation and points the future direction.

Chapter 2

Literature Review

In this chapter, we introduce related studies on detecting malware for Android platform. We also present studies focusing on Android platform security defense in Section 2.1 and application analysis techniques focusing on generating security properties of Android applications in Section 2.2.

2.1 Android Platform Security Defense

The Android platform provides several security measures that harden the attacks targeting Android, most notably the Android permission system. To perform certain tasks on the device, each application has to explicitly request permission from user during the installation. However, many users tend to blindly grant permissions to unknown applications. As a consequence, malware may conduct multiple attacks (e.g., confused deputy attack and collusion attack) by manipulating other applications and the Android system as observed by [34] [40] [43] [101]. Existing studies have developed several security extensions to defend against specific types of attacks.

Quire [36] designs a provenance system to prevent confused deputy attacks. As a further extension of Quire, Bugiel et al. [18] propose a security framework with pre-defined security policies to prevent both confused deputy attacks and collusion attacks. Saint [84], Porscha [83], and CRePE [28] achieve application isolation and protection with the assistance of context-related policies and policy-oriented access control mechanism. AppFence [59] retrofits the Android OS to impose privacy controls on installed applications. SEAndroid [102] achieves flexible manadatory access control to Android platform by enabling the effective use of SELinux [103] and by developing a set of middleware extensions to the Android permission system. Aurasium [121] protects the Android platform by repackaging applications to hook system APIs and enforcing practical policies.

Checking at install time, Kirin [41] performs lightweight certification of applications by checking whether the permissions required by applications break certain pre-defined security rules. Apex [81] allows users to selectively grant permissions to applications as well as impose constraints on the usage of resources using a policy enforcement framework.

Virtualization approaches have been used to enhance the security of Android platform. AirBag [116] boosts defense capability against malware infection using a lightweight OS-level virtualization approach. Paranoid Android [91] design a scalable security architecture that is able to apply multiple security checks simultaneously in virtual environments. L4Android [70] encapsulates the original OS in a virtual machine and ensures the isolation between the virtual machine and secure applications. Cells [8] enables multiple virtual smartphones to run simultaneously on the same physical device in a securely isolated manner.

2.2 Android Application Security Analysis

In this section, we present related studies that focus on analyzing the security mechanism of Android applications. Previous works in this direction have used both static analysis and dynamic analysis. With the former, the application's code is decompiled in order to extract features without actually running the application. The latter involves real-time execution of the applications, typically in an emulator or protected environment.

2.2.1 Static Analysis

Felt et al. [43] analyze API calls to identify over-prvileged applications. CHEX [75] automatically identifies component hijacking vulnerabilities from application by analyzing data flows. FlowDroid [11] provides highly precise static taint analysis for applications. It proposes a precise model of Android's lifecycle which allows the analysis to properly handle callbacks invoked by the Android framework. DroidChecker [24] uses inter-procedural Control Flow Graph (CFG) searching and static taint checking to detect exploitable data paths in applications. Epicc [82] and IccTA [72] analyze properties of messages sent between components of applications to detect privacy leakages. Klieber et al. [68] propose a taint analyzer by combining FlowDroid and Epicc to analyze both inter-component and intra-component dataflow. ScanDal [66] detects privacy leakages in applications.

Some static analysis works focus on detecting hidden activities by analyzing triggers and behaviors of applications. AppIntent [124] identifies user-intended sensitive data transmissions by analyzing a sequence of GUI manipulations corresponding to the data transmission. TriggerScope [45] introduces trigger analysis to detect logic bombs in Android applications. Similarly, HSOMINER [86] enables a large scale discovery of unknown hidden sensitive operations using machine learning techniques. Both TriggerScope and HSOMINER analyze triggers and conditions of potential harmful activities.

2.2.2 Dynamic Analysis

DroidScope [122] is a virtualization-based taint analysis system. It includes several modules to perform dynamic taint analysis at different levels. TaintDroid [39] monitors how third-party applications access or manipulate users' personal data, aiming to detect sensitive data leakages. Unfortunately, many data-flow analysis systems, such as DroidScope and TaintDroid, are designed for Dalvik environment. This makes data-flow analysis of new applications and malware infeasible since Android has replaced Dalvik environment with a new design known as Android Run Time (ART) in Android 5.0. To better analyzing new applications and malware, TaintArt [105] is proposed to dynamically track information flows on the Android OS with ahead-of-time compilation strategy. Specially, TaintART is a multi-level information flow tracking system for the new Android system implementing Android Run Time (ART).

There are several dynamic analysis systems aiming at understanding malicious behaviors. DroidChameleon [94] demonstrates the vulnerabilities of existing anti-malware tools against transformation attacks. Although transformation attacks are simple in most cases, anti-malware tools/systems make little effort to provide transformation-resilient detection. CopperDroid [106] provides an automatic VMI-based dynamic analysis system to reconstruct the behaviors of Android malware. AppsPlayground [93] integrates multiple components comprising different detection and automatic exploration techniques to provide dynamic security analysis.

2.2.3 Hybrid Analysis

Recently, more and more studies combine static analysis and dynamic analysis together as hybrid analysis to analyze Android applications. IntelliDroid [115] uses static analysis to instruct dynamic analysis in order to trigger specific behaviors during execution. Similar as IntelliDroid, DyTa [47] consists of a static phase and a dynamic phase. The static phase detects potential defects with a static checker; the dynamic phase generates test inputs through dynamic symbolic execution to confirm these potent defects. AppAudit [118] relies on the synergy of static and dynamic analysis to provide effective real-time application auditing, which helps to reveal privacy leakages. In summary, hybrid analysis reduces the number of false positives compared to purely static analysis, and performs more efficiently compared to dynamic analysis.

2.3 Android Malware Detection

Over the past few years, Android malware detection has attracted extensive attentions in both academia and industry.

A number of techniques use signatures for Android malware detection. NetworkProfiler [33] generates network profiles for applications and extracts fingerprints based on such traces, while Canfora et al. [21] obtain resource-based metrics (i.e., CPU, memory, storage, and network) to distinguish malware activity from benign one. It is challenging for these signature-based methods to effectively detect zero-day malware or newly designed attacks.

Most recent malware detection systems are learning-based systems, which rely on single classifier trained with human engineered features. A list of examples is given below. MAMADROID [76] builds a behavioral from the sequence of abstracted API calls performed by an application, and uses it to extract features and perform classification. DroidMiner [123] extracts sensitive API call graphs as detection features, while DroidAPIMiner [2] extracts relevant features at API level, and builds a detection model using the generated feature set. Gascon et al. [46] generate function call graphs for Android applications, and build a detection model relying on embedded function call graphs. DroidMat [117], StormDroid [26] and Drebin [10] use not only sensitive API calls but also other information extracted from AndroidManifest.xml file as detection features and train single classifiers afterwards. ICCDetector [120] builds a malware classifier according to ICC-related information included in applications. DRACO [13] and MARVIN [74] utilize static analysis and dynamic analysis to extract features from pre-determined feature categories, and train a detection model afterwards. MASK [23] statically analyzes attributes (including permssions, intent filters,and presence of native code) extracted from applications, and trains a detection model for detecting malware. Finally, Crowdroid [20] relies on crowdsourcing to get system calls from real users, and creates an anomaly model according to system call vector clusters.

A small portion of existing works integrate several classifiers to detect malware in different manners. Smutz and Stavrou [104] apply an ensemble classifier consisting of many basic classifiers. They perform a diversity analysis to detect unreliable prediction results, and retrain their classifiers with unreliable observations so as to improve classifier accuracy. Specially, the basic classifiers are independent to each other in this work.

DroidSIFT [127] extracts weighted contextual API dependency graph and constructs feature sets accordingly. With graph-based feature vectors, DroidSIFT builds two classifiers, while the first classifier discovers zero-day Android malware, and the second classifier uncovers the corresponding family of detected malware.

RiskRanker [53] includes two risk analysis modules, while the first-order analysis module sifts through untrusted applications and exposes risky applications, and the second-order analysis module identifies applications with encrypted native code and dynamic code loading. .

The increasing complexity of Android malware calls for new defensive techniques that are harder to evade. To this end, some efforts are made to detect mobile malware using deep neural networks. For example, DroidDetector [126] and DroidSec [125] build Deep Belief Networks to detect Android malware relying on 192 human engineered features, including required permissions, sensitive API calls, and some dynamic behaviors obtained from DroidBox [35]. Deep4maldroid [60] extracts Linux kernel system calls and constructs the weighted directed graphs which are then used to train deep neural networks. Although these systems apply deep neural networks, they still rely on features pre-determined by domain experts, such as required permissions and sensitive API calls.

Towards automatically engineering features for malware detection, FeatureSmith [131] mines the scientific literatures written in natural language to generate features that are semantically related to malicious behaviors.

Chapter 3

ICCDetector: ICC-Based Malware Detection on Android

3.1 Introduction

Many existing malware detection methods are designed to detect malwares based on required resources, such as permissions, suspicious API calls and system calls. For example, Kirin [41] detects malwares by matching their required permissions against pre-defined security rules. DroidMiner [123] and DroidAPIMiner [2] build malware detection models based on API-related features. Most of these methods treat the detected applications as standalone entities in Android platforms.

However, in order to bypass existing detection methods, malwares move to another direction by conducting malicious operations without requiring suspicious resources. As observed by [34, 40, 43, 101], malwares may conduct multiple attacks (e.g., *Confused Deputy* attacks and *Collusion* attacks) by manipulating other apps. In fact, instead of being independent to each other, Android applications may communicate through the Inter-Component Communication (ICC) mechanism provided by Android, which is designed to reduce the developers' burden and promote functionality reuse [27]. Although ICC facilitates inter-application collaboration, it can be exploited by malwares to obfuscate malicious behaviors and bypass existing detection methods. For example, consider a malware (`com.jx.theme`) which aims to install APK files during runtime. Instead of requiring the corresponding permission (`android.permission.INSTALL_PACKAGE`), which should not be used by third-party apps, and thus can be detected by most existing detection methods, this malware generates an Explicit Intent, sends it to `Package: com.android.packageinstaller, Class: com.android.packageinstaller.PackageInstallerActivity`, and manipulates the latter to install some APK files from SD cards. It is difficult to detect such malwares from their required resources without inspecting the ICC information involved.

As a pioneer to address such challenge, we systemically analyze ICC patterns of benign apps and malwares, and propose **ICCDetector**, an effective and accurate malware detection method, which detects malwares based on not their required resources, but their ICC patterns. The ICC patterns of an app represent how it use the ICC mechanism, and can be extracted from the app's APK file. ICCDetector is trained with the ICC patterns extracted from some benign apps and those from certain malwares before it outputs a detection model. The detection model is used to detect a malware based on its ICC patterns. By looking into the ICC patterns, ICCDetector not only examines the communications between applications and Android system, but also the interactions between applications. Because of this, ICCDetector is especially useful for detecting those "advanced malwares" which invalidate most existing malware detection methods by exploiting the ICC mechanism instead of requiring suspicious resources.

We collect 5,264 recent malwares and 12,026 benign apps to evaluate the effectiveness and accuracy of ICCDetector. For comparison, we choose a highly cited malware detection method [90] as a benchmark, which detects malwares according to their required permissions. With the same dataset, the evaluation result indicates that ICCDetector achieves an accuracy of 97.4%, roughly 10% higher than the benchmark. Furthermore, ICCDetector produces a false positive rate of 0.67%, which is only about one half of the benchmark. After manually analyzing false positives, we discover 43 new malwares from the benign dataset, and reduce the number false positives to seven.

For detected malwares, ICCDetector further classifies them into five new malware categories according to their ICC patterns. The classification clarifies the relationship between malware behaviors and ICC characteristics. In addition, we test the runtime performance of ICCDetector, identify the performance bottleneck and specify the directions for performance improvement.

The rest of the work is organized as follows. Section 3.2 describes ICC patterns of apps. Section 3.3 details the system design of ICCDetector. Section 3.4 evaluates ICCDetector in different aspects, including a comparison with a benchmark, a analysis of detection performance, a classification of malwares, and runtime measurement. Section 3.5 discusses some recent work on Android malware detection and the limitations of ICCDetector. Section 3.6 summarizes the related work, and Section 3.7 concludes the work.

3.2 ICC Patterns

In this section, we identify the ICC patterns of benign apps and malicious apps, provide systemic analysis of ICC patterns, and clarify how ICC patterns can be used to distinguish between benign apps and malicious apps.

3.2.1 App Components

An Android application consists of four types of components, *Activity, Service, Broadcast Receiver*, and *Content Provider. Activity* provides a screen with which users can interact in order to do something. All visible portions of applications are *Activities. Service* can perform long-running operations in the background and does not provide a user interface. *Content Provider* manages access to a structured set of data. *Broadcast Receiver* receives information sent from multiple applications. An application must declare the names of its *Activity, Service* and *Content Provider* components in its manifest file. However, application developers are allowed to register *Broadcast Receiver* at run time to listen for specific broadcasts during a specified period of time. That is, applications can declare `Broadcast Receiver` both in manifest file and in java code.

Number of Components. In reality, malwares tend to register more *Broadcast Receivers* and less *Activities, Services* and *Content Providers* than benign apps do. A large number of *Broadcast Receivers* enables malwares to monitor system-events, such as network connectivity changes and battery changes. Although some malwares provide legitimate functionalities to end users, these functionalities are limited, which means the number of *Activities, Services* and *Content Providers* declared by malwares are relatively small. For example, without declaring any *Activity* or *Content Provider*, a malware (`com.android.update`) registers only one *Service* (`com.android.update.Updater`) to stealthily download and install malicious APK files to its fetched devices.

Name of Components. Since code reuse is common in the development of malwares, the malwares belonging to the same malware family tend to conduct similar malicious behaviors and reuse some components. For example, four malwares with different package names (`BatteryUpgrade-Tap-To-Start`, `Battery_Upgrade--Tap_to_start`, `BatteryUpgrade-Tap-ToStart-2`, `com.extend.battery`) share some malicious components to conduct similar attacks, such as `com.extend.battery.Splash`, `com.extend.battery.BatteryService` and `com.extend.battery-BootReceiver`.

Interestingly, we discover that malwares are more likely to register some components with similar confusing names so as to fool end users around or evade from detections. For example, `com.gp.geekadoo` is a malware which pretends to be a card game application in markets, and is capable of gaining super user privileges, rewriting system files, and connecting to a command and control server. Specially, `com.gp.geekadoo` includes several components with confusing names, such as `com.google.update.Dialog`, `com.google.update.UpdateService`, and `com.google.update.R-eceiver`. Without expert knowledge, end users might be confused by these names, and treat this malware as an updated version of Android or a patch of Google.

Moreover, in the Android system, when multiple *Activities* match a single Implicit Intent, the user of the system will be prompted to choose which component should receive and respond to the Intent [7]. With confusing names, the user may be tricked to choose malicious applications.

3.2.2 Intents

In the ICC mechanism, Intents are used to link components, and can be sent between *Activities, Services*, and *Broadcast Receivers*. The major functionality of Intents is to start *Activities*, start and stop *Services*, and deliver broadcast information to *Broadcast Receivers*.

Explicit Intents. Explicit Intents specify the components to start with by including targeted package names and class names. Typically, Explicit Intents are used to connect components within the same application and designed for internal application communications [7].

However, malwares can abuse Explicit Intents by sending them to other applications (i.e., external components). For example, a benign application makes its components exposed in order to receive system-generated Intents. In this case, a malware can directly send Explicit Intents to these exposed components. Without strict action check and appropriate permission protections, these benign components will be directly launched and manipulated by malwares.

In order to avoid being detected by traditional malware detection approaches which can capture suspicious permission usages and API calls, malwares find an effective solution by including or dynamically installing additional APK files. These newly installed APK files are responsible to conduct actual malicious actions. In this case, the original malwares will not be detected since their malicious behaviors cannot be captured by monitoring permissions and API calls. However, these malwares may communicate with the dynamically installed malwares using Explicit Intents, which can be inferred from their ICC patterns.

In addition, malwares tend to send more Explicit Intents to Android system than benign applications do. In our experiments, we discover that many malwares send Explicit Intents to an Android component, `Package: com.android.packageinstaller, Class: com.android.packageinstaller.PackageInstallerActivity`, which is responsible to install APK files saved on SD cards. However, none of the 12,026 popular benign applications which we collected from GooglePlay create such Explicit Intents.

Implicit Intents. Unlike Explicit Intents, Implicit Intents do not name any specific components, but instead declare general actions to perform. When an application creates an Implicit Intent, the Android system finds the appropriate component to start by comparing the contents (i.e., action, category, and data) of the Intent to the declared Intent Filters. If the Intent matches an Intent Filter, the system starts that component and delivers it the Implicit Intent object. There is a variety of system Intent actions and categories defined in the `Intent` class, and applications can define their own actions using their package names as prefixes. This results in two types of actions in Android: `system action` (prefix: `android.-` or `com.android.-`) and `user-defined action` (prefix: `package_name.-`). In particular, it is unusual and suspicious for an application generating Implicit Intent containing actions defined by other applications.

Different from the standard process, malwares may send malicious Implicit Intents to the exposed components of benign applications. These components are exposed to receive system-generated Intents or internal Implicit Intents (which is not recommended by Android due to security consideration). Without necessary action check or permission protections, the exposed components may be launched and manipulated by malwares via Implicit Intents.

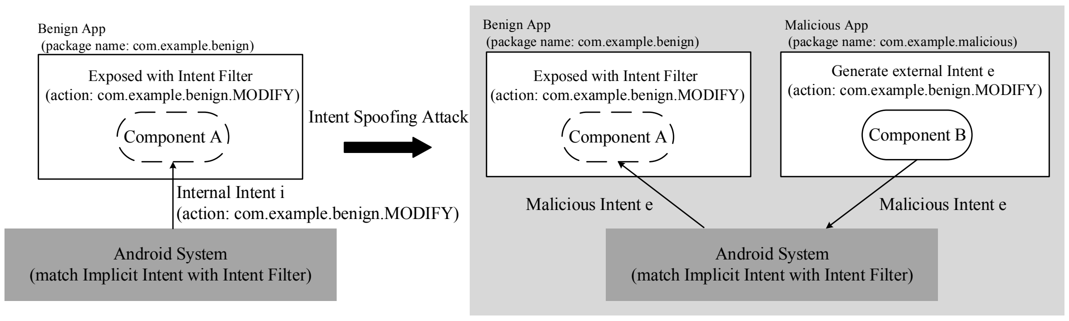

Benign App
(package name: com.example.benign)
Exposed with Intent Filter
(action: com.example.benign.MODIFY)
Component A
Internal Intent i
(action: com.example.benign.MODIFY)
Android System
(match Implicit Intent with Intent Filter)
Intent Spoofing Attack
Benign App
(package name: com.example.benign)
Exposed with Intent Filter
(action: com.example.benign.MODIFY)
Component A
Malicious App
(package name: com.example.malicious)
Generate external Intent e
(action: com.example.benign.MODIFY)
Component B
Malicious Intent e
Malicious Intent e
Android System
(match Implicit Intent with Intent Filter)

Figure 3.1: Example of *Intent Spoofing* Attack

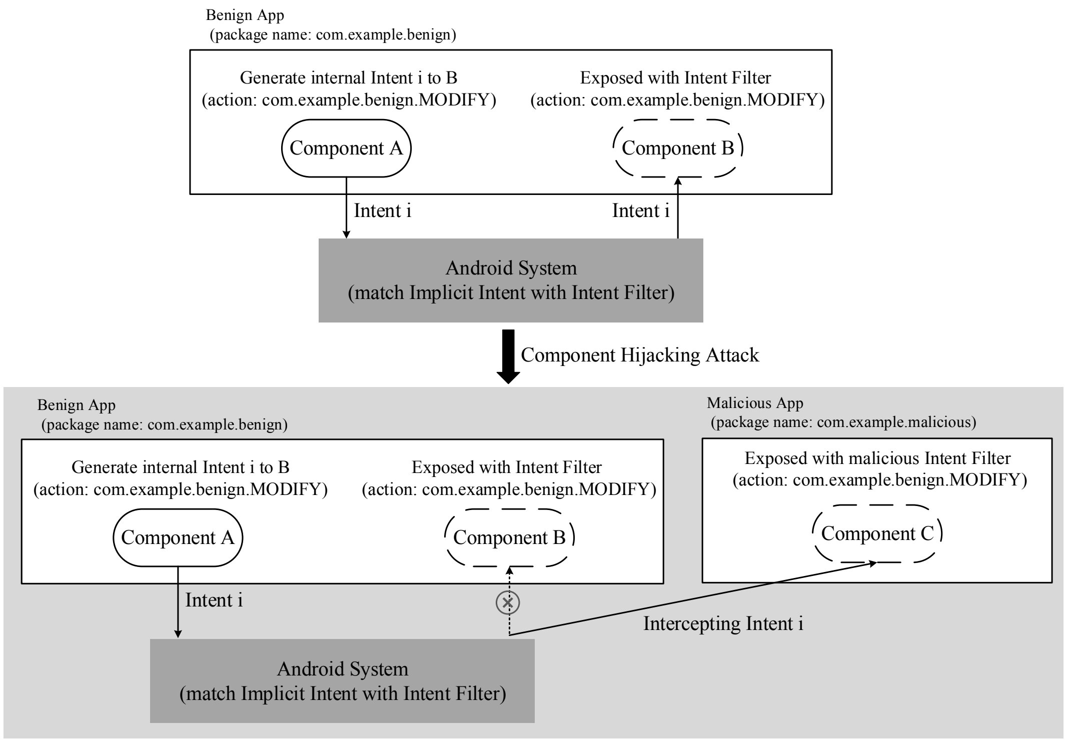
For example, a malware may launch an *Intent Spoofing* attack [27] by misusing Implicit Intents as shown in Figure 3.1. `Component A` is exposed to receive internal Intents and perform an action (`com.example.benign.MODIFY`). Misusing the Implicit Intent mechanism, malicious `Component B` may trick `Component A` to receive an external Intent (i.e., malicious Intent `e`) instead of the internal Intent (i.e., benign Intent `i`), and perform `com.example.benign.MODIFY` accordingly.

Although it is challenging for normal applications to get the knowledge of other applications' exposed components, it is relatively easy for well-prepared or colluded malwares to attain the knowledge. Malware authors may analyze popular applications and exploit the ICC vulnerabilities in designing their malwares. Also, the same malware authors may develop multiple malwares, which make use of each other's exposed components to perform *Collusion* attacks.

Although Android provides permission checks for sending out Implicit Intents with sensitive action strings (e.g., `android.intent.action.CALL`, and `android.intent.action.REBOOT`) and recommends developers to protect their exposed components (especially *Services*) with permissions, this is not an effective way to prevent Implicit Intents from being misused.

3.2.3 Intent Filters

Intent Filters are used to match with Implicit Intents in Android system. The use of Intent Filters in malwares is very differently from their use in benign applications.



Benign App
(package name: com.example.benign)
Generate internal Intent i to B
(action: com.example.benign.MODIFY)
Exposed with Intent Filter
(action: com.example.benign.MODIFY)
Component A
Component B
Intent i
Intent i
Android System
(match Implicit Intent with Intent Filter)
Component Hijacking Attack
Benign App
(package name: com.example.benign)
Malicious App
(package name: com.example.malicious)
Generate internal Intent i to B
(action: com.example.benign.MODIFY)
Exposed with Intent Filter
(action: com.example.benign.MODIFY)
Exposed with malicious Intent Filter
(action: com.example.benign.MODIFY)
Component A
Component B
Component C
Intent i
Intercepting Intent i
Android System
(match Implicit Intent with Intent Filter)

Figure 3.2: Example of *Component Hijacking* Attack

Intercepting Implicit Intents. Malwares may intercept Implicit Intents with Intent Filters in *Component Hijacking* attacks [27], where malicious components are launched in place of the expected benign components. Malwares may also intercept Implicit Intents to read the data included in the Intents, connect to certain applications, or even inject false information into the response returned. As illustrated in Figure 3.2, malwares may register appropriate Intent Filters so as to intercept external Intents generated by benign apps.

Intent Filters with Sensitive Actions. Compared with benign applications, malwares especially care about the system-wide events (i.e., system broadcast information). Some of system broadcasts can only be sent by Android system, but can be received by any components with appropriate Intent Filters. Malwares tend to register more Intent Filters for broadcast information related to

Table 3.1: Percentage of Benign Apps and Malicious Apps Registering Intent Filters with Sensitive Actions

Sensitive Actions	**Benign (12,026)**	**Malicious (5,264)**
`android.intent.action.BOOT_COMPLETED`	11.9%	60.1 %
`android.intent.action.PACKAGE_ADDED`	3.2%	16.5%
`android.intent.action.POWER_CONNECTED`	2.1%	5.5%
`android.intent.action.SMS_RECEIVED`	1.6%	33.5%
`android.intent.action.PHONE_STATE`	1.2%	10.4%
`android.intent.action.SIG_STR`	0%	12.2%

phone states, such as `android.intent.action.BOOT_COMPLETED`, and `android.intent.action.SMS_RECEIVED`. Some of the Intent Filters that are often misused by malwares are given in Table 3.1, from which we observe that the percentage of malwares registering such Intent Filters is significantly different from the percentage of benign applications registering such Intent Filters. For example, none of the benign application in the benign dataset (including 12,026 benign applications) register Intent Filter to receive broadcast information related to changes of signal strength (i.e., `android.intent.action.SIG_STR`), while 12.2% of malwares intend to intercept such event.

