ADVERSARIAL-RESILIENCE ASSURANCE FOR MOBILE SECURITY SYSTEMS

BY

WEI YANG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

ABSTRACT

As mobile phones become an increasingly critical part of our world, ensuring the security and privacy of mobile applications (in short as apps) becomes increasingly important. For too long, researchers have often tackled security in an attack-driven, ad hoc, and reactionary manner with large manual efforts devoted by security analysts. In the efforts of making security systems automated and systematic, multiple intelligent techniques, such as program analysis and machine learning, have been introduced in the mobile security systems for better security decision making. However, these intelligent techniques are originally proposed for domains such as image recognition, Virtual Personal Assistants, and software testing without considering the presence of adversaries.

In this dissertation, we present three main bodies of research on adversarial resiliency of intelligent techniques used in mobile security systems. We first present how intelligent techniques can be adapted for automated decision making in mobile security systems. Then we investigate the possibility to design and implement systematic attack strategies that are specifically adversarial to these newly-proposed intelligent techniques. Last, based on the findings that the intelligent techniques are indeed susceptible to the adversarial attacks, we develop techniques to further strengthen the adversarial resiliency of intelligent techniques toward these adversarial attacks.

In particular, we use mobile malware detection as a representative of security systems for our investigation. To show how a malware detection approach can be enhanced by intelligent techniques such as machine learning and static program analysis, we propose AppContext, an approach that identifies malware with 87.7% precision and 95% recall. To show the possibility to attack intelligent techniques such as machine learning, we propose MRV, an approach that automatically constructs more than hundreds of new malware variants compromising state-of-the-art learning-based malware detectors. To strengthen the adversarial resiliency against obfuscation techniques used by malware to confuse static analysis, we propose EnMobile, which detects malware with substantially higher precision and recall than four state-of-the-art approaches, namely Apposcopy, Drebin, MUDFLOW, and AppContext.

To my family, for their love and support.

ACKNOWLEDGMENTS

I thank my advisors Professors Tao Xie and Carl Gunter. Tao has provided me with enormous support throughout my Ph.D. study, and I have had a great privilege to work with him for the past seven years. If it were not for Tao, I would not even come to Illinois to finish my Ph.D.. Tao has given me the freedom to discover and explore my true research passions, even when they fell outside of his core emphases. He has supported me in my efforts to take our research beyond academia and make it relevant in broader contexts. Carl has provided me his great supervision and guidance over the past four year. He is always available to discuss research, and share his technical insights and life advice. He has encouraged me to become a member of the research community and challenged me to be a more effective teacher. His kindness and wisdom have motivated and inspired me a lot through my Ph.D. study, and will be motivating and inspiring me in the future.

I would like to offer my special thanks to two other members of my thesis committee, Dr. Mukul Prasad and Professor Darko Marinov. Mukul has been a great source of help and advice since we collaborated on the work published in my first paper. I spent three wonderful summers working as an intern with him. Darko constantly provides valuable feedback on my work during my preliminary exam, final defense, and on other occasions. Both of them are very supportive in helping me to accomplish and publish my work in this dissertation.

I would like to thank ChengXiang Zhai and Dawn Song for their generous help during my Ph.D. study and my academic job searching. ChengXiang took the time to give me much valuable advice on our collaborative project and job searching. Dawn invited me to spend the last summer of my Ph.D. study working with her at UC Berkeley. She has been very supportive throughout my stay at Berkeley.

I am extremely grateful to all my current and past group mates: Dengfeng Li, Zhengkai Wu, Wenyu Wang, Wing Lam, Zexuan Zhong, Angello Astorga, Siwakorn Srisakaokul, Liia Butler, Jiayi Cao, Yue Leng, Mingming Zhang, Zelin Zhao, Xusheng Xiao, Tianyu Wo, Yuan Yao, Blake Bassett, Xiao Yu, Sihan Li, Ruowen Wang, JeeHyun Hwang, Rahul Pandita, John Majikes, Qi Wang, Soteris Demetriou, Karan Ganju, Yunhui Long, Guliz Seray Tuncay, Vincent Bindschaedler, Avesta Hojjati, Whitney Merrill, and Aston Zhang,.

I would like to thank other current or former Illinois students in the area of PL/FM/SE, who spent their time and efforts in conducting technical discussion and providing advice to me, especially Alex Gyori, Owolabi Legunsen, August Shi, Farah Hariri, Qingzhou Luo, Milos Gligoric, Yu Lin, Lamyaa Eloussi, Peiyuan Zhao, Yi Zhang, Cosmin Radoi, Daejun Park, Xueqing Liu, and Si Liu.

Many thanks to my other collaborators, including Deguang Kong, William Enck, Lingming Zhang, Cong Liu, Benjamin Andow, Yuetang Deng, Huoran Li, Xuanzhe Liu, Haoyu Wang, Yao Guo, Jiaqi Guo, Haibing Zheng, Beihai Liang, Xia Zeng, Wujie Zheng, Fan Xia, Jian-Guang Lou, Ting Liu, and Dongmei Zhang.

Many thanks to undergraduate students who I have been working with, including Ximin Lin, Evan N. Johnson, Dean Lin, Sherry Wu, Xiang Li, Chaeyun Jung, Shelby Doty, Yurui Cao, Lucas J. Hsiung, Jerry R. Guo, and Rittika Adhikary. I learned a lot while mentoring them. We have together built a great research environment.

I would like to thank many friends who helped me during my Ph.D. study: Ling Chen, Da Young Lee, Xin Zhang, Yitao Hu, Xuan Tang, Bin Liu, Alex Zhong, Jie Luo, Zhengyang Jin, Yu Feng, Yonghwi Kwon, Weihang Wang, and Cheng Zhang.

I would like to thank my undergraduate advisor at SJTU, Jianjun Zhao. Without him, I would never have the motivation or opportunity to start my Ph.D. study. He also provided me help and advice every time I met him during my Ph.D. study years.

Finally and above all, my girlfriend, Yan, tirelessly and patiently supported me during my Ph.D. study years. I thank her from the bottom of my heart; it simply would not be possible without her constant love and unwavering support. I am extremely grateful to my parents for supporting me through all these years. I could not thank them enough for everything they have done for me.

I apologize to those who have helped me but are not explicitly mentioned here. You all have my gratitude for your help.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

The increasing popularity of smartphones has made them a target for malware. App markets that distribute software (i.e., applications, in short as apps) to smartphones leverage both automated and manual app analyses to detect malware (e.g., Google Bouncer [1] and Apple App store [2]). To improve the effectiveness of app analysis, existing research proposes approaches that extract features from apps and compare those features against predefined sets of signatures or patterns of malicious or privacy-infringing behaviors, such as method calls, permissions, and information flows [3, 4, 5, 6, 7, 8, 9, 10].

Similar to PC malware, mobile malware has begun taking steps to evade detection by camouflaging as benign apps [11]. For example, an app can hide malicious intentions by using APIs that are appropriate for its expected functionality. As another example, an app may present itself as a messaging app that sends SMS messages when the user clicks the "send" button. However, it also sends SMS messages containing the user's contact information in the background without notifying the user. Since both of these apps use the same SMS APIs, existing automated tools that consider method calls and information flows are unlikely to distinguish between these cases. Notably, the key difference between these two apps is that the malicious app uses the SMS APIs under an *unexpected context*.

In this dissertation, we present three main bodies of research on adversarial resiliency of intelligent techniques used in mobile security systems. We first present how intelligent techniques can be adapted for automated decision making in mobile security systems. Then we investigate the possibility to design and implement systematic attack strategies that are specifically adversarial to these newly-proposed intelligent techniques. Last, based on the findings that the intelligent techniques are indeed susceptible to the adversarial attacks, we develop techniques to further strengthen the adversarial resiliency of intelligent techniques toward these adversarial attacks.

This dissertation showcases the three main bodies of research on a specific problem: using contextual information in the mobile system to automate security decision making. Contextual information is information used to describe the contexts of a security-sensitive behavior (e.g., who initiate the behavior, when the behavior happens, what factor controls the triggering of the behavior). The motivation to use contextual information in making security decision is to improve the usability of the security system. We want to automate the security decision making rather than bothering the user every time when security-sensitive behaviors occur.

The first observation that we have on this problem is that malicious behaviors may share the same security-sensitive operations as benign behaviors, and the differentiating factors to distinguish malware and benign apps are the contexts under which the security-sensitive operations happen. Chapter 3 (AppContext) illustrates our malware detection techniques and findings based on such observation.

In Chapter 3, we identify contexts as differentiating features and present a technique to extract contexts. However, we do not know which context feature can distinguish malware from benign apps. In AppContext, we feed all context feature values as feature vectors to train a machine learning model and the resulting machine learning model is used to decide which feature vector corresponds to malware.

In Chapter 4 (EnMobile), we take an alternative route to extract contexts and leverage the extracted contexts. We recognize entities in external environments of an app, and the app's interactions with such entities, as the key to comprehensively characterizing contexts of security-sensitive behaviors in mobile apps. We develop an entity-based static analysis to precisely extract such contexts. Then, to leverage contexts in detecting malware, instead of using a machine learning model, we propose a signature language that characterizes an app's interactions with external entities, and compile a malware signature for each malware family to detect malware.

After using both learning-based techniques and signature-based techniques to capture distinguishing context features, we come to question the robustness of the proposed approach. We make our second observation: a context-based detection system could identify the differentiating contexts between current malware and benign apps. However, some of the identified differentiating contexts might not be robust. Malware authors can easily change these contexts while maintaining malicious behaviors. We name these contexts as differentiating features but inessential features for malicious behaviors.

Contexts can be differentiating features but inessential features for many reasons. For example, malware payloads usually copy/paste code from one payload to another. By doing so, although a machine learning model could identify that some specific patterns occur in malware much more often than in benign apps. These patterns are there only due to the copy-paste practices but being inessential for malicious behaviors. Chapter 5 introduces MRV, which leverages such observation and develops two major attacks to automatically mutate malware to evade existing malware detection.

1.1 THESIS STATEMENT

The thesis statement is three-pronged:

- (1) Adopting intelligent techniques in existing mobile security systems can enhance the accuracy of security decision making.
- (2) It is possible to design and implement systematic attack strategies that are specifically adversarial to these newly-proposed intelligent techniques.
- (3) It is possible to further strengthen the adversarial resiliency of intelligent techniques toward these adversarial attacks.

In this dissertation, we first investigate how intelligent techniques can be adapted for automated decision making in a mobile security system. Specifically, we propose an approach, AppContext, to detect malware based on the insight that the context of a security-sensitive behavior is a strong indicator of the maliciousness of the behavior. AppContext includes a static-analysis technique for context extraction, which accurately identifies activation conditions and guarding conditions for security-sensitive behaviors, and an abstraction to model the contexts of security-sensitive behaviors based on the two unique characteristics of malware (activation conditions and guarding conditions). AppContext uses machine learning techniques to further leverage the extracted context features to detect malware.

Then, to investigate the possibility to systematically attack existing intelligent techniques, we propose Malware Recomposition Variation (MRV), which includes two practical attacks (feature evolution attack and feature confusion attack) to effectively mutate existing malware for evading detection. We develop a transplantation framework capable of inter-method, inter-component, and inter-app transplantation to automatically mutate app features. We evaluate the robustness of detection models and the differentiability of selected features of malware detectors by systematically and automatically applying proposed attacks to existing malware detectors. We also propose and evaluate three defense mechanisms to strengthen the robustness of malware detectors against MRV attacks.

Last, to investigate the possibility to strengthen the adversarial resiliency of intelligent techniques, we propose a static-analysis approach named EnMobile, to derive the entity-based characterization by analyzing bytecode of a given app, including identifying entities and entity references, extracting provenance information for flows, and matching against signatures in the face of segmented flows. We identify malware interaction patterns with entities and provenance information of the interactions as a corner stone of comprehensively characterizing mobile malware, especially carefully-designed evasive malware. We also propose a novel signature-specification language, based on this characterization, that enables security analysts to create robust, abstract specifications.

1.2 CONTRIBUTIONS

To confirm the thesis statement, this dissertation makes the following main contributions:

- The dissertation presents a new abstraction to model the contexts of security-sensitive behaviors based on the two unique characteristics of malware (activation conditions and guarding conditions). Such abstraction of the contexts should be detailed enough to reflect the intentions of security-sensitive behaviors, but not too redundant to include all the low-level detailed information about system states. Our *context* definition is based on the observation that activating conditions (e.g., events triggering the execution of payloads) and guarding conditions (e.g., environmental attributes controlling the execution of payloads) are the key elements of context information to differentiate malicious behaviors and benign behaviors. Thus, we define the *context* for a security-sensitive behavior as a tuple containing an *activation events* (the event that triggers the security-sensitive behavior), and a series of *context factors* (environmental attributes controlling the execution of the security-sensitive behavior).

- The dissertation introduces a novel approach for malware detection, AppContext, which statically analyzes the security-sensitive behaviors in an Android app. To extract activation events, AppContext chains all inter-component communications (ICCs) [12] within the app and constructs an extended call graph (ECG) to infer activation events. To compute context factors, AppContext combines the control flows of all components from entry points triggered by activation events to the method calls that trigger security-sensitive behaviors in a reduced inter-procedure control flow graph (RICFG) [13], and leverages information flow analysis [14] to identify the environmental attributes that affect the control flows.

- The dissertation presents entity-based characterization of mobile app behaviors. Such characterization summarizes malware interaction patterns with entities and provenance information of the interactions; such characterization is a corner stone of comprehensively characterizing mobile malware. Introducing the concept of entity allows security analysts to express entity interactions in an *end-to-end* fashion, making it much more independent of specific realizations of that interaction, and hence more robust. We also propose a novel signature-specification language, based on this characterization, that enables security analysts to create robust, abstract specifications. Security analysts can specify an information flow by just using the two end-point entities without enumerating all possible intermediate-point entities (*e.g.,* files, databases).

- The dissertation introduces the first approach that conducts semantic analysis of existing malware to systematically construct new malware variants for malware detectors to test and strengthen their detection signatures/models. In particular, we use two variation strategies (i.e., malware evolution attack and malware confusion attack) following structures of existing malware to enhance feasibility of the attacks. Upon the given malware, we conduct semantic-feature mutation analysis and phylogenetic analysis to synthesize mutation strategies. Based on these strategies, we perform program transplantation to automatically mutate malware bytecode to generate new malware variants.

1.3 DISSERTATION ORGANIZATION

The main parts of this dissertation are organized as follows.

Chapter 2. Background. This chapter introduces three main types of security threats in mobile apps and related techniques in addressing corresponding security issues.

Chapter 3. Differentiating Malicious and Benign Mobile App Behaviors Using Context. This chapter introduces AppContext, an approach of static program analysis that extracts the contexts of security-sensitive behaviors to assist app analysis in differentiating between malicious and benign behaviors.

Chapter 4. Adversarial-resilient Static Analysis: Entity-based Characterization and Analysis of Mobile Apps. This chapter introduces EnMobile, which includes new characterization of mobile-app behaviors and entity-based static analyses to accurately characterize an app's interactions with entities.

Chapter 5. Malware Detection in Adversarial Settings: Exploiting Feature Evolutions and Confusions in Mobile Apps. This chapter introduces Malware Recomposition Variation (MRV), an approach that conducts semantic analysis of existing malware to systematically construct new malware variants for malware detectors to test and strengthen their detection signatures/models.

CHAPTER 2: BACKGROUND

Along with the boom of Android apps come severe security challenges. Existing techniques fall short when facing emerging security problems in Android apps, such as zero-day or polymorphic malware, deep and complex vulnerabilities, and untrustworthy app descriptions. To fight against these threats, we propose a semantics/context-aware approach, and design and develop a series of advanced techniques.

2.1 SECURITY THREATS IN MOBILE APPS

Android has dominated the smartphone market and become the most popular operating system for mobile devices. In the meantime, security threats in Android apps have also quickly increased. In particular, four major classes of problems, malware, program vulnerabilities, privacy leaks, and insecure app descriptions, bring considerable challenges to Android app security. Although a lot of research efforts have been made to address these threats, they have fundamental limitations and thus cannot solve the problems.

2.1.1 Malware Attacks

Malware steals and pollutes sensitive information, executes attacker-specified commands, or even totally roots and subverts the mobile devices. To fight against malware, a signature-based approach extracts malicious behaviors as signatures (such as bytecode or regular expression) while a more complicated machine-learning-based approach learns discriminant features from analyzing semantics of malware. Unfortunately, existing automated approaches for Android malware detection and classification can be evaded both in theory and practice. One major challenge for both signature-based approaches and learning-based approaches is to form an informative feature set for the signature or detection model. To address the challenge, existing approaches of malware detection tend to include as many features as possible. For example, Drebin [15], a recent approach of malware detection, uses the feature set containing 545,334 features. A recent study [16] shows that such large feature set has numerous non-informative or even misleading features. The extracted features are thus associated with volatile application syntax, rather than high-level and robust program semantics. As a result, these approaches are also susceptible to evasion.

2.1.2 Software Vulnerabilities

Apps may also contain security vulnerabilities, such as privilege escalation [17], capability leaks [6], permission re-delegation [18], component hijacking [19], and inter-component communication vulnerabilities [12, 20, 21, 22]. These vulnerabilities are largely detected via automatic static analysis [17, 18, 19, 12, 20, 21, 22] to guarantee the scalability and satisfactory code coverage. However, static analysis is conservative in nature and may raise false positives. Therefore, once a potential vulnerability is discovered, it first needs to be confirmed; once it is confirmed, it then needs to be patched. Nevertheless, it is fairly challenging to programmatically accomplish these two tasks because there is a need of automated interpretation of program semantics. So far, upon receiving a discovered vulnerability, the developers have no choice but to manually confirm whether the reported vulnerability is real. It may also be nontrivial for the (often inexperienced) developers to properly fix the vulnerability and release a patch for it. Thus, these discovered vulnerabilities may not be addressed for long time or not addressed at all, leaving a big time window for attackers to exploit these vulnerabilities.

2.1.3 Untrustworthy Descriptions

Unlike traditional desktop systems, Android provides end users with an opportunity to proactively accept or deny the installation of any app to the system. As a result, it is essential that the users become aware of each app's behaviors so as to make appropriate decisions. To this end, Android markets directly present the consumers with two classes of information regarding each app: (1) permissions requested by the app and (2) textual descriptions provided by the developers. However, neither can serve the needs. Permissions are not only hard to understand [18] but also incapable of explaining how the requested permissions are used. For instance, both a benign navigation app and a spyware instance of the same app can require the same permission to access GPS location, and yet use it for completely different purposes. While the benign app delivers GPS data to a legitimate map server upon the user's approval, the spyware instance can periodically and stealthily leak the user's location information to an attacker's site. Due to the lack of context clues, a user is not able to perceive such differences via the simple permission enumeration. Textual descriptions provided by developers are not security-centric. There exists very little incentive for app developers to describe their products from a security perspective, and it is still a difficult task for average developers to write dependable descriptions. Besides, malware authors deliberately deliver misleading descriptions so as to hide malice from innocent users. Previous studies [23, 24] have revealed that the existing descriptive texts are deviated considerably from requested permissions. As a result, developer-driven description generation cannot be considered trustworthy.

2.2 ANDROID MALWARE DETECTION

Existing automated approaches of Android malware detection and classification fall into two general categories: (1) signature-based and (2) machine-learning-based. Signature-based approaches [5, 25] look for specific patterns in the bytecode and API calls, but they are easily evaded by bytecode-level transformation attacks [26]. Machine-learning-based approaches [15, 27, 16, 28] extract features from an app's behavior (such as permission requests and critical API calls) and apply standard machine learning algorithms to perform binary classification. Because the extracted features are associated with application syntax, rather than program semantics, these detectors are also susceptible to evasion.

2.2.1 Signature-based Mobile Malware Detection and Analysis

Many of the existing Android anti-malware products use signature-based approaches for malware detection. Signature-based approaches extract syntactic or semantic features [25] to find signature(s) that matches with an existing database. DroidRanger [5] proposed permission-based footprinting and heuristics-based schemes to detect new samples of known malware families and identify certain behaviors of unknown malicious families, respectively. Risk- Ranker [3] developed an automated system to uncover dangerous app behaviors, such as root exploits and assess potential security risks. Signature-based approaches become ineffective if variants of existing malware are generated through polymorphism.

2.2.2 Learning-based Mobile Malware Classification

Many efforts have also been made to automatically classify Android malware via machine learning. Peng et al. [27] proposed a permission-based classification approach and introduced probabilistic generative models for ranking risks for Android apps. Juxtapp [29] performed feature hashing on the opcode sequence to detect malicious code reuse. DroidAPIMiner [30] extracted Android malware features at the API level and provided light-weight classifiers to defend against malware installations. DREBIN [15] took a hybrid approach and considered both Android permissions and sensitive APIs as malware features. To this end, it performed broad static analysis to extract feature sets from both manifest files and bytecode programs.

It further embedded all feature sets into a joint vector space. As a result, the features contributing to malware detection can be analyzed geometrically and used to explain the detection results. Despite the effectiveness and computational efficiency, these machine-learning-based approaches extract features from solely external symptoms and do not seek an accurate and complete interpretation of app behaviors.

2.3 MOBILE APP VULNERABILITIES

Although the permission-based sandboxing mechanism enforced in Android can effectively confine each app's behaviors by only allowing the ones granted with corresponding permissions, a vulnerable app with certain critical permissions can perform security-sensitive behaviors on behalf of a malicious app. It is so called confused deputy attack. Mobile security vulnerabilities can present in numerous forms, such as privilege escalation [17], capability leaks [6], permission re-delegation [18], component hijacking [19], and inter-component communication vulnerabilities [12, 20, 21, 22].

Prior work primarily focused on automatic discovery of these vulnerabilities. Once a vulnerability is discovered, it can be reported to the developers and a patch is expected. Some patches can be as simple as placing a permission validation at the entry point of an exposed interface (to defeat privilege escalation and permission re-delegation attacks), or withholding the public access to the internal data repositories (to defend against content leaks and pollution).

2.3.1 Component Hijacking Vulnerabilities

Component hijacking may fall into the latter category. When receiving a manipulated input from a malicious Android app, an app with a component hijacking vulnerability may exfiltrate sensitive information or tamper with the sensitive data in a critical data repository on behalf of the malicious app. In other words, a dangerous information flow may happen in either an outbound or inbound direction depending on certain external conditions and/or the internal program state. A prior effort has been made to perform static analysis to discover potential component hijacking vulnerabilities [19]. Static analysis is known to be conservative in nature and may raise false positives. For instance, static analysis may find a viable execution path for information flow, which may never be reached in actual program execution; static analysis may find that interesting information is stored in some elements in a database, and thus has to conservatively treat the entire database as sensitive. As a result, upon receiving a discovered vulnerability, the developers have to manually confirm whether the reported vulnerability is real. However, it is nontrivial for average developers to properly fix the vulnerability and release a patch.

2.3.2 Bytecode Rewriting

In principle, these aforementioned patching techniques can be leveraged to address the vulnerabilities in Android apps. Nevertheless, to fix an Android app, a specific bytecode rewriting technique is needed to insert patch code into the vulnerable programs. Previous studies have utilized this technique to address various problems. I-ARM-Droid [31] rewrote Dalvik bytecode to interpose on all the API invocations and enforce the desired security policies. Aurasium [32] repackaged Android apps to sandbox important native APIs so as to monitor security and privacy violations. Livshits and Jung [33] implemented a graph-theoretic algorithm to place mediation prompts into bytecode program and thus protect resource access. However, due the simplicity of the target problems, prior work did not attempt to rewrite the bytecode programs in an extensive fashion. In contrast, to address sophisticated vulnerabilities, such as component hijacking, a new machinery has to be developed, so that inserted patch code can effectively monitor and control sensitive information flow in apps.

2.3.3 Instrumentation Code Optimization

The size of a rewritten program usually increases significantly. Thus, an optimization phase is needed. Several prior studies attempted to reduce code instrumentation overhead by performing various static analyses and optimizations. To find error patterns in Java source code, Martin et al. [34] optimized dynamic instrumentation by performing static pointer alias analysis. To detect numerous software attacks, Xu et al. [35] inserted runtime checks to enforce various security policies in C source code, and remove redundant checks via compiler optimizations. As a comparison, due to the limited resources on mobile devices, there exists an even more strict restriction for app size. Therefore, a novel approach is necessary to address this new challenge.

2.4 TEXT ANALYTICS FOR MOBILE SECURITY

Recently, efforts have been made to study the security implications of textual descriptions for Android apps. WHYPER [23] used techniques of natural language processing to identify the descriptive sentences that are associated to permissions. It implemented a semantic engine to connect textual elements to Android permissions. AutoCog [24] further applied machine learning techniques to automatically correlate the app descriptions to permissions, and therefore was able to assess description-to-permission fidelity of apps. These studies demonstrate the urgent need to bridge the gap between the textual descriptions and security-related program semantics.

2.4.1 Automated Generation of Natural Language Description

There exists a series of studies on software description generation for traditional Java programs. Sridhara et al. [36] automatically summarized method syntax and function logic using natural language. Later, a similar approach [37] was proposed to improve the method summaries by also describing the specific roles of method parameters and integrating parameter descriptions. Such approach offered heuristics to generate comments and describe the specific roles of different method parameters. A further approach [38] was proposed to automatically identify high-level abstractions of actions in code and described them in natural language. The approach can also identify code fragments that implement high-level abstractions of actions and express them as a natural language description.

In the meantime, Buse and Weimer [39] leveraged symbolic execution and code summarization techniques to document program differences, and thus synthesize succinct human-readable documentation for arbitrary program differences. Moreno et al. [40] proposed a summarization approach that determines class and method stereotypes and uses them, in conjunction with heuristics, to select the information to be included in the class summaries. The goal of these approaches is to improve the program comprehension for developers. As a result, these approaches focused on documenting intra-procedural program logic and low-level code structures. On the contrary, they did not aim at depicting high-level program semantics and therefore could not help end users to understand the risk of Android apps.

CHAPTER 3: DIFFERENTIATING MALICIOUS AND BENIGN MOBILE APP BEHAVIORS USING CONTEXT

3.1 OVERVIEW

A fundamental difference between malicious and benign apps is that their design principles are different. The principles guiding the design of benign apps are to meet requirements from users. However, two basic principles [41] guide the design of most malware are to (1) trigger the execution of their malicious payload (i.e., the part of malware carrying malicious behaviors) frequently to seek maximal benefits; (2) evade detection to prolong their lifetime. Guided by these principles, mobile malware leverages two major features of mobile platforms as below.

Frequent occurrences of imperceptible system events. Unlike traditional software, where events typically come from standard user inputs (keyboards and mice), a large portion of behaviors in mobile apps are driven by events from the mobile system and its sensors [42]. Compared to UI-triggered events, which rely on the user to perform a specific sequence of UI interactions in a specific app, system events are much more frequently triggered. Thus, malware often leverages system events to increase the chances of invoking its malicious payloads [4]. Moreover, system events can occur when the user is not using the app or the device itself, malicious behaviors triggered by system events can easily evade the user's attentions, concealing the signs and traces of the malicious behaviors.

Informative external-environment states. Mobile apps can access numerous attributes of the external environment (e.g., locations and system time). These attributes often convey useful information about the current states of the environment. Such environment states are frequently exploited by malware to actively control the execution of malicious behaviors. For example, the DroidDream [43] malware family suppresses its malicious behaviors during the day and invokes its malicious payload only at night. Since app reviewers or automated tools, such as Bouncer [1], can analyze apps for only a short period of time and with limited variations of environmental conditions, it is very likely that the reviewers and the tools cannot detect the malware when the environmental conditions that trigger the malicious behaviors are not met.