Number of Intent Filters. Malwares may conduct *Component Hijacking* attacks by exposing their malicious components with Intent Filters. Therefore, it is more likely for malwares to register Intent Filters for their *Activities* and *Services*. In our experiments, 29.32% of malwares declare Intent Filters for *Services*, while only 7.0% of benign apps make *Services* exposed. Furthermore, it is common for malwares to register more Intent Filters, which allows malwares to reliably launch malicious components or payloads.

Registration Mode of Intent Filters. The Registration mode of Intent Filters can serve as an indicator to differentiate between benign apps and malicious apps. The registration of Intent Filters for *Activities* and *Services* must be recorded in the manifest file, while the registration of Intent Filters for *Broadcast Receivers* is flexible, which can be static and dynamic. Android enforces dynamic registration of Intent Filters to keep applications informed with system changes during runtime. However, the dynamic Intent Filters make it possible for malwares to capture specific events in runtime and make necessary responses as required to perform malicious operations.

In our experiments, it is extremely common that malwares dynamically register Intent Filters to capture sensitive broadcasts, such as `android.intent.action.BOOT_COMPLETED`, `android.intent.action.BATTERY_CHANGED` and `android.intent.action.PACKAGE_ADDED`.

Table 3.2: ICC Patterns of Benign Apps and Malicious Apps

ICC Patterns	**Benign App**	**Malicious App**
Component	Sufficient *Activities, Services, Providers*, and few *Receivers*	Few *Activities, Services, Providers*, and sufficient *Receivers*
Explicit Intent	Send to internal components	Send to internal components and external components
Implicit Intent	Send to internal components, and external components with `system action`	Send to internal components, external components with `system action` and `user defined action`
Intent Filter	Receive internal Implicit Intents, and few system-wide broadcasts	Receive internal Implicit Intents, various system-wide broadcasts and external Implicit Intents

Table 3.2 summarizes the ICC patterns of benign apps and malicious apps. Benign applications use the ICC mechanism mainly for linking internal components and communicating with the Android system. However, malwares usually manipulate the ICC mechanism for monitoring system events, and creating Intents and Intent Filters to interact with external components.

3.3 System Design

ICCDetector consists of two phases, including Training Phase and Detection Phase as shown in Figure 3.3. In the training phase, ICCDetector extracts ICC-related fea-

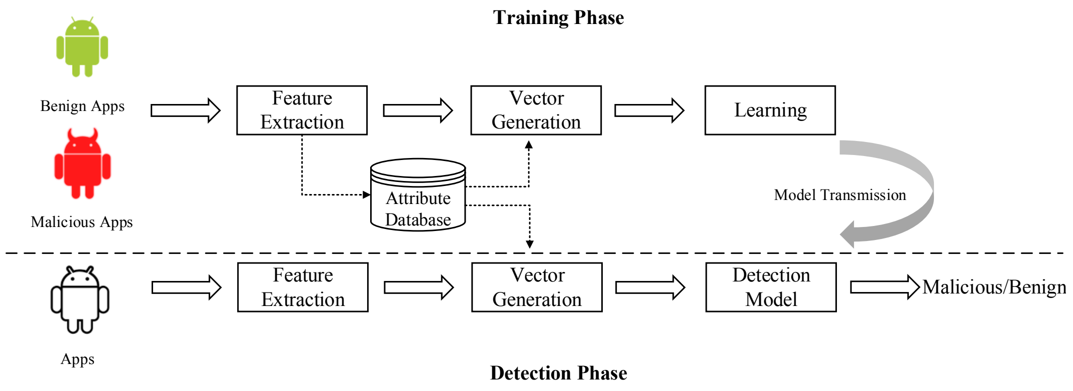


Figure 3.3: ICCDetector System Architecture

tures by analyzing the ICC sources and sinks of certain benign apps and malwares, and generates feature vector for every processed app. A classification method is used to take its input from the generated feature vectors of benign apps and malwares, and outputs a detection model. This detection model can be used to differentiate between benign apps and malwares, and it is transmitted to the detection phase. In the detection phase, ICCDetector generates a feature vector for each app being detected and feeds the feature vector into the detection model, which outputs whether the detected app is benign or malicious.

3.3.1 Training Phase

Feature Extraction. In the first step of the training phase, ICCDetector extracts all of the ICC-related features from a given app. To achieve this, we develop a tool named Parser on top of any ICC analysis tool which outputs all ICC sources and sinks from the app's APK file. Examples of such ICC analysis tools include ComDroid [27], Amandroid [113] and EPICC [82]. We choose EPICC for Parser in this work. Parser defines various categories of ICC-related features, and formats of these features. Given an app's APK file, Parser extracts the ICC-related features for each category, and represents the extracted features in corresponding formats. ICC-related features are defined in the following four categories:

Components

Given an application, Parser extracts the names and types of its components. For *Broadcast Receivers*, Parser also records the registration modes (i.e., static or dynamic). After that, Parser represents the ICC-related featuresin the following format: `component_name(activity/service/provider)`, `component_name(receiver_static)`, `component_name(receiver_dynamic)`, `num_of_activity/service/provider`, `num_of_receiver(static)`, `num_of_receiver(dynamic)`. For example, if an app dynamically registers a `BroadcastReceiver`: `com.bwx.bequick.receivers.AirplaneModeReceiver`, then Parser extracts an ICC-related feature `com.bwx.bequick.receivers.AirplaneModeReceiver(receiver_dynamic)` from this app.

Explicit Intents

In this category, Parser records the total number of generated Explicit Intents and the number of external Explicit Intents. As explained in Section 3.2, it is important to check the Explicit Intents' targets, including internal components and external components. Parser labels an Explicit Intent as `external` if it is sent to another app (i.e., the targeted package name is not included in the sender's APK file). The ICC-related features in this category are represented as `num_of_explicitintent`, `num_of_external_explicitintent`, `external_package_name(external_explicitintent)`. For example, the package name of an app is `com.jx.theme`, which sends out an Explicit Intent to `package:com.android.packageinstaller`, `class: com.android.packageinstaller.PackageInstallerActivity`, then Parser retrieves an ICC-related feature as `com.android.packageinstaller(external_explicitintent)`. Note that, Parser does not record the package names of internal Explicit Intents. Since internal Explicit Intents are designed for intra-application communications, they are not very useful for detecting malwares.

Implicit Intents

For each Implicit Intent of an app, Parser matches it with the Intent Filters retrieved from the same app following the process defined by Android. If a match exists, Parser labels the Implicit Intent as `internal`, which is used to connect components within the same app. Otherwise, Parser regards this Implicit Intent as `external`, and checks its action field. If the action is defined by other apps (i.e., the action is `user-defined` and its prefix is different from the sender's package name), Parser labels it as `external_userdefined_action`; otherwise, Parser labels it as `external_system_action`.

For each Implicit Intents, Parser retrieves its action string and identifies the potential target, and outputs the following ICC-related features: `num_of_implicitintent`, `num_of_internal_implicitintent`, `num_of_external_implicitintent(userdefined_action)`, `num_of_external_implicitintent(system_action)`, `action_string(nternal)`, `action_string(external_userdefined_action)`, and `action_string(external_system_action)`. For instance, a malware (package name: `net.mujee.www`) generates an Implicit Intent with `android.intent.action.DIAL`. However, none of its Intent Filter is registered to receive this Intent. Parser retrieves an ICC-related feature `android.intent.action.DIAl(external_system_action)` in this case.

Intent Filters

Similar to Implicit Intents, Intent Filters are represented with the included action strings. In addition, Parser records the types of components which Intent Filters are registered for. Especially, if an Intent Filter is registered for a *Broadcast Receiver*, Parser checks whether the registration is dynamic or static. The ICC-related features in this category include `action_string(for_activity/service)`, `action_string(for_receiver_static)`, `action_string(for_receiver_dynamic)`, `num_of_intentfilter_for_activity/service`, `num_of_intentfilter_for_receiver(static)`, `num_of_intentfilter_for_receiver(dynamic)`, and `num_of_total_intentfilter`. For example, given an application which dynamically registers two Intent Filters with the same action string (`android.provider.Telephony.SMS_RECEIVED`), Parser extracts an ICC-related feature `android.provider.Telephony.SMS_RECEIVED(for_receiver_dynamic)`, and sets its corresponding value to two.

Table 3.3 summarizes the formats of ICC-related features. For all the ICC-related features retrieved from benign apps and malwares by Parser, ICCDetector stores them separately in the *Attribute Database*. In addition, ICCDetector stores the output of Parser for each analyzed application, which includes the extracted ICC-related features and the corresponding values.

Feature Vector Generation. In the training phase, ICCDetector leverages any two-class classification method (e.g., SVM [19], Decision Tree [99] and Random Forest [73]) to learn the ICC patterns from benign apps and malwares, respectively. In particular, ICCDetector treats each of the extracted ICC-related features as a detection feature. Therefore, the number of detection features is equal to the size of *Attribute Database*. If the size of *Attribute Database* is X, ICCDetector defines an X-dimensional vector space. For each app, ICCDetector constructs a feature vector by mapping its Parser output to the X-dimensional vector space as shown in Figure 3.4.

Usually, a typical Android app generates roughly 100 none-zero ICC-related features; therefore, its feature vector is sparse. ICCDetector represents the feature vectors sparsely using hash tables [16]. In comparison, existing works [123] [10] use Boolean expression to represent feature vectors, which indicates whether an

Table 3.3: Summary of ICC-related Features

Feature Category	Formats of ICC-related Features
Component	`component_name(activity/service/provider)`
	`component_name(receiver_static)`
	`component_name(receiver_dynamic)`
	`num_of_activity/service/provider`
	`num_of_receiver(static)`
	`num_of_receiver(dynamic)`
Explicit Intent	`num_of_explicitintent`
	`num_of_external_explicitintent`
	`external_package_name(external_explicit)intent`
Implicit Intent	`num_of_implicitintent`
	`num_of_internal_implicitintent`
	`num_of_external_implicitintent(userdefined_action)`
	`num_of_external_implicitintent(system_action)`
	`action_string(internal)`
	`action_string(external_system_action)`
	`action_string(external_userdefined_action)`
Intent Filter	`num_of_intentfilter`
	`num_of_intentfilter_for_activity/service`
	`num_of_intentfilter_for_receiver(static)`
	`action_string(for_receiver_dynamic)`
	`num_of_intentfilter_for_receiver(dynamic)`
	`action_string(for_activity/service)`
	`action_string(for_receiver_static)`

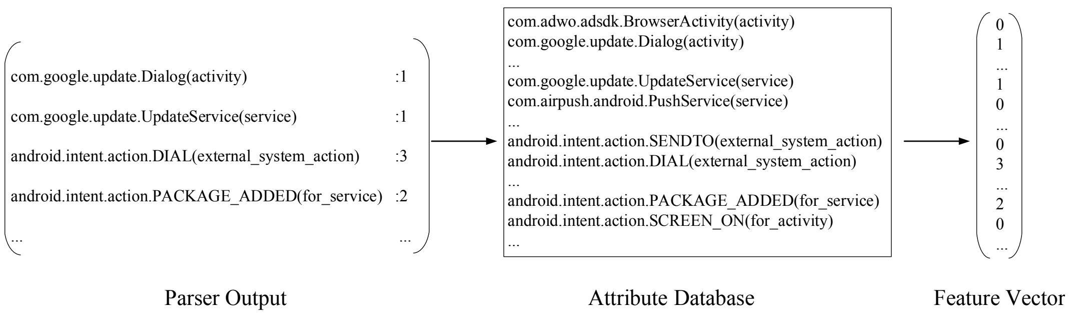

com.google.update.Dialog(activity) :1
com.google.update.UpdateService(service) :1
android.intent.action.DIAL(external_system_action) :3
android.intent.action.PACKAGE_ADDED(for_service) :2
... ...
com.adwo.adsdk.BrowserActivity(activity)
com.google.update.Dialog(activity)
...
com.google.update.UpdateService(service)
com.airpush.android.PushService(service)
...
android.intent.action.SENDTO(external_system_action)
android.intent.action.DIAL(external_system_action)
...
android.intent.action.PACKAGE_ADDED(for_service)
android.intent.action.SCREEN_ON(for_activity)
...
0
1
...
1
0
...
0
3
...
2
0
...
Parser Output
Attribute Database
Feature Vector

Figure 3.4: Process of Feature Vector Generation

app includes a feature or not. Since ICCDetector records the exact value for each ICC-related feature in generating feature vectors, ICCDetector has more accurate information to differentiate between benign apps and malicious apps.

Learning. As described in **Feature Extraction** and **Feature Vector Generation**, ICCDetector extracts ICC-related features as many as possible from given apps and constructs feature vectors by mapping Parser out to the vector space. Some of these extracted ICC-related features, however, are correlated to each other. Note that the number of extracted ICC-related features decides the dimensionality of feature vectors. If ICCDetector extracts too many ICC-related features in **Feature Extraction**, it leads to high dimensional feature vectors in **Feature Vector Generation**, which may contain a high degree of irrelevant and redundant information, and thus degrade the performance of learning algorithms. As a preprocessing step to machine learning, feature selection is effective in reducing dimensionality, removing irrelevant and redundant features, and mitigating overfitting. In this work, we apply a well-known feature selection method, Correlation-based Feature Selection (CFS) [55], in ICCDetector. CFS identifies and removes irrelevant and redundant features according to the correlation between features. After the process, CFS keeps a subset of original features, which are sufficient for the classification of Android applications. Consequently, CFS effectively reduces the dimensionality of feature vectors.

Given the input of reduced-dimension feature vectors generated from benign apps and malwares, respectively, ICCDetector applies any two-class classification method and outputs a detection model, which separates the feature vectors from benign and malicious. The detection model is then transmitted to detection phase.

3.3.2 Detection Phase

In the detection phase, ICCDetector extracts the ICC-related features from an app being detected, generated its feature vector, and feeds the feature vector to the detection model. The detection model decides whether the detected app is benign or malicious.

3.4 Evaluation

We evaluate the performance of ICCDetector in different aspects with real data, including comparison with a benchmark, analysis of detection performance, classification of detected malwares, and runtime measurement.

3.4.1 Data Collection

We built an initial dataset of 14,264 benign apps by crawling GooglePlay from July 2014 to August 2014. To exclude potential malicious apps from this initial dataset, we sent each app to VirusTotal [110], which is an antivirus service with fifty-four antivirus scanners. We labeled an app in the original dataset as benign if and only if no antivirus scanner raises any alarm for the app. We also excluded potential malicious apps such as adwares, and spywares from the initial dataset so as to generate the benign dataset, which consists of 12,026 apps. An existing malware dataset [10], which consists of 5,264 malwares is used as the ground truth in our evaluation. The collected dataset includes 12,026 benign apps and 5,264 malicious apps, which is sufficient to evaluate ICCDetector. The reason is that ICCDetector builds detection model applying SVM, which does not require large number of training samples to fine-tune model parameters.

3.4.2 Feature Selection and Analysis

From 12,026 benign apps and 5,264 malwares, ICCDetector extracts 121,621 ICC-related features in total. Since the extracted features contain redundant information due to correlation between features, we apply CFS to identify and remove the redundant features according to the correlation between features. After the process, CFS chooses 5,000 ICC-related features, which are subsequently used for the classification of malicious and benign applications.

A majority of the ICC-related features that are removed by CFS belongs to *Component*. When design names for components, app developers usually include package names in components names. For example, an application with package name `com.bwx.bequick` includes several components named as `com.bwx.bequick.EulaActivity`, `com.bwx.bequick.ShowSettingsActivity`, and `com.bwx.bequick.MainSettingsActivity`. Since it is common to include package names in component names in mobile application development, our ICCDetector extracts numerous unique ICC-related features from component names. We notice that a majority of ICC-related features in this category only appear once in single applications and are correlated to some other features. Consequently, CFS can effectively reduce the dimensionality of feature vectors after removing these redundant features.

While CFS filters out a majority of ICC-related features belonging to *Component*, it keeps the features in this category that are useful for distinguishing malwares from benign apps. For example, CFS keeps an ICC-related feature `com.allen.txtxcb.Settings(activity)`, which is extracted from an *Activity* named `com.allen.txtxcb.Setting`. This *Activity* can only be found in a malware family called DroidKungFu, and is shared by the malwares in this family. Another example is `com.android.installer.full.AndroidInstaller2Activity(activity)`, which is an ICC-related feature selected by CFS. This feature, which is extracted from an *Activity* called `com.android.installer.full.AndroidInstaller2Activity`, is shared by several Russian malwares. These malwares pretend to be legitimate package installers provided by Android, but are capable of manipulating SMS, taking pictures, and directly installing arbitrary applications. This *Activity* has a confusing name so as to fool end users and evade from detections, which is normal in malwares but rare in benign applications. It is thus helpful to keep these ICC-related features for differentiating between malwares and benign applications.

We also analyze the selected ICC-related features belonging to other categories. In general, a majority of the selected features can be used to describe the communications among components within or cross application boundaries. Due to the differences between malicious ICC patterns and benign ICC patterns, these selected features can be used to distinguish benign apps and malwares. For example, `android.intent.action.PACKAGE_CHANGED(external_system_action)` is a selected ICC-related features extracted from an external Implicit Intents `android.intent.action.PACKAGE_CHANGED`. This Implicit Intent is widely used by malwares for monitoring system events that are related to package and downloading additional APK files during runtime; however, it is barely generated by benign applications. In the Training phase, our model learns from the training dataset that this selected feature usually appears in malicious feature vectors but rarely appears in benign feature vectors. In the Detection phase, all of the selected ICC-related features are used to distinguish between malicious and benign applications.

3.4.3 Experiment Result

We compare the detection performance of ICCDetector with a benchmark, which is a highly cited Android malware detection method proposed in recent years [90], using the same dataset. The benchmark is a typical Android malware detection method, which detects malwares based on their required permissions and its accuracy is up to 88.2% using the dataset mentioned in Section 3.4.1.

In the experiments, ICCDetector leverages on a widely used two-class classification method, *Support Vector Machine (SVM)* [19], to train a detection model. *SVM* is suitable for processing multidimensional data like the feature vectors and capable of producing a model efficiently. Given the feature vectors of benign apps and malwares, *SVM* discovers the hyperplane to separate them with the maximum margin, where the margin is the sum of (i) the minimum distance between the hyperplane and the boundary of benign feature vectors, and (ii) the minimum distance between the hyperplane and the boundary of malicious feature vectors.

We conduct a series of experiments using *ten-fold cross validation* [69] to measure the performance of ICCDetector and the benchmark. In particular, we randomly split the benign dataset and the malicious dataset into ten subsets, respectively. The detection model is trained and tested in ten rounds. In each round, we mix one benign subset and one malicious subset as the testing dataset (i.e., unknown dataset), and the remaining subsets as the training dataset (i.e., known dataset). The testing dataset is tested using the classifier trained on the training dataset. In each round, there is no overlap between the testing dataset and the training dataset. Each application of the whole dataset is classified once so the accuracy of cross validation is the percentage of the applications that are correctly classified. We evaluate the performance of ICCDetector using three metrics, True Positive Rate (TPR), False Positive Rate (FPR), and Accuracy, where TPR is the percentage of malwares being detected correctly, FPR is the percentage of benign apps being detected as malwares, and Accuracy is the percentage of all apps being detected correctly in our experiments.

Table 3.4: Experiment Result

Metrics	**True Positive Rate**	**False Positive Rate**	**Accuracy**
ICCDetector	93.1%	0.67%	97.4%
Benchmark	65.0%	1.71%	88.2%

Table 3.4 shows the evaluation results of ICCDetector and the benchmark. The accuracy of the benchmark is up to 88.2%, while ICCDetector achieves an accuracy of 97.4%, roughly 10% higher than the benchmark, with a lower false positive rate of 0.67%, which is only a half of the benchmark. Through manually analyzing detected false positives, we discover that only seven benign applications are falsely

identified as malware. The true positive rate of ICCDetector is also considerably better than the benchmark, roughly 30% higher than the benchmark. More importantly, ICCDetector discovers 1,708 more "advanced malwares" than the benchmark (i.e. these malwares can only be detected by ICCDetector), while it misses 220 "obvious malwares" which can be easily detected by the benchmark.

Table 3.5: Percentage of Benign Apps and Malicious Apps Requiring Sensitive Permissions

Permission	**Benign App (12,026)**	**Malicious App (5,264)**
`android.permission.INTERNET`	92.5%	97.5%
`android.permission.ACCESS_NETWORK_STATE`	81.4%	67.4%
`android.permission.ACCESS_COARSE_LOCATION`	23.6%	32.9%
`android.permission.CAMERA`	14.2%	4.2%
`android.permission.CALL_PHONE`	11.3%	13.4%

3.4.4 True Positive Analysis

Looking into the 1,708 "advanced malwares" which are correctly detected by ICCDetector but not by the benchmark, we discover that the differences between their permission usage patterns and those of benign apps are not very significant. Table 3.5 shows the percentage of benign apps and malwares which require some sensitive permissions. Since the permission patterns are similar, it is difficult for the benchmark to distinguish between benign apps and malwares.

In comparison, the ICC patterns can be used to distinguish benign apps and malwares in such case. In general, benign apps mainly use ICC for internal communications, in a sense that Intents and Intent Filters are mainly used to link the components within the same apps. However, malwares tend to interact with external components and monitor Android system via the ICC mechanism. For example, benign apps barely register any Intent Filters for package-related information, while the malwares usually register several such Intent Filters in order to properly download APK files at runtime.

One example of the "advanced malwares" is `com.safesys.viruskiller`, which pretends to be antivirus app in markets. Without requiring any sensitive permissions, `com.safesys.viruskiller` downloads APK files by generating external Implicit Intents and monitoring the system events related to `package`, such as `android.intent.action.PACKAGE_ADDED` and `android.intent.action.PACKAGE_CHANGED`. Another example is `com.accutracking`, which is a malware intercepting private information and accessing personal files. One characteristic of `com.accutracking` is its rich variants. Although its variants are given different package names, they share the same ICC-related features and same ICC patterns, which can be easily detected by ICCDetector. These "advanced malwares" are still available in alternative markets, such as *kekaku* [65], *coolAPK* [29], *appchina* [9], and *amazon* [5].

3.4.5 False Negative Analysis

ICCDetector misses 364 malwares, while 220 of them can be easily detected by the benchmark. After manually checking how these malwares use ICC and permissions, we discover that these malwares barely use ICC. Instead of stealthily conducting malicious actions, these malwares attack in a straightforward way by simply requiring a bunch of sensitive permissions, and sometimes even permissions not for use by third-party apps. For example, with only four non-zero ICC-related features, a malware requires three permissions `android.permission.READ_LOGS`, `android.permission.INSTALL_PACKAGES`, and `android.permission.MODIFY_PHONE_STATE`, which should not be used by third-party apps. Since benign apps merely require such system-level permissions, the benchmark can easily detect such malwares based on their required permissions.

It is obvious that ICCDetector and the benchmark are complementary. A hybrid approach combining ICCDetector and the benchmark would produce better results.

3.4.6 False Positive Analysis

In the experiment, ICCDetector labels 81 benign applications as malicious (i.e., false positives). After manually analyzing these false positives, we discover that 31 of them were mislabeled by VirusTotal before. Besides them, 43 of other false positives are manually identified as malicious because they are capable of conducting malicious actions. In the end, the false positives of ICCDetector boil down to seven benign applications falsely classified as malware. We classify the false positives into three categories as follows:

Mislabeled by VirusTotal. In order to construct benign dataset, we excluded potential malwares by sending each app in the original dataset to VirusTotal, and labeled an app as benign if and only if no antivirus scanner raises any alarm for the app. However, the detection result of VirusTotal should be updated and corrected over time. After resending the 81 false positives to VirusTotal, we discover that 31 applications, which had been labeled as benign when constructing benign dataset in August 2014, received alarms from at least one antivirus scanners in May 2015. For these 31 applications, the detection results of ICCDetector and those of the updated version of VirusTotal are consistent.

Since VirusTotal has not released any technical details related to its updating process, it remains unknown that how these 31 malwares bypassed the scanning of VirusTotal in August 2014. Fortunately, these malwares can be easily detected by ICCDetector according to their ICC patterns. For example, `com.tobyyaa.superbattery` is an application which includes several malicious components, such as `com.millenialmedia.-`, `com.admob.-`, `com.flurry.-` and `com.appbrain.-`. This application is capable of manipulating SMS, and making phone calls by generatingsuspicious Intents such as `android.intent.action.DIAL`, and `android.intent.action.CALL`, and certainIntent Filters such as `android.intent.action.NEW_OUTGOING_CALL`, and `android.provider.Telephony.SMS_RECEIVED`. These suspicious ICC patterns can be captured by ICCDetector in malware detection.

New Identified Malwares. Not only can ICCDetector product consistent results with the updated version of VirusTotal, but also can identify new malwares. After manually analyzing the 81 false positives, 43 of them are identified as malicious, which have not been identified by VirusTotal before. Most of these newly identified malwares are capable of leaking private information, manipulating SMS, connecting to remote servers, and monitoring system state. For example, `com.tunewiki.lyricplayer.android.quicklaunch` is a newly identified malware discovered in GooglePlay which is designed to make phone calls, monitor and leak phone states and system settings. To achieve its goal, this application generates an Intent `android.intent.action.DIAL` to make phone call, and registers several Intent Filters to monitor any phone state and setting changes.