Based on the above-mentioned fundamental differences between malware and benign apps, we propose that *the context in which a security-sensitive behavior occurs* is a strong indicator of whether the security-sensitive behavior is malicious or benign. Malware executes its malicious payloads only under certain unique contexts to reach a balance between prolonging its life time and increasing the chance of being invoked. Such contexts are unique because a balance can be reached only when malicious behaviors are invoked frequently enough to meet the needs (e.g., a certain number of clicks per day to improve search engine rankings of a website), but not too frequently for reviewers/users or tools to notice the abnormal behaviors of the app. On the contrary, most of the contexts for benign behaviors are user interactive, and thus are exploited less frequently by malware.

Expressing contexts in mobile apps is a non-trivial task. In mobile apps, various elements could be used to describe the contexts in which security-sensitive behaviors occur. However, due to the complex event-driven nature of mobile apps, expressing the contexts using all the factors determining the invocation of security-sensitive behaviors would incur huge overhead in extracting the context information and extra burden in differentiating benign behaviors from malicious ones. Consider the example that a security-sensitive behavior can occur only when an app component enters into the lifecycle method that invokes the behavior. Android apps are component-based and each component has a lifecycle [44]. Any factor changing the component's state will determine the invocation of the lifecycle method, thus determining the invocation of the security-sensitive behavior. Since there are a large number of these factors, such as messages sent by other components, remote procedure calls by other components, UI operations of users, and system events, incorporating all these factors into the definition of context would make the analysis for extracting contexts expensive and bring noisy data in differentiating benign behaviors from malicious ones.

To express contexts concisely and yet capture the essence to reflect intentions, we propose an abstraction of the contexts. Such abstraction of the contexts should be detailed enough to reflect the intentions of security-sensitive behaviors, but not too redundant to include all the low-level detailed information about system states. Our *context* definition is based on the observation that activating conditions (e.g., events triggering the execution of payloads) and guarding conditions (e.g., environmental attributes controlling the execution of payloads) are the key elements of context information to differentiate malicious behaviors and benign behaviors. Thus, we define a *context* for a security-sensitive behavior as a tuple containing an *activation events* (the event that triggers the security-sensitive behavior), and a series of *context factors* (environmental attributes controlling the execution of the security-sensitive behavior).

Although our context abstraction reduces the burden in inferring context information, we still need to address two challenges posed by mobile apps. First, inferring activation events requires the analysis of the entry points of the app. Unlike desktop programs that have only one entry point for a program execution (i.e., the main function), a mobile app usually has multiple entry points due to its event-driven nature. Also, not all entry points of an app are triggered by external events, and some of them are triggered by inter-component communications (ICCs) [12] within the app. It is possible that the program execution path from the entry point triggered by an activation event to the invocation of a security-sensitive behavior goes through a chain of components of the app. Existing analysis can identify only the entry point of each component, and thus cannot be directly applied to infer activation events. Second, computing context factors requires the analysis of control flows from the activation events to the invocations of the security-sensitive behaviors. The ICCs in apps complicate the analysis because a conditional statement controlling the execution of ICCs may further control the security-sensitive behaviors in the target component of ICCs.

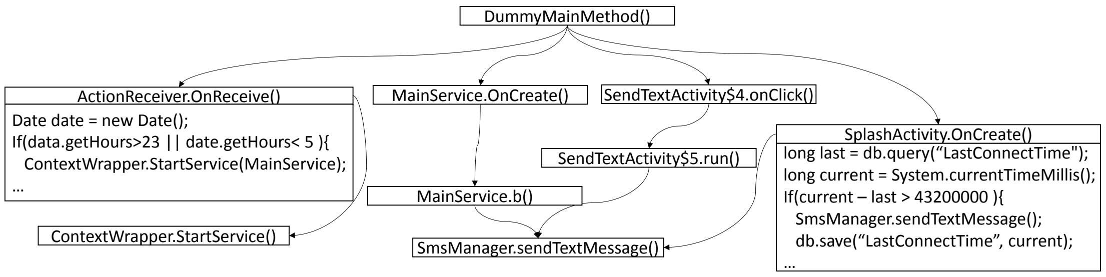
To address these challenges, we propose AppContext, an approach that statically analyzes the security-sensitive behaviors in an Android app. To extract activation events, AppContext chains all ICCs within the app and constructs an extended call graph (ECG) to infer activation events. To compute context factors, AppContext combines the control flows of all components from entry points triggered by activation events to the method calls that trigger security-sensitive behaviors in a reduced inter-procedure control flow graph (RICFG) [13], and leverages information flow analysis [14] to identify the environmental attributes that affect the control flows.

To leverage the extracted contexts for differentiating malicious behaviors and benign ones, we transform these contexts as features and use machine learning techniques, such as support vector machine (SVM) [45], to classify security-sensitive behaviors as malware or benign ones. We use machine learning techniques because the reasoning about the maliciousness of a behavior is vague and subjective by nature. Simply using a static threshold (e.g., the frequency of contexts) to differentiate malicious and benign behaviors does not perform well because it is difficult to determine a proper threshold. For many subtle cases, machine learning techniques are desirable to detect malware by taking multiple factors into account and making decisions based on rich data sets statistically.

3.2 A MOTIVATING EXAMPLE

To illustrate our approach, we use a simplified malware example named MoonSms. MoonSms is a repackaged app that carries both benign functionality and injected malicious DroidDream [43] payloads. The benign functionality provides a variety of festive greetings for SMS messages. Thus, it is rational that MoonSms requests the SEND_SMS permission. Figure 3.1 shows that *SmsManager.sendTextMessage* (i.e., an API method that uses the SEND_SMS permission) is invoked under three contexts. Each invocation of this method is a security-sensitive behavior of the app.

The first invocation of *SmsManager.sendTextMessage* occurs when the user clicks the


DummyMainMethod()
ActionReceiver.OnReceive()
MainService.OnCreate()
SendTextActivity$4.onClick()
SendTextActivity$5.run()
SplashActivity.OnCreate()
MainService.b()
ContextWrapper.StartService()
SmsManager.sendTextMessage()

```
Date date = new Date();
If(data.getHours>23 || date.getHours< 5 ){
  ContextWrapper.StartService(MainService);
…
```

```
long last = db.query("LastConnectTime");
long current = System.currentTimeMillis();
If(current – last > 43200000 ){
  SmsManager.sendTextMessage();
  db.save("LastConnectTime", current);
…
```

(a) Part of the MoonSms's call graph

```
1  <activity android:label="@string/app_name" android:name=".SplashActivity">
2      <intent-filter>
3          <action android:name="android.intent.action.MAIN" />
4          <category android:name="android.intent.category.LAUNCHER" />
5      </intent-filter>
6  </activity>
7  <service android:name="com.android.main.MainService" android:process=":main" />
8  <receiver android:name="com.android.main.ActionReceiver">
9      <intent-filter>
10         <action android:name="android.intent.action.SIG_STR" />
11     </intent-filter>
12 </receiver>
```

(b) Code snippet of MoonSms's manifest file

Figure 3.1: Motivating Example in MoonSMS App

"Send" button in an activity component named "SendTextActivity". When the "Send" button is clicked, its *onClick* event handler spawns a new thread that invokes *SmsManager.sendTextMessage*.

The second invocation of *SmsManager.sendTextMessage* occurs when the signal strength of the device changes. When the signal strength changes, the system broadcasts an Intent containing the "SIG_STR" action. MoonSms registers a broadcast receiver component named "ActionReceiver" (Lines 8-12 in Figure 3.1(b)) to receive this Intent. When this Intent is broadcasted, ActionReceiver is activated and its *onReceive* method begins execution. ActionReceiver's *onReceive* method starts a service component named "MainService" by invoking the *startService* API method (when the current time is between 11 pm and 5 am), which begins executing MainService's *onCreate* lifecycle method. Finally, MainService's *onCreate* method invokes another method named *b*, which calls *SmsManager.sendTextMessage*.

The third invocation of *SmsManager.sendTextMessage* occurs when MoonSms is launched. When the MoonSms is launched, its main activity component, "SplashActivity" (Lines 1-6 in Figure 3.1(b)), begins execution in its *onCreate* lifecycle method. SplashActivity's *onCreate* method invokes *SmsManager.sendTextMessage* when the current time is at least 12 hours after the "LastConnectTime" is saved in a database.

In the preceding example, the first invocation is not malicious because reviewers can analyze the content on the screen and confirm that the security-sensitive behavior is expected to occur. However, the second and third invocations cannot be justified by the functionality that MoonSms is expected to provide. By inspecting the behaviors, we find that the second and third invocations are malicious because these invocations send SMS to a confirmed malicious server.

This example demonstrates that the contexts of security-sensitive behaviors are essential to differentiate between benign and malicious behaviors, especially when the benign functionality provided by apps may rationalize the requested permissions, and the security-sensitive method calls allowed by the requested permissions can also be used by malicious functionality. AppContext focuses on exposing the contexts of security-sensitive behaviors. We refer back to this example in the rest of the chapter to illustrate how AppContext formalizes the abstraction of contexts of security-sensitive behaviors and extracts these contexts from app binary code.

3.3 CONTEXT OF SECURITY-SENSITIVE BEHAVIOR

In this section, we formally define the context of a security-sensitive behavior.

We consider a **security-sensitive behavior** as an invocation of a security-sensitive method under a certain context. A security-sensitive method is a method that meets at least one of the following three requirements: (1) Permission-protected methods. Some methods in the Android API require permissions to be invoked. Such methods usually access security-sensitive resources and data (the detailed list of the methods is specified in PScout [46]). (2) The methods that is either a source method or a sink method (output channel) of an information flow. An information flow consists of a source from which the security-sensitive data may originate and a sink to which the data may be sent (the detailed list of sources and sinks are specified in Susi [47]). Sources and sinks are not always protected by permission; for example, *FileOutputStream.write* is a sink method to write the data to a file but does not require Android permissions to be invoked. A permission-protected method may not be a source/sink method; for example, *ContextWrapper.setWallpaper* is protected by permission SET_WALLPAPER, but is neither a source nor a sink. (3) Reflection methods [48] and dynamic code-loading methods [49]. Resolving reflection or dynamic loading methods in static analysis is a known difficult problem with fundamental limitations [50]. For this reason, we do not attempt to resolve these methods, but rather treat them as being security sensitive. In doing so, we are being conservative, because these methods may result in invoking other security-sensitive methods. There are a few methods in the Android API allowing apps to load and invoke code at runtime that has also been leveraged by existing malware [4] (a detail list is listed on our project website [51]).

Our definition of context (Definition 4.6) includes two important characteristics that determine the invocations of security-sensitive method calls: **activation events** (Definition 4.2) and **context factors** (Definition 4.5). Such definition represents a set of essential elements for decision making in app inspection.

The activation events are the *external events that trigger the security-sensitive methods*. The external events include UI events (events triggered by interactions on apps' graphical user interfaces), SYSTEM events (events initiated by the system-state changes such as receiving SMS), and HARDWARE events (events triggered by the interactions on the device interfaces, such as pressing the HOME or BACK button). Activation events connect security-sensitive behaviors to the behaviors' "initiator" in the external environment (e.g., users or system), as the events are triggered when the external environment changes or the mobile system reaches a certain state.

To infer activation events of security-sensitive method calls, we analyze the entry points (e.g., *ActionReceiver.OnReceive()* and *SendTextActivity$4.onClick()* in Figure 3.1(a)) of call graph that contains the security-sensitive method calls. In an Android app, not all entry points are triggered by activation events, and some of entry points can be triggered only by inter-component communications. For example, *MainService.OnCreate()* is triggered by *startService()* in the component *ActionReceiver*. An analysis needs to trace back a chain of entry-point methods executed before the invocation of the security-sensitive methods to identify the entry points that can be used to infer activation events.

To assist the analysis to locate entry points triggered by activation events, we first define an extended call graph that connects all the ICCs in an app.

Definition 4.1. An *extended call graph* $ECG = (N, E)$ for an app p is a directed graph in which each node $n \in N$ denotes a method in p, and each edge $e(a, b) \in E$ denotes either a calling relationship from a to b or a in one component A calls b in another component B. An entry point of the ECG is a node n_e that has no incoming edges (i.e., for each nodes $n \in N$, $e(n, n_e) \notin E$).

An extended call graph (ECG) is a call graph with edges representing ICCs. The entry point of ECG can be triggered by activation events. For example, Figure 3.2 shows part of MoonSms's ECG. Compared to the corresponding call graph (CG) shown in Figure 3.1(a), the ECG has an ICC edge from *ActionReceiver.OnReceive* to *MainService.OnCreate*, connecting the component *ActionReceiver* to component *MainService*. ECG enables our approach (Section 3.4) to link the security-sensitive method call (*SmsManager.sendTextMessage*) to the entry point *ActionReceiver.OnReceive*, and the activation

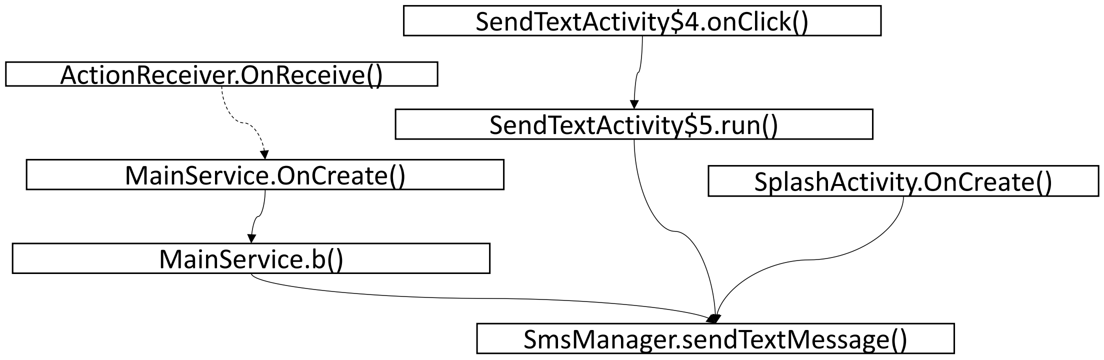

SendTextActivity$4.onClick()
ActionReceiver.OnReceive()
SendTextActivity$5.run()
MainService.OnCreate()
SplashActivity.OnCreate()
MainService.b()
SmsManager.sendTextMessage()

Figure 3.2: ECG of CG shown in Figure 3.1(a)

event (signal strength changes) can be further inferred from the entry point. We next define the activation event.

Definition 4.2. An *activation event* act_{n_e,n_k} of a method call n_k is the event that triggers the entry point n_e in an extended call graph $ECG = (N, E)$ and there exists a call path $P = n_e n_1 n_2 ... n_k$ such that $e(n_e, n_1) \in E$ and for $i = 1, 2, ..., k$, $1 \leq k$, $e(n_{i-1}, n_i) \in E$.

Activation events are identified by their *action types*, which can be inferred from entry points. Specifically, the action types of UI events are their corresponding operation types (e.g., click, long click), the action types of system events are state changes that trigger the events (e.g., signal strength changes), and the action types of hardware events are the component lifecycle phases that the events lead to (e.g., onPause, leaving the component; onResume, re-entering the component).

The context factors are *environmental attributes that control the execution of security-sensitive method calls.* The values of context factors can affect control flows from entry points triggered by activation events to security-sensitive method calls. To precisely describe the control flows in an Android app, we adopt and simplify the definition of an inter-procedure control-flow graph (ICFG) from Harrold et al. [13] and define a *reduced inter-procedure control-flow graph* (RICFG).

Definition 4.3. Given an $ICFG$, an entry point n_e, and a method call n_k, a reduced inter-procedure control-flow graph $RICFG_{n_e,n_k}$ is a subgraph of $ICFG$ that contains all the paths from n_e to n_k.

For example, Figure 5.9(a) shows an $RICFG_{n_e,n_k}$ where the entry point n_e is *ActionReceiver.OnReceive()* in the ECG (shown in Figure 3.2) and the security-sensitive method call n_k is *sendTextMessage*.

Apps usually obtain the values of the environmental attributes by using certain Java/Android API methods (e.g., *currentTimeMillis()*, *getInstalledApplications()*). We denote such API methods as *environment-property methods*. We next define control dependence among statements and use control dependence and environment-property methods to define context factors.

Definition 4.4. In a program, if a statement n_s controls whether a statement n is executed, n is *control dependent* on n_s.

Definition 4.5. Given an $RICFG_{n_e,n_k}$ and a set of conditional statements S_{n_e,n_k} in $RICFG_{n_e,n_k}$ that n_k is control dependent on, a *context factor* f_{n_e,n_k,s_i} is an environmental attribute whose value is used in a conditional statement s_i where $s_i \in S_{n_e,n_k}$.

The context factors are computed by analyzing the information flows (data dependence) from environment-property methods to conditional statements that control the execution of security-sensitive method in the RICFG. Based on these definitions, we formally define a *context*:

Definition 4.6. A *context* C_{n_e,n_k} for method call n_k is a tuple consisting of the activation event act_{n_e,n_k} and the set of context factors $\{f_{n_e,n_k,s_i} | s_i \in S_{n_e,n_k}\}$ where S_{n_e,n_k} is the set of conditional statements in $RICFG_{n_e,n_k}$.

3.4 APPROACH

We next present AppContext, our approach that extracts the values of elements in the context definition defined in Section 3.3. First, AppContext constructs a call graph from an app's binary and performs static analysis to locate its security-sensitive behaviors. Next, AppContext identifies activation events by the entry points of the computed call graph, and converts the call graph into an ECG by using ICC information. Then, AppContext constructs RICFGs for each security-sensitive method calls in the ECG and traverses each RICFG to find conditional statement sets. Next, AppContext finds context factors whose values are used in conditional statements via information flow analysis and then generates the complete contexts using identified activation events and context factors. Finally, AppContext classifies the security-sensitive behaviors by using the features of the extracted contexts.

3.4.1 Locating Security-Sensitive Behaviors

AppContext locates security-sensitive method behaviors by constructing call graphs and locating security-sensitive method calls within the call graphs (we leverage Flowdroid's call graph building [52]; please check their paper [14] for details). Security-sensitive method calls are divided into three groups by the information used to identify them, as illustrated below.

First, the permission-protected API methods, source or sink methods, reflection methods, and dynamic code-loading methods are all identified by using a method signature. If a method matches a method signature in this group, AppContext extracts and saves the method name, permission, and the entry points for later analysis.

Second, the methods that read or write security-sensitive Content Providers are identified by the URIs of the content providers. To access a content provider, the URI designating the recipient content provider is passed to a *ContentResolver* class (Section 2). Only the method calls using the URIs of security-sensitive content providers are security sensitive. The list of URIs designating security-sensitive content providers is provided in PScout [46]. If the URI parameter of a method is in the URI list, AppContext saves the URI, permission, and the entry points for later analysis.

Finally, the methods that send or receive security-sensitive Intents are identified by the Intent-action strings. An app can call *sendBroadcast* or *registerReceiver* with Intent action strings to send or receive specified Intent messages. The list of Intent-action strings requiring permissions to send or receive is provided in PScout [46]. If the intent parameter in the method is in the list, AppContext saves the Intent-action string, permission, and the entry points for later analysis.

3.4.2 Identifying Activation Events

As discussed in Section 3.3, the activation events are represented by their action types. Action types can be extracted from the app's entry points. AppContext identifies activation events by analyzing two types of entry points. (1) For system events handled by intent filters and hardware events, their entry points are *lifecycle methods*. If the components of the lifecycle methods have intent filters specified for system Intent messages, the entry points are invoked by system events. Otherwise, the entry points are invoked by hardware events. (2) For both system events captured by event-handling methods and UI events, their entry points should be *event-handling methods*.

Algorithm 3.1 presents the analysis used to extract activation events for the given security-sensitive method calls. The analysis returns a list of activation events ($\mathcal{E}$) for each security-sensitive method call. The analysis takes security-sensitive method calls and their corresponding entry points as input. An entry point belongs to one of two above-mentioned categories: lifecycle methods and event-handling methods.

For the first category of entry points, lifecycle methods, the analysis first decides whether the activation event could be a system event captured by intent filters (Line 6). If the component that the lifecycle method belongs to has intent filters, for each intent filter, the attributes in the intent filters are used to represent the activation events of the contexts. For each activation event, AppContext create a tuple and saves activation event along with

Algorithm 3.1: IdentifyActivationEvent

Inputs : $\mathcal{B}$: A set of contexts without context factors and activation events (i.e., tuples consisting of security-sensitive method calls and their entry points in call graphs)
CG: The call graph of the whole app
$\mathcal{A}$: App binary code

Outputs: $\mathcal{E}$: A set of contexts without context factors (i.e., tuples consisting of security-sensitive method calls, their activation events, and corresponding entry points)
ECG: The extended call graph of the whole app

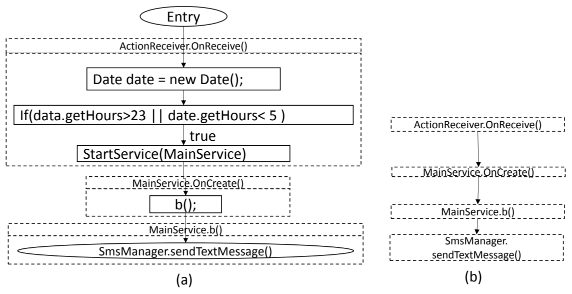
```
1  begin
2      $\mathcal{E} \leftarrow \emptyset$
3      foreach $b \in \mathcal{B}$ do
4          $entrypoint \leftarrow getEntrypoint(b)$
5          if $isLifeCycleMethods(entrypoint)$ then
6              if $hasIntentFilters(entrypoint, A)$ then
                   // System events (by intent filters)
7                  $Filters \leftarrow getFilters(entrypoint, A)$
8                  foreach $filter \in Filters$ do
9                      $\mathcal{E}.addFilter(b, filter)$
10                 end
11             end
12             $ICC \leftarrow findICCs(CG, entrypoint)$
13             if $ICC \neq \emptyset$ then
                   // adding ICC edges
14                 $CG.add(ICC)$
                   // Recursively invoke the algorithm
15                 $E \leftarrow replaceEntryPoint(b, CG)$
16                 $\mathcal{E}.addAll(identifyActivationEvent(E, CG, A))$
17             end
18             else
                   // Hardware events
19                 $\mathcal{E}.addLifeCycle(c)$
20             end
21         end
22         if $isEventHandler(entrypoint)$ then
23             $\mathcal{E}.addHandler(c)$
24         end
25     end
26     $ECG \leftarrow CG$
27     return $\mathcal{E}$
28 end
```

the method call and the entry point in the tuple to the $\mathcal{E}$ list for later analysis (Line 9).

The analysis then decides whether the lifecycle method can be invoked by ICC calls (e.g., *startService*, *sendBroadcast*) (Line 13). If there are method calls invoking the lifecycle method, the analysis adds ICC edges to the CG (Line 14), and replaces entry points of the ICC calls with the original entry points (Line 15). Then Algorithm 3.1 is invoked recursively with the augmented CG (i.e., ECG) and new entry points to cover all activation events. The activation events are then saved in the tuples for later analysis (Line 16).

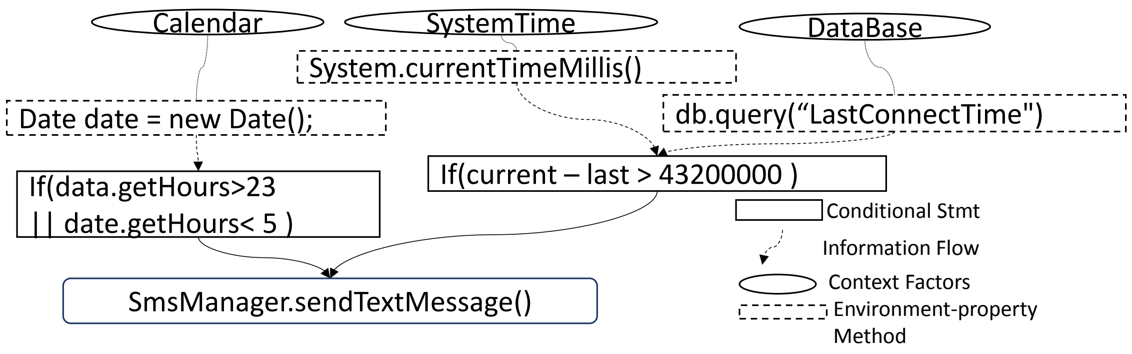
If the lifecycle method cannot be invoked from app code, then the security-sensitive method call is triggered by hardware events. We use the attributes of the lifecycle methods to represent the activation events, and save the activation events in the tuples for later analysis (Line 19).

For the second category of entry points, event-handling methods, the analysis uses the



Entry
ActionReceiver.OnReceive()
Date date = new Date();
If(data.getHours>23 || date.getHours< 5)
true
StartService(MainService)
MainService.OnCreate()
b();
MainService.b()
SmsManager.sendTextMessage()
(a)
ActionReceiver.OnReceive()
MainService.OnCreate()
MainService.b()
SmsManager.sendTextMessage()
(b)

Figure 3.3: An RICFG (a) and its corresponding ECG subgraph (b)



Calendar
SystemTime
DataBase
System.currentTimeMillis()
Date date = new Date();
db.query("LastConnectTime")
If(data.getHours>23 || date.getHours< 5)
If(current – last > 43200000)
SmsManager.sendTextMessage()
Conditional Stmt
Information Flow
Context Factors
Environment-property Method

Figure 3.4: Context factors of MoonSms in Figure 3.1

attributes of the UI event-handling methods or system event-handling methods to represent the activation events, and save the activation events in the tuples for later analysis (Line 23).

3.4.3 Extracting Context Factors

After computing the ECG and activation events for a security-sensitive method call, AppContext constructs and traverses the RICFGs to extract context factors. As shown in Section 3.3, the RICFGs need to be constructed based on the ECG. Thus, for each security-sensitive method call, AppContext identifies the ECG's entry points that can lead to the invocation of the method. Then AppContext obtains the ICFG of the app by connecting the CFG of each node on the ECG. Based on the ICFG, AppContext constructs the RICFGs from each entry point to the security-sensitive method call. For each RICFG, AppContext traverses the RICFG to identify the conditional statements on which the security-sensitive method is control-dependent. Finally, AppContext saves the set of extracted conditional statements with the security-sensitive method call and the corresponding activation events.

Figure 3.4 presents the analysis used to extract context factors. For each conditional statement extracted in the previous step, AppContext tracks the information flow from the environment-property methods (Section 3.3) to the conditional statement. The sources of

Table 3.1: Feature categories for classification

Features of Behavior Information		
Permission	Security-sensitive method call	
Features of Activation Event		
Hardware event	System event	UI event
Features of Context Factors		
List of environmental attributes		

the information flows indicate which context factors control the invocation of the security-sensitive behaviors. In the MoonSms example, the context factors are Calendar information, system time, and database information. By combining the context factors with corresponding activation events of the security-sensitive method calls, AppContext generates the complete context tuples.

3.4.4 Classifying Security-Sensitive Behaviors

Leveraging the extracted contexts to classify security-sensitive behaviors as malicious and benign, we formulate the detection of malicious behaviors as a classification problem. AppContext leverages a supervised learning approach to train a classifier to compute the conditional likelihood of a security-sensitive behavior being malicious versus benign given context features. Specifically, AppContext uses a support vector machine (SVM) as the classifier because SVM is very resilient to over-fitting even with a large number of values.

Classification is performed using a set of features. A feature is a function that associates a training example with a value, i.e., a function evaluates a certain single domain-specific criterion for the example. AppContext leverages the list of features in Table 3.1 for classifying security-sensitive behaviors. The list consists of the features about the security-sensitive behavior itself, and the features describing the contexts of the behavior: the activation events and the context factors. With this list of features, AppContext generates a feature vector for each context of a security-sensitive behavior.