During the manual analysis, we discover that these newly identified malwares not only manipulate the ICC mechanism, but also abuse sensitive permissions. For instance, `com.thukhakyaw.calllocator` is a newly identified malware which may make phone calls, send out SMS, and modify system states. In particular, this malware requires several permissions which are not allowed to use by any third-party applications, such as `android.permission.MODIFY_PHONE_STATE` and `android.permission.UPDATE_DEVICE_STATS`. This discovery further demonstrates the accuracy of ICCDetector.

Benign Applications. Among the 81 false positives, seven benign applications are falsely identified as malware. After manually analyzing these benign applications, we discover that they barely use any ICC mechanism, therefore it is difficult for ICCDetector to correctly identify them.

3.4.7 Classifications

The classifications of malwares in different malware families facilitate better understanding and analyzing of malwares [10, 123]. On the other hand, some of the existing classifications have the following limitations:

- Some malware families are named by different mobile security software vendors and researchers. The naming scheme is confusing and inconsistent. For example, *BaseBridge* is also named as *AdSMS*, and *LeNa* is a variant of *DroidKungFu* [129].
- Some malwares belonging to different families have similar malicious behaviors. For example, the malwares in *Lovetrap* and *NickyBot* are similar in terms of sending premium SMSes and starting malicious services right after Android system boot-up.
- Some malware families contain too few samples. For example, *FakeInstaller* contains about 1,000 malicious samples, while *GGTracker, DroidCoupon* and *GamblerSMS* only have one malicious sample. More importantly, the majority of existing malware families (i.e., more than 200 malware families) contain less than thirty samples per family.

Motivated to overcome such limitations, we propose five new malware categories based on ICC patterns and classify detected malwares into corresponding categories. In order to conduct certain malicious operations, malwares need to use the ICC mechanism accordingly. Therefore, these newly defined malware categories are closely related to malware behaviors.

Server Connector. Malwares in this category mainly conduct malicious actions by connecting to command and control servers, dynamically downloading and installing APK files, and executing remote commands. These malwares usually register several *Broadcast Receivers* and *Services* to receive `c2dm` (Cloud to Device Messaging [52]) related Intents and to execute received commands.

Especially, in order to effectively download and install APK files from the remote servers, these malwares leverage Intent Filters to monitor events related to `package`, such as `android.intent.action.PACKAGE_CHANGED`, `android.intent.action.PACKAGE_ADDED`, and `android.intent.-action.PACKAGE_REMOVED`. Several well-known malware families, including `DroidKungFu`, `DroidRooter`, `RootSmart` and `ExploitLinuxLotor`, belong to this category.

Telephony Abuser. This category includes malwares which conduct attacks targeting at telephonic functionalities, such as making phone calls, blocking incoming phone calls and SMSes, and sending SMSes to premium numbers. To effectively manipulate telephonic functionalities, malwares need to generate corresponding Intents, such as `android.intent.action.DIAL`, and register certain Intent Filters to intercept SMS-related information and new outgoing calls, such as `android.provider.telephony.SMS_RECEIVED` and`android.intent.action.NEW_OUTGOING_CALL`. Malware families in this category include `Opfake, Dialer, MobileSpy`, and etc.

System Monitor. Malwares in this category especially care about system-wide broadcast information that is relevant to phone states and settings, such as battery state, power state, and connectivity setting. From phone states and settings, these malwares can infer whether a phone is in use or not, and pick the appropriate time to perform malicious actions without user's awareness. To achieve their malicious objectives, these malwares tend to register several Intent Filters to capture broadcasts with special actions, such as `android.settings.SIG_STR`, `android.net.conn.CONNECTIVITY_CHANGE`, `android.intent.a-ction.POWER_CONNECTED` and `android.intent.action.PHONE_STATE`. Moreover, some malwares in this category generate external Intents so as to change phone states and settings.

Effective Launcher. This category contains malwares which leverage a special system-wide Intent with action `android.intent.action.BOOT_COMPLET-`

`ED` to effectively launch their malicious *Activities* and *Services* when the Android system completes its booting process. Moreover, some malwares in this category can immediately bootstrap their *Services* before starting the host app's primary *Activity* by intercepting an Intent with action `android.intent.action.MAIN`.

Advertiser. Instead of including dangerous and sensitive Intent Filters or Intents, malwares in **Advertiser** usually include more than one advertisement libraries, which are mainly used by malwares. Such libraries include `Airpush`, `LeadBolt`, `Appenda` and `SendDroid`.

Table 3.6 summarizes the categories we defined according to ICC characteristics, which demonstrates similar malicious behaviors within each category. We also looked into the 1,708 "advanced malwares" which can be detected by ICCDetector but not the benchmark, and the 43 newly identified malwares detected from false positives. We discovered that most of them belong to **Server Connector, Telephony Abuser** and **System Monitor**, which are more dangerous than the other two categories.

3.4.8 Runtime Measurement

We ran our experiments on a machine with 4×3.20GHz Intel-Core and 12 GB of RAM, and measured the runtime of ICCDetector. In each of ten rounds in our experiments, ICCDetector is trained with 15,561 applications (i.e., 90% of datasets), and tested with a mix of 1,203 benign apps and 526 malwares. In the training phase, an ICC analysis tool, EPICC, is used to analyze the APK file of each app, which outputs all ICC sources and sinks. Our Parser is then used to extract all ICC-related features, including their names and values. After processing all the apps in training dataset, ICCDetector generates a feature vector for each processed app, and outputs a *SVM* detection model. In the detection phase, each app is processed using APK analysis, feature generation, and vector generation as in the training phase. The detection model labels an app being detected as "benign" or "malicious".

Table 3.6: Malware Categories based on ICC Patterns

Category	**Num of Apps**	**Malicious Behaviors**	**ICC Characteristics**	**Example of Malware Family**
Server Connector	1258	Connect to command and control servers, dynamically install APK files, and execute remote commands	Register Intent Filters for package-related information, register *Broadcast Receivers* for `c2dm`, and register *Services* for executing commands	DroidKungFu, DroidRooter, RootSmart, ExploitLinuxLotoor
Telephony Abuser	2270	Conduct attacks aiming at telephonic functionalities, such as making phone calls and sending SMS	Register *Services* for controlling SMS, and register Intent Filters for monitoring SMS-related information and new-outgoing calls	Opfake, Mobile-Spy, SendPay, Dialer, SMSreg
System Monitor	348	Monitor system-wide broadcasts relevant to phone state changes	Register Intent Filters for monitoring phone states, such as battery states and power states, and generate external Intents to change connectivity	AccuTrack, Anti, Gamex, SafeKid-Zone
Effective Launcher	1016	Leverage ICC mechanism to effectively launch malicious *Services* and *Activities*	Register Intent Filters and Intents with `android.intent.action.BOOT_COMPLETED` and `android.intent.action.MAIN`	Boxer, Fakelogo, Lifake, Iconeyss, Adrd
Advertiser	8	Include more than one malicious advertisement libraries	Register *Services* and *Activities* related to advertisement libraries, such as *Airpush, LeadBolt, Appenda, SendDroid.*	Opfake, Fidall, Stiniter, Kidlogger

Table 3.7: Time for Processing an App

Step	APK Analysis	Feature Extraction	Vector Generation	Model Detection
Average	36.00s	2.6×10^{-3}s	0.79s	1.2×10^{-3}s

Table 3.7 shows the average time for processing each app in our experiments. The performance bottleneck is at the APK analysis. In the future, the performance of ICCDetector would be improved with the development of more efficient ICC analysis tools.

3.5 Discussions

Besides the benchmark, ICCDetector is also compared with some recent malware detection methods, including Drebin [10], and DroidMiner [123]. Since the source codes and datasets of Drebin and DroidMiner are not open to the public, we provide our comparison qualitatively.

Drebin [10] is a lightweight malware detection method directly working on smartphones. Due to the limited resources of mobile devices, Drebin conducts a broad static analysis to gather many detection features, including permissions, APIs, network addresses, app component names, and Intent Filters extracted from manifest files. In comparison, ICCDetector extracts more ICC-related features, including the number of Intents, the names and actions of each Intent, the potential internal and external receivers of each Intent, and the Intent Filters extracted from bytecode. Therefore, ICCDetector is more accurate in capturing ICC-related features and patterns in malware detection.

On the other hand, DroidMiner [123] detects malwares based on not only the frequency and names of sensitive APIs, but also the connections of multiple sensitive APIs. Unlike ICCDetector, DroidMiner does not inspect any ICC-related functions. Therefore, it is less effective to capture the communications and interactions between components within or cross application boundaries.

Limitations. Lacking dynamic inspection of malware behaviors, ICCDetector may be bypassed by malwares using Java reflection and bytecode encryption [94]. This encourages us to incorporate dynamic analysis in future versions of ICCDetector. Another limitation of ICCDetector, which is due to the use of classification methods, is its vulnerability to mimicry and pollution attacks [108], where malwares may include more benign features and poison the training dataset to lower their suspicions.

3.6 Related Works

Mobile Malware Detection. Static malware detection methods analyze app codes and manifest files without running the apps. For instance, Kirin [41] detects malwares based on the permissions required by the Android apps which break certain pre-defined security rules. Stowaway [42] detects overprivileges in Android apps by mapping API calls to permissions. Peng et al. [90] proposed a malware detection model based on app categories and declared permissions. RiskRanker [53] captures risky apps based on known malicious behaviors and existing vulnerabilities in Android, and detects malwares from risky apps based on manual efforts. DroidMiner [123] and DroidAPIMiner [2] use sensitive API calls in detecting malwares, while DroidMat [117] and Drebin [10] use not only sensitive API calls but also other information extracted from manifest files as detection features. These previous works capture the communications between apps and Android system based on the required resources of detected apps, while ICCDetector captures not only the communications between apps and system, but also the interactions among apps based on ICC-related features.

Another class of malware detection, including TaintDroid [39], DroidScope [122], CrowDroid [20], Paranoid Android [91], and DroidRanger [130], employs dynamic analysis to detect malwares at runtime. These dynamic approaches are complementary to the static analysis based approaches, including ICCDetector.

ICC Analysis. Much work has been done on ICC analysis. For example, ComDroid [27] investigates the attack surfaces related to ICC. CHEX [75] focuses on detecting *Component Hijacking* attacks by analyzing information flows. AppSealer [128] generates vulnerability-related patches for preventing *Component Hijacking* attacks. Epicc [82] is a static analysis tool for identify ICC precisely and scalably. Amandroid [113] conducts static analysis for security vetting of Android apps based on inter-component control and data flows. Pscout [12] produces a permission specification, which is a set of mappings between API calls (including ICC APIs) and permissions. These works focus on identifying ICC-related attack surfaces for Android apps, while ICCDetector focuses on detecting malwares based on ICC-related features. The ICC analysis tools developed in these works can be applied by ICCDetector in constructing its Parser.

3.7 Conclusion

ICCDetector detects malwares based on ICC-related features which capture the interaction between components within or cross application boundaries. The performance of ICCDetector is better than the benchmark in our experiments. The malwares detected by ICCDetector are classified into five new malware categories according to their ICC characteristics, which clarifies the relationship between malware behaviors and ICC patterns. Furthermore, after manually analyzing false positives, we discover 43 new malwares from the benign dataset. In the future, we plan to apply ICCDetector to detect new malwares in various application markets. We also plan to build a dataset which can be used to evaluate and compare different malware detection methods on a common platform.

Chapter 4

DeepRefiner: Multi-Layer Android Malware Detection System Applying Deep Neural Networks

4.1 Introduction

Android malware running on most mobile devices severely violates end users' security and privacy. Most previous research in Android malware detection relies on pre-defined features and single classification models. Some existing systems provide efficient detection according to the presence or absence of certain permissions or components, but are less resilient towards obfuscation techniques. Other detection systems perform effective detection by implementing semantic analysis of Android bytecode. These approaches, however, require complicated feature extraction which are conducted in static analysis and/or dynamic analysis such as those performed in Soot [107], FlowDroid [11], Epicc [82] and TaintDroid [39].

While facing the challenges, we found that xml files have enough features to efficiently identify common malware [10] while using bytecode semantics improves the robustness of detection systems [95]. In this work, we propose an efficient and effective malware detection system, DeepRefiner, by applying deep neural networks to automatically extract features to detect malware on both xml files and bytecode.

In the first detection layer, DeepRefiner retrieves xml values by performing lightweight preprocessing on xml files, which are included in apk files of Android applications. The xml values provide detailed information about resources required by Android applications. DeepRefiner applies Multilayer Perceptron (MLP) [58] to detect malware based on xml values. The fully-connected structure of MLP is particularly suitable for processing xml values that have no sequential information among them. With the help of MLP, DeepRefiner effectively detects most malware by looking into the resources required in xml files. For certain malware that DeepRefiner cannot provide reliable prediction results, DeepRefiner generates a temporary label `uncertain`, and feeds uncertain applications into the second detection layer for a refined inspection and detection.

As shown in our experiments, with the dataset of 110,440 applications, including 62,915 malicious applications and 47,525 benign applications, DeepRefiner provides reliable and efficient classification for more than 73% of applications after processing each application for about 0.22s on average. Using the first layer only, DeepRefiner achieves an accuracy of 98.32%, with a true positive rate of 98.33% and a false positive rate of 1.70%, excluding the uncertain applications in the calculation.

In the second detection layer, DeepRefiner conducts effective detection for uncertain applications according to bytecode semantics, which provide comprehensive information about programming behaviors conducted by applications. Detecting malware according to bytecode semantics has the following advantages. First of all, bytecode semantics are different in malware and in benign applications. Malware and benign applications may use same bytecode in different context to achieve different purposes [45]. The second benefit of using bytecode semantics is that bytecode semantics are robust to typical existing obfuscation techniques. To bypass detection, malware often renames identifiers and reassembles multiple times to re-arrange classes, methods and strings. However, such obfuscation techniques affect no bytecode semantics within methods. Another advantage is that many bytecode sequences are shared across malware, since malware developers often adopt malicious libraries from some open source platforms such as Github [49] and PUDN [92].

Different from existing works (e.g., DroidMiner [123], MAMADROID [76] and DroidSIFT [127]) which extract semantics from data flows and control flows, DeepRefiner makes the effort to capture comprehensive bytecode semantics at both method-level and application-level by performing the combination of a Bytecode2Vec technique and a Long Short Term Memory neural network [56]. Using the Bytecode2Vec technique, DeepRefiner captures method-level bytecode semantics by representing each line of bytecode with a dense Bytecode Vector according to its context (i.e., nearby bytecode in the same method). Two different lines of bytecode with similar functionalities or contexts are represented with similar Bytecode Vectors, and will be treated similarly during classification. Since typical obfuscation techniques affect no bytecode semantics within methods, method-level bytecode semantics encoded in Bytecode Vectors help DeepRefiner perform robust detection. After the process of Bytecode2Vec, DeepRefiner represents each application as a variable-length Bytecode Vector Sequence by combining all Bytecode Vectors according to the original bytecode sequence in source code.

DeepRefiner further applies stacked LSTM hidden layers on top of variable-length Bytecode Vector Sequence. During the process, LSTM hidden layers are used to capture historical information about the Bytecode Vector Sequence and iteratively update it to learn the application-level bytecode semantics without losing method-level bytecode semantics due to the special structure of LSTM called memory cell. With this novel combination, DeepRefiner successfully captures the triggers of malicious behaviors even if they are separated far away from each other in the source code as shown in our experimental results.

Applying both method-level and application-level bytecode semantics to detect malware is proved to be highly effective in our experiments. For the 29,320 uncertain applications that cannot be reliably classified in the first detection layer, DeepRefiner further refines the classification results according to their bytecode semantics, and provides an accuracy of 96.14% and a false positive rate of 4.10%. More importantly, with the help of bytecode semantics, DeepRefiner successfully detects 10,991 malicious applications from 11,000 obfuscated malicious applications, which are generated by various obfuscation techniques.

Another problem we investigate regards to the practicality of malware detection. The practicalities of most learning-based malware detection systems are constrained by laborious human feature engineering and complicated feature extraction. As Android framework evolves over years, so do benign applications and malware. As a result, it becomes increasingly difficult for domain experts to identify the features for malware detection and assimilate the fast evolving knowledge about Android system and malware detection. For example, Android added more than 5,000 new APIs from version 4.4 to 5.0 [7], and both benign applications and malware target at Android versions with the updated APIs and features [105]. In response, extensive domain knowledge and manual investigation are required for existing approaches to update.

More importantly, malware evolves especially rapidly. Mobile malware attacks increased more than three times between 2015 and 2016 [64]. New techniques proposed in the academia have been quickly adopted by malware authors. For example, anti-emulation techniques proposed in HITCON 2013 [1] were found in real-world malware later [86]. If the human engineered features in malware detection cannot catch up with the evolution of malware, malware detection may be evaded. For example, DroidAPIMiner [2] has been proved less effective in detecting newly identified malware based on frequently used API calls in [76]. Such rapid evolution results in increasing demands on highly automatic feature engineering and new learning approaches to provide practical detection.

To show the advantage of practical detection provided by DeepRefiner, we remove expert's domain knowledge and complicated human feature engineering from the whole process. Instead, DeepRefiner applies deep neural networks with multiple hidden layers to perform automatic feature engineering on top of lightweight preprocessing. In preprocessing, DeepRefiner retrieves xml values from xml files in the first detection layer and captures bytecode semantics from disassembled classes.dex file in the second detection layer. DeepRefiner then represents applications as vectors, which are used as inputs for deep neural networks. The hidden layers in neural networks automatically engineer detection features from input vectors through nonlinear transformation. The whole process of DeepRefiner is fully automated and is proved to be highly efficient as shown in our experiments.

To show the advantage of the whole system, we evaluate the overall detection performance of DeepRefiner and compare it with a state-of-the-art single classifier-based detection method, StormDroid [26], as well as ten commercial anti-virus scanners. DeepRefiner shows superior performances over StormDroid and all anti-virus scanners. With the dataset of 62,915 malware and 47,525 benign applications, DeepRefiner achieves an accuracy of 97.74% with a true positive rate of 97.96% and a false positive rate of 2.54%, while StormDroid's accuracy is 89.62% with a true positive rate of 87.51% and a relatively high false positive rate of 7.59%. Most signature-based anti-virus scanners' detection rates are in the range between 70% and 80%, which are significantly lower than DeepRefiner.

The robustness of DeepRefiner is evaluated from two perspectives. First of all, we test DeepRefiner against typical existing obfuscation techniques [94], including Repacking, Renaming Identifier and Data Encryption. For 11,000 obfuscated malware generated by typical obfuscation techniques, DeepRefiner successfully detects 99.92% of them. In addition, for adversarial crafting [89] [88] against neural network-based malware detection systems, the defensive approaches proposed by Grosse et al. [54] can be similarly applied to the first detection layer of DeepRefiner, which also relies on xml files. For the second detection layer, we conduct preliminary research on the robustness of DeepRefiner following the three constrains that are identified by Grosse et al. Our experimental results show that DeepRefiner is robust even after injecting 400% code into obfuscated malware.

The rest of the work is organized as follows. Section 4.2 presents the DeepRefiner system. Section 4.3 introduces the settings of experiments. Section 4.4 evaluates DeepRefiner in different aspects. Section 4.5 discusses the limitations. Section 4.6 summarizes the related work, and Section 4.7 concludes the work.

4.2 The DeepRefiner System

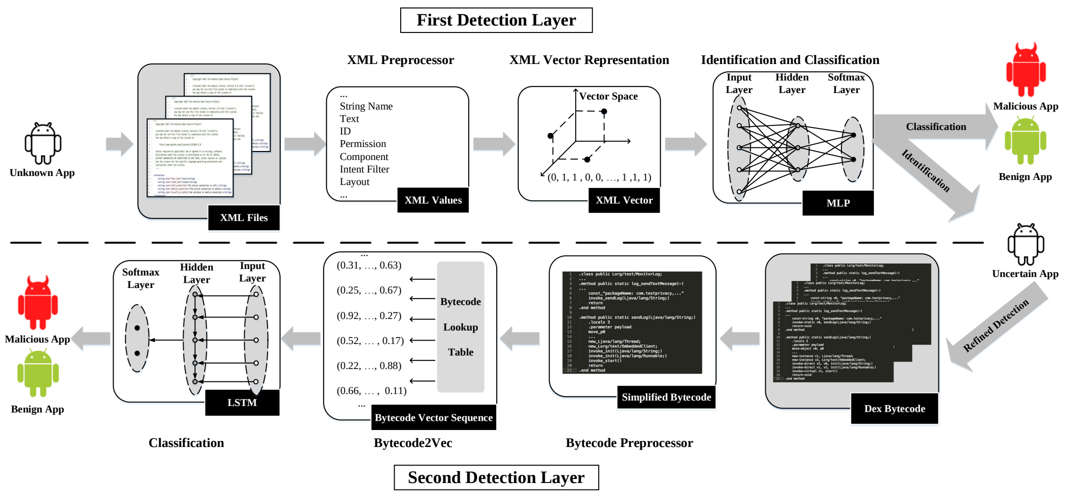


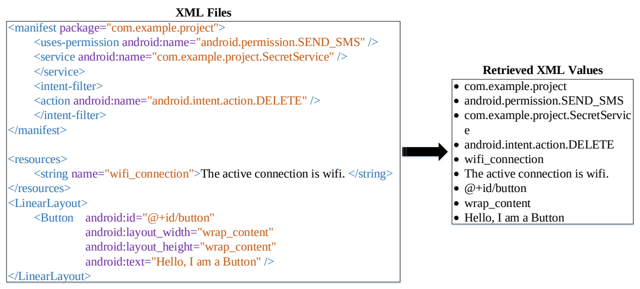
Figure 4.1: Architecutre of DeepRefiner

4.2.1 Overview

As depicted in Figure 4.1, DeepRefiner consists of two detection layers to achieve both efficient and effective detection. In the **First Detection Layer**, DeepRefiner efficiently classifies a majority of malware from benign applications, and identifies the applications that cannot be reliably classified based on the resources required in their xml files. For applications that cannot be reliably classified, DeepRefiner generates a temporary label uncertain and further feeds them into the second detection layer. In the **Second Detection Layer**, DeepRefiner performs refined inspection and effectively detects malware from uncertain applications based on bytecode semantics at both method-level and application-level. In the rest of this section, we discuss each detection layer in detail.

4.2.2 The First Detection Layer

In the first detection layer, DeepRefiner performs efficient process to distinguish complexities of applications and classify a majority of malware from benign applications. The first detection layer consists of three phases, including XML Preprocessor, XML Vector Representation, and Identification and Classification.



```
<manifest package="com.example.project">
    <uses-permission android:name="android.permission.SEND_SMS" />
    <service android:name="com.example.project.SecretService" />
    </service>
    <intent-filter>
    <action android:name="android.intent.action.DELETE" />
    </intent-filter>
</manifest>

<resources>
    <string name="wifi_connection">The active connection is wifi. </string>
</resources>
<LinearLayout>
    <Button    android:id="@+id/button"
               android:layout_width="wrap_content"
               android:layout_height="wrap_content"
               android:text="Hello, I am a Button" />
</LinearLayout>
```

- com.example.project
- android.permission.SEND_SMS
- com.example.project.SecretServic
  e
- android.intent.action.DELETE
- wifi_connection
- The active connection is wifi.
- @+id/button
- wrap_content
- Hello, I am a Button

Figure 4.2: Example of Input and Output of XML Preprocessor

XML Preprocessor

In XML Preprocessor, DeepRefiner decompiles apk files and performs lightweight preprocessing to retrieve xml values from AndroidManifest.xml and all xml files under `/res/`. As the example shown in Figure 4.2, from `<string name="wifi_connection"> The active connection is wifi</string>`, XML Preprocessor retrieves two unique xml values `wifi_connection` (a string name) and `The active connection is wifi` (a text shown to end users). The retrieved xml values provide detailed information about the resources required and included in an application, and show different patterns in benign applications and malware. For example, in GUI texts shown to end users, benign applications usually include instructions to guide end users performing application's functionalities. However, malware often shows texts requiring users to input Credit Card numbers, download new services, or provide login credentials. In addition, as shown in our experiments, some xml values are shared among different malware. For example, `Call Customer Service: 15210066773` is a text only observed in malware. This phone number pretends to be a customer service number, while in fact it is used to conduct telephone fraud, and is shared among many malware. Therefore, xml values are able to capture the differences between malware and benign applications.

XML Vector Representation

Next, DeepRefiner represents each application as an XML Vector. To this end, DeepRefiner builds an `XML Database` according to the unique xml values retrieved from training dataset. If the size of XML Database is X, DeepRefiner defines an X-dimension vector space accordingly. For each application, DeepRefiner represents it as an XML Vector by mapping its retrieved xml values to the X-dimension vector space, such that for each xml value the respective dimension is set to one and all other dimensions are zeros.

Identification and Classification

In the last phase, DeepRefiner identifies uncertain applications from unknown applications, and classifies the rest of applications into malicious or benign according to the input X-dimension XML Vectors.