Table 3.2 shows an example of feature vectors. For features describing behavior information (i.e., Permission, Method Call), the feature values are the name of the permission or method. For methods such as source/sink, reflection, or dynamic loading methods that do not have corresponding permissions (i.e., do not require permissions to be invoked), the permission names are predefined strings such as “SOURCE”, “SINK”, “REFLECTION”, “DYNLOADING”. For features describing activation events, the feature values are the action types (Section 3.3) of the events. For features describing the context factors

Table 3.2: Feature vectors for MoonSms example

Permission	Method Call	Hardware	System	UI	F_3*	F_4*	F_5*	...	F_{142}
SEND_SMS	*sendTextMessage*	N/A	SIG_STR	N/A	1	0	0	...	0
SEND_SMS	*sendTextMessage*	EnterApp	N/A	N/A	0	1	1	...	0
SEND_SMS	*sendTextMessage*	N/A	N/A	Click	0	0	0	...	0

* F_3 = Calendar, F_4 = System Time, F_5 = Database

$(F_1, F_2, ..., F_{142})$, the feature values are either "1" (the context contains the context factor) or "0" (the context factor is not part of the context).

3.5 RESULTS

To evaluate the effectiveness of AppContext and using context information to detect malware, we have conducted three evaluations.

We seek to answer the following research questions:

- **RQ1**: How effective is AppContext in identifying malware? How does AppContext compare to the approach without context information in terms of the effectiveness of malware identification?
- **RQ2**: How do activation events and context factors in our context definition contribute to the effectiveness of malware identification?
- **RQ3**: How accurate is our static analysis in inferring contexts?

3.5.1 Study Subjects

Our subject apps include 846 Android apps in total (633 benign apps, 202 malicious apps, and 11 open-source apps). To collect malicious apps, we randomly select 130 malicious apps from a malware dataset collected by Zhou et al. [4], 30 malicious apps from the VirusShare dataset [53], and 50 malicious apps from the Contagio dataset [54]. We also select 17 malicious apps identified by VirusTotal [55] that were posted on Google Play in 2013 but were later removed by Google. Our final malware dataset contains 202 malicious apps. These malicious apps cover the majority of existing Android malware families from 2011 to 2014, which are rapidly evolving to circumvent detection by various mobile security software.

To collect benign apps, we download the top 500 apps for each category from Google Play as of January 2013. Because FlowDroid runs out of memory on large apps, to ensure that enough apps can be analyzed without errors, for each category, we randomly select 20 apps under 5 MB and 20 apps without size restriction from these top 500 apps. We also exclude the apps identified as malware by VirusTotal and the apps that cause FlowDroid to throw exceptions or timeout. The final benign dataset contains 633 apps. To collect open-source apps, we randomly select 15 apps from F-Droid [56]. Among these 15 apps, we exclude 4 apps that do not have security-sensitive behaviors. Our open-source dataset contains 11 apps.

We apply AppContext to extract contexts from the subject apps. AppContext runs on a desktop with 3.4 GHz Intel Core i7 processor and 8 GB of memory. We set the timeout of AppContext as 80 minutes, and AppContext exceeds the timeout limit for 162 apps, which are then excluded from the later study. For the 846 apps being used as subjects, AppContext takes on average 647 seconds to finish the analysis of one app.

3.5.2 RQ 1: Overall Effectiveness

To answer RQ1, we label the extracted contexts from the subject apps, and perform a ten-fold cross-validation to evaluate the overall effectiveness of AppContext. To make a fair comparison with the existing approaches that do not use context information, we apply the supervised learning approach using all the features of AppContext, and then apply the same supervised learning approach using the features containing only the behavior information shown in Table 3.1 (i.e., security-sensitive method calls and permissions). The results are shown in Table 3.3 and Table 4.4, respectively (the second and third rows).

Labelling security-sensitive method calls. Because there is no ground truth for determining a security-sensitive method call as malicious or benign, as a best-effort solution, we systematically label security-sensitive method calls as malicious based on the existing malware signatures [25, 57, 55]. Specifically, we label a security-sensitive method call as malicious if the class/package name of the method call matches any class/package name that we collected from the existing malware signatures. We label the rest of security-sensitive method calls as benign.

We collect class/package names from malware signatures of three sources. (1) Apposcopy [25] includes a list of semantic signatures for existing malware along with a tool to check apps' binaries against the signatures. We run all of the subject apps using a tool that we reproduced based on Apposcopy and record the names of the packages and classes that match the signatures. (2) We use class names in Androguard's signature database [57]. (3) The VirusTotal [55] service inspects malware by using a number of antivirus software and reports the family that the malware belongs to. We identify the malware family that each of our malicious apps belongs to using VirusTotal, and we identify the package/class names of the malicious payloads from the online technical reports provided by the antivirus software vendors for each malware family.

Cross Validation. We use the labeled behaviors (i.e., method calls) both as training and test data in a ten-fold cross-validation [58], which is a standard approach for evaluating machine-learning techniques. It works by randomly dividing all data into 10 equally sized buckets, training the classifier on 9 of the buckets, and classifying the remaining bucket for testing. This process is repeated 10 times, with each of the 10 buckets used exactly once as the testing data. We report the average precision and recall in Table 4.4.

Results. We evaluate the effectiveness of AppContext in identifying both malicious behaviors and malicious apps. An app is identified as a malicious app if any of its security-sensitive method calls is identified as malicious. Table 3.3 and Table 4.4 show that AppContext (the row of Complete Context) has higher precision and recall in both identifying malicious behaviors and identifying malware than the existing approach that does not use context information (the row of Behavior Information). We next present two major reasons that cause such misidentification.

First, AppContext misidentifies a number of security-sensitive method calls triggered by UI events and without context factors. This result suggests that compared to system events and hardware events, UI events have less indication of the maliciousness of a security-sensitive method call.

Second, a few method calls are incorrectly identified as malicious because we mistakenly label similar benign behaviors as malicious. In malicious payloads, a small number of security-sensitive method calls may not have malicious intentions, such as *MediaPlayer.pause* protected by the WAKE_LOCK permission in malicious payloads. However, as we label all security-sensitive method calls in a malicious payload, AppContext incorrectly identifies such benign method calls as malicious. This result suggests that the identification results can be improved if the training set for the classifier is labeled more accurately.

We also evaluate the effectiveness of AppContext in identifying malicious reflective calls or dynamic code-loading method calls. AppContext shows high precisions and recalls in identifying malicious method calls. AppContext correctly identifies 872 out of 922 malicious method calls but also misidentifies 180 benign method calls as malicious (i.e., 82.9% precision, 94.5% recall). AppContext correctly identifies 710 out of 787 malicious dynamic code-loading method calls but misidentifies 137 benign method calls as malicious (i.e., 83.8% precision, 90.2% recall). For all 56 malicious apps using root exploits (which are commonly launched by dynamic code loading [4]), only one malicious app (i.e., AsRoot) was not identified by AppContext. As the detailed behaviors of reflective calls and dynamically-loaded code

Table 3.3: Malicious security-sensitive behaviors identified by AppContext

Features Used	**P(%)**	**R(%)**
Complete Context (C)	94.8	84.8
Behavior Information (B)	79.0	37.3
Activation Events (E)	83.2	49.5
Context Factors (F)	90.6	71.2
B & E	88.0	71.3
B & F	90.2	76.9
E & F	92.5	77.3

Table 3.4: Identification of malware by AppContext

Features Used	**TP**	**FP**	**FN**	**P(%)**	**R(%)**
Complete Context (C)	192	27	10	87.7	95.0
Behavior Information (B)	169	78	33	68.4	83.6
Activation Events (E)	163	78	39	67.6	80.6
Context Factors (F)	150	26	52	85.2	74.2
B & E	193	63	9	75.3	95.5
B & F	180	46	22	79.6	89.1
E & F	187	27	15	87.3	92.5

TP = True Positive, FP = False Positive, FN = False Negative
P = Precision, R = Recall

were unobtainable in static analysis, such results show the advantage that AppContext can differentiate benign and malicious security-sensitive method calls without knowing the detailed behaviors being triggered.

3.5.3 RQ2: Effectiveness of Activation Events and Context Factors

RQ2 evaluates the effectiveness of both activation events and context factors in identifying malicious app behaviors. To answer RQ2, we use only partial features listed in Table 3.1 to train the classification model. We apply the same supervised learning approach used in RQ1 with the features being the activation events (the row of Activation Events), context factors (the row of Context Factors), behavior information and activation events (the row of B & E), behavior information and context factors (the row of B & F), and activation events and context factors (the row of E & F), respectively. The results are shown in Table 3.3 and Table 4.4.

Results. We evaluate the effectiveness of activation events by comparing the result of the analysis using activation events (the rows of Complete Context, B & E, and E & F) to the result of the analysis not using activation events (the rows of B & F, B, and F) in Table 3.3 and Table 4.4. The comparison shows that adding the features of activation events to the analysis improves both the precision and recall of the identification results. We find that the improvements are mainly because activation events help effectively identify malicious method calls that have no context factors. The activation events in some of these malicious method calls are often used by benign apps to update the UI to inform users that certain events have occurred. For example, UMS_DISCONNECTED is used to inform users that the device has been disconnected from USB mass storage, SIG_STR is used to inform users that the phone signal strength changes, and ACTION_POWER_CONNECTED is used to inform users that external power has been connected to the device. Because these events are seldom used in benign apps to trigger security-sensitive method calls, the activation events can effectively differentiate benign and malicious behaviors with no context factors.

We also evaluate the effectiveness of context factors by comparing the results of the analysis using context factors (the row of Complete Context, B & F, and E & F) with the result of analysis not using context factors (the row of B & E, B, and E). The result shows that the analysis using context factors has relatively higher precisions (over 90% for identifying malicious behaviors and around 80% for identifying malware). We find that the improvement in the precision is mainly because context factors effectively help identify the malicious behaviors whose activation events are UI events. We also find that context factors can disambiguate the malicious and benign intentions for certain vague cases when security-sensitive method calls are protected by commonly-used resources (e.g., Internet). For example, we find that some of benign apps and malware will both connect to servers (*URL.openConnection*) after the apps start, and thus the activation events and behaviors for both apps are the same. However, the context factors of malware include data from an Intent message (*Intent.getExtras*) and data from the Internet (*URL.openStream*), suggesting that whether the apps connect to the server or not is determined by whoever sends the Intent message or the Internet data. Such context factors demonstrate the command & control nature of certain families of malware.

In addition, context factors also reflect controls of security-sensitive method calls in benign apps. For example, we find that a few benign apps and malware obtain device information (*TelephonyManager.getDeviceId* etc.) after the apps start. The difference between two types of apps is that the benign apps invoke *getDeviceId* only when auto logins are successful (i.e., the context factors for *getDeviceId* include information from the database or the Internet). But malware directly sends device information to the server (i.e, no context factors).

Finally, we further evaluate the effectiveness of contexts by running analysis using features of activation events and context factors (the row of E & F). The precision and recall of the analysis are comparable as the precision and recall of the analysis using complete context. Such results suggest that contexts can identify a number of malicious method calls without knowing the detailed behaviors being triggered, consistent with the analysis result for behaviors that invoke reflection or dynamic code-loading methods. Both results indicate that the maliciousness of a security-sensitive method call is more closely related to the behavior's intention (reflected via contexts) than the type of the security-sensitive resources that the behavior accesses.

3.5.4 RQ3: Accuracy of Static Analysis

To evaluate the effectiveness of the extracted contexts, we dynamically verify whether the security-sensitive method is invoked by triggering the activation events and configuring context factors based on the contexts. The execution path triggered by the activation events may vary when the context factors are assigned different values. In this evaluation, we use only open-source apps as the subjects. The main reason is that these apps come with source code, which can be used to easily infer the correct values of context factors in controlling the execution of the security-sensitive method calls. AppContext is applied on 11 open-source apps to extract contexts and the analysis time is logged.

To verify the correctness of context factors, we analyze the source code to check whether a security-sensitive method call is control dependent on each context factor. If the control dependence exists, we determine the values of the context factors that lead to the execution of the security-sensitive method call. We then configure the external environment based on the inferred values of context factors and trigger the activation events of 88 security-sensitive behaviors of these apps.

We use the activity manager through the Android Debug Bridge (ADB) to simulate system events, and manually simulate hardware and UI events. We configure the values of each context factor by changing configuration of emulators. Then, we use the profiler of the activity manager to log the executions of the apps. To monitor the execution traces, we start the profiler before firing the activation events and stop the profiler 5 seconds afterwards.

The preceding evaluation process has some limitations. The profiler cannot trace the invocations of the *onCreate* or *onDestoy* methods, because the profiling must be started after the creation of an app's process and be stopped before the destruction of the app's process. We also exclude events that cannot be simulated through ADB such as error events (e.g., triggering the *onError* method in *MediaPlayer.OnErrorListener*) and the context factors whose value we cannot manipulate such as data from URL connection).

Results. Table 3.5 shows our evaluation results. Among the 88 generated contexts, we are able to confirm 82 contexts (i.e., 93.2% accuracy). Six contexts cannot be verified because the activation events could not trigger the security-sensitive method calls. The context factors

Table 3.5: Effectiveness of context extraction

# App	**# Context**	**# Verified Context**	**Time(sec)**
11	88	82	291

of all the contexts whose activation events could trigger the security-sensitive method calls are accurate. The average analysis time is 291 seconds, which is acceptable for the app reviewing process. Note that the evaluation result is conservative since the inferred values for context factors may not be accurate.

3.6 CONCLUSION

In this chapter, we introduce AppContext, an approach of static program analysis that extracts the contexts of security-sensitive behaviors to assist app analysis in differentiating between malicious and benign behaviors. We implement a prototype of AppContext and evaluate AppContext on 202 malicious apps from various malware datasets, and 633 benign apps from the Google Play Store. AppContext correctly identifies 192 malicious apps with 87.7% precision and 95% recall. Our evaluation results suggest that the maliciousness of a security-sensitive behavior is more closely related to the intention of the behavior (reflected via contexts) than the type of the security-sensitive resources that the behavior accesses.

CHAPTER 4: ADVERSARIAL-RESILIENT STATIC ANALYSIS: ENTITY-BASED CHARACTERIZATION AND ANALYSIS OF MOBILE APPS

4.1 OVERVIEW

Malware detection based on behaviors represents a prominent class of malware-detection approaches where characteristics of known malware samples are used as a basis of identifying new malware [25, 4, 59, 60, 61, 62, 63, 64]. These approaches typically work in two phases: malicious-behavior characterization and malicious-behavior detection. The behavior-characterization phase is used to derive a specification of the malicious behaviors, which may either be manually specified as malware signatures or automatically mined as malware models in a suitable representation. In the behavior-detection phase, techniques of static or dynamic program analysis are used to analyze a given mobile app for possible matches against the specified signatures or mined models.

Limitations of existing approaches. Existing malware detection approaches [25, 65, 14, 22, 66, 67, 28, 68, 15, 30, 25] (based on specified signatures or mined models) suffer from the overfitting problem (*i.e.,* tailored to be capable of detecting only the malware samples used for deriving the signatures or models) for two main reasons: limited expressiveness and limited accuracy.

Limited expressiveness. The existing approaches primarily use information-leaking dataflows within the app as the basis of a malicious behavior. This characterization is ill-equipped to capture the roles of each party (*e.g.,* initiator) for the malicious behavior and the provenance of the malicious behavior (*e.g., who* controls the flows). So the existing approaches typically fail to capture malicious behaviors initiated and controlled by malicious servers, such as initiating spams or launching denial-of-service attacks. Without a proper characterization for these behaviors, the existing approaches turn to easily mutable features (shared across malware samples of the same family) such as network addresses or other string constants to detect these behaviors, allowing malware developers to easily change these features to evade detection.

Limited accuracy. Using *implementation-specific* structures (*e.g.,* API methods, objects), the existing approaches fail to accurately express common malicious behaviors consisting of interactions between the malware and entities in its environments. For example, a malware signature can be specified to express a `GingerMaster` malware sample's malicious behavior segmented into four phases: (1) the app retrieves and preprocesses a phone number from the telephony manager (entity A); (2) the app writes the preprocessed phone number into a temporary file (entity B); (3) the app reads the preprocessed phone number from the same temporary file (entity B); (4) the app sends the preprocessed phone number to a (malicious) server (entity C). For Phases 2 and 3, the existing approaches can recognize that the app interacts with some files (*i.e.,* the *type* of entities) based on the app's invoking API methods on a Java `File` object, but cannot recognize that the file in Phase 2 is indeed the same as the file in Phase 3.

Therefore, the existing approaches can produce false positives by matching the malware signature with a benign app where the preprocessed phone number is saved to a file, and non-sensitive information being read from a different file is sent to a server. In addition, the `GingerMaster` malware family has seven variations (i.e., different implementations) during the period from 2011 to 2013 [69]. Malware samples of these variations can either (1) skip Phases 2 and 3 by directly sending the preprocessed phone number to the malicious server or (2) replace the temporary file entity in Phases 2 and 3 to be a temporary database. Thus, the existing approaches can also produce false negatives (by including in the malware signature the specific behavior of Phases 1-4).

To address such significant limitations of the existing approaches, in this chapter, we present a novel approach, EnMobile, consisting of techniques for malware-behavior characterization and detection, respectively.

Malware-behavior characterization. Our approach is motivated by the finding [4, 69, 15] that more than 90% of current mobile malware have a command-and-control (C&C) architecture, where the malware receive and respond to commands from an external actor, *e.g.,* a remote server. Thus, our approach directly characterizes the underlying C&C structure of the malware. In particular, EnMobile improves the existing malware-behavior characterization in two main aspects.

Entity-based characterization. Without using implementation-specific structures or easily mutable features (*e.g.,* API methods, objects) in an app, EnMobile expresses the app's behaviors in terms of interactions among *entities*. Entities are a host of actors on the mobile platform including mobile system components (*e.g.,* the telephony manager, SMS manager, contacts provider), local on-device resources (*e.g.,* files, databases), other mobile apps and libraries, human users, and network locations, *etc.* EnMobile recognizes entities through their identities (*e.g.,* files with different names are different entities).

Furthermore, introducing the concept of entity allows security analysts to express entity interactions in an *end-to-end* fashion, making it much more independent of specific realizations of that interaction (*e.g.,* specifics in Phases 2 and 3), and hence more robust. For example, for the malware family of the example sample, security analysts can specify the information flow from the telephony manager (entity A) to the malicious server (entity C)

by just using the two end-point entities (entities A and C) without enumerating all possible intermediate-point entities (*e.g.,* files, databases).

Flow-provenance predicates. Going beyond using information-leaking dataflows within the app, EnMobile enriches interactions among entities with *provenance* information. Provenance in our context refers to *who* controls the flow, and why, *i.e.,* the specific *intended purpose* of the flow [28, 70, 71, 72]. For example, the existing approaches may produce an information flow (`file` $\rightarrow$ `sendTextMessage`). Such flow can match both the behavior of sending contents of a file out through SMS (a benign action of sending predefined messages) and the behavior of specifying which phone numbers that the SMS should be sent to (a malicious action of sending premium messages). To address such issue, we propose a set of data flow predicates (Section 4.3.1) to reflect the purpose (*e.g.,* passing configuration parameters vs. purely transmitting information to another entity) that an information flow within an app can serve for, a set of control flow predicates (Section 4.3.1) to present the ownership of information flows (*i.e.,* the entities that initiate/control the information flows).

Malware-behavior detection. As the entity-based characterization is more abstract than the existing approaches using implementation-specific features, EnMobile includes various static analyses enabling the instantiation of the characterization (*i.e.,* extracting program information from malware samples and matching such information against the signatures) in the following two main aspects (besides supporting flow-provenance predicates).

Identification of entities and entity references. In order to characterize an app's behaviors directly in terms of its interactions with entities of the app, one challenge is to extract the correspondence between an in-program object (named as an entity reference) and the entity with which the object may interact in a given execution context (*e.g.,* calling context). To infer the entities that each Java object can point to, we develop an identity-propagation algorithm that conducts a flow- and context-sensitive analysis extended from taint analysis [14]. Such algorithm addresses two main issues. First, multiple objects could point to the same entity. Second, a given program object can interact with different entities under different execution contexts.

Matching against signatures. As discussed earlier, malware typically perform malicious behaviors segmented into multiple phases (*e.g.,* downloading, preprocessing), storing intermediate computation results in temporary files or databases. Such segmentation gives rise to multiple *segments* of information flow, punctuated with interactions with entities (*e.g.,* files or databases). These segments would need to be "stitched together" in order to be properly matched against a signature specified to characterize the end-to-end interaction. To address the challenge, we propose a *flow-sensitive* stitching algorithm to ensure that the connected information flows are feasible in the actual execution.

This chapter makes the following main contributions:

- **Characterization.** We identify malware interaction patterns with entities and provenance information of the interactions as a corner stone of comprehensively characterizing mobile malware. We also propose a novel signature-specification language, based on this characterization, that enables security analysts to create robust, abstract specifications.

- **Detection.** We design static analyses to derive the entity-based characterization by analyzing bytecode of a given app, including identifying entities and entity references, extracting provenance information for flows, and matching against signatures in the face of segmented flows.

- **Implementation and Evaluations.** We present a practical implementation of our approach and evaluate its effectiveness, on a set of 6614 apps consisting of malware (from Genome [4] and Drebin [15]) and benign apps (from Google Play). Our results show that EnMobile achieves substantially higher precision and recall than four state-of-the-art approaches.

Related Work. Existing malware detection approaches characterize malware behaviors by features that commonly exist in malware but not in benign apps. These approaches include mining (MUDFLOW [65]), clustering (CHABADA [66]), classification (AppContext [28]), graph matching (Astroid [68], Apposcopy [25]), and natural language processing (AsDroid [73], WHYPER [23]) etc. However, non-essential features (e.g., component type, file name, unrelated information flows) in code clones are often mistaken by these approaches as discriminative features. Copy-paste practice is prevalent in malware industry, resulting in many code clones in malware samples [74]. Because the same code snippet has appeared in many malware samples, these approaches may regard those non-essential features in code clones as major discriminant factors (because the same pieces of code snippet have appeared in many malware samples but not in benign apps). EnMobile provides security analysts a way to directly characterize malware behaviors through the high-level interactions among entities instead of leveraging a specific implementation difference in the malware.

EnMobile also falls into the general category of information flow analysis. Much work has been proposed to enhance static analysis of mobile apps [75, 76, 77, 22, 78, 20, 79, 80, 81, 82, 83, 19, 84, 85]. Information flow analysis tracks whether privacy-sensitive data (*i.e.,* sources) flows to outgoing channels or sensitive outlets (*i.e.,* sinks). EnMobile complements existing information flow analysis by adding entity-based characterization to the information flow. AAPL [77] uses enhanced data flow analysis techniques to increase the number of data flows that can be detected by information flow analysis and then uses the peer-voting mechanism

```
TrickMe controls three behaviors via commands from a C&C server:
B1: Sending the user's SMS to the C&C server through Internet
B2: Performs click fraud based on coordinates provided by the C&C server
B3: Downloading malicious payloads from downloading servers whose addresses are specified by
    the C&C server, and renaming the downloaded files based on the names provided by the C&
    C server.
```

Figure 4.1: Natural language description of Malware `TrickMe`

```
public void onCreate(Bundle b) {
...
String v0 = "http://www.malicious.com";
URL url = new java.net.URL(v0); //<url, CON_1>
HttpURLConnection n = url.openConnection(); //<n, CON_1>
f_s = new File("server.xml"); //<f_s, FILE_1>
f_c = new File("commandFile");//<f_c, FILE_2>
f_info = new File("infoFile");
f_n = new File("coordinateFile"); //<f_n, FILE_3>
f_d = new File("downloadFile");
f_f = new File("fileNameFile");
read(f_s, n); // Reading message from n to f_s
parse(f_s, f_c, f_n, f_d, f_f); //Parsing f_s into four files
readSMS(f_info); ... }
```

Figure 4.2: `onCreate` method of `MainActivity`

to lower the false positive rate to report illegitimate information leakages. AAPL fails to handle obfuscation techniques such as string encryption (by using constant propagation analysis) and produces high false positives (by matching all sources with all potential sinks). EnMobile resolves these two limitations by precisely computing the identity of an entity. SPARTA [86] and FlowDroid [14] are two general information flow analysis frameworks. SPARTA enables the flow-policy checking by providing an integrity type system to annotate source code with information-flow type qualifiers. FlowDroid is a static taint analysis tool for Android apps based on Soot [87] and Heros [88]. EnMobile complements SPARTA and FlowDroid by analyzing all types of data flows to detect malicious behaviors that are not information leakage (*e.g.,* bot-driven C&C behaviors).

4.2 A MOTIVATING EXAMPLE

We illustrate our approach using a simple malware example `TrickMe`, shown in Figure 4.2, Figure 4.3, and Figure 4.4, which is derived from several real pieces of malware. Its C&C

```
public void onStop() {
...
for(String command: readLine(f_c)){
if(command.equals("click")){
float [] axis = getAxis(readLine(f_n));
MotionEvent down = MotionEvent.obtain(...,0, axis[0],axis[1],0);
MotionEvent up = MotionEvent.obtain(...,1, axis[0],axis[1],0);
...
Activity adActivity = ...; //<s,AdsPlatform>
Webview adView = adActivity.findViewbyID (...);
adView.dispatchTouchEvent(down);
adView.dispatchTouchEvent(up);
}
if(command.equals("sendInfo")){
sendFile("http://www.malicious.com",f_info);
}
if(command.equals("install")){
String [] filename = readLine(f_f);
int i = 0;
for(String url: readLine(f_d)){
File f_i = new RandomAccessFile(filename[i++],"rw");
read(f_i, new java.net.URL(url).openConnection());
}}}...}
```

Figure 4.3: `onStop` method of `MainActivity`

```
public String [] readLine(File file){ //<file, FILE_2>, <file, FILE_3>
FileReader r = new FileReader(file); //<r, FILE_2>, <r, FILE_3>
BufferedReader br = new BufferedReader(r);//<br, FILE_2>, <br, FILE_3>
String line = br.readLine(); ...
return line; }
```

Figure 4.4: `readLine` method of `MainActivity`

structure comes from Geinimi [89], its downloading behavior mimics Answerbot [90], and its information leakage behaviors follow BeanBot [91]. TrickMe has three malicious behaviors driven by a remote malicious server as described in Figure 4.1. All three behaviors reside in an Activity component `MainActivity`, the MAIN Activity component of `TrickMe`, which is invoked when the malware is launched.

`TrickMe` receives commands and prepares necessary information for future malicious behaviors in the `onCreate` method of `MainActivity` (Figure 4.2). It first opens a network connection to a malicious server (Lines 3 - 5), and reads a message from the server to file `server.xml` (Line 12). It then parses the `server.xml` into the four files `commandFile`, `coordinateFile`, `downloadFile`, and `fileNameFile` (Line 13). Finally, it reads a list of SMSs into file `infoFile` (Line 14).

In the `onStop` method of `MainActivity` (Figure 4.3), `TrickMe` launches one of three different malicious behaviors based on different commands received earlier from the server. On command `sendInfo` it sends the content of `infoFile` to the server, on command `click` it computes and clicks on the X-Y coordinates computed based on the numbers in `coordinateFile` to incur click fraud [92], and on command `install` it downloads malicious payloads from URL addresses listed in `downloadFile`, and names the downloaded files according to the list of names in `fileNameFile`. The downloaded payloads are used by `TrickMe` to launch other malicious behaviors.