Model Design. Before introducing the details of model architecture, we first present the critical points of the model design. As illustrated in Figure 4.3, DeepRefiner builds the first detection model composing of multiple hidden layers followed by a Softmax layer. In order to precisely process XML Vectors and perform automatic feature engineering, DeepRefiner applies Multilayer Perceptron (MLP) [58] with multiple hidden layers as the deep neural network used in the first detection layer. The fully-connected structure of MLP is particularly suitable for processing XML Vectors that have no sequential information among them. In particular, the multiple

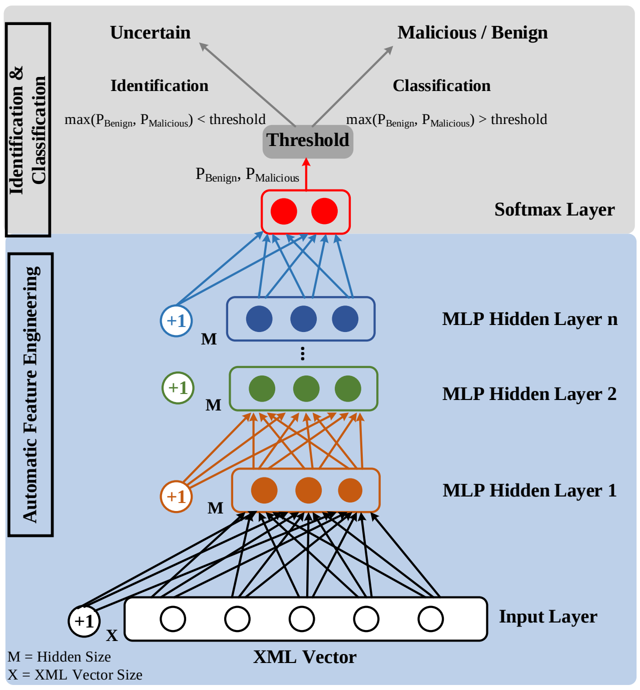


Figure 4.3: Identification and Classification Phase in the First Detection Layer of DeepRefiner

hidden layers of MLP are designed to perform automatic feature selection through non-linear transformation [50]. After the process of MLP hidden layers, the resulting Hidden Vector, which encodes automatically engineered features, is passed to a Softmax layer. The relationships among automatically engineered features are detected and used for classification by the Softmax layer.

Model Architecture. The first detection model takes each XML Vector along with an activation function as input. Due to the fully-connected structure of MLP, every neuron (or, node) in previous layer connects with a certain `weight` and `bias` to every neuron in the following layer. At the last MLP hidden layer, the input X-dimension XML Vector is represented as an M-dimension Hidden Vector along with calculated `weights` and `biases`, where M is the number of neurons in the last MLP hidden layer. Given the generated M-dimension Hidden Vector of an application as input, the Softmax layer adds up the calculated `weights` and `biases` of this application, and normalizes the calculated results into predictive probabilities, $P_{\text{Malicious}}$ and P_{Benign}, by assigning a probability to each class (i.e., malicious or benign). In particular, the Softmax layer takes the option that has higher probability as the classification result.

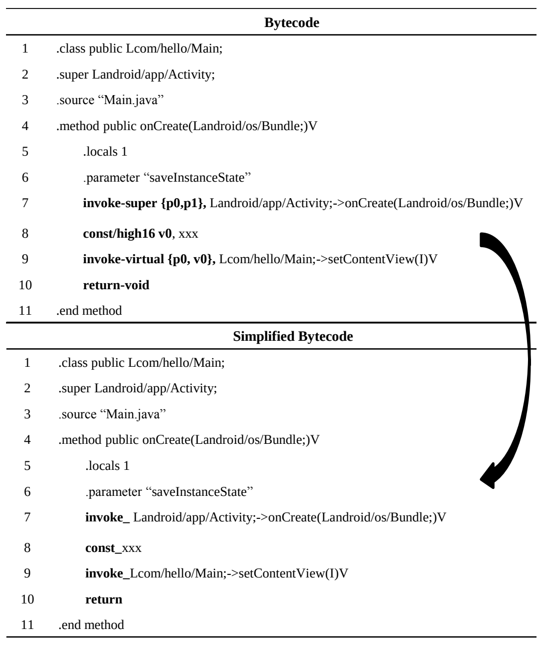
However, in the case that the input XML Vectors do not provide sufficient information for classification, the Softmax layer might generate relatively low $P_{\text{Malicious}}$ and P_{Benign}, and outputs unreliable classification result. In order to provide reliable detection result, DeepRefiner adds a threshold on top of the Softmax layer and introduces a temporary label called uncertain.The process is as follows:

$$\text{Label} = \begin{cases} \text{Uncertain} & \text{if} \max(P_{\text{Malicious}}, P_{\text{Benign}}) < \text{threshold} \\ \text{Malicious} & \text{elif}(\max(P_{\text{Malicious}}, P_{\text{Benign}}) \geq \text{threshold}) \\ & \text{and}(P_{\text{Malicious}} > P_{\text{Benign}}) \\ \text{Benign} & \text{else} \end{cases}$$

The uncertain applications are the ones that DeepRefiner cannot reliably classify according to their xml values in the first detection layer, and thus require further inspection from different perspectives. The threshold is set to 1.00 through most of this work. Tuning of this threshold is discussed in Section 4.3. As shown in our experiments, DeepRefiner at the first layer identifies a small portion (i.e., 26.5%) of applications in the dataset as uncertain, and effectively classifies the rest of applications into malicious or benign with a high accuracy.

4.2.3 The Second Detection Layer

In the second detection layer, DeepRefiner performs refined classification for uncertain applications identified in the first detection layer according to bytecode semantics at both method-level and application-level. The second detection layer also consists of three phases, including Bytecode Preprocessor, Bytecode2Vec and Classification.


Bytecode

```
.class public Lcom/hello/Main;
.super Landroid/app/Activity;
.source "Main.java"
.method public onCreate(Landroid/os/Bundle;)V
      .locals 1
      .parameter "saveInstanceState"
      invoke-super {p0,p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V
      const/high16 v0, xxx
      invoke-virtual {p0, v0}, Lcom/hello/Main;->setContentView(I)V
      return-void
.end method
```

Simplified Bytecode

```
.class public Lcom/hello/Main;
.super Landroid/app/Activity;
.source "Main.java"
.method public onCreate(Landroid/os/Bundle;)V
      .locals 1
      .parameter "saveInstanceState"
      invoke_ Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V
      const_xxx
      invoke_Lcom/hello/Main;->setContentView(I)V
      return
.end method
```

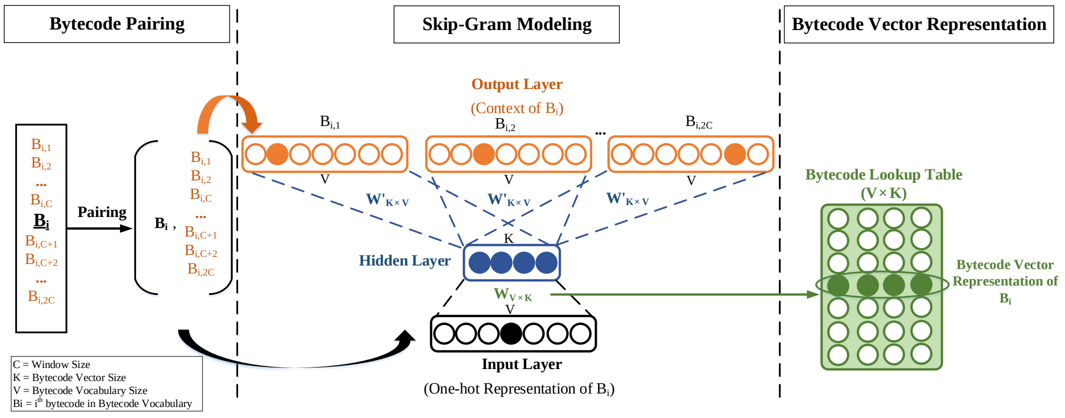
Figure 4.4: Example of Input and Output of Bytecode Preprocessor

Bytecode Preprocessor

In Bytecode Preprocessor, apk files are disassembled and dex bytecode is obtained by directly decompiling classes.dex file using apktool [114]. Dex bytecode contains more than 200 dex instructions, while a majority of dex instructions are similar with little differences, such as number of bits reserved for operands and operand type in operators. Although dex instructions provide detailed information to the mobile device, they significantly increase the complexities of bytecode analysis. Simplifying dex instruction is widely used as a preprocessing step of bytecode analysis, such as information flow tracking [105], dex bytecode symbolic execution [62], and call graph generating [46]. As a result, DeepRefiner then performs lightweight preprocessing by replacing the original dex instructions with fifteen instruction categories as given in [46].

As shown in Figure 4.4, two similar dex instructions `invoke-super {p0,p1}` and `invoke-virtual {p0,v0}` are replaced with the same dex instruction category `invoke`. In particular, DeepRefiner only simplifies bytecode instructions without removing any other information, such as class names, superclass names, implemented interfaces, fields, and methods.

Bytecode2Vec



Bytecode Pairing
Skip-Gram Modeling
Bytecode Vector Representation
Output Layer
(Context of Bi)
Bi,1
Bi,2
Bi,2C
Pairing
Bi ,
Bi,C
Bi,C+1
Bi,C+2
V
W'K×V
K
Hidden Layer
WV×K
Input Layer
(One-hot Representation of Bi)
Bytecode Lookup Table (V×K)
Bytecode Vector Representation of Bi
C = Window Size
K = Bytecode Vector Size
V = Bytecode Vocabulary Size
Bi = ith bytecode in Bytecode Vocabulary

Figure 4.5: Bytecode2Vec consists of three steps: (1)Bytecode Pairing; (2) Skip-Gram Modeling; (3) Bytecode Vector Representation

In this phase, DeepRefiner represents each line of simplified bytecode with a dense and real-valued Bytecode Vector, which captures bytecode semantics by tracking its contexts within its method. For a target line of bytecode, its context is the nearby bytecode within a chosen window size in the same method. As shown in Figure 4.5, Bytecode2Vec consists of three steps, including Bytecode Pairing, Skip-Gram Modeling, and Bytecode Vector Representation.

Bytecode Pairing. DeepRefiner firstly builds a `Bytecode Vocabulary` according to unique bytecode obtained from known applications, and generates bytecode pairs by pairing the target bytecode with its contexts. Let B_i be the i^th bytecode in Bytecode Vocabulary, and C be the chosen window size. For the target bytecode B_i, its context is the nearby bytecode $\{B_{\text{i},1}, B_{\text{i},2}, ..., B_{\text{i},2\text{C}}\}$ including C lines of bytecode ahead and C lines of bytecode behind. Accordingly, DeepRefiner generates $2C$ bytecode pairs by pairing B_i with each bytecode in its context. Since some lines of bytecode, such as class names, superclass names, and filed names, do not belong to any method, DeepRefiner treats these bytecode as an individual method, and generates bytecode pairs accordingly. Note that, DeepRefiner chooses method as the granularity of generating bytecode pairs, since method-level bytecode semantics are more resilient towards typical obfuscation techniques [61].

Skip-Gram Modeling. With the generated bytecode pairs, DeepRefiner then builds a Skip-Gram neural network [78] by taking each bytecode pair as a training instance. Bytecode in the training instance is one-hot encoded and represented as a V-dimension vector with a value one at the index corresponding to the bytecode and zeros in all other indexes, where V is the size of Bytecode Vocabulary. In particular, with B_{i} as input and its context $\{B_{\text{i},1}, B_{\text{i},2}, ..., B_{\text{i},2\text{C}}\}$ as output, the Skip-Gram neural network projects input and output into an embedding space so as to learn the appropriate representation for each bytecode as a dense vector.

After training the Skip-Gram neural network with bytecode pairs, DeepRefiner obtains a $V \times K$ weight matrix **W** between the input layer and the hidden layer, where V is Bytecode Vocabulary size and K is Bytecode Vector size set by DeepRefiner. In particular, the weight matrix **W** contains the vector encodings (i.e., Bytecode Vectors) of all bytecode in Bytecode Vocabulary as its rows. For example, Bytecode Vector of B_{i} is the i^{th} row in **W**. The weight matrix **W** is then applied as the Bytecode Lookup Table to search for Bytecode Vector given a bytecode in the Bytecode Vocabulary.

More importantly, the generated Bytecode Vectors encode method-level bytecode semantics. If two different bytecode have similar contexts or perform similar functionalities, they are represented by similar Bytecode Vectors and should be treated similarly by the detection model, which is significantly different from existing works. For example, malware often uses `getDeviceId()` and `getScriberId()` in a sequence before sending out sensitive information. As shown in our experiments, the similarity score between the Bytecode Vectors of `getDeviceId()` and `getScriberId()` is 0.94, while the similarity score between `getDeviceId()` and an irrelevant bytecode `WebChromeClient.onRequestFocus()` is -0.24.

Bytecode Vector Representation. For each line of bytecode included in an application, DeepRefiner retrieves the corresponding row from Bytecode Lookup Table as its Bytecode Vector, and represents each application as a variable-length Bytecode Vector Sequence $\{BV_1, BV_2, ..., BV_n\}$ by combining the Bytecode Vectors according to the original bytecode sequence in the source code, where BV_i is a K-dimension Bytecode Vector and n is the number of bytecode lines in source code.

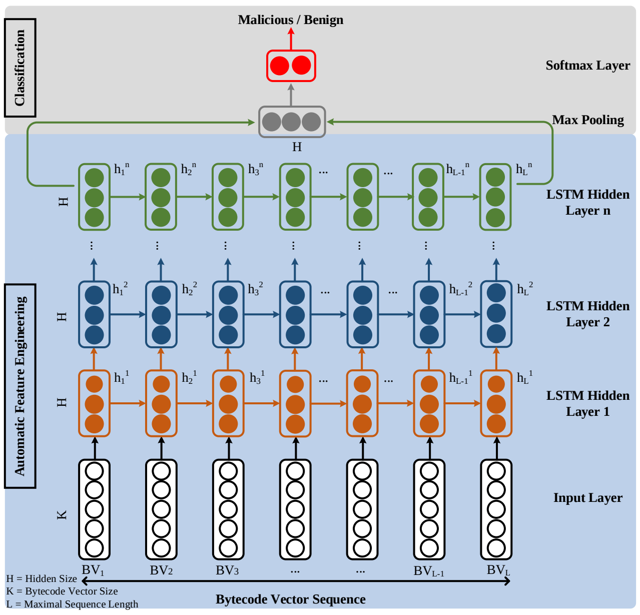


Figure 4.6: Classification Phase in the Second Detection Layer of DeepRefiner

Classification

In the last phase, DeepRefiner performs refined inspection on top of Bytecode Vector Sequences, and detects malware from uncertain applications.

Model Design. DeepRefiner builds the second detection model composing of multiple stacked LSTM hidden layers followed by a Max pooling layer and a Softmax layer as illustrated in Figure 4.6. For modelling variable-length Bytecode Vector Sequence with sequential information, DeepRefiner chooses Long Short Term Memory [56] over traditional Recurrent Neural Networks (RNN) since LSTM mitigates the gradient vanish problem of RNN. In particular, LSTM introduces a new structure called memory cell. A memory cell is composed of four main elements: an input gate, a neuron with a self-recurrent connection, a forget gate and an output gate. The gates allow the memory cell to store and access information over long periods of time without the loss of short time memory. This structure is especially effective in capturing application-level bytecode semantics without losing method-level semantics. Similar as in the first detection layer, automatic feature engineering is conducted by multiple LSTM hidden layers. Specially, DeepRefiner includes a Max pooling layer between the last hidden layer and the Softmax layer. Using Max pooling allows an input Bytecode Vector Sequence with arbitrary length to be represented by a fix-length hidden vector. Max pooling also helps the Softmax layer to focus on certain bytecode combinations that are most relevant to the final classification task.

Input Layer. As described in Section 4.2.3, each uncertain application is represented by a Bytecode Vector Sequence $\{BV_1, BV_2, ..., BV_n\}$, where n is the sequence length. Since the length of Bytecode Vector Sequence are variable in different applications, the second detection model firstly pads short Bytecode Vector Sequences with several K-dimension Bytecode Vector of zeros in the input layer to reach the longest length. As a result, the input of detection model is a Bytecode Vector Sequence $\{BV_1, BV_2, ..., BV_L\}$, where L is the length of longest Bytecode Vector Sequence in the dataset.

Stacked LSTM Hidden Layers. LSTM hidden layers in the second detection layer are connected and process the input sequence in forward direction. During the process, each LSTM hidden layer produces a Hidden Vector Sequence $\{HV_1, HV_2, ..., HV_L\}$, where HV_i is an H-dimension hidden vector and H is the hidden size of current hidden layer. The output of the previous LSTM hidden layer is taken by the following LSTM hidden layer as input. Stacked LSTM hidden layers are used to build up historical information about the input Bytecode Vector Sequence, and iteratively update it to learn the bytecode semantics at application-level. Finally, the Hidden Vector Sequence produced by the last LSTM hidden layer is fed into Max pooling.

Max Pooling and Softmax Layer. From the input Hidden Vector Sequence, Max pooling extracts maximum values and generates an H-dimension vector as the final hidden representation for the input Bytecode Vector Sequence. At the end of the detection model, the Softmax layer is used to perform the classification task on the hidden representation, and outputs predictive probabilities for each uncertain application based on calculated `weights` and `biases`.

As shown in our experiments, with Bytecode Vector and LSTM precisely capturing bytecode semantics at different level, DeepRefiner understands the bytecode semantics of uncertain applications in a more comprehensive manner. As a result, DeepRefiner then produces accurate prediction results for uncertain applications, which are challenging for DeepRefiner to classify correctly in the first detection layer.

4.3 Experiments

DeepRefiner's detection performance is empirically evaluated with a series of experiments. We describe experimental settings in this section and report the evaluation results along with detailed analysis in Section 4.4.

4.3.1 Data Collection

We built an initial benign dataset by crawling GooglePlay from March 2016 to May 2016. To exclude potential malware from this initial dataset, we sent each application to VirusTotal [111], which is an antivirus service with over fifty antivirus scanners. We discard applications from the original benign dataset if any anti-virus scanner raises alarm for it so as to generate the benign dataset, which consists of 47,525 benign applications. The malicious applications came from VirusShare [109] and MassVet [25]. Duplicated applications were removed if they share the same SHA-256 hash values. The final malicious dataset includes 38,074 malware before 2015 and 24,841 malware from 2015 to 2016. Same datasets are used to evaluate Deep-

Refiner, StormDroid, and anti-virus scanners in the experiments. DeepRefiner is evaluated with a relatively large dataset since DeepRefiner requires a large number of training applications in order to fine-tune parameters in the deep neural networks and generate mature detection models to effectively detect malware from testing applications.

4.3.2 Measurements and Metrics

We conduct a series of experiments using `ten-fold cross validation` to measure the performance of DeepRefiner. The dataset including 110,440 applications is shuffled and randomly divided into ten equal-size subsets. Since hyperparameters (e.g., learning rate, batch size and hidden size) have significant influences on deep neural networks' performance, it is inappropriate using one testing dataset for both hyperparameter selection and evaluation. This is because the trained neural network could be over-fitted and might perform poorly on other new datasets besides certain testing dataset. Simply choosing nine subsets for training and the remaining subset for testing is not applicable in our experiments.

To prevent over-fitting and provide fair experimental environment, DeepRefiner is trained and tested in ten rounds. In each round, we choose eight subsets as training set, one subset as validation set, and the remaining subset as testing set. Validation set is used to select hyperparameters for deep neural network. After that, a detection model is trained with the training set and the selected hyperparameters. Testing set is then used to evaluate the detection performance of the detection model. Especially, since DeepRefiner includes two detection models and performs refined detection in a pipeline process, most applications in testing set will be reliably classified by the first detection model, and only a small portion applications will be identified as uncertain and go through the second detection model. As a result, the first detection layer is evaluated with the original testing set, while the second detection model is evaluated using the uncertain applications identified from the original testing set.

We repeat the process ten times and get detection result for every application in our dataset.

We evaluate the performance of DeepRefiner using three metrics, True Positive Rate (TPR), False Positive Rate (FPR), and Accuracy, where TPR is the percentage of malware being detected correctly, FPR is the percentage of benign applications being detected as malware, and Accuracy is the percentage of all applications being classified correctly in our experiments.

4.3.3 Parameter Tuning

We now present the tuning process of parameters which have influences on the performance of DeepRefiner.

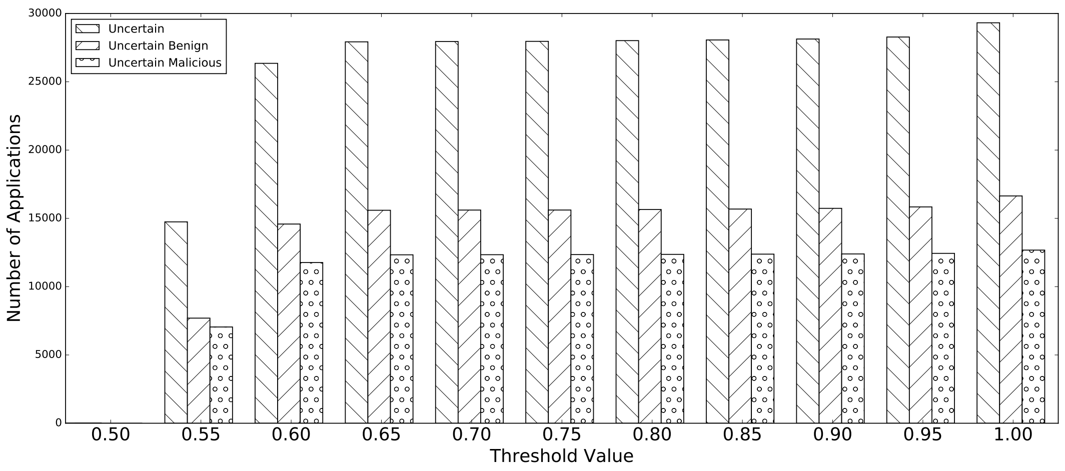


Figure 4.7: Distribution of Uncertain Applications over Different Uncertain Threshold Value

Threshold Tuning

For most of our experiments, we choose the threshold value in such a way that the first detection layer identifies as many uncertain applications as possible. As a result, the threshold is set to 1.00. For DeepRefiner users who wish to have a lower amount of uncertain applications may choose a lower threshold at the range of $[0.50, 1.00]$.

In Figure 4.7, we present the distribution of uncertain applications under different threshold values. The number of uncertain applications is stable when threshold is changed from 0.60 to 1.00, and reaches the maximum value when threshold is set to 1.00. Since the largest value of $P_{\text{Malicious}}$ and P_{Benign} must be in the range of $(0.50, 1.00]$, the first detection layer identifies zero uncertain application when the threshold is set to 0.50. In this case, DeepRefiner relies on only the first detection layer to generate final detection results for all unknown applications.

Surprisingly, for each threshold value, the first detection layer identifies more benign applications as uncertain than malicious applications, even though the dataset contains more malicious applications than benign applications. One reason is that xml patterns of benign applications are not straightforward compared with xml patterns contained in malware. Benign applications tend to require more and more xml resources commonly used by malware, but also include unique benign elements. As a result, the first detection layer is confused by benign applications' behaviors only according to their xml values. The effects of different threshold values are analyzed in details in Section 4.4.

Hyperparameter Tuning

In the first detection layer and the second detection layer, `weights` and `biases` of MLP and LSTM are initialized and optimized end-to-end with Stochastic Gradient Decent [17] and backpropagation [98] with shuffled mini-batches in the training phase, and used for detection afterwards. The learning rate of each parameter is automatically scheduled by Adam method [67], with the initial learning rate of 0.001 and a decay ratio of 0.95 for each iteration. The remaining hyper-parameters, such as number of hidden layers, mini-batch size, hidden size, Bytecode Vector size, are fine-tuned and set empirically with the ideal values by evaluating with the validation set.

According to the resulting values, the detection model in the first detection layer is set to include three MLP hidden layers with 3,000 neurons in each hidden layer, while the second detection model is set to contain three stacked LSTM hidden layers with 20 as the hidden size for each hidden layer. For both MLP and LSTM, the mini-batch size is set to 100.

For Bytecode2Vec in the second detection layer, the Bytecode Vector size and window size are set in such a way to strike a balance between efficiency and effectiveness. Although larger Bytecode Vector size and window size allow Bytecode Vector to include more comprehensive method-level bytecode semantics, they also significantly increase the computing complexities of Skip-Gram modeling and detection model training. Due to such consideration and experimental evaluation, Bytecode Vector size is set to ten and window size is set to five in our experiments. The `weights` in Skip-Gram model are also initialized randomly at first and updated by backpropagation in the training phase.

4.4 Evaluation

In this section, we present evaluation results along with detailed analysis of DeepRefiner's detection performance in each detection layer, robustness evaluation against obfuscation techniques, and runtime performance.

4.4.1 Evaluation of the First Detection Layer

In the first detection layer, from the dataset of 110,440 applications, DeepRefiner identifies about one fourth of applications as uncertain, including 16,644 uncertain benign applications and 12,676 uncertain malicious applications. For the rest 81,120 applications, including 30,881 benign applications and 50,239 malicious applications, DeepRefiner achieves an accuracy of 98.32%, with a true positive rate of 98.33% and a false positive rate of 1.70%.

False Positives

From 30,881 benign applications, DeepRefiner classifies 1.70% of them as malicious. In particular, 75% of these false positives include less than 25 non-zero values in their XML Vectors. Although these false positives are not identified as uncertain, they still do not provide sufficient information to DeepRefiner. In addition, these false positives include xml values that are usually observed in malware. For example, more than 10% of false positives require permissions to read phone state, access user location, write to external storage, and change phone settings. Some false positives even require signature-level permissions or permissions not for use by third-party applications. Similar as malware, some false positives also require end users to input phone numbers and Credit Card number into text filed, and show payment-related text to users. The xml patterns in these false positives are very similar as malware, which is difficult for DeepRefiner to correctly classify them.

False Negatives

In the first detection layer, DeepRefiner generates prediction labels for 50,239 malicious applications, while mistakenly labels 838 of them. After resending the 838 false negatives to VirusTotal, we discover that 6% of false negatives only receive alarms from one anti-virus scanner. 63% of false negatives generated in the first layer are adware, which are repackaged to include a bunch of advertisement libraries. The xml values of adware do not show significant differences from benign applications' xml values. For the rest of false negatives, we discover that most of their XML Vector include insufficient information for DeepRefiner to provide correct detection.

Evaluation under Different Thresholds

We also evaluate the detection performance of the first detection layer under different threshold values and present the evaluation results in Table 4.1. When the threshold is set to 0.60, the first detection layer already achieves an accuracy of 97.51% and a false positive rate of 1.74%. The accuracy slightly increases from 97.51% to 98.32% when the threshold changes from 0.60 to 1.00. As shown in our experiments, higher threshold may induce more uncertain applications, which helps to filter out unreliable classification and results in better detection performances in the first detection layer.

Table 4.1: Performance of The First Detection Layer under Different Threshold Values

Threshold	**Performance of The First Detection Layer**			**Number of Uncertain Applications**
	Accuracy	**TPR**	**FPR**	
0.50	87.30%	86.91%	12.17%	0
0.55	92.80%	89.35%	2.36%	14745
0.60	97.51%	97.03%	1.74%	26351
0.65	98.14%	98.10%	1.79%	27925
0.70	98.15%	98.11%	1.79%	27948
0.75	98.15%	98.12%	1.79%	27960
0.80	98.16%	98.14%	1.79%	28014
0.85	98.17%	98.15%	1.79%	28063
0.90	98.19%	98.16%	1.78%	28131
0.95	98.21%	98.19%	1.76%	28278
1.00	**98.32%**	**98.33%**	**1.70%**	**29320**

The experimental results also show that the first detection layer alone (when threshold is set to 0.50) produces reasonable yet not outstanding detection performances, which demonstrates the necessities of introducing uncertain applications and detecting malware according to comprehensive bytecode semantics as in the second detection layer.

Case Studies of Identified Uncertain Applications

Since it is difficult to manually analyse all uncertain applications, we randomly choose one round classification results obtained from the ten-fold cross validation to perform detailed analysis. In the chosen round, DeepRefiner identifies 3,116 uncertain applications, including 1,547 uncertain benign applications and 1,569 uncertain malware.

Looking into these uncertain applications, we discover that information included in their XML Vectors are insufficient for the first detection model to precisely capture their behaviors and perform reliable classifications. Specifically, after mapping to the XML Vector space, 52% of the uncertain applications have less than 50 non-zero values in their XML Vectors, while 27% of them have less than 10 non-zero values.

In addition, we find that the differences between XML Vectors of uncertain benign applications and uncertain malware are not significant. We analyse the top 50 xml values included in uncertain benign applications and uncertain malware, and discover that 44 of these popular xml values are shared among uncertain applications. These shared xml values not only include sensitive permissions such as `android.permission.WRITE_SETTINGS` and `android.permission.MOUNT_UNMOUNT_FILESYSTEM` , but also include strings such as `Cancel`, `Number` and `Settings`.

Identifying some applications as uncertain does not mean that DeepRefiner cannot provide accurate prediction results for them in the first detection layer. Actually, from the 3,116 uncertain applications, DeepRefiner correctly detects all of the uncertain malware and several uncertain benign applications. Although these classification results are correct, we do not include them into the final prediction results since the corresponding predictive probabilities are below the chosen threshold.

4.4.2 Evaluation of the Second Detection Layer

In the second detection layer, for 29,320 uncertain applications, DeepRefiner correctly classifies 96.14% of them, with a true positive rate of 96.47%.

False Positives

From 16,644 uncertain benign applications, DeepRefiner mistakenly predicts 683 of them as malware. Most false positives include bytecode to perform sensitive behaviors, such as reading SMS, making phone calls or recording audio. In particular, a calender application frequently records user's location, reads user's sensitive information and encrypts these information using `TelephonyManager.getDeviceId()`, `TelephonyManager.getScriberId()`, and `Cipher.getInstance()` in a sequence. This bytecode sequence is usually observed in malware, and thus leading DeepRefiner labels this application as malicious. Furthermore, these false positives also include advertisement libraries that are widely used in malware, such as AdMob, InMobi, StartApp. These libraries usually require sensitive permissions and leak user's personal information such as device IDs, contacts and location [132].

False Negatives

We also check the 448 malware missed by DeepRefiner in the second detection layer. One half of these false negatives are adware and do not perform obviously malicious behaviors, while the rest of false negatives perform malicious behaviors regarding SMS and phone calls, such as blocking SMS from a specific phone number and placing calls to premium services.

In conclusion, DeepRefiner's false positives are mainly benign applications that behave similar as malware, while false negatives are malware that do not perform clearly malicious behaviors.

4.4.3 Overall Evaluation

We evaluate the overall detection performance of DeepRefiner and compare it with a state-of-the-art research work StormDroid [26] and ten commercial anti-virus scanners. We note that our dataset is significantly unbalanced along the time line, with benign applications crawled from March 2016 to May 2016 and malware mainly downloaded before 2015. This unbalanced dataset poses challenges to the malware detection systems which need to catch up with the rapid evolution of both Android framework and malware. As shown in the experiments, DeepRefiner achieves reasonably superior performances when compared with the baseline method and anti-virus scanners.

Table 4.2: Detection Performance of DeepRefiner and StormDroid

	DeepRefiner	**The First Detection Layer**	**The Second Detection Layer**	**StormDroid**
Accuracy	97.74%	98.32%	96.14%	89.62%
TPR	97.96%	98.33%	96.47%	87.51%
FPR	2.54%	1.70%	4.10%	7.59 %

Comparison with StormDroid

We first compare the detection performance of DeepRefiner with StormDroid, which detects malware using a fine-tuned SVM model trained with sensitive permissions and sensitive API calls according to a pre-determined feature set. As shown in Table 4.2, DeepRefiner achieves an accuracy of 97.74%, roughly 10% higher than StormDroid, with a lower false postive rate. The superior performance of DeepRefiner results from automatic feature engineering and the collaboration of multiple detection layers to capture malicious behaviors from different perspectives.

Since both StormDroid and the first detection layer of DeepRefiner retrieve permissions during detection, we also compare their detection performance. As shown in Table 4.1, the first detection layer is used to classify all unknown application when the threshold is set to 0.50. Evaluated with the same dataset, the first detection layer produces reasonable detection performances with an accuracy of 87.30% and a true positive rate of 86.91%. By only looking into xml files, the first detection layer achieves comparable detection performances with StormDroid, which extracts carefully-selected features from permissions and sensitive APIs that are commonly used by malware. These carefully-selected features benefit StormDroid in such a way that StormDroid does not need to process raw data with a lot of noises as the first detection layer does.

More importantly, conducting detection according to xml files not only helps the first detection layer achieve comparable performance, but also speeds up the detection process. For an unknown application, StormDroid takes on average 1.2s [26] to generate detection result, while the first detection layer only requires less than 0.22s as shown in Section 4.4.5.

Comparison with Anti-virus Scanners

We also compare DeepRefiner against ten widely-used anti-virus scanners using the same malicious dataset. The detection performance of each scanner is crawled from the VirusTotal service [111].

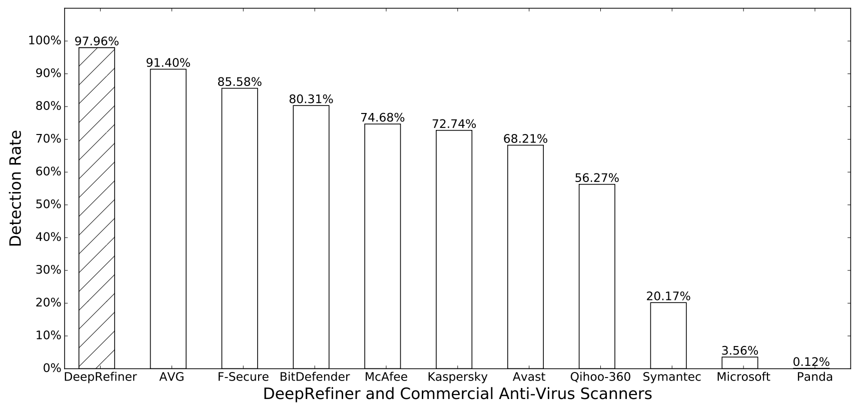


Figure 4.8: Detection Rate of DeepRefiner and Ten Anti-Virus Scanners

As shown in Figure 4.8, the detection rate (i.e., TPR) of anti-virus scanners varies considerably. While the best scanner detect 91.4% of malware, some other scanners discover less than 5% of malware. DeepRefiner provides a detection rate of 97.96% and outperforms all ten anti-virus scanners. By relying on human-crafted signatures from known malware, these anti-virus scanners have limitations of detecting zero-day malware and perform less effective if extracted signatures cannot catch up with the rapid evolution of malware. Different from human crafted signatures, detection features in DeepRefiner are automatically generated according to required xml resources and bytecode semantics included in applications, which are proved to be effective than human engineered signatures.

4.4.4 Robustness

In this section, we first test the robustness of DeepRefiner against typical existing obfuscation techniques and show how resilient DeepRefiner is against various obfuscation techniques. In addition, as a malware detection system applying deep neural networks, we also test the robustness of DeepRefiner against adversarial sampling.

Against Typical Obfuscation Techniques

Obfuscated malware poses challenges to malware detection systems. Malware developers apply obfuscation techniques to manipulate detection systems and evade from detection by transforming malware in different forms but still with the same behavior. For ease of use and without requiring comprehensive domain knowledge about malware detection and Android system, most malware developers apply typical obfuscation techniques which can be easily performed, such as Repacking, Identifier Renaming and Data Encryption.

We apply DroidChameleon [94], a framework implementing eleven typical obfuscation techniques, to 1,000 malicious applications randomly selected from our malicious dataset. DroidChameleon generates 11,000 obfuscated malicious applications, which are then used as a new testing set to evaluate the robustness of DeepRefiner. Since DroidChameleon obfuscates malware by modifying bytecode without changing meta-data stored in xml files, we evaluate the robustness of DeepRefiner with its second detection layer only.

Table 4.3: Detection Performance of DeepRefiner against Various Common Obfuscation Techniques

Obfuscation Techniques	**DeepRefiner Performance**		
		TPR	FPR
Disassembling and Reassembling	✓	100%	0%
Class Renaming	✓	100%	0%
Method Renaming	✓	100%	0%
Field Renaming	✓	100%	0%
String Encryption	✓	100%	0%
Array Encryption	✓	100%	0%
Call Indirection		99.1%	0.9%
Code Reordering	✓	100%	0%
Junk Code Insertion	✓	100%	0%
Instruction Insertion	✓	100%	0%
Debug Information Removing	✓	100%	0%
Overall		**99.92%**	**0.08%**

As shown in Table 4.3, for ten out of eleven obfuscation techniques, DeepRefiner detects all 10,000 obfuscated malicious applications. For the 1,000 obfuscated malicious applications generated by Call Indirection, DeepRefiner successfully detects 99.1% of them, while misclassifies nine obfuscated malicious applications as benign.

Since DeepRefiner does not detect malware according to the presence or absence of certain strings, APIs or methods, it is not vulnerable to the obfuscation techniques such as Class Renaming, Field Renaming, Method Renaming, String Encryption and Array Encryption. Interestingly, although DeepRefiner is based on bytecode semantics observed from bytecode sequence, applying obfuscation techniques, such as Disassembling and Reassembling, Code Reordering, and Junk Code Insertion, to change bytecode sequence and/or insert bytecode sequences does not affect DeepRefiner's detection. The reason is that DeepRefiner detects malware according to bytecode semantics at both method-level and application-level. For example, Disassembling and Reassembling rearranges the classes order and methods order within the whole application without changing code order within methods. In this case, DeepRefiner still can successfully detect obfuscated malware according to stable method-level bytecode semantics.

Another interesting finding is that although both Call Indirection and Junk Code Insertion introduce new pieces of code into obfuscated malware, DeepRefiner successfully detects all obfuscated malware generated by Junk Code Insertion, while misses nine obfuscated malicious samples generated by Call Indirection. After manually checking these obfuscated malware, we find that the new bytecode segments inserted by Junk Code Insertion include `goto`, `const/16`, `nop` and `add-int`. These bytecode segments are not closely related to malicious behaviors and have no effect on bytecode semantics [95]. On the other hand, Call Indirection introduces previously non-existing methods to break down original methods, which might affect method-level bytecode semantics.

More importantly, unlike other obfuscation resilient detection systems which require complicated feature extraction process to generate data dependence graphs and information flows, DeepRefiner relies on lightweight process and automatic feature engineering. The evaluation results also demonstrate the importance of applying both application-level and method-level bytecode semantics to detect obfuscated malware.

Against Adversarial Sampling

As shown in recent research works, deep neural networks are vulnerable to adversarially crafted samples, which may fool deep neural networks to generate adversary-desired misclassifications [88] [54] [89]. Different from crafting adversarial samples against image classification systems which are represented on a continuous scale of real numbers, Grosse et al. [54] study the specialises of performing adversarial crafting attacks on neural networks for malware detection, which is challenging due to the constraint of preserving the functionalities of adversarially crafted applications. They propose additional constraints that appear in malware detection:

(1) Individual features must be fully added or removed without gradually changes. (2) The utility of the modified application must be preserved. (3) Only a restricted amount of features should be added.

Following the constraints, Grosse et al. study the robustness of a typical deep neural network for malware detection which is designed by them, where adversaries craft adversarial samples by adding features (including hardware components, permissions, components, and intents), to AndroidManifest.xml file. In particular, they study the effectiveness of three defensive approaches, including (1) feature reduction, (2) distillation, and (3) re-training with adversarially crafted samples. Their defensive approaches can be similarly applied to the first detection layer of DeepRefiner, which relies on AndroidManifest.file and other xml files for malware detection.

For the second detection layer, we conduct preliminary research on the robustness of DeepRefiner following the three constraints of crafting adversarial samples given by Grosse et al. After adding a restricted amount of junk code into malware without changing its functionalities, our experimental results show that DeepRefiner accurately detects modified malicious applications which are added with 400% of junk code. We plan to conduct in-depth study on other approaches to craft adversarial samples targeted on the second detection layer of DeepRefiner in the near future.

4.4.5 Runtime Evaluation

As the above evaluation has shown the effectiveness of DeepRefiner, we now evaluate the computational overhead incurred by each phase of DeepRefiner. We run the experiments for preprocessing and vector representation on a desktop with 4×3.2GHz Intel-Core and 12 GB of RAM, and conduct the experiments for classification on a desktop with 16GB GPU.

On the large dataset, training the detection models takes about ten minutes and one hour for the first detection layer and the second detection layer, respectively. Although applying deep learning algorithms to train detection models is time-consuming than applying traditional machine learning algorithms (such as SVM and Decision Trees), the trained detection model is capable of performing automatic feature engineering and has outstanding performances of processing raw data with a lot of noises (as shown in Section 4.4.3). In other words, existing detection systems using traditional machine learning algorithms are faster in training models than DeepRefiner, but spend more time to extract useful features from each application during training phase and detection phase.

Table 4.4: Average Time for Processing an Application to Generate Detection Result

The First Detection Layer			
XML Preprocessor	**XML Vector Representation**	**Identification & Classification**	**Overall**
0.17s	0.05s	2×10^{-5}s	0.22s
The Second Detection Layer			
Bytecode Preprocessor	**Bytecode2Vec**	**Classification**	**Overall**
2.09s	0.33s	1.64×10^{-4}	2.42s

Table 4.4 shows the average time of processing an unknown application to generate detection result in our experiments. The performance bottleneck is at the Bytecode Preprocessor. Although the process of Bytecode Preprocessor is lightweight, it takes more time than other phases since there are too many lines of bytecode in apk files.

Overall, for an unknown application, DeepRefiner takes on average 0.22s to finish the entire process in the first detection layer to identify uncertain applications and generate final detection results for most of unknown applications (about 74%). For the identified uncertain applications, DeepRefiner takes on average 2.42s to generate the final classification results according to both method-level and application-level bytecode semantics. Although DeepRefienr spends more time in the second layer, its detection is much faster than some detection methods [120] [76], which take about 30s to process an unknown application by applying complicated feature extraction process.

In conclusion, by distinguishing the complexities of applications and applying automatic feature engineering, DeepRefiner takes on average 0.22s to provide reliable detection results for most of unknown applications, while spends 2.64s to accurately detect malware from uncertain applications.

4.5 Limitations

As a detection system applying static analysis, DeepRefiner suffers from the inherent limitations from static analysis and might fail to detect malicious behaviors which are loaded and executed at runtime. Although DeepRefiner could capture triggers about such behaviors when `java.lang.reflect` and `DexClassLoader` are used in bytecode, this limitation encourages us to incorporate dynamic analysis as new detection layers in future versions of DeepRefiner.

DeepRefiner may make ambiguous predictions after operating for a period of time unless it is updated with new labeled applications. A promising direction in this area is to applying online learning techniques to speed up the update process [6].

Another limitation is inherited from automatic feature engineering. Although automatic feature engineering does not rely on expert domain knowledge and complicated feature extraction, it generates detection features that are not straightforward for human to understand. We plan to understand generated detection features and make classification decisions interpretable by visualizing deep neural networks [134].

4.6 Related Work

Over the past few years, Android malware detection has attracted extensive attentions in both academia and industry. In this section, we mainly review learning-based mobile malware detection systems which are more relevant to DeepRefiner.

A small portion of existing works integrate several classifiers to detect malware in different manners compared with DeepRefiner. Smutz and Stavrou [104] apply an ensemble classifier consisting of many basic classifiers. They perform a diversity analysis to detect unreliable prediction results, and retrain their classifiers with unreliable observations so as to improve classifier accuracy. The basic classifiers are independent to each other, while the two detection layers in DeepRefiner are complementary. For unreliable observations captured in the first detection layer, DeepRefiner feeds them into the second detection layer for further inspection.

DroidSIFT [127] extracts weighted contextual API dependency graph and constructs feature sets accordingly. With graph-based feature vectors, DroidSIFT builds two classifiers, while the first classifier discovers zero-day Android malware, and the second classifier uncovers the family of detected malware. Unlike DroidSIFT, the second detection layer in DeepRefiner is used to perform refined detection over uncertain applications, instead of identifying malware families.

RiskRanker [53] includes two risk analysis modules, while the first-order analysis module sifts through untrusted applications and exposes risky applications, and the second-order analysis module identifies applications with encrypted native code and dynamic code loading. RiskRanker detects malware according to certain behaviors, while DeepRefiner is a learning-based detection system without human engineered features.

Different from DeepRefiner, most learning-based mobile malware detection systems rely on single classifier trained with human engineered features. A list of examples is given below. MAMADROID [76] builds a behavioral model from the sequence of abstracted API calls performed by an application, and uses it to extract features and perform classification. DroidMiner [123] extracts sensitive API call graphs as detection features, while DroidAPIMiner [2] extracts relevant features at API level, and builds a detection model using the generated feature set. Gascon et al. [46] generate function call graphs for Android applications, and build a detection model relying on embedded function call graphs. DroidMat [117], StormDroid [26] and Drebin [10] use not only sensitive API calls but also other information extracted from AndroidManifest.xml file as detection features and train single classifiers afterwards. ICCDetector [120] builds a malware classifier according to ICC-related information included in applications. DRACO [13] and MARVIN [74] utilize static analysis and dynamic analysis to extract features from pre-determined feature categories, and train a detection model afterwards. MASK [23] statically analyzes attributes (including permssions, intent filters,and presence of native code) extracted from applications, and trains a detection model for detecting malware. Finally, Crowdroid [20] relies on crowdsourcing to get system calls from real users, and creates an anomaly model according to system call vector clusters.

The increasing complexity of Android malware calls for new defensive techniques that are harder to evade. To this end, some efforts are made to detect mobile malware using deep neural networks. For example, DroidDetector [126] and DroidSec [125] build Deep Belief Networks to detect Android malware relying on 192 human engineered features, including required permissions, sensitive API calls, and some dynamic behaviors obtained from DroidBox [35]. Deep4maldroid [60] extracts Linux kernel system calls and constructs the weighted directed graphs which are then used to train deep neural networks. Mclaughlin et al. [77] produce a deep neural network-based detection system only according to 218 dex instructions of applications. Although these systems apply deep neural networks, they still rely on features pre-determined by domain experts, such as required permissions and sensitive API calls. Different from these existing works, DeepRefiner combines several detection models in a complementary way and relies on automatic feature engineering without domain knowledge about malware detection. Furthermore, instead of using one-hot or boolean vector representations, DeepRefiner represents applications with dense Bytecode Vector Sequences in the second layer.

Towards automatically engineering features for malware detection, FeatureSmith [131] mines the scientific literatures written in natural language to generate features that are semantically related to malicious behaviors, while DeepRefiner learns detection features from xml files in the first layer and dex bytecode in the second layer.

Some recent works study the robustness of deep neural networks against adversarial crafting. Papernot et al. [88] formalize the space of adversaries against deep neural networks and introduce a class of algorithms to craft adversarial samples. In an application to computer vision, they show that the proposed algorithms can produce adversarial samples and bypass deep neural networks. As a defensive mechanism, Papernot et al. [89] introduce defensive distillation to reduce the effectiveness of adversarial samples on deep neural networks. Grosse et al. [54] investigate adversarial crafting attacks to Android malware detection, and propose defensive mechanisms to thwart the attacks.

Enormous signature-based detection methods have been proposed in the literature for detecting malicious mobile applications. For example, Kirin [41] detects malware based on required permissions which break certain pre-defined security rules. Since they are not closely related to this work, we refer readers to a survey [3] for more details.

4.7 Conclusions

This work presented DeepRefiner, an Android malware detection system combining multiple detection layers in complementary way to provide refined detection. To catch up with the rapid evolution of both Android system and malware, DeepRefiner removes laborious human feature engineering and complicated feature extraction from the process. It applies two different detection layers, where the first detection layer achieves efficient detection according to xml files and the second detection layer performs effective and robust detection based on comprehensive bytecode semantics at different scales. The robustness of DeepRefiner against typical obfuscation techniques and adversarial samples is evaluated and demonstrated in our experiments. In the future, we plan to apply DeepRefiner to detect new malware in various application markets and include more detection layers, such as dynamic analysis layer, to better capture and understand malware behaviors.

Chapter 5

DroidEvolver: Self-Evolving and Scalable Malware Detection System

5.1 Introduction

Android malware running on most mobile devices severely violates end users' security and privacy. As Android framework evolves over years, so do benign applications and malware. Both benign applications and malware target at newer Android versions with the updated APIs and features [105]. As a result, it becomes increasingly difficult to build detection systems that are trained with older applications (both benign and malicious) and make outstanding performance when faced with modern applications after operating for long periods of time.

Rapid-aging detection system is a huge concern in both industry and research area. As shown by Baidu in BlackHat 2016 [71], the recall rate of their detection model trained on January 2016 drops by 7.6% in only six months. Without the ability of self-learning, Baidu's detection model requires continuous retraining with labeled applications. As in research area, most existing detection systems need to periodically retrain their detection models with labeled applications.

These solutions, however, face several challenges. First of all, it is difficult to manually decide when to retrain detection models. If the model is retrained too frequently, there will be little novelty in the information obtained to enrich the detection model; otherwise, the detection model cannot capture some emerging threats in a timely way. In addition, the retraining process requires manual labeling of all the processed applications, which is constrained by available resources. Requiring manual labeling also induces to a loose retraining frequency [100], which results in periods of time where the model cannot be trusted and might be bypassed by malware. Last but not least, the retraining process is performed by cumulating original training dataset with newly labeled applications. Consequently, the retraining process is expensive and unscalable, especially in the scenario where the number of new applications grows rapidly from time to time.

While facing the challenges, we propose a novel malware detection system for Android that relies on the ability of self-evolving and real-time update to achieve both effective and scalable detection over time. The proposed system, DroidEvolver, automatically and efficiently updates itself to adapt to new changes discovered from both Android framework and applications without requiring true labels of applications.

DroidEvolver maintains a model pool, which includes several detection models trained with different learning algorithms, to detect malware based on API calls. The intuition is that different detection models are less likely to be outdated at the same time even they are initialized with same training dataset.

Upon processing every new application, DroidEvolver generates a Juvenilizing Indicator. The indicator is generated by calculating the distances between the new application and other processed applications after mapping them into the feature space corresponding to each detection model according to extracted API calls. The distance represents the similarity between the new application and other processed applications. A small Juvenilizing Indicator value indicates that this new application has few common features with processed applications and will be identified as drifting application.

The model processing drifting application is identified as aging model. Both feature set and model structure of aging model have limitations of representing the identified drifting application. As a result, the identification of drifting applications and aging models indicates that these aging models might generate ambiguous predictions and need to be updated by learning from drifting applications. The intuition is that drifting applications usually include API changes, which reflect application evolvement and Android evolvement.

Furthermore, instead of requiring true labels of applications, DroidEvolver uses pseudo labels generated by the model pool along with the corresponding drifting applications to update aging models in order to improve predictions in the future. Specially, when aging models are identified from the model pool, these aging models are not allowed to perform classification for application. The model pool then generates the final detection results through weighted voting by young models. For drifting applications, the detection results then used as pseudo label for aging model update. If no aging model is identified, all models contribute to the final detection result without further update.