Comparison of signatures in Apposcopy and EnMobile. In a signature-based scheme for malware detection, such as Apposcopy [25], security analysts can specify the control-flow and data-flow properties shown in Figure 4.5 as the signature for the `TrickMe` malware.

The `activity(a)` predicate declares an activity component `a`. The `icc(SYSTEM, a, MAIN, _)` predicate states an inter-component communication from the system to the activity `a` and the content of the communication is a "MAIN" intent message. The `flow(a, SMS, a, file)` predicate represents an information flow from source `SMS` in component `a` to sink `file` in component `a` (Figure 4.2, Line 14).

Although the specified predicates do represent valid information flows and triggering events in `TrickMe`, they are insufficient for representing the unique characteristics of the malware, and therefore may be unable to differentiate malware from a benign app. For example, a benign SMS manager app can sync the app's configuration with a server (`BufferedReader` $\leadsto$ `URLConnection`), (`URLConnection` $\leadsto$ `file`), and back up SMSs (`SMS` $\leadsto$ `file`) where $\leadsto$ indicates an information flow. Such an app also possesses the same control-flow and data-flow properties as `TrickMe`, as per Apposcopy's characterization, and would therefore be indistinguishable from `TrickMe`.

```
activity(a), icc(SYSTEM, a, MAIN, _),
flow(a, URLConnection, a, file),
flow(a, BufferedReader, a, URLConnection),
flow(a, BufferedReader, a, file),
flow(a, SMS, a, file)
```

Figure 4.5: Characterization of `TrickMe` in Apposcopy [25]

Figure 4.6 presents the signature of `TrickMe` specified by security analysts in EnMobile. EnMobile allows to accurately characterize `TrickMe` in three perspectives. First, EnMobile allows to designate the entities that certain behaviors may be attributed to, and thereby precisely characterize the purpose of the behaviors. For example, a unique behavior of `TrickMe` is its use of the "command" read from "commandFile" sent from the C&C server, to direct the launching of different malicious behaviors. Identifying and implicating the entity of the remote C&C server, rather than the local file "commandFile" (as the existing approaches would do), are key to recognizing the true nature of this behavior.

Second, EnMobile allows to stitch segments of the end-to-end flow behavior exhibited by `TrickMe`. For example, the signature in Apposcopy includes only the benign-looking flows (`SMS` $\rightsquigarrow$ `file`) and (`BufferedReader` $\rightsquigarrow$ `URLConnection`), while EnMobile infers these flows as segments of a larger, and potentially malicious flow (*i.e.,*`Transmit*(s, n_2)` of `UploadMessage` in Figure 4.6 indicating the behavior of sending SMSs to a web server).

Third, EnMobile detects malicious behaviors other than information leakage. EnMobile captures several non-leakage behaviors in `TrickMe` (click fraud, downloading and renaming file). Such detection needs a more nuanced characterization of information flows. As our results show (Sec. 4.5.3), characterization can significantly impact the accuracy of malware detection.

4.3 ENTITY-BASED CHARACTERIZATION

Broadly, EnMobile aims to characterize an app in terms of its *relevant* interactions with entities. To this end, it tracks and precisely characterizes information flows associated with the security-sensitive behaviors of an app. We next illustrate some preliminaries before presenting characterization in EnMobile.

Security-sensitive Behavior. A security-sensitive behavior is an invocation of a security-sensitive method. We consider two types of methods as security sensitive: permission-protected methods and other source/sink methods that read/write information.

Permission-protected methods are API methods that require permissions to access security-sensitive resources and data. We use the list of permission-protected methods specified in PScout [46], and the list of source/sink methods specified in Susi [47]. Further, we follow PScout [46] to label each security-sensitive method call with one of a small set of abstract actions (*e.g.,*`SEND, RECEIVE, READ, WRITE`), based on its overall behavior. These action labels are used in our characterization (Section 4.3.1).

Entity. An entity is an external resource that an app interacts with during its execution. Entities may include network locations (*e.g.,* URLs or phone numbers) external to the device running the app, such as the URL of a C&C server. They may also include on-device intermediate storage sites (*e.g.,* files, databases) or specific Android system resources (*e.g.,* the SMS Manager), with which the app may interact during execution. Entities form the sources (providing information) or targets/sinks (consuming information) of information flows to/from the app. An entity is defined by a tuple: < *entity type*, *entity identifier* >.

Entity type. The type identifies the category of entity, such as a file or a network location (`File`, `UrlConnect`), as well as the type of communication channel of the app with the entity, such as an SMS communication with a phone number (`SmsTarget`) and a phone call to a number (`PhoneTarget`).

Entity identifier. The identifier is the name or address of the entity, such as a filename, a URL, or a phone number, which together with the entity type can be used to uniquely identify the entity. In EnMobile, entity identifier values are stored and propagated in the program via (primitive-type or string-type) constants or symbolic expressions (involving variables provided by the external input, *e.g.,* network message, user input).

Entity reference. An entity reference is an in-program object or variable that serves as a proxy of the entity within the app and through which the app communicates with the entity. For example, variable `f_s`, in Figure 4.2, Line 6, is a reference of a `File` entity with identifier "server.xml". An entity may have multiple references. Conversely, a single object, such as the Android SMS Manager, may instantiate different entities (*e.g.,* SMSs to different phone numbers) at different points during the app's execution.

4.3.1 Language Specification

We propose a language to characterize an app based on its interactions with entities. One use of such characterization is to write signatures for recognizing malware. Figure 4.6 shows a signature characterizing the `TrickMe` example.

The characterization is a set of Datalog rules. Each rule, of the form: `head :- predicate1, predicate2, ...` , is a horn clause, defining a predicate `head` as the conjunction (logical

```
TrickMe(a) :− Download(a), SendMessage(a), UploadMessage(a)

ClickAds(a):− Connection(n), AdsPlatform(s), SysUIEvent(e),
Config∗(n, s, TOUCH), Control∗(n, s, TOUCH), Trigger(e, s, TOUCH).

Download(a) :− Connection(n), SysUIEvent(e), Connection(n_i),
Initiate∗(n, n_i), File(f_i), Initiate∗(n, f_i), Transmit(n_i, f_i),
Trigger(e, f_i, WRITE), Control∗(n, f_i, WRITE).

UploadMessage(a) :− Connection(n), SysUIEvent(e), Connection(n_2),
SmsInbox(s), Transmit∗(s, n_2), Control∗(n, n_2, WRITE),
Trigger(e, n_2, WRITE).
```

Figure 4.6: Characterization of `TrickMe` in EnMobile

Table 4.1: App-behavior description language

Type	Syntax	Definition
Event Predicate	`SysEvent(v)`, `UiEvent(v)`, `SysUiEvent(v)`	v: event of appropriate type (one of three event types: System event, UI event, or System UI event)
Entity Predicate	`Entity(e)`, `File(e)`, `UrlConnect(e)`, `SmsTarget(e)`, `SmsInbox(e)`	e: entity of appropriate type (partial list of potential entity types)
Data-flow Predicate	Transmit(e_{source}, e_{target})	e_{source}: source entity, e_{target}: target entity
	Transmit*(e_{source}, e_{target})	Transmit*(e_s, e_t) :- Entity(e_i), Transmit*(e_s, e_i), Transmit(e_i, e_t)
	Config(e_{source}, e_{target}, a)	e_{source}: source entity, e_{target}: target entity, a: target entity's action
	Config*(e_{source}, e_{target}, a)	Config*(e_s, e_t, a) :- Entity(e_i), Transmit*(e_s, e_i), Config(e_i, e_t, a)
	Initiate(e_{source}, e_{target})	e_{source}: source entity, e_{target}: target entity
	Initiate*(e_{source}, e_{target})	Initiate*(e_s, e_t) :- Entity(e_i), Transmit*(e_s, e_i), Initiate(e_i, e_t)
Control-flow Predicate	Trigger($v_{trigger}$, e_{target}, a)	$v_{trigger}$: triggering event, e_{target}: implicated entity, a: security sensitive action
	Control($e_{control}$, e_{target}, a)	$e_{control}$: controlling entity, e_{target}: controlled entity, a: security sensitive action
	Control*($e_{control}$, e_{target}, a)	Control*(e_c, e_t, a) :- Entity(e_i), Transmit*(e_c, e_i), Control(e_i, e_t, a)

AND) of one or more other predicates (*e.g.,*`predicate1`). A predicate is a relation name with variables or constants as arguments.

Table 4.1 provides an informal specification of our proposed language. It consists of four kinds of predicates, namely *event*, *entity*, *data-flow*, and *control-flow* predicates, described next.

Event predicate. Event predicates declare relevant events, as one of three types: *System*, *UI*, or *System UI* events. A system event is one initiated by the system-state changes, *e.g.,* receiving an SMS, a UI event is triggered by interactions on an app's graphical user interface, and a system UI event is triggered by the interactions on the device interfaces, such as pressing an app's icon on the system's screen to launch the app. This categorization follows previous work on Android testing [28].

Entity predicate. Entity predicates declare specific entities, each of a specific type, with which the app interacts during its execution. For example, in Figure 4.6, `File(f)` denotes a file entity `f`. Table 4.1 lists a few examples of currently recognized types.

Data-flow predicates

We make the observation that the *intent* of an information flow can be determined based on a specific parameter of the sink method that it flows into. The reason is that each parameter of a (sink) method plays a specific role in executing its behavior. Thus, our characterization categorizes each parameter of a sink method into one of three types: (1) *transmit* parameters, which receive data to be written to a target entity, (2) *config* parameters, which are used to configure security-sensitive behaviors, and (3) *initiate* parameters, which carry identifiers, *e.g.,* the file name, to initialize a target entity. Based on this characterization, information flows can also be categorized as Transmit, Config, or Initiate, and represented using the corresponding predicates as explained below. To implement this characterization, we pre-compile lists of transmit, config, initiate parameters and their corresponding methods for common entities in the Android SDK, as a one-time effort for Android.

Predicate *Transmit* (*Transmit).** The *Transmit* predicate encodes data transmission from a source entity e_{source} to a target entity e_{target}, where the app reads information from e_{source} and writes it to e_{target}. An information flow satisfies a *Transmit* predicate if it flows into a designated transmit parameter of a sink method. For example, in the *TrickMe* characterization (Figure 4.6), predicate `Transmit(n_i, f_i)` encodes the behavior of downloading payloads from given URLs (`n_i`) to files (`f_i`).

We also define the predicate `Transmit*(`e_{source}`, `e_{target}`)`, to represent information transitively flowing from e_{source} to e_{target} through a sequence of *Transmit* flows. For example, the `Transmit*(s, n_2)` (Figure 4.6) encodes the behavior of reading an SMS from SMSInbox `s`, storing it into file "f_info" (Line 14, Figure 4.2) and subsequently forwarding it to a given URL `n_2` (Line 15, Figure 4.3).

Predicate *Config* (*Config).** This predicate encodes information flows from a source entity e_{source} to target entity e_{target} initiated exclusively for configuring the behavior of a security-sensitive action a performed by e_{target}. Similar to *Transmit*, the definition is extended to define predicates *Config**, as per Table 4.1. For example, in the `TrickMe` malware, the number saved in the "coordinateFile" is used to configure the behavior of *dispatchTouchEvent* (Lines 4-7 in Figure 4.3), as the content in the "coordinateFile" is from network connection n, the configuration relationship is represented by `Config*(n, s, TOUCH)`.

Predicate *Initiate* (*Initiate).** This predicate represents behavior where the entity identifier (*e.g.,* a file name) of a target entity e_{target} is read from a source entity e_{source} and

Table 4.2: Identity propagation logic

Statement Type	**Format**	**Flow Functions**	**Propagation Description**
Entity Initialization	$x = new(a_{init}, a_0, ..., a_n), n \in \mathcal{N}$	ⓘ $I_{out} \overset{s}{=} I_{in} \cup x,\ a_{init} \in I_{in}$	Indicative parameter → Left-hand side (LHS)
Assign	$x = y$	ⓘⓘ $I_{out} \overset{s}{=} I_{in} \cup x\ ,\ y \in I_{in}$	Right-hand side (RHS) → LHS
Identity Setter	$x.set(y)$	ⓘⓘⓘ $I_{out} \overset{s}{=} I_{in} \cup x\ ,\ y \in I_{in}$	Tainted parameter → Caller object (e.g., y→x)
Call	$c.m(a_0, ..., a_n), n \in \mathcal{N}$	ⓘⓥ $I_{out} \overset{s}{=} I_{in} \cup \{a'_i\},\ \forall a_i \in I_{in}$	Caller parameter → Callee (Context switching)
Return	return $y; x = c.m(a_0, ..., a_n), n \in \mathcal{N}$	ⓥ $I_{out} \overset{s}{=} I_{in} \cup x\ ,\ y \in I_{in}$	Returned object → LHS

flows into an initialization statement used to instantiate e_{target}. *Initiate* can be extended to predicate *Initiate**, as defined in Table 4.1. In the *TrickMe* signature (Figure 4.6), predicate `Initiate*(n, f_i)` represents behavior that file `f_i` is instantiated using its identifier read from file "filenameFile" (Lines 13-16, Figure 4.3), which itself is downloaded from network connection `n` (Lines 11-14, Figure 4.2).

Control-flow predicates

These predicates capture the "*who*" of security-sensitive behaviors, *i.e.,* which entity or event controls them, a key determinant of the maliciousness of behaviors.

Predicate *Trigger.* The *Trigger* predicate asserts that a given security-sensitive behavior is triggered by a certain event. Specifically, predicate `Trigger(`$V_{trigger}$`, `e_{target}`, `A`)` is true if event $V_{trigger}$ triggers the execution path to a method call performing an action A (*e.g.,* upload information), where e_{target} (*e.g.,* URL connection) is the target entity whose reference in the program makes the security-sensitive method call. For example, the `onStop` method of `TrickMe` (Figure 4.3), which can be *triggered* by a System UI event such as pressing the HOME button, contains three specific behaviors. The three `Trigger(e,*,*)` predicates in Figure 4.6 capture this triggering relationship.

Predicate *Control.* The *Control* predicate asserts that a security-sensitive action a, performed by a reference of entity e_{target}, is control-dependent on another entity $e_{control}$. Specifically, predicate `Control(`$e_{control}$ `, `e_{target}`, `a`)` is true if and only if there exists an information flow from a reference of $e_{control}$ to a conditional statement guarding the execution of a security-sensitive method call, performing action a. *Control* can also be extended to predicate *Control**, as defined in Table 4.1. In the `TrickMe` example, the command sent from URLConnection `n` *controls* the three malicious behaviors. The predicates `Control*(n, *, *)` in Figure 4.6 encode this control relationship.

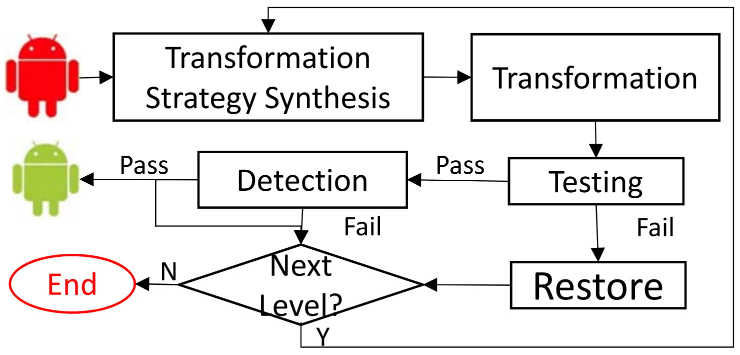


Figure 4.7: Overview of EnMobile

4.4 ENTITY-BASED STATIC ANALYSIS

In this section, we present how EnMobile matches an Android program against the given malware signatures specified with the entity-based characterization. Figure 5.5 presents the overview of EnMobile. EnMobile takes the bytecode of an app as input and outputs an entity-based characterization of the app's data and control flows, expressed in terms of the predicates defined in Section 4.3.1. At the meta level, such process takes four steps: (1) identify entities (*i.e.,* entity type and entity identifier); (2) map entities to program objects (*i.e.,* entity references); (3) extract entity-based flow facts through analysis on entity references and augment the extracted flow facts with provenance information; (4) match the flow facts against the malware signatures.

4.4.1 Identifying Entities and Entity References

For the purpose of analysis, EnMobile categorizes entity references into two types: initial entity reference and alias entity reference. Normally identifying an entity reference depends on a parameter (*e.g.,* file name) in the statement that initializes the reference. We call such a parameter as an *indicative parameter*. If an indicative parameter is constant or external input (*e.g.,* user input, network message), we term the entity reference initialized by the parameter as *initial entity reference*. If an indicative parameter is a variable that in turn points to the initial entity reference (*e.g.,* variable `file` and `r` in Lines 2-3, Figure 4.4), we term the entity reference initialized by the parameter as *alias entity reference*.

For brevity, we use only entities related to the SMS-sending behavior in `TrickMe` as examples to illustrate the techniques in the rest of the section. For each entity reference involved in sending SMS, we use red comments in Figure 3.1 to show the pair of the entity reference (*i.e.,* variable) and the entity that the reference points to. For example, in `TrickMe`, `url`, `f_s`, `f_c`, `f_info`, `f_n`, `f_d`, `f_f` in Figure 4.2 are initial entity references, while `n` in Figure 4.2 and `r`, `br` in Figure 4.4 are alias entity references.

Identifying entities. EnMobile identifies entities via initial entity references. In particular, EnMobile identifies the entity type through the Java types of initial entity references. For example, in Figure 4.2, `f_s` has java Type `File` indicating the entity type as file.

EnMobile extracts the entity identifier through the indicative parameter of the initial entity reference. An indicative parameter can either be constant or external input. For a constant identifier, EnMobile records the constant value as the identity of the entity. For an external-input identifier, EnMobile computes a symbolic expression as the identity of the entity. The symbolic expression is computed by a combination of sources of the variable (constant or user input) and the propagation paths from the sources to the variable. The reason why we choose to compute the symbolic expression instead of using constant propagation analysis to infer the actual value of the identifier is to deal with the situations where the identifier value goes through an encryption scheme.

Mapping entities to entity references. Initial entity references are naturally mapped to entities after identifying the entities. Mapping alias entity references to corresponding entities is still challenging for two main reasons. First, multiple references could point to the same entity. In Figure 4.4, `r` and `br` point to the file entity referred to by `file`. The identity of the entity can be *propagated* from one reference to another as one reference is used to initialize another object. Second, an entity reference may point to different entities under different execution contexts. In Figure 4.4, `r` and `br` can point to "commandFile", "coordinateFile", or "downloadFile" in different executions.

We develop an *identity propagation* algorithm to compute the entities that each alias entity reference points to. For a given initial entity reference, the identity propagation computes a set of entity references that point to the same entity as the initial entity reference; we refer to this set as *reference set*. The idea of identity propagation extends the idea of the taint propagation. The identity taints are generated at each initial entity reference. Any entity reference being tainted points to the same entity as the initial entity reference of the taint.

Table 4.2 informally presents the flow functions used in the identity propagation algorithm. A flow function of a statement maps the set of dataflow facts **in** that hold before the statement to the set of dataflow facts **out** that hold after the statement. Here a dataflow fact is the reference set of identities. In identity propagation, the flow function maps I_{in} (*i.e.,* reference set before the statement) to I_{out} (*i.e.,* reference set after the statement). In our implementation, I is a set of pairs <var, entity-ID>. We categorize program statements that can propagate identity taints into five types: entity initialization statement, assignment statement, identity setter method (*i.e.,* method that sets the identity of an entity), normal method (*i.e.,* method except entity initialization and identity setter methods), call statement, and return statement. Each type of statements is represented as a type of edges in "exploded supergraph" [93] of IFDS framework. We conservatively assume that reference sets remain the same for other edges (*e.g.,* edges do not belong to any of these five statement types) in the exploded supergraph [93]. Note that we omit formal details (*e.g.,* object sensitivity, context sensitivity) in the table. After the reference set has been calculated for each identity, EnMobile iterates through all identities and merges the reference sets if two identities are identical (*i.e.,* two identities with the same identifier value and same type).

In the `TrickMe` example, `n` in Figure 4.2 and `r`, `br` in Figure 4.4 are alias entity references. The identity taint `CON_1` is generated from `url` and propagated to `n` by applying $\textcircled{i}$ [1]. For `r` and `br`, the identity `File_2` first propagates from Line 3 in Figure 4.3 to variable `file` on Line 1 in Figure 4.4 by applying $\textcircled{iv}$. Then the identity further propagates to `r` and `br` by applying $\textcircled{i}$. Note that identity `File_3` also propagates (Line 5 in Figure 4.3) to `file`, `r`, and `br`. However, because our analysis is context-sensitive, the later information-flow analysis is able to tell that the variable `command` on Line 3 in Figure 4.3 is tainted by the information from `File_2`, and variable `axis[]` on Line 5 in Figure 4.3 is tainted by the information from `File_3`.

We perform two customizations in our information flow analysis. First, the sources of our identity propagation are based on a certain type of variables (*i.e.,* certain primitive type or string type of variables in initialization methods) instead of certain methods (*i.e.,* source methods). So in addition to method matching, identity generation requires an additional checking on method parameters. Second, the identity propagation is field-insensitive through certain methods (*e.g.,* initialization methods, setter methods). For example, in an identity setter method, an identity taint propagates from the method parameter directly to the receiver object rather than to the class field that is assigned by the taints in the setter method. To address such difference, we feed predefined knowledge (*e.g.,* initialization methods and parameters of entities) to help EnMobile perform identity propagation according to the high-level semantics.

4.4.2 Augmenting with Provenance Information

We omit the description of extracting flow facts through entity references given that this process is a standard information-flow analysis. In this section, we illustrate how we augment the extracted flow facts with provenance information in two steps.

[1]`URL` is used to initialize a new `HttpURLConnection` object in the implementation of `HttpURLConnectionImpl`.

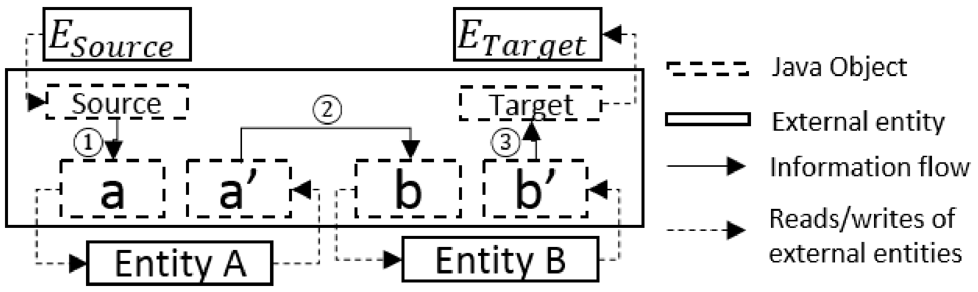


Figure 4.8: Malicious server configures `TrickMe` to perform click frauds

Classifying the type of information flows. In this step, we classify the type of flows based on the three types of data flows defined in Section 5.3. To differentiate the type of data flows, EnMobile needs to track which parameter of the sink method is tainted in the computation. EnMobile first performs traditional information flow analysis and takes the computed flows and sink variables as input, and checks them with the predefined method signatures to determine whether the information flow is *transmit*, *config*, or *initiate*. EnMobile takes lists of method signatures that contain the information of transmit, config, and initiate parameters in the methods. For each information flow, EnMobile derives the flow type based on the sink variable that the information flows into. For example, in the *SmsManager.sendTextMessage* method, the first parameter (destinationAddress) indicates that the flow is a config flow, and the third parameter (text) indicates that the flow is a transmit flow.

Computing control-flow predicates. To connect Event $V_{trigger}$ to Target Entity e_{target}, we first locate the security-sensitive method called by references of e_{target}. Each security-sensitive method corresponds to an action of the entity (*e.g.,* SEND for *sendTextMessage*). Then, we analyze the call path's entrypoints that lead to the method calls. The entrypoints are the top nodes in the call graph of the app. EnMobile follows the inter-component communications to link the E_{target}'s method call to the entrypoints, and the events $E_{trigger}$ can be further inferred from the entrypoints.

To compute control dependencies among entities. EnMobile tracks information flows from entities to conditional statements through inter-procedure control-flow graphs. The value of a conditional statement decides which program branch to take in runtime executions, and thus decides invocations of methods on one of the program branches. For a given method M invoked by an entity e_{target} (M corresponds to Action A), EnMobile computes the information flows from all entities to conditional statements (that control the invocations of M). If an information flow from an entity $e_{control}$ to the conditional statements exists, then $e_{control}$ controls the Action A of e_{target} (*i.e.,*`Control (`$e_{control}$`, `e_{target}`, A)` holds).

Table 4.3: Differentiating Malware and Benign Apps

Apps	#T	#AE	#AA	EnMobile(%)		Base1(%)		Base2(%)		Appo(%)	
				M.	B.	M.	B.	M.	B.	M.	B.
Benign	2716	1717	1592	1.0	99.0	0.9	99.1	4.8	95.2	59.1	40.9
Malware	5098	4897	4062	97.2	2.8	92.2	7.8	97.3	2.7	67.6	32.4

#T: Total #apps; #AE: #apps analyzable by EnMobile; #AA: #apps analyzable by Apposcopy
M.: % analyzed apps. predicted as malicious; B.: % analyzed apps. predicted as benign

4.4.3 Matching Against Signatures

To perform malware detection, EnMobile compares the set of flow facts $\mathcal{P}(M)$ extracted from an app M, using the aforementioned analysis, against a pre-compiled library of signatures of known malware. Specifically, for a malware signature $\mathcal{S}$ (as a set of predicates) from the malware library, the comparison checks whether the predicates in $\mathcal{S}$ are a subset of the flow facts (also as a set of predicates) in $\mathcal{P}(M)$, modulo renaming of variables. In the process of signature matching, EnMobile enumerates all feasible combinations of the segmented flows to match the end-to-end characterizations in the signatures. EnMobile determines the feasibility of combinations of segmented flows by incorporating the flow-sensitive information (*i.e.,* taking into account the order of the statements) in the extracted flow facts. For example, in Figure 4.8, for the flow from E_{Source} to E_{Sink} to occur, the flow ① from `URLConnection` to `InputStreamReader` must precede flow ② from `FileReader` to `FileOutputStream`, which must precede flow ③ from `FileReader` to `AdActivity`. Basically, whether two flows can be connected depends on the order between the sink of the previous flow and the source of the next flow (*i.e.,* the read from the entity should happen after the write to the entity).

We simplify this flow stitching problem to a graph reachability problem in which we transform the inter-procedure control-flow graph into a directed graph. The direction of the edges represents the order of the execution. For each sink in the computed flows, EnMobile searches in other flows of the same entity to check whether the sink can reach sources of these flows. If a source is reachable to the sink, then the flow B that the source belongs to can be connected with the flow A that the sink belongs to. We name B as a reachable flow of A. EnMobile maintains a list of reachable flows for each computed flow for further computation. Note that although EnMobile incorporates the Android lifecycle model into the flow computations, EnMobile considers only the sequential execution across Android components (*i.e.,* no clicks on back button) to lower false positives. Also, EnMobile does not consider constraints on the order of user-event callback methods (*e.g.,* onClick). EnMobile assumes that user-event callback methods for the same Activity component can be triggered in any order.

Table 4.4: Identification of malware by variations of AppContext, MUDFLOW, Drebin, and Apposcopy

	AppContext		**Drebin**		**MUDFLOW**	**Apposcopy**	**EnMobile**
	O.	**S.**	**O.**	**S.**			
P.(%)	95.42	76.52	98.47	92.80	97.61	74.48	99.64
R.(%)	97.65	95.15	97.48	89.21	53.46	67.60	97.24

P. = Precision, R. = Recall, O.= Result with Original Training Sample
S.= Result with Smaller Number of Training Sample

4.5 EVALUATION

We evaluate EnMobile in characterizing malware behaviors, by investigating the following research questions:

RQ1: How effective is EnMobile in characterizing malicious behaviors in existing malware?