To achieve effective and scalable detection over time, the aging model needs to automatically and quickly adapt to new changes in drifting applications. To this end, all detection models included in the model pool are initialized and updated with online learning algorithms, which perform incremental training over streaming data in a sequential manner. In contrast to batch learning algorithms (e.g., SVM, Decision Trees), online learning algorithms are not only more efficient and scalable, but also able to avoid expensive retraining cost when handling new labeled data. Furthermore, with the ability of learning from drifting applications, DroidEvolver fine-tunes itself during detection and automatically adapts to new changes over time.

To provide detailed evaluation, we carefully build a dataset, including 34,722 malicious applications and 33,294 benign applications, which has fairly balanced ratio between malicious applications and benign applications from 2011 to 2016.

To show the advantage of the whole system, we evaluate the detection performance of DroidEvolver and compare it with a state-of-the-art detection system MA-

MADROID [76]. MAMADROID builds a behavioral model by abstracting API calls to their packages or families in order to maintain resilience to API changes. Although MAMADROID makes efforts to achieve detection over time, it requires Markov chain modeling and complicated feature extraction which is conducted in static analysis such as those performed in Soot [107] and FlowDroid [11]. Furthermore, expensive retraining with manually labeled applications is also required to update MAMADROID in order to detect new malware.

The comparison is evaluated from two perspectives. First of all, we test the ability of detecting unknown malware from applications which are published in the same year (i.e., when DroidEvolver and MAMADROID are trained and tested with applications from same year). The detection accuracies of DroidEvolver are in the range between 95.19% and 96.18% from 2011 to 2016, while MAMADROID's accuracies are in the range between 80.04% and 85.49%. More importantly, we evaluate the ability of performing effective detection over the years (i.e., when DroidEvover and MAMADROID are trained with older applications and tested with newer ones). As shown in the experimental results, the accuracy of DroidEvolver drops from 95.28% to 86.87% over five years (i.e., trained with applications from 2011 and tested with applications from 2016), while the accuracy of MAMADROID drops from 80.04% to 58.03% over these years. The experimental results demonstrate that DroidEvolver significantly outperforms MAMADROID without complicated process.

DroidEvolver is not only effective, but also scalable in both training and detection. The training time required by DroidEvolver linearly varies from 3 seconds to 27 second when dataset size increases from 10,000 to 50,000. Under the same experimental setting, MAMADROID requires 26 seconds and 1,207 seconds to finish training on 10,000 and 50,000 applications, respectively. Furthermore, DroidEvolver takes on average 1.37 seconds to process an unknown application to generate classification results and update itself if certain standards are satisfied, while MAMADROID requires 39.15 seconds to classify an unknown application.

5.2 Overview of DroidEvolver

As depicted in Figure 5.1, DroidEvolver consists of two phases to achieve detection over time. In the ***Initialization Phase***, DroidEvolver extracts features from applications with true labels (i.e., benign or malicious) to initialize the feature set and the model pool. Specially, the model pool is initialized by learning from these labeled applications using several online learning algorithms. Each online learning algorithm is applied to initialize one detection model. Both Feature Set and model pool are transmitted to the next phase for classification and evolvement. In the ***Detection Phase***, DroidEvolver generates a Feature Vector for each unknown application being detected, and feeds the Feature Vector into the model pool, which outputs whether the detected application is benign or malicious. For applications and models that satisfy update policy, DroidEvolver automatically updates models and corresponding feature sets without requiring true labels of applications, and applies the updated model pool and feature sets to perform classification for new applications.

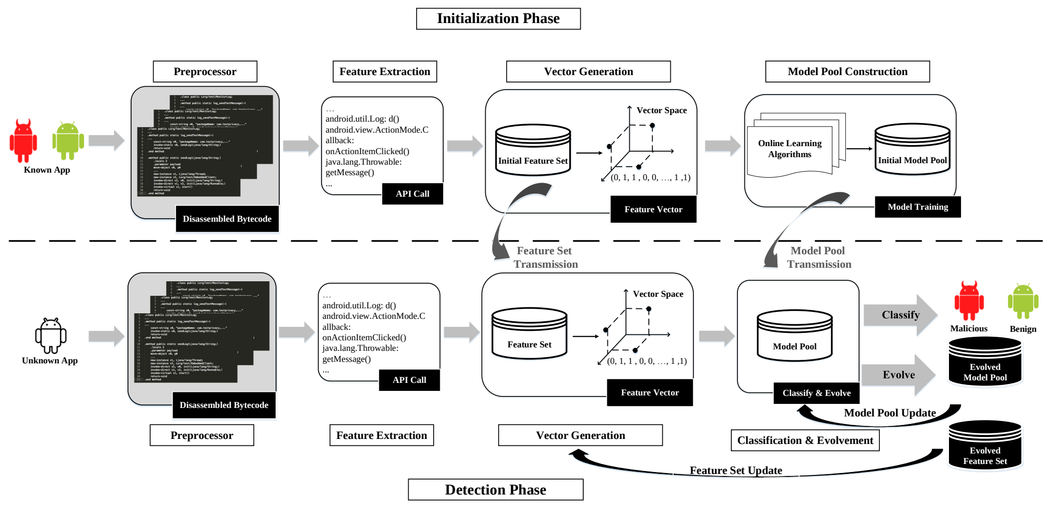


Figure 5.1: Overview of DroidEvolver System

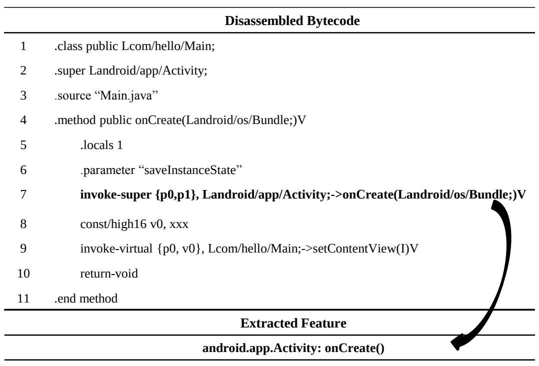
5.3 Initialization Phase: Initialize with Labeled Applications

In the Initialization Phase, DroidEvolver initializes feature set and model pool using training dataset. The Initialization Phase consists of four modules, including Preprocessor, Feature Extraction, Vector Generation, and model pool Construction.

5.3.1 Preprocessor

In Preprocessor, DroidEvolver applies apktool [114] to decompile apk files of application and obtain disassembled dex bytecode. Bytecode provides information about API calls and data used in an application.

5.3.2 Feature Extraction



Disassembled Bytecode

```
.class public Lcom/hello/Main;
.super Landroid/app/Activity;
.source "Main.java"
.method public onCreate(Landroid/os/Bundle;)V
    .locals 1
    .parameter "saveInstanceState"
    invoke-super {p0,p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V
    const/high16 v0, xxx
    invoke-virtual {p0, v0}, Lcom/hello/Main;->setContentView(I)V
    return-void
.end method
```

Extracted Feature

android.app.Activity: onCreate()

Figure 5.2: Example of Input and Output of Feature Extraction

In Feature Extraction, DroidEvolver performs lightweight process to extract raw Android API calls from disassembled bytecode. Specifically, for every application includes API call from Android families, the included API call will be extracted by DroidEvolver and treated as a detection feature. As the example shown in Figure 5.2, from a piece of disassembled bytecode, DroidEvolver extracts one unique feature `android.app.Activity: onCreate()`.

DroidEvolver extracts Android API calls as detection features for the following reasons. First of all, Android API calls allow application to access system resources and provide valuable information about application behaviors. In addition, although Android API packages and classes change frequently in different Android versions, API families are consistent. For example, at API level 1 (Android version 1.0 released in October 2008), Android includes 96 packages. The package number increases to 196 at API level 27 (Android version 8.1 released in November 2017). However, in different Android versions, API package and class changes do not affect API families, which include `android, java, javax, junit, apache, json, dom, xml` corresponding to android.*, java.*, javax.*, org.apache.*, org.json.*, org.w3c.dom.* and org.xml.* packages. Extracting features according to family list instead of existing package list helps DroidEvolver quickly adapt system changes. Last but not least, the feature extraction process is lightweight and efficient without requiring complicated static analysis and/or dynamic analysis such as those performed in Soot [107], FlowDroid [11], Epicc [82] and TaintDroid [39].

For all features extracted from training applications (including both benign and malicious applications), DroidEvolver stores them in the initial feature set. Specially, although the feature set is fixed in the Initialization Phase and is shared by all detection models in model pool, it will be automatically and separately updated to include new features extracted from unknown applications for each detection model during detection. As a result, the feature set is not fixed and the size of feature set is not bounded in the Detection Phase.

5.3.3 Vector Generation

Next, DroidEvolver represents each application in training dataset as a feature vector. To this end, DroidEvolver defines a vector space according to the initial feature set. If the size of initial feature set is X, DroidEvolver defines an $X-$dimension vector space accordingly. For each training application, DroidEvolver represents it with a feature vector by mapping its extracted features to the $X-$dimension vector space, such that for each feature the respective dimension is set to one and all other dimensions are zeros.

5.3.4 Model Pool Construction

In this module, DroidEvolver initializes a model pool by learning from the feature vectors of labeled applications applying several different online learning algorithms.

DroidEvolver utilizes the model pool instead of a single detection model for the following reasons. First of all, a single detection model may not be able to provide accurate responses for some observations due to its limited capability [104]. The model pool discards single classifier's bias and generates trustworthy predictions. In addition, unlike traditional online learning models which require true labels of applications for further update, DroidEvolver relies on the pseudo labels generated by the model pool for self-evolve. The model pool contains several detection models initialzed with different online learning algorithms to generate reliable pseudo labels and direct DroidEvolver to evolve towards the right direction.

Online Learning Algorithms

Online learning algorithms operate sequentially to process one sample at a time. The standard process of applying online learning for binary linear classifications is described as following. Consider $\{(x_t, y_t)|t \in [1, T]\}$ be a sequence of training data example, where $x_t \in R^d$ is a d-dimension vector and $y_t \in \{+1, -1\}$. At each time step t, the algorithm receives an incoming sample x_t and then predicts its class label $\hat{y}_t$:

$$\hat{y}_t = sgn(w.x_t)$$

Afterwards, the true label y_t is revealed and the learning algorithm suffers a loss

$l_t(y_t, \hat{y}_t)$. The hinge loss and logistic loss are commonly used for binary classification:

$$l_t(y_t, \hat{y}_t) = \begin{cases} max(0, 1 - y_t \cdot \hat{y}_t) & HingeLoss \\ log(1 + e^{-y_t \cdot \hat{y}_t}) & LogisticLoss \end{cases}$$

At the end of each learning step, the online learning algorithm decides when and how to update the model. The update function is often dependent on gradient of the weight vector w with respect to loss.

DroidEvolver applies LIBOL [57], a library for online learning algorithms, which implements both first order learning [96] [30] and second order learning [37] [112] as shown in Table 5.1. Specifically, first order learning only exploits the first order information (i.e., mean of weight vector) and adopts the same learning rate for all features, while second order learning aims to dynamically incorporate knowledge of observed data in earlies iteration to perform more informative gradient-based learning.

Model Pool Initialization

Given the feature vectors generated from labeled applications (both benign and malicious applications) as input, DroidEvolver applies online learning algorithms mentioned above and outputs the initial model pool. These initialized detection models are linear classifiers which fit linear decision boundary (i.e., hyperplane) between benign applications and malware. Every model is a weight vector, $w = [w^1, w^2, ..., w^d]$, which indicates the weight (i.e., relative importance) of each of the features used to generate prediction labels. The initial model pool is then transmitted to the Detection Phase for classification and evolvement.

Table 5.1: Online Learning Algorithms

First Order Learning	Perceptron [96]	Update whenever a mistake is made by the model in each iteration
	OGD [133]	Update the weight vector by applying the Gradient Descent principle
	PA [30]	Update the model by correcting the prediction mistake and avoid the new model deviating too much from the existing one
	ALMA [48]	A large margin variant of the Perceptron algorithm
	RDA [119]	Exploit the regularization structure more effectively in a stochastic online setting
Second Order Learning	SOP [22]	Perform prediction using the current instance in the correlation matrix
	CW [37]	Exploit confidence of weights when making updates in online learning processes
	ECCW [31]	A variant of CW by solving CW's constraint
	AROW [32]	Perform adaptive regularization of the prediction function upon seeing each new instance and perform especially well in the presence of label noise
	Ada-FOBOS [38]	Incorporate knowledge of the geometry of the data observed in earlies iterations to perform more informative gradient-based learning
	Ada-RDA [38]	Modify the proximal function adaptively at each iteration

5.4 Detection Phase: Classify and Learn from Unknown Applications

In the Detection Phase, DroidEvolver performs classification for unknown applications and automatically updates itself to adapt to new features discovered from unknown applications. The Detection Phase consists of four modules, including Preprocessor, Feature Extraction, Vector Generation, and Classification and Evolvement. Since the process of the first three modules follows the same steps described in Section 5.3, this section focuses on the last module Classification and Evolvement, which includes Drifting Application Identification, Classification and Pseudo Label Generation, and System Update.

5.4.1 Drifting Application Identification: When to Evolve

Models that are built through training on older applications often make poor and ambiguous decisions when faced with modern applications (i.e., applications with new features). This phenomenon commonly known as *concept drift*. In order to achieve effective detection over time, it is important to identify which application is "modern" application and when the model shows signs of cannot handling "modern" application. In this work, DroidEvolver identifies applications that are different from older applications as *drifting applications*. Furthermore, DroidEvolver identifies the model shows signs of failing to represent drifting applications as *aging model*. The identification of drifting applications and aging models indicates update is required to maintain effectiveness of the detection system.

DroidEvolver firstly measures the difference between a new application and a group of processed applications with same prediction label (i.e., benign or malicious) using an indicator. This indicator is called *Juvenilizing Indicator* (*JI*) in this work. Let X_i be the Feature Vector of i^{th} unknown application A_i given to DroidEvolver, M_j be the j^{th} detection model in current model pool and W_j be the weight matrix of M_j, the detailed operations of generating ξ_{ij} for each X_i and M_j are as follows:

App Buffer Construction

Since JI value calculation involves the comparison of unknown application and other processed applications, and DroidEvolver is designed to handle large scale of applications, it is expensive to generate JI value by comparing unknown application with every processed application. As a result, DroidEvolver constructs an App Buffer to store a subset of processed applications in order to not only speed up the process, but also to be timely and relevant up to date. When App Buffer does not exist at the beginning during detection (which means X_i is the first unknown Feature Vector given to M_j), DroidEvolver initializes an App Buffer of size k, $B = \{B_1, B_2, ..., B_k\}$, by randomly selects k Feature Vectors from training dataset. To keep the existing App Buffer up-to-date, DroidEvolver randomly replaces one Feature Vector in App Buffer each time when given a new unknown Feature Vector X_i to maintain a functional App Buffer $B = \{X_{i-1}, B_2, ..., X_i, ..., B_k\}$.

Distance Calculation

As illustrated in Section 5.3.4, M_j performs classification by computing a hyperplane represented by W_j. DroidEvolver calculates the distance from each object of App Buffer to the hyperplane by $B_t \times W_j$, where $B_t \in B$. DroidEvolver then generates a Distance Set $D = \{D_{X_{i-1}}, D_{B2}, ..., D_{X_i}, D_{Bk}\}$. Specially, M_j will predicts applications as malicious if the distance is larger than zero and predicts the ones as benign if the distance is less than zero.

JI Value Generation

It is inappropriate to compare new unknown application with processed applications without considering classes (i.e., malicious or benign). Due to the different functionalities performed by malware and benign applications, new malicious application might be similar with known malicious applications while significantly different from known benign applications. By taking prediction class into consideration, the computation of ξ_{ij} for X_i by M_j is shown in Equation (1). Here σ is an indicator function, where $\sigma(true)$ equals 1 and $\sigma(false)$ equals 0.

$$\xi_{ij} = \begin{cases} \frac{\sum\limits_{B_t \in B} \sigma(x_i \times w_j \geq B_t \times w_j)}{\sum\limits_{B_t \in B} \sigma(B_t \times w_j \geq 0)} & \text{if } x_i \times w_j \geq 0 \\ \\ \frac{\sum\limits_{B_t \in B} \sigma(x_i \times w_j < B_t \times w_j)}{\sum\limits_{B_t \in B} \sigma(B_t \times w_j < 0)} & \text{else} \end{cases} \tag{5.1}$$

For each unknown application A_i, DroidEvolver calculates the JI value corresponding to every model in the model pool and generates a JI Value Set $\{\xi_{i1}, \xi_{i2}, ..., \xi_{im}\}$ by following the operations described above.

It should be noted that the JI value is generated by calculating the distance of a application from the model hyperplane since the models in the model pool perform classification based on a hyperplane. For other classification algorithms, the calculation of JI might be different in order to adjust to the classification algorithms.

An unknown application is different from other processed applications if it is too close or too far away from the hyperplane. The difference induces JI values which are too large or too small (i.e., outliers). As a result, DroidEvolver employs two thresholds, τ_0 and τ_1, to identify abnormal JI value. In this work, if a JI value is in range of [τ_0, τ_1], the value is identified as valid; otherwise, it is identified as invalid. The threshold values are chosen during Initialization Phase by computing on the training dataset and are enforced on identification in Detection Phase.

For a new application A_i, it is identified as drifting application if its JI value set $\{\xi_{i1}, \xi_{i2}, ..., \xi_{im}\}$ includes invalid value. Certain model that corresponds to the invalid JI value is identified as aging model accordingly.

5.4.2 Unknown Application Classification & Pseudo Label Generation: What to Evolve With

For each new unknown application, the classification result is generated by detection models included in the model pool through weighted voting. Specially, if the unknown application has been identified as drifting application in the first step, its classification is performed by excluding aging models. The reason is that aging models are the ones that cannot represent the drifting application and may make unreliable prediction results. Furthermore, the prediction label of drifting application is also used as pseudo label for update. In the case that all models are identified as aging models, DroidEvolver produces a prediction label for drifting application through weighted voting of all aging models and skips the update step.

5.4.3 Aging Model Juvenilization: How to Evolve

The identification of drifting applications and aging models indicates that aging models should be juvenilized for the following reasons. First of all, the differences between drifting application and known applications may be caused by new features or new patterns included in drifting application. It is thus important to include these new features in order to keep the whole system up-to-date. In addition, the detection capabilities of aging models are constrained by their “aging” feature set and model structure. Aging models need update to keep functional in the future.

For each identified drifting application, DroidEvolver updates the identified aging model separately.

Feature Set

DroidEvolver automatically updates the current feature set of each aging model by including new features extracted from drifting applications. These new features usually indicates the trend of application evolvement and Android system evolvement.

Model Structure

DroidEvolver then updates the model structure of each aging model using drifting application and corresponding pseudo label as a new training sample. As described in Section 5.3.4, after aging model generating prediction label $\hat{y}_t$ for drifting application x_t, the pseudo label y_{t_p} is revealed and the classifier suffers a loss $l_t(y_{t_p}, \hat{y}_t)$. At the end of update process, the online learning algorithm (which is used to initialize the aging model) decides when and how to update the aging model in order to improve prediction in the future. By learning from drifting application, DroidEvolver adjusts the model structure of aging model to adapt to new patterns observed from drifting application.

The dynamic updating manner of both feature set and model structure helps DroidEvolver rapidly capture changes in both applications and Android system. With the ability of adapting itself to new changes and trends, DroidEvolver achieves detection overtime.

5.5 Experimental Settings and Parameter Tunning

DroidEvolver's detection performance is empirically evaluated with a series of experiments. We describe experimental settings in this section and report the evaluation results along with detailed analysis in Section 5.6.

5.5.1 Data Collection

We built an initial benign dataset and malicious dataset by crawling applications from an open Android application collection project [4]. To exclude potential malware from initial benign dataset, we sent each application to VirusTotal, which is an antivirus service with over sixty antivirus scanners. We discard applications from the initial benign dataset if any anti-virus scanner raises alarm for it so as to generate the benign dataset, which consists of 33,294 benign applications in total. In addition, each application in initial malicious dataset was also sent to VirusTotal and was excluded from the dataset if it received alarms from less than 15 anti-virus scanners out of 63 scanners in VirusTotal. The way we clean up our dataset is consistent with previous research on malware detection. According to Roy et al. [97], malicious samples that only receive one alarm from VirusTotal will be considered as "low quality" and those receive more than 10 alarms out of 54 scanners will be considered as "high quality".

More importantly, the built dataset was carefully collected to cover different date (from 2011 to 2016) in order to fairly evaluate DroidEvolver's detection performance over time (i.e., DroidEvolver should be trained with old applications and be evaluated with new applications). The date of each application is determined by the date of packaging APK file, which is included in the dex file of APK file [85]. The dataset distribution over different years is described in Table 5.2.

Table 5.2: Data Distribution over Different Years

Year	2011	2012	2013	2014	2015	2016	Total
Benign	4,414	5,789	5,784	5,793	5,750	5,764	33,294
Malicious	6,063	5,777	5,685	5,760	5,657	5,780	34,722
Total	10,477	11,566	11,469	11,553	11,407	11,544	68,016

5.5.2 Metrics and Measurements

We evaluate the performance of DroidEvolver using three metrics, True Positive Rate (TPR), False Positive Rate (FPR) and Accuracy, where TPR is the percentage of malware being detected correctly, FPR is the percentage of benign applications being detected as malware, and Accuracy is the percentage of all applications being classified correctly in our experiments. We conduct a series of experiments to measure the performance of DroidEvolver from two perspectives, including detection in same time period and detection over time.

Performance in Same Time Period

Since parameters have significant influences on DroidEvolver's performance, it is inapproapriate using one testing dataset for both parameter selection and evluation. This is because the trained model could be overfitted and might perform poorly on other new datasets besides certain testing dataset. Simply using ten-fold cross validation is not applicable in our experiments.

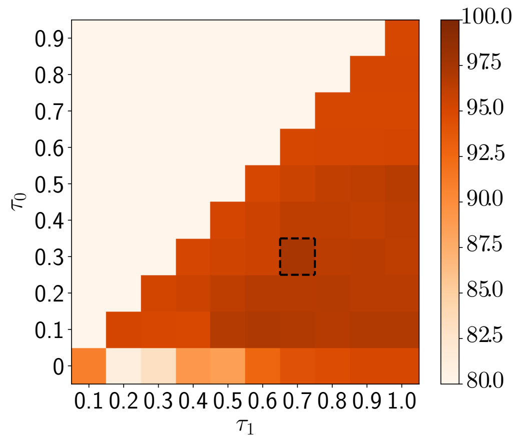
To prevent over-fitting and provide fair experimental setting, we follow the standard dataset division process when parameter tunning is involved in training. The dataset developed in the same period (i.e., 2011, 2011-2012, 2011-2013, 2011-2014, 2011-2015) is shuffled and divided into five equal-size subsets. We randomly choose three subsets as training set, one subset as validation set, and the remaining subset as testing set. Validation set is used for parameter selection. For each time period, certain parameter values are selected when DroidEvolver initialized with the training set achieves best performance on the validation set. After that, DroidEvolver is initialized with the training set and applies the selected parameters for detection. The testing set is then used to evaluate the detection performance of DroidEvolver.

Performance over Time

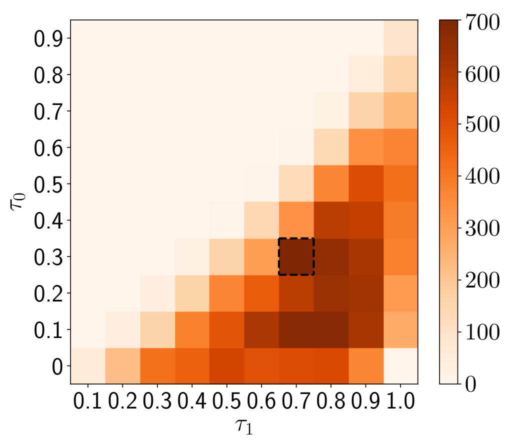
DroidEvolver's detection performance over time is evaluated in a series of experiments in which the model pool is initialized with the training set and the selected parameters in one time period and tested with all applications developed in later time periods.

5.5.3 Parameter Tunning

We now present the tunning process of parameters which have influences on the performance of DroidEvolver. It should be demonstrated that all parameter values are chosen during Initialization Phase by computing on the training set and the



(a) Detection Accuracy under Different Threshold Settings



(b) Number of Drifting Applications under Different Threshold Settings

Figure 5.3: The Effects of Threshold Values on Detection Accuracy and Identified Drifting Applications

validation set, and are enforced in Detection Phase. Since DroidEvolver has several training sets for different time periods, the parameter values corresponding to each training set are selected separately by DroidEvolver in Initialization Phase. In this section, we choose the experimental setting in which applications developed in 2011 are treated as the training set and the validation set to explain the parameter tunning process.

Threshold Tunning

Thresholds, τ_0 and τ_1, are critical to identify drifting applications and aging models, and thus have significant influences on the detection performance of DroidEvolver. In our experiments, the threshold values are set empirically in such a way that DroidEvolver initialized with training set achieves maximize detection accuracy on validation set.