RQ2: How do entity-identity analysis and richer data flow predicates in entity-based characterization contribute to the effectiveness of malicious-behavior identification?

RQ3: What is the effectiveness of EnMobile compared to other state-of-the-art approaches of malware detection?

4.5.1 Evaluation Setup

Evaluation Subjects. Our subject set consists of a malware dataset and a benign app dataset. Our malware dataset starts with all malware from the Genome [4] and Drebin [15] malware datasets, which are commonly used in malware detection research [65, 25, 66, 73, 22, 28]. The Malware Genome dataset comprises $1,260$ malware samples organized into 49 malware families and the Drebin dataset comprises $5,560$ malware samples organized into 178 malware families. We remove families containing fewer than 20 malware samples as well as malware samples duplicated across Genome and Drebin, yielding 27 families with $5,098$ malware samples in total. To collect benign apps, we download a total of $2,700$ apps (100 randomly selected apps for each of the 27 categories) from Google Play, as of December 2016. We implement EnMobile using several third-party static analysis frameworks, including Soot [94], FlowDroid [14], and AppContext [28]. To isolate and remove the effects of potential limitations of these frameworks on our conclusions, we further pre-run EnMobile on the complete subject set and filter out any apps that cause any of the third-party frameworks to throw exceptions or time out. This step gives us a final analyzable dataset of 4897 malware samples and 1717 benign apps.

Given the large number of unanalyzable benign apps (999), we reassess the distribution of our final benign app dataset. We find that it retains 52 to 77 apps in each Google Play category (originally 100); the size range (42KB to $51,192$KB) of a final app's file and the size range (16KB to $8,829$KB) of a final app's classes.dex file (bytecode without resource files) remain the same compared to the original dataset. This distribution suggests that our benign app dataset remains broadly representative of real benign apps even after removing unanalyzable apps. All runs of EnMobile have been performed on a desktop with 4 Intel Xeon 3.2 GHz E3-1225 processors and 16 GB of memory with a timeout of 20 minutes per app (the same default timeout set by Apposcopy [25]).

Malware Signature Library. For the purpose of this evaluation, we develop a library of malware signatures, one per family, for each of the 27 malware families (Table 4.5) in our dataset. For each malware family, we develop a signature characterizing that family in the signature language introduced in Section 4.3.1 using a small set of malware samples and benign apps. For a given malware family, this set consists of 10 randomly selected malware samples from the family and 100 randomly selected benign apps. To compose the malware signature, we first collect the security-sensitive behaviors (*i.e.,* data-flow and control-flow facts) that commonly exist in the malware samples and then remove the behaviors that match with benign behaviors. The signature-creation procedure entails fewer than six man-hours of effort per new malware signature, as a one-time effort for each malware family.

4.5.2 RQ1: Entity-Based Characterization

To evaluate the effectiveness of EnMobile's entity-based characterization on our malware dataset, we run EnMobile on all malware samples and benign apps except those that we used to develop the malware signatures, and perform two evaluations.

In the first evaluation, we record which malware family signatures (if any) each app matches[2]. Ideally, each malware sample should match its family's signature and no other signatures. Note that this classification problem is qualitatively harder than simply classifying a given app as malware or benign, and the true test of the accuracy of a signature-based approach, such as EnMobile. Column "EnMobile" in Table 4.5 shows the results of this evaluation. Here, for a given malware family, false negative rate (FN) refers to (among all samples in the malware family) the percentage of malware samples that are *not* matched

[2]Despite being theoretically possible, no apps end up matching multiple signatures in our current evaluation.

by EnMobile to that family's signature. Conversely, false positive (FP) refers to (among all benign apps and malware of *other* families) the percentage of apps that are (incorrectly) matched by EnMobile to this family's signature. As shown in Table 4.5, EnMobile can effectively classify malware instances into their appropriate families with on average 2.2% false negatives and around 0.05% false positives (shown as 0.1% in the table due to limited significant digits). For most malware families, EnMobile has under 5% false negatives and 0.1% false positives.

The second evaluation assesses the effectiveness of EnMobile and other approaches, in broadly differentiating malware from benign apps (vs. the family-based classification in Table 4.5), *i.e.,* classifying malware as malware (vs. benign) and benign apps as such (vs. malware). As shown in columns "EnMobile" in Table 4.3, here too EnMobile performs quite well, correctly classifying over 97% of the malware and 99% of the benign apps. Further on manually inspecting the 1% (*i.e.,* 17 out 1717) benign apps being classified as malware, we find that 8 apps actually possess malicious or highly suspicious behaviors. For example, a popular app (`com.genericsnippet.funnyecards`) contains code to download/execute payload from an unknown server[3] and contains a potentially unwanted library (MobClix). The other 9 benign apps misclassified by EnMobile contain some interesting suspicious-looking behaviors; for example, an app that turns a deprecated smartphone into a baby camera, sending SMS to parents whenever the phone signal changes. In future work, we plan to use app descriptions to check whether such suspicious behaviors are in fact desirable.

4.5.3 RQ2: Entity Identities and Flow Predicates

Two of the key contributions of EnMobile are (1) its entity-based characterization, built on top of entity-identity analysis (Section 4.4.1), and (2) the rich set of data-flow predicates (Section 4.3.1) to identify malicious intents. In this evaluation, we assess the effectiveness of these specific features by comparing EnMobile against two baseline versions: EnMobile without entity-identity analysis (Base1), and EnMobile without rich data-flow types (Base2). Note that the core information flow analysis in *both* Base1 and Base2, and indeed in EnMobile itself, is at least as precise as the type and/or API-based information flow analysis in previous work [65, 25, 14, 20, 22], albeit implemented in our own framework.

EnMobile without entity identities (*Base1*). To realize Base1, we turn off the entity-identity analysis in EnMobile. Specifically, the analysis retains the type of the entities, but ignores the identities of the entities in the flows. Note that we still need to perform identity

[3]We find that the app removes this behavior in its recent versions, potentially confirming this behavior as malicious or unwanted.

propagation to some extent to infer the entity type for some entity references. Of course, without entity identities, stitching segmented information flows cannot be performed either.

For fair comparison, we also modify EnMobile's malware signature library to make it suitable for Base1. Specifically, in each of the signatures, we remove entity identities but retain entity types. Further, we study malware reports from major anti-virus vendors as well as flows extracted by EnMobile to identify (segmented) information flows common to a majority of the samples in a malware family. We replace the original data-flow predicates, representing a connected flow in the signature, with a set of predicates representing each of the segmented flows. When no flows match a majority of the malware samples, we use flows with the best (highest) match.

Table 4.5 (evaluating characterization of malware by family) and Table 4.3 (evaluating basic malware detection of malware vs. benign) show a comparison of EnMobile (column EnMobile in both tables) to Base1 (column Base1). The results show that Base1, *i.e.,*EnMobile without entity identities, produces more false negatives for most malware families (7.1% on average vs. 2.2% for EnMobile) as well as in overall malware detection (7.8% vs. 2.8% in Table 4.3).

One main reason is that different samples in a malware family typically have different implementations of the same end-to-end flow through varied sets of segmented flows. Without the benefit of the entity-identity analysis, and the flow stitching that it enables, no single signature can characterize all samples of a malware family, *even* with the preceding custom retrofitting of the signature library for Base1. These results demonstrate the benefit of our entity-identity analysis for accurate malware characterization.

EnMobile without types of data flows (*Base2*). To realize Base2, we simply represent the three types of data flows as a single basic information flow, in both the signatures and in the extracted flow facts for each app. We then perform signature matching based on the extracted flow facts and signatures.

As shown in Tables 4.5 and 4.3, Base2 produces more false positives for some malware families and incorrectly marks more benign apps as malware than EnMobile (4.8% vs. just 1% in Table 4.3). The reason is that the signatures lacking our provenance information incur wrong matching of data flows. For example, the analysis may match a Transmit flow (*e.g.,* a flow sending an SMS) with a possible Config flow (*e.g.,* flows specifying the SMS recipient's number).

4.5.4 RQ3: Comparison with Related Approaches

We compare EnMobile with three related state-of-the-art approaches: one signature-based approach (Apposcopy [25]) and three learning-based approaches (MUDFLOW [65], AppContext [28], and Drebin [15]).

Comparison with a signature-based approach (Apposcopy). Apposcopy leverages a list of manually-specified signatures (*e.g.,* Figure 4.5) to match malware samples. Because Apposcopy provides signatures for only several malware families in our dataset, we use the following methodology to generate the best possible Apposcopy signatures uniformly for *all* malware families. We generate Apposcopy signatures for each family by two means: (i) we follow the same procedure as in creating EnMobile's signatures to manually create the signature based on 10 malware samples and 100 benign samples; (ii) We run Astroid [68], an automatic signature generator for Apposcopy, 10 times for each family. Each time Astroid randomly selects five samples from the malware family and produces a signature. We pick the best signature (in terms of the least total number of FP and FN) from among the preceding 11 signatures to report the results. All runs of Apposcopy are on the same machine as EnMobile with the same timeout threshold per app (20 minutes).

The last two columns ("Appo") of Tables 4.5 and 4.3 report the results of Apposcopy in detecting malware. As shown in Table 4.5, Apposcopy performs much worse than EnMobile for most of the malware families, especially the malware families whose most malware samples are from the Drebin malware database. Such effectiveness worsening is likely due to the evolution of malware. For example, in the `Kmin` malware family, the functionality of a receiver `com.km.HoldMessage` in some malware samples is replaced by a service `com.km.charge.CycleServic` in some other malware samples. This kind of evolution changes the type of the Android component hosting the malicious behavior. Such changes can easily evade Apposcopy's detection because Apposcopy's signatures heavily rely on the internal component structure (including a component's type) to characterize malware. However, EnMobile does not suffer from the same issue because such structural changes do not affect the end-to-end communications among entities.

Another issue that we observe by investigating the FN and FP produced by Apposcopy is that Apposcopy fails to characterize the essential malicious behavior shared across all samples in a malware family. For example, in the `Jifake` malware family, the only flow expressed through Apposcopy's signature is sending the current system time through SMS. However, some malware samples in Jifake do not possess such behavior. In fact, the behavior of installing another app is universal in this malware family. The incapability of Apposcopy to characterize such installation behavior results in high false negatives in this malware family.

Comparison with learning-based approaches. We also compare EnMobile with state-of-the-art learning-based detection approaches: AppContext [28], Drebin [15], and MUDFLOW [65].

Both AppContext and Drebin require a large number of malware samples as training data to train a machine learning model, but many malware families have very few known samples (only 42% of malware families have more than 5 samples [68]). So in addition to evaluate AppContext and Drebin[4] following traditional ten-fold cross-validations (**O.** in Table 4.4), we also evaluate their effectiveness on a smaller training set (**S.** in Table 4.4) by following the evaluation methodology used in Astroid [68]. Following such evaluation methodology, instead of training malware from all families as a whole, we perform the training and testing family by family. For each malware family, the training set consists 10 randomly selected samples from the family, all samples from other malware families, and a similar number of benign apps as in the original training set. The testing set consists of the rest of samples from the malware family and the rest of benign apps. We report the average results of all families in Table 4.4).

MUDFLOW detects malware by identifying abnormal information flows for each category of sensitive sources. To produce the input that MUDFLOW accepts, we use FlowDroid [14] to extract information flows from all of our subjects. We feed the extracted information flows with the SUSI category [47] of sources and sinks of these information flows and the permission list of each app to MUDFLOW to compute the final result.

Table 4.4 shows the effectiveness of the existing approaches and EnMobile. As shown in the table, EnMobile outperforms all the existing approaches. Note that although AppContext and Drebin reach similar effectiveness as EnMobile when training with the original dataset (*i.e.,* 90% training data and 10% testing data), their effectiveness downgrades a lot when using a smaller number of training samples. This result is especially impressive for EnMobile, considering that the difference between the smaller and original training datasets comes from much reduced malware samples in a single malware family. The downgrade indicates the overfitting nature of these learning-based approaches. Such result suggests that EnMobile can be a great substitute for learning-based approaches for malware detection when security analysts have access to only a small number of malware samples. EnMobile also outperforms MUDFLOW by much higher recall. The advantage of EnMobile over MUDFLOW lies in

[4]Since Drebin is not open source, we leverage the public feature vectors of 2,742 malware and 58,097 benign apps produced by Drebin to evaluate the effectiveness of Drebin in detecting malware. To establish a fair comparison, we randomly select feature vectors of 3,000 benign apps to make Drebin's dataset possess similar distribution (*i.e.,* percentages of malware and benign apps) as our dataset.

detecting those malware samples that have C&C behaviors or behaviors of dynamic code loading (*e.g.,* BaseBridge). Because of the dynamic nature of such behaviors (*i.e.,* the loaded code is unknown before the execution), traditional information-flow analysis often fails to detect these behaviors. Via the entity-based characterization, EnMobile can accurately identify the controlling entity of the downloading behavior and the command-and-control nature of the malware. Thus, EnMobile can outperform the existing approach by accurately identify these malware samples without requiring to know the details of dynamically loaded code.

4.6 DISCUSSION

Limitations. Intentional obfuscations of the entity identity may sabotage our analysis. For example, creating an alias entity by using symbolic links (*e.g.,* Ink, Shortcut), or using different copies of the same encryption scheme to encrypt the entity identity. In these cases, the malware may evade detection of EnMobile. However, since these camouflage attempts have clear patterns and are likely to be suspicious, other techniques such as dynamic analysis [95] can be used to complement EnMobile. Attackers can also hide malicious behaviors matched by our signature into dynamic loaded code to evade EnMobile's detection. However, security analysts can leverage EnMobile to further characterize the behaviors of dynamic code loading to detect the evolved malware. In our evaluation, signatures characterizing dynamic code loading can successfully match malware of corresponding families (*e.g.,* basebridge).

Threats of Validity. The tuning of malware signatures could affect the results of the evaluation. To prevent EnMobile's signatures from being overfitting for our subjects, when constructing the malware signatures, we strictly constrain ourselves in analyzing no more than 10 malware samples per family. Also, EnMobile is based on behavioral signatures rather than syntactic structures used in much of previous work [25, 15], and doing so further mitigates against overfitting. To avoid creating unfair signatures for Apposcopy, we further use Astroid [68] to generate signatures with different numbers of malware samples as input. We compare these signatures and demonstrate that our signatures selected for Apposcopy performs better.

4.7 CONCLUSION

We have presented EnMobile, a novel approach for accurately characterizing mobile apps' interactions with entities. We have demonstrated a practical application of EnMobile for detecting malware. Our results suggest the effectiveness of EnMobile in characterizing differential characteristics of malware and benign apps, and robustness of EnMobile's specification-driven signature (*i.e.,* based on intrinsic definitions of malware) over implementation-driven ones (*i.e.,* based on features of low-level program structures). We envision a number of applications of EnMobile: with increasing uses of IoT apps, EnMobile can be extended for characterizing broader interactions between the physical world and apps; for human-assisted app auditing, entity-based characterization can enhance security analysts' understanding of app behaviors.

Table 4.5: Categorization of Malware by EnMobile

Malware Family	#T	EnMob(%)		Base1 (%)		Base2 (%)		Appo(%)	
		FN	FP	FN	FP	FN	FP	FN	FP
ADRD	91	0.0	0.0	2.3	0.0	0.0	0.0	36.4	0.0
AnserverBot	184	0.6	0.1	2.2	0.0	0.6	0.4	0.0	0.0
BaseBridege	331	10.4	0.2	33.4	0.2	10.4	0.5	50.0	0.1
Boxer	27	0.0	0.2	7.4	0.2	0.0	0.2	25.9	0.4
DroidDream	97	0.0	0.1	4.4	0.1	0.0	0.2	3.1	8.5
DroidDreamLight	46	1.1	0.0	1.1	0.0	1.1	0.0	0.0	0.0
DroidKungFu	668	1.8	0.0	5.1	0.0	1.6	0.7	8.1	0.0
ExploitLotoor	70	10.4	0.1	17.9	0.1	10.4	0.4	85.0	1.9
FakeDoc	132	2.3	0.1	11.0	0.1	2.4	0.2	6.3	0.3
FakeInstaller	925	1.8	0.0	5.8	0.0	1.8	0.1	68.4	0.1
FakeRun	61	0.0	0.1	4.9	0.1	0.0	0.3	11.1	0.0
Gappusin	58	8.6	0.0	12.0	0.0	8.6	0.1	58.6	0.0
Geinimi	94	0.0	0.2	4.4	0.2	0.0	0.9	0.0	0.0
GingerMaster	342	3.8	0.1	7.1	0.1	3.8	0.2	70.4	0.0
GoldDream	70	0.0	0.0	1.6	0.0	0.0	0.0	3.2	0.0
Hamob	28	4.5	0.0	18.1	0.0	4.5	0.0	3.7	5.7
Iconosys	152	0.6	0.0	2.6	0.0	0.6	0.2	69.7	0.0
Imlog	43	0.0	0.0	0.0	0.0	0.0	0.0	76.7	29.1
Jifake	29	0.0	0.0	3.4	0.0	0.0	0.0	50.0	0.5
KMin	148	3.7	0.1	5.9	0.1	3.7	0.1	34.7	19.1
MobileTx	69	0.0	0.1	8.8	0.1	0.0	0.6	31.9	0.0
Opfake	613	0.8	0.0	5.6	0.0	0.8	0.1	33.7	19.2
Pjapps	58	0.0	0.0	0.0	0.0	0.0	0.2	0.0	0.0
Plankton	625	2.1	0.0	3.7	0.0	2.1	0.2	32.1	3.0
SendPay	59	6.9	0.0	8.6	0.0	6.9	0.0	22.4	7.9
SMSreg	41	0.0	0.0	9.8	0.0	0.0	0.3	16.2	44.6
YZHC	37	0.0	0.0	3.6	0.0	0.0	0.1	0.0	5.2
Average	188.8	2.2	0.1	7.1	0.1	2.2	0.2	29.5	5.4

#T = Total number of apps, EnMob = Enmobile, Appo = Apposcopy
FN = False negative rate, Base1 = EnMobile without entity identity
FP = False positive rate, Base2 = EnMobile without data-flow types

CHAPTER 5: MALWARE DETECTION IN ADVERSARIAL SETTINGS: EXPLOITING FEATURE EVOLUTIONS AND CONFUSIONS IN MOBILE APPS

5.1 OVERVIEW

To fight against malware, a signature-based technique extracts malicious behaviors as signatures (such as bytecode or regular expression) while a more complicated machine-learning-based technique learns discriminant features from analyzing semantics of malware. One major challenge for both signature-based and learning-based malware detection approach is to form an informative feature set for signature or detection model. To address challenge, existing malware detection tends to include as many features as possible. For example, Drebin, a recently published malware detection work [15], uses the feature set containing 545,334 features. Recent study [16] shows that such large feature set has numerous non-informative or even misleading features. Therefore, in this chapter, we investigate the question: *can a malware be mutated to evade detection by changing its feature values while maintaining its malicious behaviors*[1] *?* More formally, we name such "mutations of malware based on feature values" as *Malware Recomposition Variation (MRV).*

A key observation made in our research is that, features, which abstract concrete malicious behaviors, are fragile, and they could be easily mutated (*i.e.,* changed). The susceptibility of such features makes it possible to evade detection if malware are properly mutated [96, 97, 98]. Our research suggests that *features that are unique to malware are not necessary needed for forming malicious behaviors.* Such result is mainly due to two factors.

First, learning-based detectors often confuse non-essential features (*i.e.,* features that are not essential for forming malicious behaviors) in code clones as discriminative features. Copy-paste practice is prevalent in malware industry which result in many code clones in malware samples [74]. Because the same code has appeared in many malware instances, learning-based detectors may regard non-essential features (*e.g.,* minor implementation detail) in code clones as major discriminant factors (because the same pieces of code appeared in many malware samples but not in benign apps). Learning-based detector place higher weight on these features not because these features are essentail to malicious behaviors but because these features appeared in malware much more frequently than in benign apps. Adversaries could simply leverage such fact to mutate some of these non-essential features with higher weight in detecting model to evade detection.

[1]We define malicious behaviors as the invocations of security-sensitive method calls in malware, more specifically the invocations of permission-protected methods in Android.

Second, the features essential to malicious behaviors are different for each malware family. Almost all existing learning-based malware detection using a universal feature set to detect malicious samples for all malware families. However, based on recent research result [99] mined from 1,068 research papers and malware documents, each malware family associates with a distinct set of malware behaviors and concrete features. Using a universal set of features for all malware families would result in a large number of non-essential features to characterize each family. Furthermore, as previously mentioned, if these non-essential features are unique in some malware samples, the trained detection model can be evaded by mutation the value of the non-essential features.

In this work, we focus on synthesizing mutation strategies (*i.e.,* what kind of features we should mutate to evade detection) and automating program transformation (*i.e.,* how to apply mutations on malware bytecode to ensure the robustness of the app while preserving malicious behaviors). Different from existing work [96, 97, 98], we explore the capability of attackers in a more realistic attacking scenario: the attackers can feed any app as the input to the detector and know the binary detection result (*i.e.,* detected as malware or not) without any additional information.

There are three major challenges to conduct MRV.

Evading Malware Detectors. To evade a malware detection model, an adversary need to identify the non-essential features and compute the mutated feature value that can evade detection. This usually requires an adversary to possess internal knowledge and understanding of malware detectors. Unfortunately, generally an adversary may have little (or even no) knowledge about the malware-detection model (such as features and algorithms). Moreover, the particular knowledge to a single malware-detection model is too specific and conducting MRV is unlikely to succeed, especially if the model (*e.g.,* the one in VirusTotal) is based on combining multiple techniques.

Preserving Malicious Behaviors. The mutated malware should maintain the original malicious purposes and therefore simply converting malware's feature values to another app's feature values is likely to break the malicious . For example, the malicious behaviors are usually designed to be triggered under certain contexts (to avoid user attention and gain maximum profits [28]), and the controlling logic of the malware is too sophisticated (*e.g.,* via logic bombs and specific events) to be changed.

Maintaining the Robustness of Apps. The mutated malware should be robust enough to be installed and executed in mobile devices. Automatically mutating an app's feature values is likely to break the code structures and therefore cause the app to crash at runtime.

To tackle these challenges, MRV employs two mutation strategies, *Malware Evolution Attack* and *Malware Confusion Attack* (Section 5.4), that generally reflects the susceptibil-

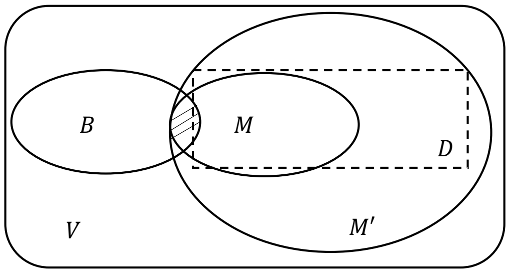


Figure 5.1: A feature vector space V, the feature vectors of existing benign apps B, the feature vectors of existing malware M, the feature vectors can be detected by detection model D, the feature vectors of all potential malware M' and their relationships.

ity of a detection technique to the mutations of malware feature values. To applying the mutation strategies without breaking the dependencies and functionalities in the program, we develop a new technique, inspired by program transplantation [100] to reuse the existing implementations of desired features instead of randomly mutating or synthesizing the code. In particular, we develop a transplantation framework capable of inter-method, inter-component, and inter-app transplantation[2]. By leveraging the existing implementations, this technique enables systematic and automatic mutations on malware samples while aiming to produce a well-functioning app.

Main Contributions. This work makes the following main contributions.

• **Observation.** We identify differentiability of selected features and robustness of detection models as two fundamental limitations of malware detection, from which we demonstrate the feasibility of producing effective attacks (Sec. 5.2).

• **Attacks.** We propose malware recomposition variation (MRV) to produce two attack types (*feature evolution attack* and *feature confusion attack*) to effectively mutate existing malware for evading detection (Sec. 5.4).

• **Framework.** We develop a transplantation framework capable of inter-method, inter-component, and inter-app transplantation to automatically mutate app features (Sec. 5.5).

5.2 MRV DESIGN

5.2.1 Limitations of Malware Detection

MRV leverages two fundamental limitations of malware detection: *differentiability of selected features* and *robustness of detection model.* To better illustrate the limitations, we model the vector space of features used by any given malware-detection technique as V (shown in the Venn diagram in Figure 5.1).

[2]Transplanting a feature in one app/component/method (*i.e.,* donor) to a different app/component/method (*i.e.,* host).

```
public class User extends Application{
public String androidid;
public String tel;}
```

Figure 5.2: `User` class of DougaLeaker malware

The differentiability of selected features can be represented by the intersection of the feature vector space (denoted as B) for the existing benign apps and that (denoted as M) of the existing malware. In an ideal case, if the selected features are perfect (*i.e.,* all differences between benign apps and malware are captured by features), no malware and benign apps should be projected to the same feature space, *i.e.,* $B \cap M = \varnothing$. Such perfect feature set, however, is difficult or even impossible to get in practice. For example, to detect a malware that loads a malicious payload at runtime, a malware detector could use the name of the payload file as a feature for the detection. Unfortunately, the name of the payload file can be easily changed to a common file name used by benign apps to evade the detection, therefore resulting in false negatives. If the detector removes such a feature in fighting malware, the detector produces false positives by incorrectly catching benign apps that may have behaviors of dynamic code loading. In either way, the selected feature set is imperfect to differentiate such malware and benign apps.

The robustness of a detection model can be represented by the difference between the feature vectors (denoted as M') of all potential malware and the feature vectors (denoted as D) that can be detected by the detection model[3]. Such difference can be denoted as $M' \setminus D$. A perfect detection model should detect all possible malicious feature vectors (*i.e.,* M'). In practice, detection models are limited in detecting existing malware because it is hard to predict the form of potential malware (including zero-day attacks). In this work, we argue that a robust malware-detection model should aim to detect malware variants produced through known mutations. Such mutations should employ not only syntactic and semantic obfuscation techniques, but also feature mutations based on analyzing the evolutions of malware families.

[3]We safely assume that for a reasonable malware-detection model, $D \subseteq M'$. A reasonable malware-detection model produces false positives on a benign app only because the feature vector of the benign app is shared by some malware.

```
public class MainActivity extends Activity{
public void onCreate(android.os.Bundle b){
super.onCreate(b);
this.requestWindowFeature(1);
User u = (User) getApplication();
u.androidid = Settings.Secure.getString(getContentResolver(), "android_id");
u.tel = getSystemService("phone").getLine1Number();
if(isRegisterd(u.androidid)){
Cursor cursor = managedQuery(ContactsContract.Contacts.CONTENT_URI, 0, 0, 0, 0);
while (cursor.moveToNext() != 0) {
this.id = cursor.getString(cursor.getColumnIndex("_id"));
this.name = cursor.getString(cursor.getColumnIndex("display_name"));
this.data = new StringBuilder(String.valueOf(this.data)).append("name:").append(this.name).
    toString();
}
cursor.close();
}else{
startService(new Intent(getBaseContext(), MyService.class));
}
}
this.exec_post(this.data);}} //sending contacts through HttpPost
```

Figure 5.3: `MainActivity` of DougaLeaker malware

5.2.2 Threat Model

We assume that an attacker has only black-box access to the malware detector under consideration. Under such assumption, the attacker can feed any malware sample as the input to the detector and know whether the sample can be detected or not, but the attacker has no internal knowledge (*e.g.,* detection model, signature, feature set, confidence score) about the detector. The attacker is capable of manipulating the malware's binary code, but has no access to the malware's source code. We assume that the attacker has access to the existing malware samples (*i.e.,* samples that are correctly detected by the malware detector), and the goal of the attacker is to create malware variants with the same malicious behaviors, but can evade the detection.

5.2.3 Overview of MRV

We propose *Malware Recomposition Variation (MRV)*, the *first* work that systematically reconstructs new types of malware using decompositions of features from existing malware to evade detection. Fig 5.5 illustrates the whole framework used to generate app mutants. MRV first performs *mutation-strategy synthesis* (Sec. 5.4) including both *feature evolution attack* and *feature confusion attack*, and then MRV leverages program transplantation to mutate the existing malware (Sec. 5.5) where program testing and malware detection are used to find the survival app mutations. Note that MRV is an iterative process. When MRV finishes the initial round of mutation, the evasive sample set and detected sample set get updated. The update of evasive samples enable the generation of new mutation strategy (trough confusion attack). The update of detected samples provides new candidates to produce evasive malware. Note that the set of evasive samples at the beginning of first of iteration is empty, MRV generate the initial set of evasive samples through evolution attack,

```
public class MyService extends Service{
public int onStartCommand(Intent intent, int flags, int startId){
User u = (User) getApplication();
String text = "android_id = " + u.androidid + "; tel =" + u.tel;
Date date = new Date();
if(date.getHours>23 || date.getHours< 5 ){
android.telephony.SmsManager.getDefault().sendTextMessage(this.number, null, text, null, null);
}
return;}}
```

Figure 5.4: `MyService` of DougaLeaker malware

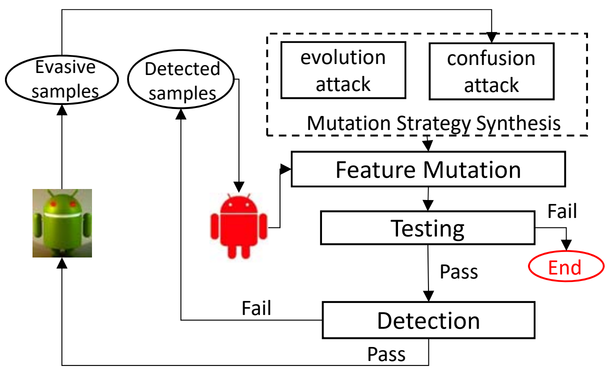


Figure 5.5: Illustration of mutant construction in evolution MRV. Key steps: (1) mutation-strategy synthesis; (2) program mutation/feature mutation; (3) program testing.

then iteratively generate new evasive samples through confusion attack.

Feature evolution attack is based on the insight that reapplying the feature mutations in malware evolution can create new malware variants that may evade detection (*i.e.,* the feature vectors fall into the area of $M' \setminus D$). As Figure 5.5 shows, the attack mutates feature values iteratively at each level (following the sequence of temporal feature, locale feature, and dependency feature).

Feature confusion attack is based on the insight that malware detection usually performs poorly in differentiating the malware and benign apps with the *same* feature vector. As discussed earlier, if we simply mutate malware feature vectors to benign feature vectors (*i.e.,* feature vectors in space B), such mutation would generally break or weaken the malicious behaviors (*i.e.,* turning the malware into benign apps). So our design decision is converting malware with unique malicious feature vectors (*i.e.,* $M \setminus (B \cap M)$) to possess the feature vectors shared with benign apps (*i.e.,* $B \cap M$). Because some malware already possess such feature vectors, we could leverage the program transplantation technique to transplant the existing implementation to the host malware. Using program transplantation greatly decreases the likelihood of breaking the original malicious behaviors in the host malware. Instead of mutating an individual feature value iteratively at each level, feature confusing attack mutates the whole feature vector.

Use Cases. Although we present our techniques as attacks to malware detection, the techniques can also be used in assisting the assessment or testing of existing malware-detection techniques, to enable the iterative design of a detection system. The main idea is to launch feature evolution attack and feature confusion attack on each revision of the detection system, so that security analysts can further prune their selection of features in the next revision. *Feature evolution attack* can be used to evaluate the robustness of a detection model. The more robust the detector model is (*i.e.,* the larger D is), the more difficult for

```
public class User extends Application{
  public String androidid;
  public String tel;}
```

Figure 5.6: `User` class of DougaLeaker malware

```
public class MainActivity extends Activity{
  public void onCreate(android.os.Bundle b){
    super.onCreate(b);
    this.requestWindowFeature(1);
    User u = (User) getApplication();
    u.androidid = Settings.Secure.getString(getContentResolver(), "android_id");
    u.tel = getSystemService("phone").getLine1Number();
    if(isRegisterd(u.androidid)){
      Cursor cursor = managedQuery(ContactsContract.Contacts.CONTENT_URI, 0, 0, 0, 0);
      while (cursor.moveToNext() != 0) {
        this.id = cursor.getString(cursor.getColumnIndex("_id"));
        this.name = cursor.getString(cursor.getColumnIndex("display_name"));
        this.data = new StringBuilder(String.valueOf(this.data)).append("name:").append(this.
            name).toString();
      }
      cursor.close();
    }else{
        startService(new Intent(getBaseContext(), MyService.class));
      }
    }
    this.exec_post(this.data);}} //sending contacts through HttpPost
```

Figure 5.7: `MainActivity` of DougaLeaker malware

a mutated malware to evade detection (*i.e.*, the smaller $M' \setminus D$ can be). The detail of each step is elaborated in subsequent sections. *Feature confusion attack* can be used to evaluate the differentiability of selected features. The more differentiable a feature is, the less the opportunity is for a malware to confuse the detector (*i.e.*, smaller $B \cap M$ is desirable).

5.3 RTLD FEATURE MODEL

In this work, we characterize semantic features using our proposed RTLD feature model, which aims to reflect the essential malicious behaviors while balancing between the computational efficiency and accuracy. The RTLD feature model is a general model summarizing the essential features (*i.e.*, security-sensitive resources) and contextual features (*e.g.*, when, where, how the security-sensitive resources are obtained and used) commonly used in malware detection.

The RTLD model covers four main aspects: *Resource* (what are the security-sensitive resources obtained by malicious behaviors), *Temporal* (When are the malicious behaviors triggered), *Locale* (Where do the malicious behaviors occur), and *Dependency* (How are the malicious behaviors controlled).

We use the simplified code snippet of the DougaLeaker malware[4] shown in Figure 3.1 to illustrate the feature model. The code snippet shows two malicious behaviors of the DougaLeaker malware. First, the malware saves the Android ID and telephone number of the victim device to global class `User` when the app starts (Lines 5-7 in Figure 4.2). Then, the malware reads the contacts on the victim device (Lines 8-13 in Figure 4.2) and sends the contacts to a malicious server (Line 20 in Figure 4.2). The malware also starts a service that sends the Android ID and telephone number to the malicious server through text messages between 11PM and 5AM.

The **resource** features describe the security-sensitive resources exploited by malicious behaviors while the dependency features further represent how the malicious behaviors are controlled. We locate resource features by constructing call graphs and identifying call graph nodes of the security-sensitive methods (including methods for accessing permission-protected resources and methods for executing external binaries/commands). We compile the list of security-sensitive methods based on PScout [46] and construct the call graphs using the SPARK callgraph algorithm implemented in Soot [101]. The call graphs represent the invocation relationships between the app's entrypoints and permission invocations. We save the entrypoints of the call graphs in this step to trace back to the other features in

[4]MD5 of the malware is e65abc856458f0c8b34308b9358884512f28 bea31fc6e326f6c1078058c05fb9.

```
public class MyService extends Service{
  public int onStartCommand(Intent intent, int flags, int startId){
    User u = (User) getApplication();
    String text = "android_id = " + u.androidid + "; tel =" + u.tel;
    Date date = new Date();
    if(date.getHours>23 || date.getHours< 5 ){
      android.telephony.SmsManager.getDefault().sendTextMessage(this.number, null, text, null,
          null);
    }
    return;}}
```

Figure 5.8: `MyService` of DougaLeaker malware

later steps. For the DougaLeaker example, we can locate the `HttpPost` method invocation (not shown in Figure 3.1) in `exec_post` and `sendTextMessage` method invocation (Line 7 in Figure 5.8) in `onStartCommand` in the call graph. Due to space limit, we omit many details here. For the detailed algorithm that we used for extracting RTLD features, please refer to our accompanying technical report [102].

The **temporal** features describe the contexts when the malicious behaviors are triggered. To extract temporal features, we identify three categories of temporal features based on the attributes of their entrypoints. (i) For system events handled by intent filters, their entrypoints are lifecycle methods. The components of the lifecycle methods should have intent filters specified. (ii) For both system events captured by event-handling methods and UI events, their entrypoints should be event-handling methods. (iii) For lifecycle events, their entrypoints are lifecycle methods, and these lifecycle methods have not been invoked by other events (due to inter-component communication).

The **locale** features describe the program location where the malicious behavior occurs. The location of the execution is either an Android component (*i.e.,* `Service`, `Activity` and `Broadcast Receiver`) or concurrency constructs (*e.g.,* `AsyncTask` and `Handler`). Malicious behaviors get executed when these components are activated. Due to the inter-component communication (ICC) in an Android program, the entrypoint component of a malicious behavior could be different from the component where the behavior resides in.

The locale features in general reflect the visibility of a task (*i.e.,* whether the execution of the task is in the foreground or background) and continuity (*i.e.,* whether the task is once-off execution or a continuous execution, even after exiting the app). For example, if a permission is used in a Service component (that has not been terminated by `stopService`), then the permission use is running in the background, and also it is a continuous task (even after exiting the app).

The **dependency** features describe the control dependencies of the invocation of the malicious behavior. A control dependency between two statements exists if the truth value of the first statement controls whether the second statement gets executed. Malware frequently leverage external events or attributes to control malicious behaviors. For example, the `DroidDream` malware leverages the current system time to control the execution of its malicious payload. It suppresses its malicious payload during the day but allows the payload executions at late night when users are likely sleeping.

We construct inter-procedure control-flow graph (ICFG) to extract dependency features. Based on the ICFG, we construct the subgraphs from each entrypoint to the resource feature (*i.e.,* security-sensitive method call). For each subgraph, we traverse the subgraph to identify the conditional statements that the security-sensitive method invocation is control-dependent on. The value of a conditional statement is used to decide which program branch to take in runtime executions, and thus decide whether a security-sensitive method invocation on one of the program branches can be executed or not. We say that such conditional statement controls the invocation of the method. Finally, we save the set of extracted conditional statements as dependency features with the resource features and the corresponding location/temporal features. Figure 5.9 shows the ICFG of the `onCreate` and `onStartCommand` methods. As shown in the Figure, the `sendTextMessage` method in `onStartCommand` (Line 7) is controlled by the conditional statement on Line 6 in `onStartCommand` and the conditional statement on Line 8 in `onCreate`. On the other hand, the `exec_post` method in `onCreate` is not controlled by any conditional statement, and thus the security-sensitive behavior in `exec_post` does not have any dependency feature.

5.4 MUTATION STRATEGY SYNTHESIS

We present our techniques of synthesizing strategies to mutate program features based on two scenarios: *black-box scenario* and *informative scenario.* In *black-box scenarios*, adversaries have no knowledge about malware-detection techniques (*e.g.,* features, models, algorithms). So instead of developing targeted malware to evade specific detection techniques, we propose a more general defeating mechanism called **evolution attack**: *mimicking and automating the evolution of malware.* Such defeating mechanism is based on the insight that the evolution process of malware reflects the strategies employed by malware authors to achieve a malicious purpose while evading detection. In *informative scenarios*, adversaries know the type of algorithms used in the detection, and therefore we develop the targeted attack called **confusion attack**. The main idea of malware confusion attack is to mimic the malware that can generally evade detection, *i.e.,* confusing the malware detectors by modifying the feature values that can be shared by malware and benign apps. In implementation, to find the features that can cause confusion and evolution, we first project all apps to the RTLD feature spaces) and then we follow the following steps to generate new attacks.

To generate **evolution attack**, we identify a feature set called *evolution feature set.* In the set, each feature is evolved either at intra-family level or inter-family level. For each feature vector in the *evolution feature set*, we count the number of evolutions as *evolution weight*, where *intra-family evolution weight* is proportional to the number of evolutions at intra-family level, and *inter-family evolution weight* is proportional to that at inter-family level. The rationale is that if the feature type has already been evolved frequently under observation, it is more likely to be evolved according to the nature of the law (in biological evolution process [103]). [5].

To generate **confusion attack**, we identify a set of feature vectors that can be projected from both benign apps and malware as *confusion feature set*. For each feature in the confusion feature set, we count the number of benign apps that can be projected to the feature vector as the *confusion weight* of the feature vector. The rationale is that if more benign apps are projected to the feature, it is harder for the malware detector to label the apps with this feature as malicious.

After the preceding step, in both **evolution attack** and **confusion attack**, for each malware that we aim to mutate, we first check whether the resource feature appears in any *critical feature set*, which denotes the *evolution feature set* in the context of *evolution attack* and the *confusion feature set* in the context of *confusion attack*, respectively. If a resource feature R appears in a vector V in the critical feature set, we then mutate the original feature vector of R to be the same as vector V by mutating the contextual features. A resource feature could appear in many vectors in the critical feature set. Then we mutate top K matching vectors ranked by the corresponding evolution or confusion weight. Otherwise, we leverage *a similarity metric* [6] to find another resource feature (in the critical feature set) R' that is most likely to be executed in the same context as R. Similarly we select top K vectors (ranked by the corresponding evolution or confusion weight) matching R' as the target vectors for mutation. Finally, if any mutated malware passes the validation test (Section 5.6) and evades detection, then evolution/confusion attack successfully produces a malware variant given the fact that each malware generally corresponds to multiple mutated malware. Empirically we set $K = 10$ in our experiments.

5.5 PROGRAM MUTATION

In this section, we present how MRV mutates existing malware based on synthesized mutation strategies. The mutation process is essentially a program transformation that keeps the the malicious behavior (*i.e.,* resource feature) while mutateing the context features. To mutate the context features, we develop a program transplantation framework that satisfies two needs: (a) transplanting the malicious behavior to different contexts in the existing program; (b) transplanting the contextual features from other programs into the existing contexts. For details of the mutation, please refer to our technical report [102].

[5]The number labeled in the bottom of each phylogenetic tree denotes the distance between two nodes. The node could be a leaf node for denoting a malware, and also could be an internal node for denoting a cluster grouped from its children node. The Hungarian-type algorithm [104] is used to compute the malware distance based on the RTLD features.

[6]Please refer to our technical report for more details.

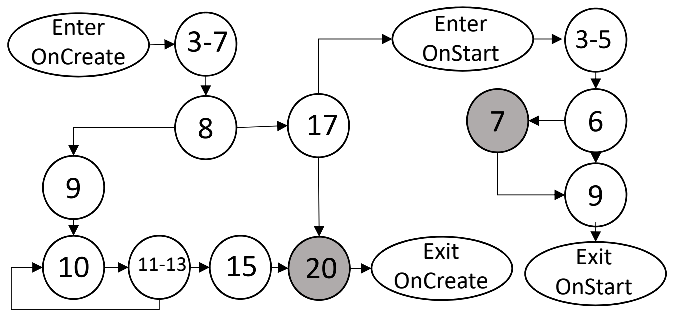


Figure 5.9: Inter-Procedural Control Flow Graph of DougaLeaker

5.5.1 Transplantation framework

Transplantation is the process that transplants the implementation of a feature (*i.e.,* organ) from one app (*i.e.,* donor app) to another app (*i.e.,* host app) [105]. We broaden the concept of transplantation to components and methods. Transplantation takes four steps: identification of the organ (*i.e.,* code area that needs to be transplanted), extraction of the organ, identification of the insertion point in the host and adaption of the organ to the host's environment.

In our transplantation framework, we take different strategies based on the type of features that need to be mutated. On one hand, to mutate the temporal features or locale features (that are usually simple to solve) of the program, we identify or construct a suitable context (that satisfies the targeted value of temporal features or locale features) in the existing program, and then transplant the malicious behavior (*i.e.,* resource feature) to the identified or constructed location. On the other hand, to alter the dependency features that usually require sophisticated ways (*i.e.,* specific method sequences) to achieve the desired control, we transplant the existing implementation of such control (*i.e.,* organ) from a donor app to the host app.

Such two-strategy design aims to simplify the existing software transplantation problem. In the first strategy, the transplantation is actually intra-app. We simply save and pass the unresolved dependency and contextual information (*e.g.,* values of parameters) in the program via setting the variables and fields global. In the second strategy, although the transplantation is inter-app, we just need to transplant a program slice that contains a few dependencies. Such transplantation is lightweight compared to transplanting the whole implementation of a functional feature in previous work [105]. Intra-app transplantation is feasible for temporal and locale features because synthesizing a new entrypoint or a new component within an existing Android program results in little or no impact to other areas of the program. Mutation of dependency features requires inter-app transplantation because synthesizing new dependencies within the program is challenging. The tight coupling of dependencies brings huge impact to other program behaviors and likely causes the mutated program to crash.

Note that although temporal features and local features all require the transplantation of the malicious behaviors, the donor (*i.e.,* area of code) that requires transplantation is different. The related code of a malicious behavior can be separated as the triggering part and the execution part. These two parts may not be in the same component. For example, in Figure 5.9, the malicious behavior of sending text message can be separated as the triggering part in the `OnCreate` method of activity component and execution part in the `OnStartCommand` method of the service component. To mutate temporal features, the donor to be transplanted is the triggering part. To mutate locale features, the donor to be transplanted is the execution part.

We categorize the transplantation based on the locality into three levels: inter-method, inter-component, and inter-program transplantation, which are illustrated below.

Listing 5.1: Code snippet of mutated DougaLeaker malware

```
public void onClick(View v) {
    User u = (User) getApplication();
    u.androidid = Settings.Secure.getString(getContentResolver(), "android_id");
    u.tel = getSystemService("phone").getLine1Number();
    if(!isRegisterd(u.androidid)){
     String text = "android_id = " + u.androidid + "; tel =" + u.tel;
     Date date = new Date();
     if(date.getHours>23 || date.getHours< 5 ){
       android.telephony.SmsManager.getDefault().sendTextMessage(MyService.number, null, text,
           null, null); } }}
```

Listing 5.1 shows the mutated code related to the SMS-sending behavior in Figure 3.1. The mutation strategy consists of two mutations: (i) to mutate the temporal feature from lifecycle event “entering the app” (*i.e.,* `onCreate` of MainActivity) to UI event “clicking the button” (*i.e.,* `onClick` of a button’s event listener), (ii) to mutate locale feature from Service to Activity.

5.5.2 Inter-method transplantation

Inter-method translation refers to the migration of malicious behaviors (*i.e.,* resource features) from a method to another method in the same component. We observe that such transplantation is commonly performed to mutate the temporal features. For example, the mutation of temporal feature in Listing 5.1 is inter-method transplantation (Lines 2-5 of `onClick` method in Listing 5.1 are transplanted from Lines 5-8 of the `onCreate` method). In the case of temporal features, the organ that needs to be transplanted is the entry of the malicious behavior and its dependencies. The entry of the malicious behavior is the first node on the call graph path leading to the malicious behavior. For example, `startService` is the entry of the SMS sending behaviors. In order to locate the entry of the malicious behavior, we construct call graphs from the `entrypoint` of the program (corresponding to the feature to be mutated) to the malicious method call. We then mark the node directly connected to the entrypoint on the call graph as the entry of the malicious behavior.

Then, we extract all dependencies related to the entry. To ensure the entry method to be invoked under the same context (*e.g.,* parameter values), we perform a backward slicing from the entry method until we reach the entrypoint of the program. For example, in Figure 5.9, nodes 3-7 and 8 are all dependencies related to the entry (*i.e.,* node 17, startService). The corresponding statements are the code snippet to be transplanted. Next, we create an entrypoint method that can provide temporal features that we need. The entrypoint creation is done by either registering an event handler for system or UI events or creating a lifecycle method in the component. We also edit the manifest file to register receiver components for some of system events. For example, in Listing 5.1, we create an event listener and an `onClick` method to provide the temporal feature that the mutation needs.

Finally, we need to remove the organ from donor methods. If some of statements are dependent on the organ, the removal can cause the donor method to crash. To avoid the side-effects of the removal, we initialize a set of global variables with the local variables in the organ. We then replace the original dependencies on the organ by making the statements dependent on the new set of global variables. We note that in some instances, the host method is invoked after the donor method, so the set of global variables may not be initialized when the donor method is invoked. So when replacing the dependencies, we add conditional statements to check for null to avoid `NullPointerException` in the donor method. For example, after transplanting Lines 5-8, we need to remove Line 7 while keeping other lines because Lines 9-13 are control-dependent on Lines 5-6.

5.5.3 Inter-component transplantation

The inter-component transplantation migrates malicious behaviors from one component to another component in the same app. Inter-component transplantation can be used to mutate the values of temporal features and locale features. For example, the mutation of the locale feature in Listing 5.1 is inter-component transplantation (Lines 6-9 in the Activity component in Listing 5.1 are transplanted from Lines 4-7 in the Service component in Figure 5.8).

Inter-component transplantation follows the same process as inter-method transplantation except for two differences. First, in addition to temporal features, inter-component transplantation is also used to mutate locale features. As previously mentioned, to mutate local features, the organ to be transplanted is the execution part of the code. To extract such organ, we find the call graph node directly linked by the entrypoint of the execution part. Note that the entrypoint of the execution part can be different from the entrypoint of the malicious behavior. For example, in Figure 5.9, the entrypoint of the execution part is `onStart`, while the entrypoint of the malicious behavior is `onClick`. After we locate the call graph node, the rest of the extraction process is the same.

The other difference of inter-component transplantation is when mutating the locale feature while maintaining the temporal feature, the regenerator needs to create inter-component communications to invoke the host method. To avoid crash caused by unmatching intent messages, the regnerator also adds conditional statements to avoid executing the existing code in the host method when such inter-component communications occur.

5.5.4 Inter-program transplantation

The inter-program transplantation is used to migrate the dependency feature of a malicious behavior in the donor app to the host app with identical malicious behavior. The extraction of the dependency feature is different from migration of the triggering/execution part of the malicious code. The organ consists of two parts. The *first* part is the implementation of the controlling behavior. We first construct the inter-component control flow graph of the app. Then we compute the subgraph containing all paths from the controlling statement (*i.e.,* the statement whose value determines the invocation of the malicious behavior) to the controlled statement (*i.e.,* malicious behavior). Such subgraph essentially represents the controlling behavior. The *second* part of the organ is the dependencies of the controlling statement. To extract these necessary dependencies, we slice backward from the controlling statement until we reach the entrypoint of the program. We then migrate both parts of the organ into the host app.

5.6 TESTING ON MUTATED APPS

We perform testing on mutated apps for two purposes: (a) whether the malicious behaviors have been preserved; (b) whether the robustness of the app has been mutated.

Checking the preserving of malicious behaviors. We develop two techniques to assist the testing. First, to simulate the environment where the malicious behaviors are invoked, we create environmental dependencies by changing emulator settings or using mock objects/events. By simulating the environment, we can directly invoke the malicious behaviors to speed up the validation process. Second, to further validate the consistency of malicious behaviors when the triggering conditions are satisfied, we apply the instrumentation technique to insert logging functions at the locations of malicious method invocations. The logging functions print out detailed information about the variables, functions, and events invoked after the triggering events. We therefore attain the log files before and after the mutation under the same context (*e.g.,* the same UI or system events and same inputs). Then, we automatically compare the two log files to check the consistency of malicious behaviors.

Checking the robustness of mutated apps. We leverage random testing to check the robustness of a mutated app. In particular, we use Monkey [106], a random user-event-stream generator for Android, to generate UI test sequences for mutated apps. Each mutated app was tested against 5,000 events randomly generated by Monkey to ensure that the app does not crash [7]

5.7 EXPERIMENT

Malware Detection Dataset. Our subject set consists of a malware dataset and a benign app dataset. Our malware dataset starts with 3,000 malware randomly selected from Genome [4], Contagio [54], VirusShare [53], and Drebin [15]. We use VirusTotal to perform sanity checking on the malware dataset (descriptions about signature-based detectors are provided later in this dissertation). We exclude the apps identified as benign by VirusTotal from the malware dataset. We also exclude any duplicate apps by comparing SHA1 hashes. For benign apps, we download the most popular 120 apps from each category of apps in the Google Play store as of February 2015 and collect 3,240 apps in total. We implement the process of extracting RTLD features using third-party static analysis frameworks, including

[7]Due to the limitation of the coverage of random testing, the mutated app passing the testing step can still be invalid. As future work, we plan to incorporate more intelligence-guided testing techniques [8, 107] in MRV testing.

Soot [101] and FlowDroid [100]. To isolate and remove the effects of potential limitations of these frameworks, we run feature-extraction analysis on the complete subject set and remove any apps that cause a third-party tool to fail. The filtering gives us a final analyzable dataset of 1,917 malware and 1,935 benign apps to perform malware detection. Our final malware dataset consists of 529 malware samples from Genome, 25 samples from Contagio, 287 samples from VirusShare, and 1,076 samples from Drebin dataset. Our final benign app dataset retains 63 to 96 apps from the original 120 apps in each Google Play category. All runs of our process of extracting RTLD features, the transplantation framework, and learning-based detection tools [28, 15] are performed on a server with Intel(R) Xeon(R) CPU 2.80GH with 38 processors and 80 GB of memory with a timeout of 80 minutes for each app.

Baseline Approaches. We implement two baseline approaches for comparison with MRV: `Random MRV` and `OCTOPUS`. We first develop a random transformation strategy (`Random MRV`) to compare against confusion and evolution attacks. Instead of following the evolution rules and similarity metrics to mutate the RTLD features, we randomly mutate RTLD features (*i.e.,* mutate the original feature value to the same-level feature value randomly selected from the available dataset) and transform the malware samples based on such mutation. Note that for `Random MRV` and evolution MRV, we follow the sequence of temporal feature, locale feature, and dependency feature to apply the transformation at different levels (Figure 5.5). We choose such sequence because the transplantation goes from inter-method to inter-app as the level increases in this sequence, likely leading to a higher success rate in the program transplantation. We leave the exploration on other possible mutation sequences to our future work.

We also implement a syntactic app obfuscation tool called `OCTOPUS` similar to Droid-Chameleon [26]. Specifically, `OCTOPUS` contains four levels of obfuscation: bytecode-sequence obfuscation (*i.e.,* repacking, reassembling), identifier obfuscation (*i.e.,* renaming), call-sequence obfuscation (*i.e.,* inserting junk code, call reordering, and call indirection), and encryption obfuscation (*i.e.,* string encryption). Then, we apply each level of obfuscation in `OCTOPUS` to each malware sample at a time, and perform testing on the sample file (Section 5.6) after each obfuscation. If the testing passes, we apply the next obfuscation to the obfuscated sample (resulted from applying the current obfuscation). If the testing fails, we apply the next obfuscation to the the sample before the current obfuscation (*i.e.,* skipping the current obfuscation). In our experiment, all semantic mutations including `Random MRV` and evolution/confusion attacks are performed after the syntactic obfuscation of `OCTOPUS`.