In Figure 5.3a, we present the detection accuracy of DroidEvolver under different threshold settings, where $0 \leq \tau_0 < \tau_1 \leq 1$. The detection accuracy is stable when τ_0 changes from 0.1 to 0.3, and τ_1 changes from 0.6 to 0.8. The detection accuracy reaches its maximum value 96.1% when $\tau_0 = 0.3$ and $\tau_1 = 0.7$.

To better understand the effects of thresholds on detection performance, we also present the number of identified drifting applications under different threshold settings in Figure 5.3b. Small difference between τ_0 and τ_1 induces to a small portion of applications from the validation set being identified as drifting applications. As a result, detection models from the model pool keep a slow update speed and may generate unreliable classification results for unknown applications. As shown in Figure 5.3, DroidEvolver achieves high detection accuracy when a large number of drifting applications are identified.

Learning from drifting applications helps DroidEvolver capture new features and new trends caused by application and Android system update. In addition, updating aging models online from pseudo labels that are generated by weighted voting help DroidEvolver adapt to the captured new features and new trends. As a result, DroidEvolver achieves high accuracy and high scalability in malware detection over time.

Buffer Size Tunning

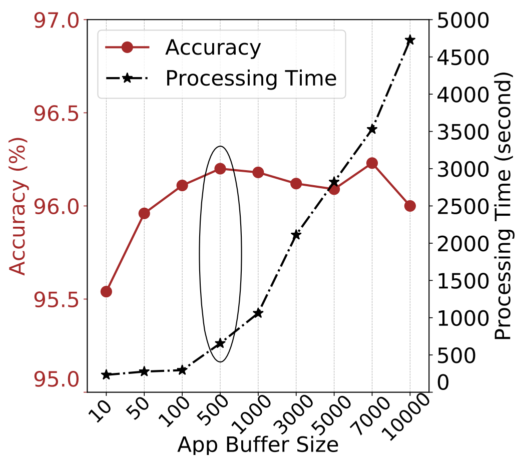


Figure 5.4: Detection Accuracy and Evolving Duration of DroidEvolver under Different Buffer Size Settings

App Buffer B is introduced to avoid comparing each unknown application with all processed applications. The size of App Buffer has influences on identification of drifting applications and aging models, and the time required to classify applications included in the validation set. In Figure 5.4, we present detection accuracy and process time of DroidEvolver under different buffer size settings for a same testing set. With the threshold values set to [0.3, 0.7], the detection accuracy of DroidEvolver slightly varies from 95.5% to 96.2% when the buffer size increases from 10 to 10,000. As shown in the experimental results, the effects of buffer size on detection accuracy is not as significant as threshold values.

Nevertheless, since it is efficient for DroidEvolver to learn from drifting applications and update drifting models accordingly, the runtime performance bottleneck in Classification and Evolvement is at the calculation of ξ, which is mainly determined by buffer size. As shown in the our results, the time required to classify 2,000 unknown applications in Classification and Evolvement rapidly increases from 200 seconds to 4,759 seconds when buffer size varies from 10 to 10,000.

To maintain efficiency and effectiveness at the same time, DroidEvolver selects the buffer size that results in highest detection accuracy with relatively low process time. In this setting, the buffer size is set to 500.

5.6 Evaluation and Analysis

In this section, we evaluate the performance of DroidEvolver with detailed analysis using the dataset summarized in Table 5.2.

5.6.1 Detection in Same Time Period

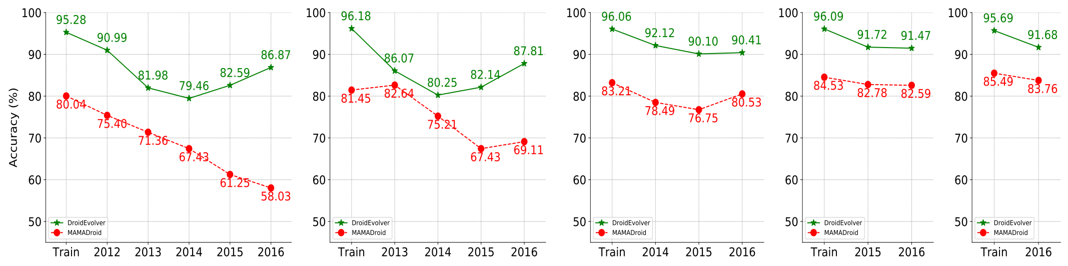
We first evaluate the detection performance of DroidEvolver and compare it with a state-of-the-art research work MAMADROID [76], which is resilient to the changes of Android framework and maintains effective after running for a long period of time. In MAMADROID, it first extracts API call sequences from the call graphs built by Soot and FlowDroid for each application, and abstracts APIs in each API sequence to their corresponding packages. Next, MAMADROID uses Markov chains to model application behaviors by evaluating every transition between API calls, and uses the probabilities of transitioning from one abstracted API call to another in the Markov chain as the feature vector. After that, MAMADROID builds a detection model using two-class classification algorithms (e.g., SVM, Decision Trees and Random Forest) along with the feature vectors obtained from training set. By abstracting API calls to their packages or families, MAMADROID maintains resilient to API changes and detects malware with high accuracy. In this work, we re-implement MAMADROID using its source code and evaluate both DroidEvolver and MAMADROID from different perspectives using the same dataset.

Table 5.3: Performance of DroidEvolver and MAMADROID in Same Time Period

	[ACC, TPR, FPR]%								
Year (App #)	2011 (10,477)			2011-2012 (22,043)			2011-2013 (33,512)		
DroidEvolver	95.28	91.51	1.98	96.18	94.72	2.57	96.06	96.10	3.98
MAMADROID	80.04	78.41	18.82	81.45	80.10	18.34	83.21	83.41	16.94
Year (App #)	2011-2014 (45,065)			2011-2015 (56,472)			2011-2016 (68,016)		
DroidEvolver	96.09	95.16	3.05	95.69	95.46	4.09	95.19	94.61	4.27
MAMADROID	84.53	84.41	15.36	85.49	86.07	15.02	85.39	84.51	13.78

We start our evaluation by measuring how well DroidEvolver and MAMADROID detect malware by training and testing using applications that are developed in the same time period. As shown in Table 5.3, DroidEvolver significantly outperforms MAMADROID by achieving on average 14.88% higher accuracy, 14.13% higher true positive rate, and 15.66% lower false positive rate.

In addition, MAMADROID's accuracy improves from 80.04% to 85.39% when the dataset increases from 10,477 to 68,016, while the performance of DroidEvolver is stable over different datasets. One reason is that larger dataset help MAMADROID correctly capture the probabilities of transitioning from one abstracted API call to another in the Markov chain while building the feature vectors for classification. The feature vectors with higher qualities result in better detection performance of MAMADROID. However, DroidEvolver does not rely on large scale of training dataset to build mature detection model and perform effective detection.



(a) Train 2011 (b) Train 20112012 (c) Train 20112013 (d) Train 20112014 (e) Train 20112015

Figure 5.5: Yearly Performances of DroidEvolver and MAMADROID

5.6.2 Detection over Time

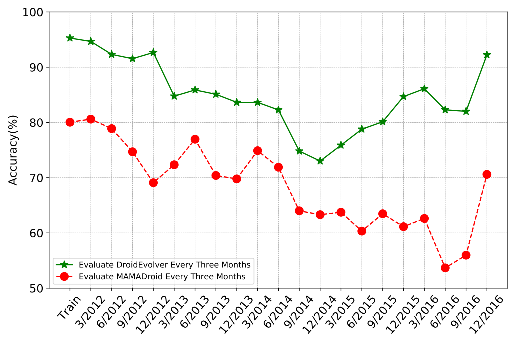
We then evaluate the performance of DroidEvolver and MAMADROID when they are trained with applications developed in one time period and tested with other applications developed in later time periods.

Coarse Granularity

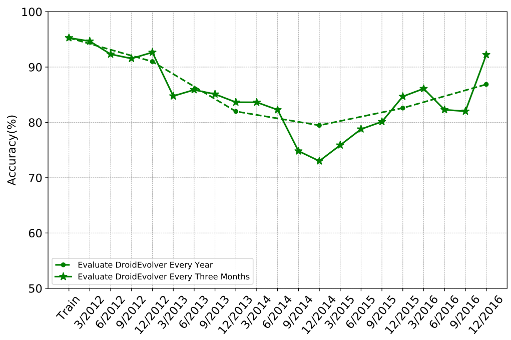
The performance of DroidEvolver and MAMADROID are evaluated on yearly basis. As shown in Figure 5.5, the detection performance of DroidEvolver is significantly and consistently better than MAMADROID in all experiments.

When DroidEvolver is evaluated on testing sets that are newer than training sets by one to five years, the average accuracy of DroidEvolver is 90.52%, 85.95%, 84.00%, 85.2% and 86.87%, respectively. In comparison, the average accuracy of MAMADROID in the corresponding cases is 80.63%, 76.48%, 71.80%, 65.18% and 58.03%, respectively. The detection accuracy of DroidEvolver is higher than MAMADROID by 16.08% on average when they are evaluated with same datasets. In addition, DroidEvolver declines by 1.68% in terms of detection accuracy per year on average over five years, while MAMADROID declines by 4.40% in the same case.

More importantly, after running for one or two years, the detection accuracy of DroidEvolver is stable and even increases instead of declining. The reason is that DroidEvolver automatically learns from newer applications to update the feature



(a) Quarterly Performances of DroidEvolver and MAMADROID



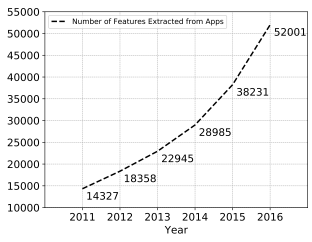
(b) Yearly and Quarterly Performance of DroidEvolver

Figure 5.6: Detection Performance of DroidEvolver and MAMADROID with Different Granularity

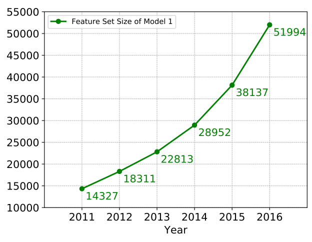
sets and model structures of aging models and keeps the whole system effective. DroidEvolver not only applies unknown applications as testing sets, but also uses some of them (i.e., drifting applications identified from unknown applications) as new training samples without true labels. By using the prediction labels of drifting applications as pseudo labels, DroidEvolver has more and more training samples to fine-tune its detection models while performing detection at the same time. In other words, unlike traditional learning-based detection systems, DroidEvover does not rely on large training set to build mature detection models in training phase. As a result, even though the detection models of DroidEvolver are initiliazed with small dataset (e.g., in Figure 5.5a detection models are initialized with 6,286 applications developed in 2011)), DroidEvolver keeps updating its detection models to capture the new trends in applications and Android framework, and performs effective detection by itself.

Fine Granularity

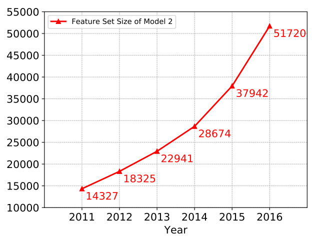
Both DroidEvolver and MAMADROID are then evaluated every three months after being trained with applications developed in 2011. We focus on this challenging case in which the "oldest applications" (i.e., applications developed in 2011) in the whole dataset are used for training.



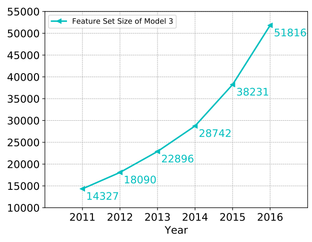
(a) Feature Included in Applications



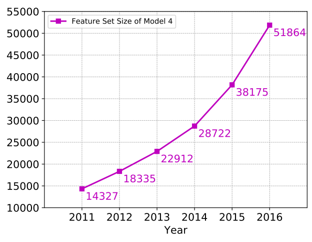
(b) Feature Set Size of Model 1



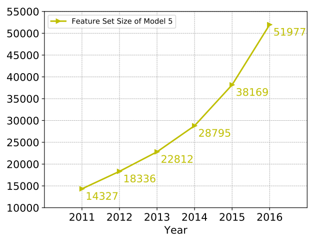
(c) Feature Set Size of Model 2



(d) Feature Set Size of Model 3



(e) Feature Set Size of Model 4



(f) Feature Set Size of Model 5

Figure 5.7: The Real Number of Features Extracted from Applications and The Number of Feature Automatically Learned by Detection Models

Figure 5.6a shows that DroidEvolver significantly outperforms MAMADROID in all quarters. The ups and downs are mainly caused by fluctuation in small sample space. As shown in Figure 5.6b, since the trends in coarse granularity and fine granularity are similar, we only show the yearly results (i.e., coarse granularity) in the rest of work.

5.6.3 Feature Evolvement

A major advantage of DroidEvolver is that its feature set is not bounded as it dynamically includes new features from unknown applications during detection. This advantage helps DroidEvolver keep evolving with application and Android system evolvement. As shown in Figure 5.7, the number of extracted features from applications increases rapidly from 14,327 to 52,001 in six years. We then randomly select five detection models from the model pool. These models are trained with 2011 applications and tested with unknown applications collected from 2012 to 2016. We now present the feature set size for each model in different years.

Although detection models are trained and tested with same dataset, their feature set sizes are different. The main reason is that DroidEvolver identifies aging model and updates aging model's feature set and model structure individually in Detection Phase. Another reason is that different detection models with different model structures are less likely to show signs of aging at the same time for same applications.

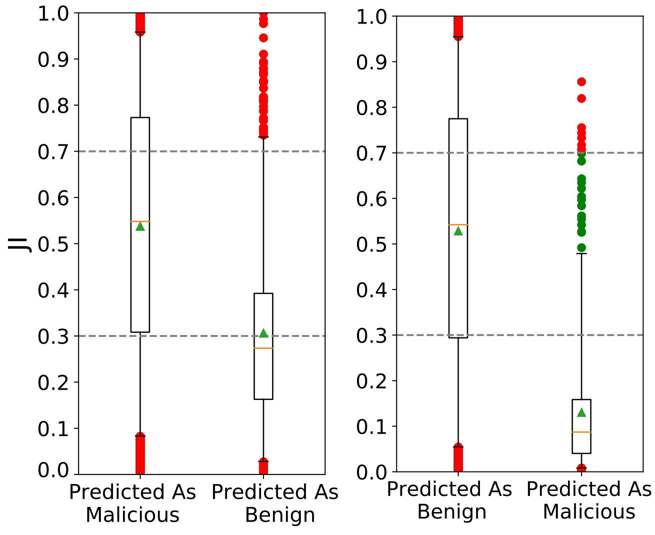
The size of feature set expanding over time demonstrates that DroidEvolver is capable of updating its models and feature sets to catch up the rapid evolution of applications and Android framework.

5.6.4 Importance of Drifting Application Identification

We now analyse the JI value distribution of unknown applications to demonstrate the importance of identifying drifting application and aging model before performing classification. In the experiment, we use the training set developed in 2011 as training set, applications developed in 2012 as testing set, and [0.3, 0.7] selected by the validation set of 2011 as threshold values. Since it is difficult to analyse JI value distribution corresponding to all detection models included in the model pool, we randomly choose one detection model as the example to perform detailed analysis.

Figure 5.8 shows a box and whisker plot for malicious applications and benign applications. The box extends from the lower to upper quartile values of the JI values, with a solid triangle at the mean and a line at the median. The whiskers extend from the box to show the $5th$ and the $95th$ percentiles of the JI values. Two grey baselines mark the τ_0 and τ_1, and each red filter point denotes a drifting application.

Figure 5.8a and Figure 5.8b show the JI value distribution of true malicious application and true benign applications, respectively. Most correct predictions (first column of Figure 5.8a and Fig 5.8b) are associated with valid JI values, while incorrect ones (second column of Figure 5.8a and Figure 5.8b) are associated with invalid



(a) True Malicious (b) True Benign

Figure 5.8: JI Distribution for One Detection Model

JI values. This indicates that detection model is able to make reliable classification when unknown application is similar as the applications has been processed. For the unknown applications that are very different from processed applications, it is highly possible that detection model will generate incorrect prediction. In other words, aging model is likely to misclassify drifting applications.

As a result, by identifying drifting applications and aging models, and excluding aging models from voting for final prediction label, DroidEvolver reaches its maximum ability of generating correct prediction labels. Furthermore, the identified drifting applications and generated reliable pseudo labels help DroidEvolver determine when to evolve and keep DroidEvolver evolving toward the right direction.

5.6.5 False Positives and False Negatives

Machine learning based malware detection solutions are usually black-box methods as they do not explain why a particular sample is classified as malicious or benign. In DroidEvolver, we address this shortcoming as follows: in addition to detection, DroidEvolver reports significant features (i.e., API calls) of an application that contribute to classification. In most cases, these significant features reveal the application's characteristics related to malicious or benign behaviors. In addition, every feature is associated with a value (known as weight) calculated by classification algorithm. According to processed applications and associated labels (both true labels and pseudo labels), the detection model gives positive values to malicious-relevant features and negative values to benign-relevant features.

The chosen experiment has the same settings as described in Section 5.6.4. With the system initialized by the training set developed in 2011, DroidEvolver achieves an accuracy of 90.99% when detecting malware from 11,566 applications developed in 2012, with a true positive rate of 87.52% and a false positive rate of 5.54%. Since the system is firstly initialized with a small dataset of 6,705 applications in total, the detection models are weak at the beginning of detection and produces reasonable detection performance.

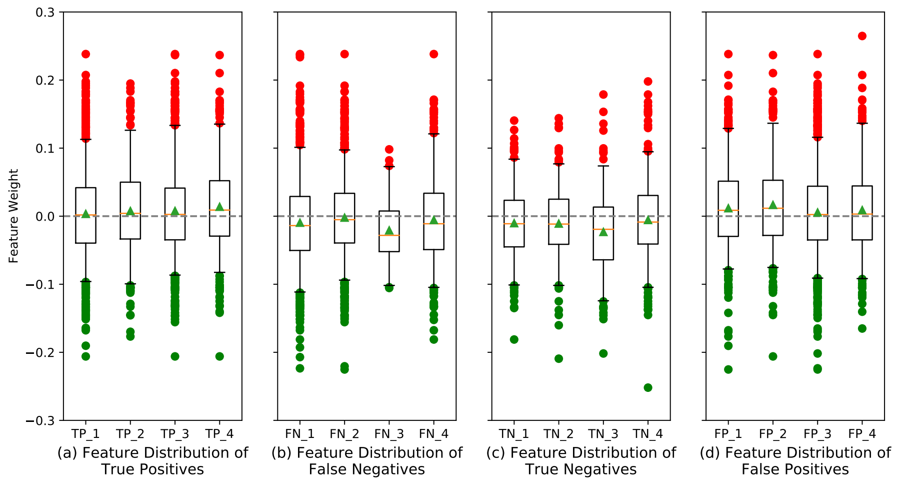


Figure 5.9: Feature Distribution of True Positives, False Negatives, True Negatives and False Positives

Feature Distribution

We first analyse the feature distribution of true positives, false negatives, true negatives and false positives. Fig 5.9 shows a box and whisker plot for 16 randomly selected applications. The box extends from the lower to upper quartile values of the feature weights. A grey baseline marks the zero value, each red filter point de-

Table 5.4: Analysis of Malicious Features Identified from False Positives

(a) Percentage of Malicious Features in All Included Features

Malicious Feature%	(40,45]	(45,50)	(50,55)	(55,60)	(60,65)	(65,70)	(70,75)
Percentage of FP	3.45%	55.17%	34.48%	3.45%	0%	0%	3.45%

(b) Percentage of Malicious Features in Top 100 Informative Features

Malicious Feature%	(50,60)	(60,70)	(70,80)	(80,90)	(90,100)
Percentage of FP	38.60%	38.60%	21.05%	1.75%	0%

notes a malicious-relevant feature and each green point denotes a benign-relevant feature.

The comparison of Figure 5.9a and Figure 5.9b indicates that DroidEvolver misclassifies malicious application with many features that are closely-related to benign behaviors. Similarly, the benign applications with many malicious-relevant features will be wrongly labeled as malicious as shown in Figure 5.9c and Figure 5.9d. More importantly, the feature distribution of false positives is more similar as true positives, rather than true negatives.

False Positives

From 5,789 benign application, DroidEvolver mistakenly predicts 321 of them as malicious. As shown in Table 5.4a, all of these false positives include more than 40% of malicious-relevant features among the total features extracted from their apk files. Although the true label of these false positives is benign, 41.38% of them include more malicious-relevant features than benign ones. In addition, we analyse the top 100 informative features that contribute to the classification. As shown in Table 5.4b, these false positives not only include more malicious-relevant features, they also include the ones that can determine the classification result of detection model. In conclusion, DroidEvolver's false positives are mainly benign applications that behave similar as malware.

Table 5.5: Analysis of Benign Features Identified from False Negatives

(a) Percentage of Benign Features in All Included Features

Benign Feature%	(40,45]	(45,50)	(50,55)	(55,60)	(60,65)	(65,70)	(70,75)
Percentage of FN	0%	0%	58.21%	34.33%	4.48%	0%	2.99%

(b) Percentage of Benign Features in Top 100 Informative Features

Benign Feature%	(50,60)	(60,70)	(70,80)	(80,90)	(90,100)
Percentage of FN	59.70%	34.33%	2.99%	2.99%	0%

False Negatives

DroidEvolver generates prediction labels for 5,777 malicious applications, while mistakenly labels 721 of them as benign. As shown in Table 5.5a, all of these false negatives include more malicious-relevant features than benign-relevant ones. 3.45% of them even include more than 70% malicious-relevant features. After analysing the top 100 most informative features than contribute to the classification result, all of these false negatives contains more malicious features in their top 100 features. Since the included features of false negatives are similar as the ones in true benign applications, it is difficult for DroidEvolver to correctly classify these false negatives by only looking to into their API usage.

5.6.6 Runtime Evaluation

As the above evaluation has shown the effectiveness of DroidEvolver, we now evaluate the computational overhead incurred by each module of Detection Phase in DroidEvolver. We run our experiments on a machine with 4×3.2 GHZ Intel-Core and 12 GB of RAM.

Runtime Performance of DroidEvolver

Table 5.6a shows the average time of processing an unknown application to generate detection result by DroidEvolver. The performance bottleneck is at the Preprocessor, which decompiles apk files of applications to get decompiled bytecode. It

Table 5.6: Average Time for Processing An Unknown Application to Generate Detection Result

(a) Runtime Evaluation of DroidEvolver

DroidEvolver	**Preprocessor**	**Feature Extraction**	**Vector Generation**	**Classification& Evolvement**	**Overall**
	1.1s	0.17s	0.05s	0.05s	1.37s

(b) Runtime Evaluation of MAMADROID

MAMADROID	**Call Sequence Abstraction**	**Feature Vector Extraction**	**Classification**	**Overall**
	37.29s	0.43s	0.0036s	39.15s

should be demonstrated that DroidEvolver takes only 0.05s on average to complete the complicated process in Classification and Evolvement, including Identification, Classification and Update. By using API calls that can be easily extracted as detection features and applying online learning algorithms to quickly update detection models, DroidEvolver takes on average 1.37s to accurately detect malware from unknown applications. As a result, DroidEvolver is suitable for large scale malware detection.

Runtime Performance of MAMADROID

We also evaluate the runtime performance of MAMADROID with same experimental settings. As shown in Table 5.6b, its first step, i.e., abstracting call graphs using Soot and FlowDroid and abstracting API calls to corresponding packages, takes 37.29s per application. In the second step of building the Markov chain model and constructing feature vectors, MAMADROID takes on average 0.43s. Finally, it takes on average 0.0036s to classify one feature vector into benign or malicious. In total, MAMAROID requires 39.15s to classify an unknown application, which is significantly slower than DroidEvolver.

Discussion

As shown in Table 5.6, DroidEvolver is significantly faster than MAMADROID in all modules except in Classification and Evolvement. Although classifying the feature vector of unknown application and updating aging model (i.e., 3.22×10^{-3}s on average) are lightweight, it is time-consuming to identify drifting application and aging model. Although DroidEvolver leverages App Buffer to strike a balance between effectiveness and efficiency, it still takes more time than other steps to calculate JI value for each unknown application, which requires comparison with all applications included in the App Buffer.

5.6.7 Scalability Evaluation

We now evaluate the scalability of DroidEvolver in both Initialization Phase and Detection Phase.

As shown in Figure 5.10, initializing the detection model pool with 10,000 applications takes about 3 seconds. The time required for initialization slightly increases from 3 seconds to 27 second when training size enlarges from 10,000 to 50,000. As a result, the process performed in Initialization Phase is efficient. In comparison, the time required by MAMADROID to train detection model significantly increases from 26 seconds to 1207 seconds under the same experimental settings. In addition, as evaluated in Section 5.6.6, it takes on average 1.37 second for DroidEvolver to process an unknown application from decompiling to classification.

In conclusion, our experiments show that DroidEvolver is efficient in both phases and is scalable enough to be deployed. The scalability is achieved by the following characteristics. First of all, DroidEvolver leverages online learning algorithms [57] which update the aging model instantly whenever drifting application is identified instead of learning from a collection of instances in a batch fashion. The simple update strategy of online learning algorithms avoids solving large-scale optimization problems, which makes DroidEvolver efficient and scalable to process applications which arrive sequentially. Moreover, since the applications are processed sequentially, the memory cost for online learning is greatly reduced and there is no retraining required. In addition, DroidEvolver applies extracted API calls as detection features, which can be easily retrieved from decompiled code without complicated process, such as those performed in Soot [107], FlowDroid [11], EPICC [41], and TaitDroid [39]. Last but not least, DroidEvolver applies App Buffer to speed up the process of calculating JI values without declining detection performance.