Malware detectors. We use a number of learning-based and signature-based malware

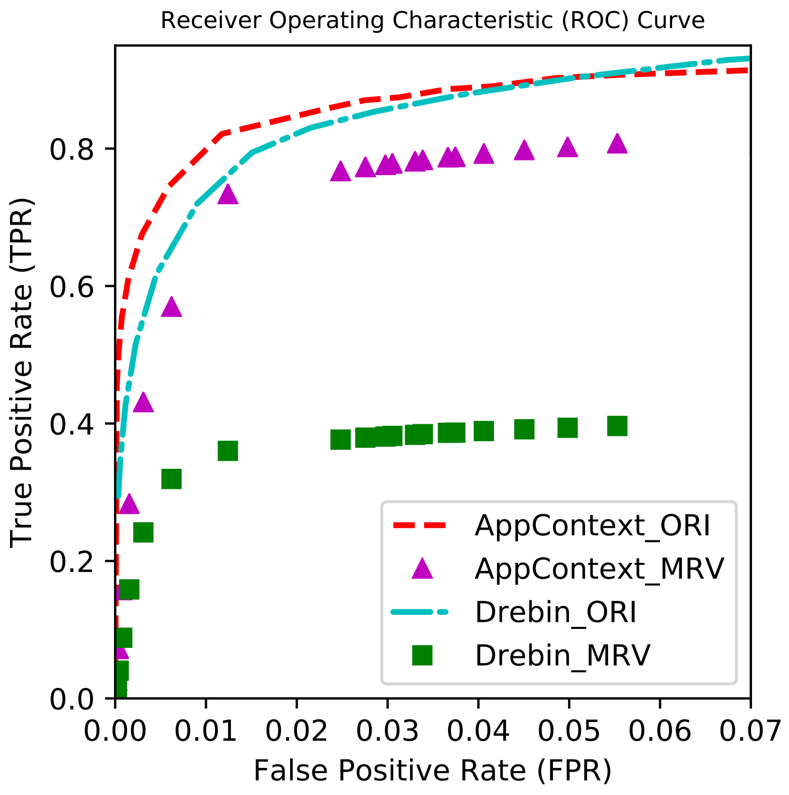


Figure 5.10: Detection results of AppContext vs. Drebin on the original dataset (ORI) and dataset with adversarial samples (MRV) produced by MRV

detectors to evaluate the effectiveness of MRV. For learning-based malware detectors, we adopt AppContext [28] and Drebin[15]. **AppContext** leverages contextual features (*e.g.,* the events and conditions that cause the security-sensitive behaviors to occur) to identify malicious behaviors. In our experiment, AppContext generates around 400,000 behavior rows on our dataset (3,852 apps), where each row is a 679-dimensional behavior vector. We conservatively label these behaviors (*i.e.,* marking a behavior as malicious only when the behavior is mentioned by existing malware diagnosis). The labeled behaviors are then used as training data to construct a classifier. **Drebin** uses eight features that reside either in the manifest file or in the disassembled code to capture the malware behaviors. Since Drebin is not open source, we develop our own version of Drebin according to its description [15]. Although Drebin extracts only eight features from an app, Drebin covers almost every possible combination of feature values resulting in a very large feature vector space. In fact, Drebin produces over 50,000 distinct feature values on our dataset (3852 apps). We perform ten-fold cross-validations to assess the effectiveness of AppContext and Drebin. Figure 5.10 shows the performance of AppContext and Drebin on all subjects in our dataset.

For signature-based malware detectors, we leverage the existing anti-virus service provided by VirusTotal. Specifically, we follow the evaluation conducted for Apposcopy [25] to pick the results of seven well-known anti-virus vendors (*i.e.,* AVG, Symantec, ESET, Dr. Web, Kaspersky, Trend Micro, and McAfee) and label an app as malicious if more than half of the seven suggest that the app is malicious. Following such procedure, only malware labeled as malicious are selected into our malware dataset, and thus all malware in our dataset can be detected by VirusTotal.

Learning algorithms. In our experiment, we leverage k-Nearest Neighbors (kNN), Decision Tree (DT), and Support Vector Machine (SVM) for malware detection in AppContext and Drebin. For confusion attack, we leverage Random Forest (RF) as the algorithm to train the substitute model[8]. The reason for us to use RF is that we want to use a different algorithm from the ones used in malware detection to validate our assumption in transferability [110].

Malware variants generation. We focus on generating malware variants by detected/-known malware samples. Among all 1,917 malware samples, 1,739 samples can be detected by all three detection tools that we used. Because many malicious servers of malware are blocked, causing malware to crash even *before* the mutations, we test the 1,739 malware with 5,000 events randomly generated by Monkey and discard the crashed apps. This step gives us a final set of **409** valid malware samples to generate malware variants. We then systematically apply `OCTOPUS`, evolution/confusion attacks, and `Random MRV` to all 409 valid malware samples.

5.8 RESULTS

5.8.1 Defeating existing malware detection

Table 5.1 shows the malware variants generated through transformation of `OCTOPUS`, `Random MRV`, malware evolution attack and confusion attack, and the detection results of VirusTotal on the variants. We also show the result of the full version of MRV (the combination of confusion and evolution attack) in the last column (**F**). Therefore, the full version includes all malware variants produced by confusion attack and evolution attack. The row "Transformable malware" refers to the number of malware samples that can be mutated to a

[8]We optimize the parameter for SVM and DT (we use C4.5 DT [108]) using `CVParameterSelection` of Weka [109]. For RF and kNN, we tune the parameters by testing on a sample set (100 malware and 100 benign apps). We set the benign/malware ratio in each subset (of an individual tree) for RF as 3 and K value for kNN as 7.

Table 5.1: Number of transformable malware samples and generated malware variants by different evasive techniques and the detection results

	O.	**R.**	**E.**	**C.**	**F.**
Transformable malware	409	121	314	58	341
Generated variants	1008	212	638	58	696
Variants undetected by VirusTotal	125	113	512	53	565
Variants Undetected by AppContext	0	2	97	56	153
Variants Undetected by Drebin	0	111	460	58	518

`O. = OCTOPUS, R. = Random MRV, E. = Malware Evolution Attack, C. = Malware Confusion Attack, F. = Full Version of MRV`

valid malware variant (*i.e.,* of all malware variants generated at different levels of an evasive technique, at least one of the malware variants can pass the testing). The row "Generated variants" shows the number of generated variants that pass the impact analysis and testing[9], and the last three rows[10] show the number of variants that can evade the detection of VirusTotal, AppContext, and Drebin, respectively.

As shown in Table 5.1, although the full MRV generates fewer malware variants than `OCTOPUS` (696 vs. 1,008), the full MRV produces much more evasive variants than both `OCTOPUS` and `Random MRV` for all three tools, especially the learning-based tools. This result indicates that the full MRV is much more effective in producing evasive malware variants than syntactic obfuscation and random transformation.

We investigate the malware variants produced by the full MRV that can still be detected by anti-virus software. We find that most variants of this kind contain extra payloads (*e.g.,* rootkit, another apk). The anti-virus software can detect them by identifying the extra payloads because our mutation transforms only the main program.

Although originally Drebin detects more malware samples than AppContext (Figure 5.10), Drebin performs worse on the full MRV dataset. Given different training malware samples, the full MRV can consistently make over 60% testing variants undetected by Drebin. One potential reason could be that AppContext leverages huge human efforts in labeling each security-sensitive behavior, while Drebin is a fully automatic approach, so overfitting is likely to occur in Drebin's model.

We also notice that `Random MRV` becomes much more effective in evading Drebin than evading AppContext (AppContext detects almost all variants produced by `Random MRV`). The reason lies in the large number of syntactic features used in Drebin. Such result indicates that although `Random MRV` is effective in befuddling the syntactic-based detection (*e.g.,* anti-

[9]The variants are generated at each level, and one malware sample may result in multiple malware variants.

[10]For AppContext and Drebin, we show the number of variants that cannot be detected by models based on all training algorithms.

Table 5.2: Details of Evolution Attack at each level (undetected vs. all)

Results	**T.**	**L.**	**D.**
Robust variants	178	316	144
Undetected by VirusTotal	77/178	296/316	139/144
Undetected by AppContext	21/178	15/316	61/144
Undetected by Drebin	73/178	272/316	115/144

T. = Temporal Features L. = Locale Features D. = Dependency Features

virus software), it is not effective in evading semantics-based detection techniques.

One noteworthy result is that confusion attack can successfully mutate only 58 malware samples into working malware variants. The reason is that confusion attack usually requires mutating more contextual features than evolution features. We observe in our experimental data that the likelihood of an attack to break the app increases as the number of mutations in the attack increases. Actually, confusion attack synthesizes more than 1,000 variants, and most of the variants are unable to run. However, such conversion rate is already high compared to `Random MRV`. `Random MRV` generates more than 320,000 variants, but only 212 of them can run without crashing (and only 2 can evade the detection of AppContext). Such result suggests that considering the feasibility of an attack is essential in generating adversarial malware samples.

5.8.2 Effectiveness of attacks at each level

For evolution attack, we also investigate the effectiveness of mutation at each RTLD level. Table 5.2 shows the detailed detection results of evolution attack at each mutation level.

Table 5.2 shows some interesting observations. For example, for anti-virus software and Drebin, the level that produces the largest number of evasive variants is on the locale-feature level, while for AppContext, the level that produces the largest number of evasive variants is on the dependency-feature level. This result indicates that mutating at the locale-feature level is more effective for the detectors using syntactic features (*e.g.,* VirusTotal, Drebin), while mutating at the dependency-feature level is more effective for semantics-based detectors (*e.g.,* AppContext). Such result also indicates that the transformation sequence used in the experiment (*i.e.,* temporal-locale-dependency) might not be the most optimal choice to evade some detectors. Ideally, we can explore different combinations of the mutation levels to maximize the number of undetected malware samples for each malware detector.

We also observe that most of unsuccessful variants produced at the dependency-feature level are due to the fact that a malicious behavior cannot be triggered in the simulated testing environment. The reason of lacking triggering is that by transplanting conditional statements from one component/method to another component/method, the internal logic of the original malware sample is broken. Some of the transplanted conditional statements may be mutually exclusive with the existing conditions in the code, thus making the malicious behavior infeasible to be triggered. As an ongoing effort, we plan to leverage a constraint solver to identify the potential UNSAT conditions when synthesizing mutation strategies.

5.8.3 Strengthening the robustness of detection

We also investigate the possibility of leveraging variants produced by MRV to strengthen the robustness of detection. We propose the following three techniques.

Adversarial Training. We randomly choose half of our generated malware variants into the training set to train the model, and put the other half of generated variants into the testing set to evaluate the model[11].

Variant Detector. We create a new classifier called variant detector to detect whether an app is a variant derived from existing malware. The variant detector leverages *mutation features* that are generated from each pair of apps' RTLD features to reflect the feature differences between the two apps. The number of mutation features is the same as the number of RTLD features. The difference is that for any RTLD feature that the two apps disagree on, the mutation feature (corresponding to the RTLD feature) is the (bidirectional) mutation between the apps on the RTLD feature. If the pair of apps are derived from same malware, we label the feature vector as "variant". Otherwise, we label the feature vector as "unrelated". Because only a small portion of all pairs of apps would have a "variant" relation, the trained model would be biased to the majority class (*i.e.,* the "unrelated" class). To resolve such issue, we use SMOTE [111] to make both classes to have an equal number of instances by creating synthetic instances. We then use the trained model on each app labeled (by malware detectors) as benign. For each of the apps, we create pairs to produce mutation features by grouping the app with each malware sample in our training set. Then the trained model determines whether the app is a variant of malware in the training set based on the mutation features.

Weight Bounding. We constrain the weight on a few dominant features to make feature weights more evenly distributed. For example, in the case of SVM, we constrain w in the cost function of SVM:

[11]We perform ten-fold cross-validation in our experiment to report TP and FP.

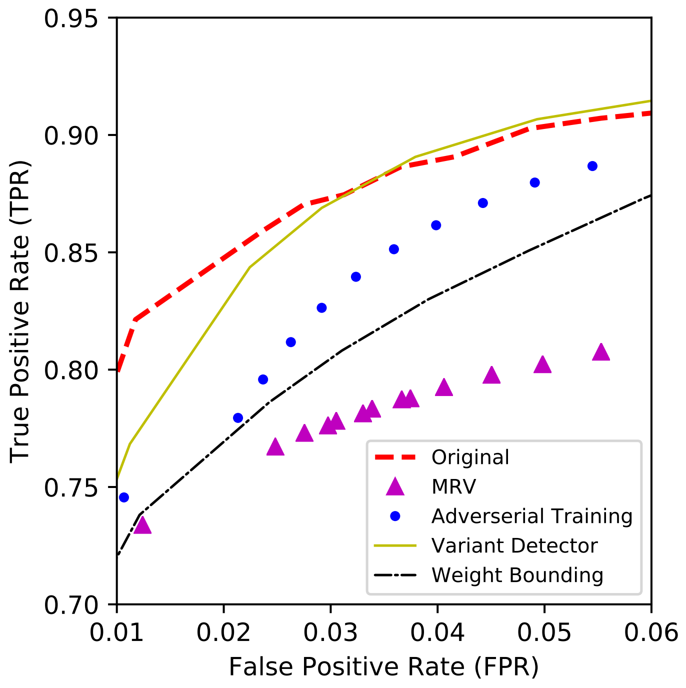


Figure 5.11: Detection results of AppContext (SVM) when different defense mechanisms are applied

$$\min_{w \in \mathcal{R}^d} \parallel w \parallel^2 + C \sum_{i}^{N} \max(0, 1 - y_i f(x_i))$$

We observe that adversaries can produce evasive malware variants by applying just a few mutations on dominant features in contrast to many more mutations on other non-dominant features. Therefore, to locate dominant features, we select all 44 malware variants produced by fewer than three mutations, and summarize 17 dominant features that enable the production of the variants. To compute the specific range of the weight, we put only 44 malware variants and their original malware samples as malicious samples in the training set, and record the range value of the weight of the 17 dominant features under different parameters. We then summarize the constraints in reasonable settings (TPR $\geq$ 0.80 and FPR $\leq$ 0.10) and put the hard constraints in the training phase.

Figure 5.11 presents the detection results of AppContext's malware detection[12]. The red line represents the detection performance on the original dataset, and the purple triangles represent the detection performance on the dataset with malware variants produced by MRV. The other three curves represent the detection performance of three proposed protection

[12]We present only SVM-based model here due to the limited space. Other learning algorithms present similar patterns as SVM. We leave investigation of specific differences across models as future work.

Table 5.3: Number of malware samples evading detection of AppContext or Drebin under different algorithms

Detector	ORI.	AT.	VD.	WB.
AppContext	178	125	106	152
Drebin	38	19	8	23

ORI. = Original detection, AT. = Adversarial training
VD. = Variant detection, WB. = Weight bounding

techniques on the dataset with malware variants. As shown in Figure 5.11, all three proposed techniques can alleviate the MRV attacks. The variant detector technique can reach almost the same performance as the original malware detector (while being more secure/robust to malware variants).

To alleviate the concerns that our proposed defenses are overfitting to MRV attacks, we also investigate whether the trained models can assist detecting not only malware variants but also unknown malware samples in general. We choose to investigate the malware samples evading the detection of the original AppContext and Drebin (178 and 38 malware samples evade the detection, respectively)[13]. As shown in Table 5.3, all the protection mechanisms can help detect evasive malware samples, and only eight of the samples can evade the detection of the variant detector technique.

5.9 CONCLUSION

In this chapter, we have proposed practical attacks that mutate malware variants to evade detection. The core idea is to leverage existing malware program structures to change the features that are non-essential to malware but important to malware detectors. To achieve this goal, we have presented the MRV approach including static analysis, phylogenetic analysis, machine learning, and program transplantation to systematically produce new malware mutations. To the best of our knowledge, our work is the first effort toward solving the malware-evasion problem by altering malware bytecode without any knowledge of the underlying detection models.

MRV opens up intriguing, valuable venues of applications. First, the proposed attacks can be used to evaluate the robustness of malware detectors and quantify the differentiability of features. Second, MRV can help discover potential attack surfaces to assist the iterative design of malware detectors. Finally, the program transplantation framework (capable of changing malware features) can be written as a malicious payload within malware and such

[13]All the numbers are counted when the false positive is within 0.06.

adaptive malware are valuable for the community to investigate.

CHAPTER 6: RELATED WORK

Static Analysis for Mobile App. Much work has been proposed to enhance static analysis on mobile apps [75, 76, 77, 22, 78, 20, 79, 80, 81, 82, 83, 19, 84, 85]. EnMobile falls into the general category of information flow analysis. Information flow analysis tracks whether privacy-sensitive data (*i.e.,* sources) flows to outgoing channels or sensitive outlets (*i.e.,* sinks). EnMobile complements existing information flow analysis by adding entity-based characterization to the information flow. AAPL [77] uses enhanced data flow analysis techniques to increase the number of data flows that can be detected by information flow analysis and then uses the peer-voting mechanism to low the false positive rate to report illegitimate information leakages. AAPL fails to handle obfuscation techniques such as String encryption (by using constant propagation analysis) and produces high false positives (by matching all sources with all potential sinks). EnMobile resolves these two limitations by precisely computing the identity of entities. SPARTA [86] and FlowDroid [14] are two general information flow analysis frameworks. SPARTA enables the flow-policy checking by providing an integrity type system to annotate source code with information-flow type qualifiers. FlowDroid is a static taint analysis tool for Android apps based on Soot [87] and Heros [88]. EnMobile complements SPARTA and FlowDroid by analyzing all types of data flows to detect malicious behaviors that are not information leakage (*e.g.,* bot-driven C&C behaviors). HARVESTER [95] is a hybrid analysis that extracts runtime value to cope with anti-analysis technique (*e.g.,* encryption) in Android apps. EnMobile can leverage HARVESTER to derive more accurate inference of entity identity by coping with the problem of reflective method calls.

Characterizing App Behaviors. Existing work uses static analyses to characterize app behaviors. DroidSift [67] characterizes malicious behaviors via program dependency graphs. However, malware can obfuscate the program by leveraging the external entities to break a program dependency graph into pieces. EnMobile can assists DroidSift to reconstruct the dependency graph from the segmented graphs. Pegasus [112] characterizes the effects of the event system and API semantics using permission event graphs. EnMobile complements Pegasus by reflecting the configuring or controlling entities of security-sensitive behaviors.

Mobile Malware Detection. There are mainly two types of techniques: signature matching and machine learning. For signature matching, a malware detector matches either a syntactic signature (*i.e.,* a sequence of instructions matched by a regular expression) or a semantic signature [25] (*i.e.,* control flows or data flows). In our evaluations, we demonstrate that EnMobile can be used to match entity-based malware signatures (which are semantic signatures). For machine learning, existing work leverages techniques such as mining (MUDFLOW [65]) clustering (CHABADA [66]), classification (AppContext [28]), and natural language processing (AsDroid [73], WHYPER [23]) to detect malicious apps. Our approach complements existing malware-detection analysis by identifying contexts that indicates the intentions of data uses. There are various approaches that perform analysis to detect malicious behaviors, such as dynamic taint analysis [7, 113], language-based information flow [114, 115, 116, 117], static analysis [118, 67, 25, 119], and Bayesian classification [120]. However, these approaches are concerned about *how* privacy-sensitive data protected by permissions are used, while our approach provides the *contexts* under which the permissions are triggered. Future work can also leverage entity-based semantic information extracted by EnMobile to train classification models to differentiate benign apps and malware.

Contexts of Permission Uses. Besides our work, prior research has leveraged the contextual information of app behaviors for various security purposes. Pegasus [112] constructs permission event graphs using static analysis to model the effects of the event system and API semantics, and performs model checking to enforce the policies specified by users. However, specifying these policies requires that users have established knowledge about the expected behavior/functionality of the app and an understanding of the Android platform. Our approach complements Pegasus by providing the contexts, which can be used to construct Pegasus' policies. AppIntent [9] presents a sequence of GUI events that lead to data transmissions and let analysts decide whether the data transmissions are intended. Their approaches focuses on GUI events, a kind of user-perceivable contexts that allows analysts to scrutinize the manipulations of UI components. Although our approach also focuses on the events that trigger app behaviors, AppIntent handles only app behaviors activated by GUI events while our approach analyzes a more comprehensive set of contexts (e.g., receivers and background services) and can complement their approach to handle data transmissions that are not triggered by sequences of GUI manipulations. Further, our approach focuses on permission uses rather than data transmissions. Moreover, in addition to the event-handlers in the code, our approach also extracts the events from Android components' attributes in the manifest files, identifying more types of contexts than their approach. AsDroid [73] detects stealthy app behaviors by identifying mismatches between API invocations and the text displayed in the GUIs. Our approach focuses on the events that trigger app behaviors rather than the textual analysis of the GUIs. Since app behaviors can occur without displaying a GUI, the textual analysis of GUIs alone is insufficient to detect all stealthy app behaviors. DroidAPIMiner [30] identifies malicious apps by performing frequency analysis of API invocations within a set of benign and malicious apps to extract the features of malware, and uses machine learning to determine the most relevant features. Our approach focuses on what causes security-sensitive API calls to be used rather than the pattern of API calls that are used. WHYPER [23] examines whether app descriptions provide any justification for the app's permission uses. WHYPER focuses on *why* apps request permissions while our approach focuses on *how* apps actually use the requested permissions. Such sentences in the app description provide explanations for the contexts of permission uses. But WHYPER cannot provide contexts for permission uses that have not been justified by app descriptions. Our approach addresses this issue by transforming the permission related behaviors and contexts into natural-language descriptions. Such descriptions can be used to explain the unjustified permission usages identified by WHYPER, complementing WHYPER to improve user-understanding of permission usages.

Studies of Permission Model. Felt et al. [121] perform usability studies to examine whether users understand the permission warnings. Their results show that user can identify (part of) the permission definition, but they are confused about the scope of permission. Users incorrectly believe that a given permission has more capabilities or less capabilities than it actually has. The context description provided by AppContext could alleviate the issue to some extent. The description explains the privileges gained by permission invocations, so users could infer what apps could do by using the permissions. Barrera et al. [122] leverage the Self-Organizing Map (SOM) algorithm to study how the developers use the Android permission system in practice. They take 1100 Android apps on the market as their study subjects, and visualize the study results in U-matrix representation of the SOM for Android permissions. Their results reveal the correlations between permissions, and the results also suggest that pairs of permissions are common. Stowaway [123] build a permission mapping to check whether Android apps follow least privilege with their permission requests. PScout [46] also build a more complete permission mapping by static analysis of Android system source code. These studies focus on correlations between permissions and permission mapping while our studies focus on contextual use of permissions.

Risk Ranking and Certification of Apps. Peng et al. [27] present the risk information of an app compared to other apps by using probabilistic generative models to calculate risk scoring of the app. MAST [124] triages Android apps by analyzing features extracted from the APKs. MAST uses machine learning techniques to measure the correlation between features and directs malware analysis resources to the apps that have the greater potential of risks. Kirin [125] performs lightweight certification of apps by identifying dangerous app configurations against a set of security rules. These approaches leverage various kinds of features or configurations in apps to identify potential risks. Unlike these approaches that present the risk scores or ranking for users, our approach analyzes the bytecode of apps to extract the contexts of permission uses. However, our approach can complement risk ranking and certification techniques by providing the extracted permission contexts as another metric for their evaluation.

Privacy Issues in Mobile Apps. Several efforts try to characterize the current mobile ad targeting process. MAdScope [126] and Ullah et al. [127] both found that ad libraries have not yet exploited the full potential of targeting. Our work is driven by such observations and tries to assess the data exposure risk associated with embedding a library in an app.

Many studies describe alternative mobile advertising architectures. AdDroid [128] enforces privilege separation by hard-coding advertising functions as a system service into Android platform. AdSplit [129] achieves privilege separation via making ad libraries and their host apps run in separate processes. Leontiadis et al. [130] proposes a client-side library compiled with the host app to monitor the real-time communication between the host app and the ad libraries to control the exposed information. MobiAd [131] suggests local profiling instead of keeping the user profiles at the data brokers to protect users' privacy. Most of these alternative architectures envision a separation of ad libraries from their host apps. This would eliminate the in-app attack channels that we demonstrate and constrain the data exposure to the ad libraries. However, none of these solutions are deployed in practice as they all disrupt the business model of multiple players in this ecosystem. We take a different approach by modeling the capabilities of ad libraries in order to proactively assess apps' data exposure risk.

There are a number of studies that aim to—or can be used to—detect and/or prevent current privacy-infringing behaviors in mobile ads. Those works mainly fall into three general categories: (1) static scanning [132, 133, 134, 135, 136], (2) dynamic monitoring [137, 138, 139, 140], and (3) hybrid techniques using both [141]. A combination of these techniques could detect and prevent some of the attack strategies of ad libraries we discussed in this work, if they are adopted in practice. However, such countermeasures can still fail to protect against all allowed behaviors. For example, TaintDroid [7] and FlowDroid [14] cannot evaluate the sensitivity of the data carried. Moreover, static code analysis will miss dynamically loaded code, and code analysis in general cannot estimate the potential reach of libraries. Further, by merely encrypting local files we cannot prevent libraries within the same process from using the key the host app uses to decrypt the files. In addition, there is no mechanism to address data exposure through app bundle information as we reveal in this work because (1) this is not considered as a sensitive API from AOSP and (2) even if marked as sensitive it is unclear how access to it by apps and/or libraries should be mediated, as there are legitimate uses of it. Our focus is not on detecting and tackling current behaviors but assessing the data exposure given the allowed behaviors. This is critical when trying to assess the privacy risk of an asset.

SUPOR [142] and UIPicker [143] seek instances where apps exfiltrate sensitive data. Like Pluto, they use NLP and machine learning techniques to find data of interest in user interfaces. Unlike Pluto, their focus is on data like account credentials and financial records, whereas Pluto is aimed at general targeted data with validation based on data of interest to advertisers. As with most of the other work in this area, SUPOR and UIPicker seek existing exfiltration instances rather than allowed instances, although some of their techniques can facilitate finding allowed instances.

Evasive Malware. Metamorphic malware [144], polymorphic malware [145] and other obfuscation techniques [146] have been developed to evade malware detection [147]. Semantic signature [148], behavior graphs [149] and other semantics aware techniques [150], [28], [151] have been developed to defeat against malware. To study how anti-malware products are resistant against transformed mobile apps, Droidchameleon [26] is developed as a systematic framework with various transformation techniques for mobile app study and they found the transformed apps can easily evade detection. Unlike MRV, DroidChameleon only performs syntactic obfuscation, which can be easily be detected by semantic-based detection tool such as AppContext. PraGUARD [152] performs assets encryption on malware samples to assess the role of external resources (i.e., assets) in the detection for the anti-malware tools PraGUARD focuses on assessing detection of external resources, while MRV focuses on assessing detection of apps behaviors. Two approaches focus on different problems, complementary to each other. Replacement attack [153] is proposed to poison behavior-based specifications by concealing similar behaviors of malware variants. Replacement attack can impede malware clustering [154]. MalGene [155] is developed to automatically locate evasive behavior in system call sequences and therefore extract evasion signatures. Different from these works, MRV can evade both anti-virus tools and machine learning based classifiers.

Recent evaluations [16] on ML-based malware detection techniques suggest that more feature number does not necessarily improve the performances due to the non-informative features [15] and noisy features. Recent study on PDFrate [96] evaluates existing learning-based malware detection techniques in different evasion scenarios. Xu et al. [97] propose an evading technique based on classification score feedback that can manipulate PDF malware samples to evade detection of PDFrate and Hidost. Carmony et al. [98] manipulate JavaScript payload in PDF malware to evade detections. The major difference between these works and MRV is that MRV requires much less knowledge about machine learning model to launch the attack. Moreover, prior work mainly focuses on pdf malware while MRV automatically generates malware variants for android apps.

Adversarial Machine Learning. Adversarial machine learning [156, 157, 158, 96] studies the effectiveness of machine learning techniques against an adversarial opponent.