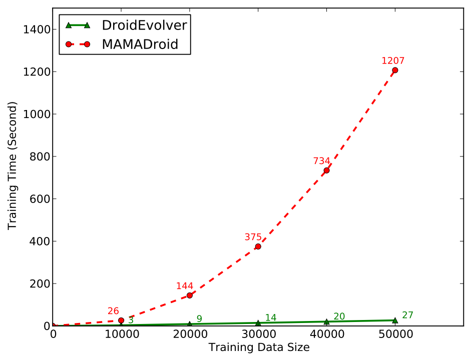


Figure 5.10: Scalability Evaluation of DroidEvolver and MAMADROID

5.6.8 Robustness Evaluation Against Typical Obfuscation Techniques

Obfuscated malware poses challenges to malware detection systems. Malware developers apply obfuscation techniques to manipulate detection systems and evade from detection by transforming malware in different forms but still with the same malicious behavior. For ease of use and without requiring comprehensive domain knowledge about malware detection and Android system, most malware developers apply typical obfuscation techniques that can be easily performed. As observed in [95], common malware transformation techniques (e.g., repackging, changing field names, and changing control-flow logic) could evade many existing commercial anti-malware tools.

However, DroidEvolver is resilient to these common evasion techniques. First of all, DroidEvolver does not rely on specific signing signatures or method names to detect malware. The simple program transformation techniques, such as resigning, repackaging and changing names, will not affect the detection model of DroidEvolver which is built according to system API calls. Another type of evasion technique is to insert junk code segments into source code. The newly inserted junk code segments include `goto`, `const/16`, `not` and `add-int`, which are not closely related to malicious behaviors. Furthermore, these junk code segments will not be extracted by DroidEvolver and therefore cannot affect the detection performance of DroidEvolver. In addition, other common obfuscation techniques that are designed to change control-flow logic also cannot evade from DroidEvolver since DroidEvolver does not rely on control-flow logic for detection.

To demonstrate the robustness of DroidEvolver, we apply DroidChameleon [95], a framework implementing eleven typical obfuscation techniques, to 100 malicious applications randomly selected from the malicious dataset developed in 2012. DroidChameleon generates 1,100 obfuscated malicious applications, which are then used as a new testing set to evaluate the robustness of DroidEvolver. To conduct fair evaluation, the model pool of DroidEvolver in this experiment is initialized with applications developed in 2011. As shown in the experimental results, DroidEvolver successfully detects 96% of obfuscated malicious applications. For every obfuscation technique, DroidEvolver misses four obfuscated malicious applications. After manually checking these obfuscated malware, we find that they are all the transformations of four malicious applications. Another interesting finding is that DroidEvolver cannot correctly classify these four malware even without obfuscation.

In summary, DroidEvolver is robust against common obfuscation techniques. More importantly, unlike other obfuscation resilient detection systems which require expensive process to generate data dependence graphs and information flows, DroidEvolver does not rely on complicated feature engineering.

5.7 Discussions

5.7.1 Limitations

As a detection system applying static analysis, DroidEvolver suffers from the inherent limitations from static analysis and might fail to detect malicious behaviors which are loaded and executed at runtime. Although DroidEvolver can capture triggers about such behaviors when `java.lang.refect` is used in source code, this limitation encourages us to integrate dynamic analysis to build the model pool in future versions of DroidEvolver.

5.7.2 Evasion

Next, we discuss possible evasion techniques and how they can be addressed. One straightforward evasion approach is that malware developers might attempt to confuse DroidEvolver and evade detection by naming their self-defined API calls in such a way to make them look similar as system API calls. Since DroidEvolver automatically makes necessary update to its feature sets without any human instructions, such attacks can not be prevent by manually whitelisting the list of legitimate system APIs. However, it is challenging to successfully perform such attacks to evade DroidEvolver, which applies model pool to general final prediction result and is capable of self-evolving.

First of all, if the carefully crafted malware only includes limited number of self-defined "system" API calls, it is difficult to successfully fool all of the detection models in the model pool and induce them to generate wrong prediction. In addition, if the crafted malware includes many self-defined "system" API call, it is highly possible that this malware will be identified as drifting application. As a result, the detection model that the malware plans to manipulate will be identified as aging model and cannot vote to the final prediction result. Therefore, DroidEvolver can still generate correct classification label. Furthermore, given the correct classification label as the pseudo label (i.e., malicious) of the crafted malware, DroidEvolver automatically includes these self-defined "system" API calls into the feature sets of aging models and marks these new features as malicious (will be reflected in feature weight). Although the crafted malware might successfully induce DroidEvolver to adopt self-defined "system" API calls, this will not affect the detection of DroidEvolver and help DroidEvolver detect similar malware.

An attacker could also try to carefully craft malicious applications [54] to mislead detection models. These malicious applications are derived from regular applications by minor yet carefully selected perturbations [88] [51] that induce models into adversary-desired misclassifications, which is known as adversarial sample crafting. The success of adversarial sample crafting requires the knowledge of trained detection models before attack, which is challenging in our case since DroidEvolver's detection models are not consistent and change a lot during detection.

5.8 Related Work

Over the past few years, Android malware detection has attracted extensive attentions in both academia and industry. In this section, we mainly review learning-based mobile malware detection detection systems which are more relevant to DroidEvolver.

Detection Over Time. A small potion of existing works focus on detection over time in different manners compared with DroidEvolver. MAMADROID [76] builds a behavioral model, in the form of a Markov chain, from the sequence of abstracted API calls performed by an application, and uses it to extract features and perform classification. Although MAMADROID maintains resilience to API changes by abstracting API calls to their packages and families, expensive retraining and experts' involvement are required to keep the whole system effective, while DroidEvolver automatically updates itself to achieve both efficient and effective detection over long time periods. On the other hand, Transcend [63] proposes a framework to identify aging classification models during deployment before the detection model's performance starts to degrade. Unlike DroidEvolver which proposes a comprehensive framework from drifting application identification to system evolvement without true labels, Transcend mainly focuses the statistical metrics to identify concept drift.

Online Learning in Malware Detection. In order to adapt to the evolution of malware over time, some efforts are made to detect malware applying online learning. DroidOL [80] and CASANDRA [79] capture security-related behaviors from applications' dependency graphs and classify applications using detection models built by online learning algorithms. Both DroidOL and CASANDRA retrain the detection model upon receiving each labeled application and make prediction of a new application using the updated model. The retraining process requires every new application being manually labeled, which is constrained by available resources. The fundamental difference between DroidEvolver and these online learning-based detection systems is that DroidEvolver does not rely on true labels of unknown applications to adjust and update its detection models.

Other Malware Detection Methods. Different from DroidEvolver, most learning-based mobile malware detection systems rely on frequent retraining with the cumulative dataset of both old labeled dataset and new labeled dataset to maintain effective after running for a period of time. A list of examples is given below.

DroidSIFT [127] extracts weighted contextual API dependency graph and constructs feature sets accordingly. With graph-based feature vectors, DroidSIFT builds two classifiers, while the first classifier discovers zero-day Android malware, and the second classifier uncovers the family of detected malware. RiskRanker [53] includes two risk analysis modules, while the first-order analysis module sifts through untrusted applications and exposes risky applications, and the second-order analysis module identifies applications with encrypted native code and dynamic code loading. DroidMiner [123] extracts sensitive API call graphs as detection features, while DroidAPIMiner [2] extracts relevant features at API level, and builds a detection model using the generated feature set. Gascon et al. [46] generate function call graphs for Android applications, and build a detection model relying on embedded function call graphs. DroidMat [117], StormDroid [26] and Drebin [10] use not only sensitive API calls but also other information extracted from AndroidManifest.xml file as detection features and train single classifiers afterwards. ICCDetector [120] builds a malware classifier according to ICC-related information included in applications. DRACO [13] and MARVIN [74] utilize static analysis and dynamic analysis to extract features from pre-determined feature categories, and train a detection model afterwards. MASK [23] statically analyzes attributes (including permssions, intent filters,and presence of native code) extracted from applications, and trains a detection model for detecting malware. Finally, Crowdroid [20] relies on crowdsourcing to get system calls from real users, and creates an anomaly model according to system call vector clusters.

The increasing complexity of Android malware calls for new defensive techniques that are harder to evade. To this end, some efforts are made to detect mobile malware using deep neural networks. For example, DroidDetector [126] and DroidSec [125] build Deep Belief Networks to detect Android malware relying on 192 human engineered features, including required permissions, sensitive API calls, and some dynamic behaviors obtained from DroidBox [35]. Deep4maldroid [60] extracts Linux kernel system calls and constructs the weighted directed graphs which are then used to train deep neural networks. Mclaughlin et al. [77] produce a deep neural network-based detection system only according to 218 dex instructions of applications. Towards automatically engineering features for malware detection, FeatureSmith [131] mines the scientific literatures written in natural language to generate features that are semantically related to malicious behaviors.

There are enormous signature-based mobile detection methods. For example, Kirin [41] detects malware based on required permissions which break certain predefined security rules. Since they are not closely related to this work, we refer readers to a survey [3] for more details.

5.9 Conclusions

This work presented DroidEvolver, an Android malware detection system automatically updating itself to catch up with the rapid evolution of both Android system and malware, and achieve both scalable and effective detection over time. Instead of relying on immutable trained detection model and fixed feature sets, DroidEvolver makes necessary update to both detection models and feature sets in order to quickly adapt to new trends identified from unknown applications. The update process is scalable by applying online learning algorithms to adjust current detection models with the reliable pseudo label generated by the model pool. The detection performances of DroidEvolver in different setting are proved to be effective. In addition, the scalability and robustness of DroidEvolver against typical obfuscation techniques are evaluated and demonstrated in our experiments.

Chapter 6

Conclusion and Future Work

6.1 Summary of Contributions

This dissertation makes contributions on automatically detecting malware for Android platform and comprehensively analyzing malicious behaviors and malicious patterns.

In the first work presented in Chapter 3, we systemically analyze ICC patterns of benign applications and malware, and propose a new malware detection system, ICCDetector, which detects malware based on not required resources, but ICC patterns. ICCDetector outputs a detection model after training with a set of benign applications and a set of malware, and employs the trained model for malware detection. ICCDetector significantly outperforms the benchmark as shown in the experiments. In addition, the detected malware are further classified into five new malware categories according to their ICC characteristics, which clarifies the relationship between malware behaviors and ICC patterns.

In the second work presented in Chapter 4, we propose DeepRefiner, an Android malware detection system combining multiple detection layers in complementary way to provide refined detection. To catch up with the rapid evolution of both Android system and malware, DeepRefiner removes laborious human feature engineering and complicated feature extraction from the process. It applies two different detection layers, where the first detection layer achieves efficient detection according to xml files and the second detection layer performs effective and robust detection based on comprehensive bytecode semantics at different scales. We compare DeepRefiner with a state-of-the-art single classifier-based detection system, StormDroid, and ten widely used signature-based anti-virus scanners. The experimental results show that DeepRefiner significantly outperforms both StormDroid and anti-virus scanners. The robustness of DeepRefiner against typical obfuscation techniques and adversarial samples is evaluated and demonstrated in our experiments.

In the third work presented in Chapter 5, we propose DroidEvolver, an Android malware detection system automatically updating itself to catch up with the rapid evolution of both Android framework and malware, and achieve scalable and effective detection over time. Instead of relying on immutable trained detection model and fixed feature sets, DroidEvolver makes necessary update to both detection models and feature sets in order to quickly adapt to new trends identified from unknown applications. The update process is scalable by applying online learning algorithms to adjust current detection model structures with the reliable pseudo label generated by the model pool. The detection performances of DroidEvolver in different settings are proved to be effective. In addition, the scalability and robustness of DroidEvolver against typical obfuscation techniques are evaluated and demonstrated in our experiments.

6.2 Future Direction

One of the main weaknesses of detection systems that employ machine learning techniques in adversarial environments is their susceptibility to evasion attacks. Most evasion attacks (e.g., [88] [87] [14] [15] [44]) take advantage of knowledge of how the detection models of detection systems operate to evade detection passively or actively.

Although it is difficult to completely protect detection systems against evasion attacks, we can still mitigate the effects of evasion attacks. Since the success of most evasion attacks require the knowledge of trained detection models, we can protect detection systems against these attacks by changing both model structures and feature sets applying online learning techniques and deep learning techniques during detection. With the ability of automatic feature engineering performed by deep learning and the ability of real-time update performed by online learning, it is challenging for attackers to get useful information about the detection models.

Another weakness of learning-based detection system is that their detection features and detection performance are not straightforward for human to understand (especially for detection systems applying deep learning techniques). As a result, most learning-based systems are used as black-box methods without explaining why a particular sample is classified as malicious or benign. We plan to make classification decisions interpretable by visualizing detection process and analyzing significant features that contribute to the classification. Furthermore, by better understanding the detection process, we can make detection systems more difficult to evade for malware.

Bibliography

[1] D. E. 201. AntiEmulator. https://github.com/strazzere/anti-emulator/blob/master/slides/DexEducation/Anti-Emulation.pdf/, 2013.

[2] Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining api-level features for robust malware detection in Android. In *Security and Privacy in Communication Networks*, pages 86–103. Springer, 2013.

[3] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith. Sok: Lessons learned from android security research for appified software platforms. *IEEE Symposium on Security and Privacy*, pages 433–451, 2016.

[4] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 468–471. ACM, 2016.

[5] Amazon. http://www.amazon.com/gp/feature.html?docId=1000625601.

[6] T. Anderson. *The theory and practice of online learning*. Athabasca University Press, 2008.

[7] Android. http://developer.android.com/guide/components/intents-filters/.

[8] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: a virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 173–187. ACM, 2011.

[9] Appchina. http://www.appchina.com/.

[10] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of Android malware in your pocket. In *Proc. of NDSS*, 2014.

[11] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[12] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the Android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 217–228. ACM, 2012.

[13] S. Bhandari, R. Gupta, V. Laxmi, M. S. Gaur, A. Zemmari, and M. Anikeev. Draco: Droid analyst combo an android malware analysis framework. *In Proceedings of the 8th International Conference on Security of Information and Networks*, pages 283–289, 2015.

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

[15] B. Biggio, B. Nelson, and P. Laskov. Poisoning attacks against support vector machines. *arXiv preprint arXiv:1206.6389*, 2012.

[16] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[17] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[18] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on Android. In *NDSS*, 2012.

[19] C. J. Burges. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2(2):121–167, 1998.

[20] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for Android. In *Proceedings of the 1st ACM workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 15–26. ACM, 2011.

[21] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio. Acquiring and analyzing app metrics for effective mobile malware detection. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, pages 50–57. ACM, 2016.

[22] N. Cesa-Bianchi, A. Conconi, and C. Gentile. A second-order perceptron algorithm. *SIAM Journal on Computing*, 34(3):640–668, 2005.

[23] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. Mast: Triage for market-scale mobile malware analysis. *In Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*, pages 13–24, 2013.

[24] P. P. Chan, L. C. Hui, and S.-M. Yiu. Droidchecker: analyzing android applications for capability leak. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 125–136. ACM, 2012.

[25] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. *USENIX Security Symposium*, 15, 2015.

[26] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu. Stormdroid: A streaminglized machine learning-based system for detecting android malware. *In 11th ASIACCS*, pages 377–388, 2016.

[27] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, pages 239–252. ACM, 2011.

[28] M. Conti, V. T. N. Nguyen, and B. Crispo. Crepe: Context-related policy enforcement for android. In *ISC*, volume 10, pages 331–345. Springer, 2010.

[29] CoolAPK. `http://www.coolapk.com/apk/com.coolapk.market`.

[30] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer. Online passive-aggressive algorithms. *Journal of Machine Learning Research*, 7(Mar):551–585, 2006.

[31] K. Crammer, M. Dredze, and F. Pereira. Exact convex confidence-weighted learning. In *Advances in Neural Information Processing Systems*, pages 345–352, 2009.

[32] K. Crammer, A. Kulesza, and M. Dredze. Adaptive regularization of weight vectors. In *Advances in neural information processing systems*, pages 414–422, 2009.

[33] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song. Networkprofiler: Towards automatic fingerprinting of android apps. In *INFOCOM, 2013 Proceedings IEEE*, pages 809–817. IEEE, 2013.

[34] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Information Security*, pages 346–360. Springer, 2011.

[35] A. Desnos and P. Lantz. Droidbox: An android application sandbox for dynamic analysis. *URL https://code. google. com/p/droidbox*, 2014.

[36] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium*, volume 31, 2011.

[37] M. Dredze, K. Crammer, and F. Pereira. Confidence-weighted linear classification. In *Proceedings of the 25th international conference on Machine learning*, pages 264–271. ACM, 2008.

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

[39] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for real-time privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[40] W. Enck, M. Ongtang, and P. McDaniel. Mitigating Android software misuse before it happens. 2008.

[41] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and Communications Security*, pages 235–245. ACM, 2009.

[42] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and Communications Security*, pages 627–638. ACM, 2011.

[43] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[44] P. Fogla and W. Lee. Evading network anomaly detection systems: formal reasoning and practical techniques. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 59–68. ACM, 2006.

[45] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in android applications. *In IEEE Security and Privacy*, pages 377–396, 2016.

[46] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of android malware using embedded call graphs. *In Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54, 2013.

[47] X. Ge, K. Taneja, T. Xie, and N. Tillmann. Dyta: dynamic symbolic execution guided with static verification results. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 992–994. IEEE, 2011.

[48] C. Gentile. A new approximate maximal margin classification algorithm. *Journal of Machine Learning Research*, 2(Dec):213–242, 2001.

[49] Github. Open Source Development Platform. `https://github.com/`.

[50] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[51] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[52] Google. `https://developers.google.com/android/c2dm/`.

[53] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day Android malware detection. In *Proceedings of the 10th international conference on Mobile Systems, Applications, and Services*, pages 281–294. ACM, 2012.

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

[55] M. A. Hall. *Correlation-based feature selection for machine learning*. PhD thesis, The University of Waikato, 1999.

[56] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[57] S. C. Hoi, J. Wang, and P. Zhao. Libol: A library for online learning algorithms. *The Journal of Machine Learning Research*, 15(1):495–499, 2014.

[58] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[59] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 639–652. ACM, 2011.

[60] S. Hou, A. Saas, L. Chen, and Y. Ye. Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs. *IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*, pages 104–111, 2016.

[61] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu. Hindroid: An intelligent android malware detection system based on structured heterogeneous information network. *In Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1507–1515, 2017.

[62] J. Jeon, K. K. Micinski, and J. S. Foster. Symdroid: Symbolic execution for dalvik bytecode. Technical report, 2012.

[63] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro. Transcend: Detecting concept drift in malware classification models. 2017.

[64] Kaspersky. Mobile Threats Report. `http://media.kaspersky.com/pdf/Kaspersky-Lab-KSN-Report-mobile-cyberthreats-web.pdf`, 2016.

[65] Kekaku. `http://m.kekaku.com/`.

[66] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center. Scandal: Static analyzer for detecting privacy leaks in android applications. *MoST*, 12, 2012.

[67] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[68] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6. ACM, 2014.

[69] R. Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145, 1995.

[70] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4android: a generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 39–50. ACM, 2011.

[71] LeiWang. Baidu Security Lab. `https://www.blackhat.com/docs/eu-16/`.

[72] L. Li, A. Bartel, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. Mcdaniel. I know what leaked in your pocket: uncovering privacy leaks on android apps with static taint analysis. *arXiv preprint arXiv:1404.7431*, 2014.

[73] A. Liaw and M. Wiener. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[74] M. Lindorfer, M. Neugschwandtner, and C. Platzer. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. *In IEEE 39th Anuual Computer Software and Applications Conference (COMPSAC)*, 2:422–433, 2015.

[75] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240. ACM, 2012.

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

[77] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, et al. Deep android malware detection. *In Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 301–308, 2017.

[78] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, pages 3111–3119, 2013.

[79] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu. Context-aware, adaptive, and scalable android malware detection through online learning. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1(3):157–175, 2017.

[80] A. Narayanan, L. Yang, L. Chen, and L. Jinliang. Adaptive and scalable android malware detection through online learning. In *Neural Networks (IJCNN), 2016 International Joint Conference on*, pages 2484–2491. IEEE, 2016.

[81] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM symposium on information, computer and communications security*, pages 328–332. ACM, 2010.

[82] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. *Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis*, 2013.

[83] M. Ongtang, K. Butler, and P. McDaniel. Porscha: Policy oriented secure content handling in android. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 221–230. ACM, 2010.

[84] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. *Security and Communication Networks*, 5(6):658–673, 2012.

[85] P. Palumbo, L. Sayfullina, D. Komashinskiy, E. Eirola, and J. Karhunen. A pragmatic android malware detection procedure. *Computers & Security*, 70:689–701, 2017.

[86] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin. Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps. *In NDSS*, 2017.

[87] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. Practical black-box attacks against deep learning systems using adversarial examples. *arXiv preprint arXiv:1602.02697*, 2016.

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

[89] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami. Distillation as a defense to adversarial perturbations against deep neural networks. pages 582–597, 2016.

[90] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of Android apps. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*, pages 241–252. ACM, 2012.

[91] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 347–356. ACM, 2010.

[92] PUDN. Programmers United Develop Net. `http://en.pudn.com/`.

[93] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and Application Security and Privacy*, pages 209–220. ACM, 2013.

[94] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334. ACM, 2013.

[95] V. Rastogi, Y. Chen, and X. Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Transactions on Information Forensics and Security*, 9(1):99–108, 2014.

[96] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[97] S. Roy, J. DeLoach, Y. Li, N. Herndon, D. Caragea, X. Ou, V. P. Ranganath, H. Li, and N. Guevara. Experimental study with real-world data for android app security analysis using machine learning. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 81–90. ACM, 2015.

[98] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[99] S. R. Safavian and D. Landgrebe. A survey of decision tree classifier methodology. 1990.

[100] D. Sahoo, C. Liu, and S. C. Hoi. Malicious url detection using machine learning: A survey. *arXiv preprint arXiv:1701.07179*, 2017.

[101] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, volume 11, pages 17–33, 2011.

[102] S. Smalley and R. Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*, volume 310, pages 20–38, 2013.

[103] S. Smalley, C. Vance, and W. Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 1(43):139, 2001.

[104] C. Smutz and A. Stavrou. When a tree falls: Using diversity in ensemble classifiers to identify evasion in malware detectors. NDSS, 2016.

[105] M. Sun, T. Wei, and J. Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. *In ACM CCS*, pages 331–342, 2016.

[106] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*, 2015.

[107] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. *In Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13, 1999.

[108] S. Venkataraman, A. Blum, and D. Song. Limits of learning-based signature generation with adversaries. 2008.

[109] VirusShare. Virus share-sharing of malware samples. *Online: https://virusshare.com/*.

[110] VirusTotal. `https://www.virustotal.com/`.

[111] VirusTotal. Virustotal-free online virus, malware and url scanner. *Online: https://www. virustotal. com/en*, 2012.

[112] J. Wang, P. Zhao, and S. C. Hoi. Exact soft confidence-weighted learning. *arXiv preprint arXiv:1206.4612*, 2012.

[113] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM, 2014.

[114] R. Winsniewski. Android–apktool: A tool for reverse engineering android apk files, 2012.

[115] M. Y. Wong and D. Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS*, 2016.

[116] C. Wu, Y. Zhou, K. Patel, Z. Liang, and X. Jiang. Airbag: Boosting smartphone resistance to malware infection. In *NDSS*, 2014.

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

[118] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. Effective real-time android application auditing. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 899–914. IEEE, 2015.

[119] L. Xiao. Dual averaging methods for regularized stochastic learning and online optimization. *Journal of Machine Learning Research*, 11(Oct):2543–2596, 2010.

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

[121] R. Xu, H. Saïdi, and R. J. Anderson. Aurasium: practical policy enforcement for android applications. In *USENIX Security Symposium*, volume 2012, 2012.

[122] L.-K. Yan and H. Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic Android malware analysis. In *USENIX Security Symposium*, pages 569–584, 2012.

[123] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in Android applications. In *Computer Security-ESORICS 2014*, pages 163–182. Springer, 2014.

[124] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.

[125] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue. Droid-sec: deep learning in android malware detection. *ACM SIGCOMM Computer Communication Review*, 44(4):371–372, 2014.

[126] Z. Yuan, Y. Lu, and Y. Xue. Droiddetector: android malware characterization and detection using deep learning. *Tsinghua Science and Technology*, 21(1):114–123, 2016.

[127] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. *In CCS*, pages 1105–1116, 2014.

[128] M. Zhang and H. Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS 2014)*, 2014.

[129] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.

[130] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *NDSS*, 2012.

[131] Z. Zhu and T. Dumitras. Featuresmith: Automatically engineering features for malware detection by mining the security literature. *In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 767–778, 2016.

[132] S. Zimmeck, Z. Wang, L. Zou, R. Iyengar, B. Liu, F. Schaub, S. Wilson, N. Sadeh, S. M. Bellovin, and J. Reidenberg. Automated analysis of privacy requirements for mobile apps. 2016.

[133] M. Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 928–936, 2003.

[134] L. M. Zintgraf, T. S. Cohen, T. Adel, and M. Welling. Visualizing deep neural network decisions: Prediction difference analysis. *arXiv preprint arXiv:1702.04595*, 2017.