Adversaries frequently attempt to break many of the assumptions that practitioners make (*e.g.,* data has various weak stochastic properties; data do not follow a stationary data distribution). Generally, the adversary is assumed to have full (or partial) information related to three components of learner: learning algorithm, feature space and training dataset. In our work, we propose a targeted attack assuming a malware developer knows the information about the feature space and classifier.

Program Transplantation. We leverage program transplantation technique for program transformation. Given an entry point in the donor and a target implantation point in the host program, the goal of automated program transplantation [105, 100, 159] is to identify and extract an organ, all code associated with the feature of interest, then transform it to be compatible with the name space and context of its target site in the host. In this work, we broaden program transplantation concept to intra-app transplantation that needs to deal with the side-effect of the removal of the organ (compared to inter-app transplantation), where effective heuristics are proposed to automatically identify the target implantation point in the host program.

CHAPTER 7: CONCLUSIONS AND FUTURE WORK

The increasing popularity of intelligent techniques such as machine learning intrigue security researchers and practitioners to adapt the intelligent techniques in security systems. Investigating and strengthening the adversarial resiliency of the intelligent techniques are crucial for such adaptation because these techniques were first proposed or developed without considering the presence of adversaries. The contributions of this dissertation address this issue from three perspectives. We first present how intelligent techniques can be adapted for automated decision making in mobile security systems. Then we investigate the possibility to design and implement systematic attack strategies that are specifically adversarial to these newly-proposed intelligent techniques. Last, based on the findings that the intelligent techniques are indeed susceptible to the adversarial attacks, we develop techniques to further strengthen the adversarial resiliency of intelligent techniques toward these adversarial attacks.

One important direction for future research is to develop software engineering infrastructures including testing infrastructures to assure the quality of learning-based security systems. The existing way to assure quality of a learning-based system is either by manual testing (*i.e.,* applying the learning-based system to several examples drawn from a test set and measuring its accuracy) or by verification in a form of statistical guarantees (*e.g.,* using statistical learning theory to suggest that the test error rate is unlikely to exceed some threshold). For manual testing, the testing procedure cannot cover most (if not all) of the possible – previously unseen – examples that may be misclassified. For verification, the statistical guarantees are expressed for points that are drawn from the same distribution as the training data (*i.e.,* the guarantee will hold when considering only "naturally occurring" inputs). While verification is challenging even from a theoretical point of view, even automatic testing can be challenging from a practical point of view.

The traditional techniques of automatic testing cannot be applied on the learning-based systems for three main reasons.

(1) **Learning instead of implementing program logics.** In traditional software, test generation is driven by the goal of covering program structures (such as all program branches/paths of a given program under test) because the logic of a traditional program is expressed in terms of control flow statements. However, core logics of learning-based systems are embedded in the machine learning models (*i.e.,* arithmetic operations of formulas) in the program. Even a single input can easily cover all program paths in a learning-based system [160] because the learning-based system automatically learns its logic from a large

amount of data with minimal human guidance (there is no need to incorporate any control-flow statements in the program itself). Due to such unique characteristic, the erroneous corner-case behaviors in learning-based systems are analogous to logic bugs in traditional software.

(2) **Non-linearity of machine learning models.** Machine learning models are fundamentally different from the models (*e.g.,* finite state machines) used for modeling and testing traditional programs. Modern learning algorithms (*e.g.,* perceptron learning, neural network) tend to learn models that are highly non-linear functions. Finding inputs that can result in high model coverage in these models is significantly more challenging due to the non-linearity.

(3) **Incapability of supporting techniques.** The Satisfiability Modulo Theory (SMT) solvers [161] that have been quite successful at generating high-coverage test inputs for traditional software are known to have trouble with formulas involving floating-point arithmetic and highly nonlinear constraints, which are commonly used in machine learning models.

Next we present our plans for possible future work that can be built upon our current contributions and results.

Testing Metrics for Learning-based Mobile Security Systems. Testing metrics are used to represent the effectiveness of test inputs. Higher values of testing metrics usually indicate better saturation of program logics in the system under test. As the values of testing metrics increase, more issues lying in the system are likely to be uncovered. In traditional software testing, various types of code coverage metrics, including line coverage, method coverage, and branch coverage, usually serve as the standard testing metrics for evaluating different testing strategies. Testing metrics of traditional software are based on program structures (*e.g.,* statements, branches, and paths) because traditional software incorporates program logics in these program structures (*e.g.,* control flows, data flows). However, program logics of learning-based systems are embedded in the arithmetic operations of formulas in the program. These facts make traditional testing metrics inappropriate to characterize the executions of such mathematical formulas. For instance, sigmoid function, as a commonly-used activation function in neural networks, can be implemented with only one line of Java code. When the function is used in neural networks, we need to observe the return value (*e.g.,* check whether the value is greater than a threshold) to tell whether the respective neuron is activated or not. In other words, if we achieve full line coverage of the method implementing a sigmoid function, we still could not guarantee that the status space of the respective neuron has been fully covered. Thus, there is a strong need of testing metrics customized for learning-based systems.

Recently, researchers developed a new systematic strategy [162] to test deep learning systems. They propose neuron coverage (*i.e.,* ratio of the number of distinct activated neurons to the total number of neurons in the neural network) as the testing metric. As their approach tries to improve neuron coverage during testing of real deep learning models, they find thousands of incorrect corner-case behaviors. The promising results show the effectiveness of their approach as well as the future of such efforts. However, considering only neuron coverage might not be sufficient for future improvements of the models under test. There is one key problem in this testing metric: it interprets neuron activation as a positive indicator for testing effectiveness, while actually it is only a status of the respective neuron. Such situation is similar to counting the number of true clauses in traditional software testing. A higher number of true clauses revealed during testing does not necessarily imply covering more situations when executing the code. In fact, it is possible for faults to be revealed even when most clauses are false. Such facts suggest that we should consider more effective testing metrics for neural networks.

For our future work, we plan to use manifold to address the testing metric problem. A manifold is a topological space, in which each point is surrounded by a locally Euclidean space. Previous research [163] speculated that lower-dimension manifolds are good models for many data-related tasks, whose data points might lie in very high-dimensional spaces. Such speculation indicates that we might be able to tell whether specific inputs are meaningful or not by checking whether they could fit in the manifold constructed from the training data.

There are multiple ways to construct manifolds from training data and check whether specific inputs could fit in the manifolds. We plan to choose the most suitable combination of approaches for each neural network model.

Manifold-reconstruction-based checking. The basic idea is to construct a new manifold $\mathcal{M}'$ with both training data and given input, and measure the distance between the new manifold and the manifold $\mathcal{M}$ constructed from only training data, which is denoted as $d(\mathcal{M}, \mathcal{M}')$. If inputs could fit in $\mathcal{M}$, then the differences between $\mathcal{M}$ and $\mathcal{M}'$ should be insignificant, resulting in a small $d(\mathcal{M}, \mathcal{M}')$. If inputs are unlikely to be meaningful with regard to the training data, it is expected to observe a somewhat different $\mathcal{M}'$, which corresponds to a big $d(\mathcal{M}, \mathcal{M}')$. This approach should be applicable to all manifold learning algorithms.

Manifold-construction-invariant-based checking. Reconstructing the manifold and deriving the manifold-to-manifold distance might lead to efficiency problems. During manifold construction, determined by the construction algorithm, some properties among the training data might be fully or partially preserved (*i.e.,* invariants), which could be utilized for faster checking of new inputs. For instance, Isomap [164] preserves the geodesic distance between each pair of points (*i.e.,* the sum of edge lengths along the shortest path between two points). This property entails that, if an input under test is together used with the training data to construct a manifold, the shortest Euclidean distance between the point corresponding to the test input and the points representing the training data in the lower-dimensional space will be the same as that in the higher-dimensional input space. Thus, if the distances of test inputs to the manifold are used to judge whether those inputs fit in the manifold, we simply need to find the shortest Euclidean distance between those inputs and training data in the input space instead of constructing manifolds and measuring the distances among manifolds.

Autoencoder-based manifold construction and checking. Autoencoders are neural networks with simpler hidden representations trained to forward inputs to outputs. Autoencoders can be leveraged to identify intrinsic properties of the data [165] and realize both manifold learning and checking. An autoencoder $ae = d \circ e$ can be viewed as two parts: an encoder $e : \mathbb{S} \rightarrow \mathbb{H}$, which is trained to map from inputs to hidden representations, and resembles to constructing the manifold from training data, and a decoder $d : \mathbb{H} \rightarrow \mathbb{S}$, which is trained to recover inputs from hidden representations, and resembles to the reverse process of manifold construction. The input space is denoted as $\mathbb{S}$, the hidden representation space is denoted as $\mathbb{H}$, and $\mathbb{H}$ has fewer dimensions than $\mathbb{S}$. Assume that $\mathbb{H}$ is large enough to embrace most hidden representations of normal inputs, then the reconstruction error for training inputs by ae, which can be defined as the average Euclidean distance from the outputs of ae to the training inputs, should be reasonably low. This process resembles to normal inputs lying on the manifold. For unintended inputs, since their hidden representations are likely to clash with those of normal inputs due to limited space dimensions of $\mathbb{H}$, and e tries to recover from hidden representations based on training inputs, the outputs of e (also as the outputs of ae) should be different from those inputs, resulting in high reconstruction errors. Such process resembles to unintended inputs being away from the manifold. Such properties enable unintended-input detection according to the reconstruction errors, essentially approximating the distances between inputs and the manifold. Previous research [166] applied this technique to defend against adversarial examples for neural network classifiers.

Test Oracle for Learning-based Mobile Security Systems. To test machine learning software, software testing techniques have been used to address two major issues: test generation (*i.e.,* generating test inputs) and test oracle (*i.e.,* determining whether the software behaviors are expected with respect to the given test inputs). Unexpected behaviors of machine learning software can come from various causes such as not having good enough training data sets, or the machine learning software itself is faulty. Here we are focusing on the latter cause: test oracles to determine whether the machine learning software is faulty.

Machine learning software is known to suffer from the "no oracle problem" [167]. Usually machine learning, especially supervised learning, learns a prediction model from training data sets and then uses the prediction model to predict the label to classify a future data instance.

A test oracle (what the prediction/label should be) is not easily obtainable. Labeling a future data instance can also be done manually, yet being ineffective. Our previous work on differential testing [168] focused on a similar problem. Differential testing is a testing approach to generate test inputs that exhibit the behavioral differences between two versions of a program. A specific implementation is chosen as a reference implementation, which can be seen as a test oracle. By using another implementation of the same functionality as the program that we want to test as a cross-reference test oracle, we can just use the output from this another implementation as our expected output for a certain test input. Since machine learning software is widely used nowadays, there are many machine learning frameworks that have the same or similar functionality to each other. Thus, it can be very useful to leverage multiple machine learning frameworks to help address the "no oracle problem" in testing machine learning software.

We plan to tackle the test oracle problem for machine learning software by using an approach of multiple-implementation testing for supervised learning software. The overall idea is that we can collect multiple implementations of machine learning software that have the same or similar functionality to the software that we want to test. It is likely that the majority of these implementations produce the expected output for a given test input. Thus, we can use the majority output from these implementations as our test oracle.

Going forward, the approaches that we have developed in this dissertation can be extended to strengthen the adversarial resiliency for a much wider range of security systems. We hope that by strengthening the adversarial resiliency of intelligent techniques, these approaches will become cost-effective enough to be used in a much wider range of practical security systems.

REFERENCES

[1] G. M. Blog, "Android and security." [Online]. Available: http://googlemobile.blogspot.com/2012/02/android-and-security.html

[2] "App review — Apple App Store," https://developer.apple.com/app-store/review/.

[3] 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.* ACM, 2012, pp. 281–294.

[4] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. S&P*, 2012, pp. 95–109.

[5] 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*, vol. 25, no. 4, 2012, pp. 50–52.

[6] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones." in *NDSS*, vol. 14, 2012, p. 19.

[7] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. OSDI*, 2010, pp. 1–6.

[8] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *Proc. FASE*, 2013, pp. 250–265.

[9] 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 *Proc. CCS*, 2013, pp. 1043–1054.

[10] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in Android apps with permission use analysis," in *Proc. CCS*, 2013, pp. 611–622.

[11] M. Hypponen, "Malware goes mobile," *Scientific American*, no. 5, pp. 70–77, 2006.

[12] 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.* ACM, 2011, pp. 239–252.

[13] M. J. Harrold, G. Rothermel, and S. Sinha, "Computation of interprocedural control dependence," in *ACM SIGSOFT Software Engineering Notes*, no. 2, 1998, pp. 11–20.

[14] 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," in *Proc. PLDI*, 2014, p. 29.

[15] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: effective and explainable detection of android malware in your pocket," in *NDSS*, 2014, pp. 23–26.

[16] 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 *ACSAC*, 2015, pp. 81–90.

[17] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *international conference on Information security.* Springer, 2010, pp. 346–360.

[18] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses." in *USENIX Security Symposium*, vol. 30, 2011, p. 88.

[19] 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.* ACM, 2012, pp. 229–240.

[20] F. Wei, S. Roy, X. Ou et al., "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proc. CCS*, 2014, pp. 1329–1341.

[21] D. Octeau, S. Jha, M. Dering, P. McDaniel, A. Bartel, L. Li, J. Klein, and Y. Le Traon, "Combining static analysis with probabilistic models to enable market-scale android inter-component analysis," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, 2016, pp. 469–484.

[22] L. Li, A. Bartel, T. F. D. A. Bissyande, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "IccTa: detecting inter-component privacy leaks in android apps," in *Proc. ICSE*, 2015, pp. 280–291.

[23] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: Towards automating risk assessment of mobile applications," in *Proc. USENIX Security*, 2013, pp. 527–542.

[24] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2014, pp. 1354–1365.

[25] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of Android malware through static analysis," in *Proc. FSE*, 2014, pp. 576–587.

[26] 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.* ACM, 2013, pp. 329–334.

[27] 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*. ACM, 2012, pp. 241–252.

[28] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: Differentiating malicious and benign mobile app behaviors using context," in *ICSE*, 2015, pp. 303–313.

[29] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among android applications," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2012, pp. 62–81.

[30] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-level features for robust malware detection in Android," in *Proc. SecureComm*, 2013.

[31] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen, "I-arm-droid: A rewriting framework for in-app reference monitors for android applications," *Mobile Security Technologies*, vol. 2012, no. 2, p. 17, 2012.

[32] R. Xu, H. Saïdi, and R. J. Anderson, "Aurasium: Practical policy enforcement for android applications." in *USENIX Security Symposium*, vol. 2012, 2012.

[33] B. Livshits and J. Jung, "Automatic mediation of privacy-sensitive resource access in smartphone applications." in *USENIX Security Symposium*, 2013, pp. 113–130.

[34] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using pql: a program query language," *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 365–383, 2005.

[35] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks." in *USENIX Security Symposium*, 2006, pp. 121–136.

[36] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 43–52.

[37] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Generating parameter comments and integrating with method summaries," in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*. IEEE, 2011, pp. 71–80.

[38] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 101–110.

[39] R. P. Buse and W. R. Weimer, "Automatically documenting program changes," in *Proceedings of the IEEE/ACM international conference on Automated software engineering.* ACM, 2010, pp. 33–42.

[40] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on.* IEEE, 2013, pp. 23–32.

[41] B. B. Rad, M. Masrom, and S. Ibrahim, "Camouflage in malware: From encryption to metamorphism," *IJCSNS*, 2012.

[42] "Android sensors overview," http://developer.android.com/guide/topics/sensors/ sensors_overview.html.

[43] T. Bradley, "DroidDream becomes Android market nightmare." [Online]. Available: https://tinyurl.com/yd222fna

[44] D. Franke, C. Elsemann, S. Kowalewski, and C. Weise, "Reverse engineering of mobile application lifecycles," in *Proc. WCRE*, 2011, pp. 283–292.

[45] J. A. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural processing letters*, no. 3, pp. 293–300, 1999.

[46] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android permission specification," in *Proc. CCS*, 2012, pp. 217–228.

[47] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," in *2014 Network and Distributed System Security Symposium (NDSS)*, 2014.

[48] S. Chiba, "Load-time structural reflection in Java," in *Proc. ECOOP*, 2000, pp. 313–336.

[49] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! analyzing unsafe and malicious dynamic code loading in Android applications," in *Proc. NDSS*, 2014.

[50] B. Livshits, J. Whaley, and M. S. Lam, "Reflection analysis for Java," in *Proc. APLAS*, 2005, pp. 139–160.

[51] "AppContext," https://sites.google.com/site/asergrp/projects/appcontext/.

[52] O. Lhoták and L. Hendren, "Scaling Java points-to analysis using Spark," in *Proc. CC*, 2003, pp. 153–169.

[53] "VirusShare," https://www.virusshare.com.

[54] "Contagio mobile - mobile marewale mini dump." [Online]. Available: http://contagiominidump.blogspot.com/

[55] "Virustotal - free online virus, malware and url scanner," https://www.virustotal.com/.

[56] F-Droid, "FOSS apps for Android." [Online]. Available: https://f-droid.org/

[57] "Androguard," https://code.google.com/p/androguard/wiki/ databaseandroidmalwares.

[58] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *IJCAI*, no. 2, 1995.

[59] "Security Threat - Symantec," https://www.symantec.com/security_response/.

[60] "Virus Information - McAfee," http://home.mcafee.com/virusinfo/.

[61] "Threat Analysis - Sophos," https://www.sophos.com/en-us/threat-center/threat-analyses.aspx.

[62] "Microsoft malware protection center," http://www.microsoft.com/security/portal /threat/Threats.aspx.

[63] "Threat Encyclopedia - Trend Micro," http://www.trendmicro.com/vinfo/us/threat-encyclopedia/.

[64] "Antiy Labs," http://www.antiy.net/.

[65] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *Proc. ICSE*, 2015, pp. 426–436.

[66] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proc. ICSE.* ACM, 2014.

[67] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware Android malware classification using weighted contextual API dependency graphs," in *Proc. CCS*, 2014, pp. 1105–1116.

[68] Y. Feng, O. Bastani, R. Martins, I. Dillig, and S. Anand, "Automated synthesis of semantic malware signatures using maximum satisfiability," in *NDSS*, 2017.

[69] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, *Deep Ground Truth Analysis of Current Android Malware*, 2017.

[70] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications," in *Security and Privacy (SP), 2016 IEEE Symposium on.* IEEE, 2016, pp. 377–396.

[71] K. O. Elish, X. Shu, D. D. Yao, B. G. Ryder, and X. Jiang, "Profiling user-trigger dependence for android malware detection," *Computers & Security*, vol. 49, pp. 255–273, 2015.

[72] J. Rubin, M. I. Gordon, N. Nguyen, and M. Rinard, "Covert communication in mobile applications," in *Automated Software Engineering (ASE)*, 2015, pp. 647–657.

[73] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "AsDroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction," in *Proc. ICSE*, 2014, pp. 1036–1046.

[74] 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." in *USENIX Security Symposium*, 2015, pp. 659–674.

[75] O. Bastani, S. Anand, and A. Aiken, "Interactively verifying absence of explicit information flows in android apps," in *Proc. SPLASH*, 2015, pp. 299–315.

[76] D. Li, Y. Lyu, M. Wan, and W. G. J. Halfond, "String analysis for Java and Android applications," in *Proc. FSE*, 2015, pp. 661–672.

[77] K. Lu, Z. Li, V. P. Kemerlis, Z. Wu, L. Lu, C. Zheng, Z. Qian, W. Lee, and G. Jiang, "Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting," in *Proc. NDSS*, 2015.

[78] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to android inter-component communication analysis," in *Proc. ICSE*, 2015, pp. 77–88.

[79] 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," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13, 2013.

[80] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "Edgeminer: Automatically detecting implicit control flow transitions through the android framework," in *Proc. of NDSS*, 2015.

[81] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, and M. D. Ernst, "Static analysis of implicit control flow: Resolving java reflection and android intents," in *Proc. ASE*, 2015, pp. 669–679.

[82] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "Static window transition graphs for android," in *Proc. ASE*, 2015, pp. 658–668.

[83] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in android applications," in *Proc. ICSE*, 2015, pp. 89–99.

[84] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard, "Information-flow analysis of android applications in droidsafe," in *Proc. NDSS*, 2015.

[85] J. Rubin, M. I. Gordon, N. Nguyen, and M. Rinard, "Covert communication in mobile applications," in *Proc. ASE*, 2015, pp. 647–657.

[86] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu, "Collaborative verification of information flow for a high-assurance app store," in *Proc. CCS*, Scottsdale, AZ, USA, November 4–6, 2014, pp. 1092–1104.

[87] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: a retrospective," in *Proc. CETUS*, 2011.

[88] E. Bodden, "Inter-procedural data-flow analysis with ifds/ide and soot," in *Proc. SOAP*, 2012.

[89] "Android.geinimi," https://www.symantec.com/security_response/writeup.jsp?docid =2011-010111-5403-99.

[90] "Android.answerbot," https://www.symantec.com/security_response/writeup.jsp? docid=2011-100711-2129-99.

[91] "Security alert: New beanbot sms trojan discovered," http://www.cs.ncsu.edu/faculty/jiang/BeanBot/.

[92] J. Crussell, R. Stevens, and H. Chen, "Madfraud: Investigating ad fraud in android applications," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services.* ACM, 2014, pp. 123–134.

[93] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proc. POPL*, 1995, pp. 49–61.

[94] "Soot: A framework for analyzing and transforming java and android applications," http://sable.github.io/soot/.

[95] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques," in *Proc. NDSS*, 2016.

[96] N. Rndic and P. Laskov, "Practical evasion of a learning-based classifier: A case study," in *IEEE S & P*, 2014, pp. 197–211.

[97] W. Xu, Y. Qi, and D. Evans, "Automatically evading classifiers," in *NDSS*, 2016.

[98] C. Carmony, M. Zhang, X. Hu, A. V. Bhaskar, and H. Yin, "Extract me if you can: Abusing pdf parsers in malware detectors," in *NDSS*, 2016.

[99] Z. Zhu and T. Dumitras, "Featuresmith: Automatically engineering features for malware detection by mining the security literature," in *CCS.* ACM, 2016, pp. 767–778.

[100] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *PLDI*, 2015, pp. 43–54.

[101] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java bytecode optimization framework," in *Proc. CASCON*, 1999.

[102] "Malware recomposition attacks," https://github.com/davidyoung8906/MRV-Report/raw/master/DSNMainTR.pdf.

[103] A. D. Baxevanis and B. F. F. Ouellette, *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins*, 2005.

[104] H. W. Kuhn and B. Yaw, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, pp. 83–97, 1955.

[105] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *ISSTA*, 2015, pp. 257–269.

[106] "Monkey," http://developer.android.com/tools/help/monkey.html.

[107] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Mobisys*, 2014.

[108] J. R. Quinlan, "Induction of decision trees," *Machine learning*, 1986.

[109] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, no. 1, pp. 10–18, 2009.

[110] N. Papernot, P. McDaniel, and I. Goodfellow, "Transferability in machine learning: from phenomena to black-box attacks using adversarial samples," *arXiv preprint arXiv:1605.07277*, 2016.

[111] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *Journal of artificial intelligence research*, pp. 321–357, 2002.

[112] K. Chen, N. Johnson, V. D'Silva, K. MacNamara, T. Magrino, E. Wu, M. Rinard, and D. Song, "Contextual policy enforcement for Android applications with permission event graphs," in *Proc. NDSS*, 2013.

[113] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "TaintEraser: Protecting sensitive data leaks using application-level taint tracking," *SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1, 2011.

[114] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J.Sel. A. Commun.*, vol. 21, no. 1, 2006.

[115] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Trans. Softw. Eng. Methodol.*, vol. 9, no. 4, 2000.

[116] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Pro. POPL*, 1999, pp. 228–241.

[117] I. Roy, D. E. Porter, M. D. Bond, K. S. Mckinley, and E. Witchel, "Laminar: Practical fine-grained decentralized information flow control," in *Proc. PLDI*, 2009, pp. 63–74.

[118] X. Xiao, N. Tillmann, M. Fahndrich, J. De Halleux, and M. Moskal, "User-aware privacy control via extended static-information-flow analysis," in *Proc. ASE*, 2012, pp. 333–346.

[119] F. Wei, S. Roy, X. Ou et al., "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proc. CCS*, 2014, pp. 1329–1341.

[120] O. Tripp and J. Rubin, "A Bayesian approach to privacy enforcement in smartphones," in *Proc. USENIX Security*, 2014.

[121] E. Chin, A. P. Felt, V. Sekar, and D. Wagner, "Measuring user confidence in smartphone security and privacy," in *Proc. SOUP*, 2012.

[122] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to Android," in *Proc. CCS*, 2010.

[123] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystified," in *Proc. CCS*, 2011.

[124] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck, "MAST: Triage for market-scale mobile malware analysis," in *Proc. WiSec*, 2013.

[125] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proc. CCS*, 2009.

[126] S. Nath, "MAdScope: Characterizing mobile in-app targeted ads," in *Mobisys*, 2015.

[127] I. Ullah, R. Boreli, M. A. Kaafar, and S. S. Kanhere, "Characterising user targeting for in-app mobile ads," in *INFOCOM WKSHPS*, 2014.

[128] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "Addroid: Privilege separation for applications and advertisers in Android," in *ASIACCS*, 2012.

[129] S. Shekhar, M. Dietz, and D. S. Wallach, "AdSplit: Separating smartphone advertising from applications," in *USENIX Security*, 2012.

[130] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo, "Don't kill my ads!: balancing privacy in an ad-supported mobile application market," in *HotMobile*, 2012.

[131] H. Haddadi, P. Hui, and I. Brown, "Mobiad: private and scalable mobile advertising," in *MobiArch*, 2010.

[132] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *WiSec*, 2012.

[133] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: Automatically detecting potential privacy leaks in Android applications on a large scale," in *TRUST*, 2012.

[134] C. Mann and A. Starostin, “A framework for static detection of privacy leaks in Android applications,” in *SAC*, 2012.

[135] 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,” in *PLDI*, 2014.

[136] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall, “Brahmastra: driving apps to test the security of third-party components,” in *USENIX Security*, 2014.

[137] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen, “Investigating user privacy in Android ad libraries,” in *MoST*, 2012.

[138]

[139] L. Vigneri, J. Chandrashekar, I. Pefkianakis, and O. Heen, “Taming the Android appstore: Lightweight characterization of Android applications,” *arXiv.org*, 2015.

[140] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, “Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones,” in *OSDI*, 2010.

[141] V. Moonsamy, M. Alazab, and L. Batten, “Towards an understanding of the impact of advertising on data leaks,” *IJSN*, vol. 7, no. 3, 2012.

[142] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang, “Supor: Precise and scalable sensitive user input detection for android apps,” in *USENIX Security*, 2015, pp. 977–992.

[143] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang, “Uipicker: User-input privacy identification in mobile applications,” in *USENIX Security*, 2015, pp. 993–1008.

[144] F. Leder, B. Steinbock, and P. Martini, “Classification and detection of metamorphic malware using value set analysis,” in *Malware*, 2009.

[145] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, “Polymorphic worm detection using structural information of executables,” in *RAID*, 2006, pp. 207–226.

[146] C. Collberg, C. Thomborson, and D. Low., “A taxonomy of obfuscating transformations,” in *Technical Report of University of Auckland*, 1997.

[147] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, “Panorama: Capturing system-wide information flow for malware detection and analysis,” in *CCS*, 2007, pp. 116–127.

[148] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, “Semantics-aware malware detection,” in *IEEE S & P*, 2005, pp. 32–46.

[149] M. Fredrikson, S. Jha, R. Sailer, and X. Yan, “Synthesizing near-optimal malware s & pecifications from sus & picious behaviors,” in *IEEE S & P*, 2010, pp. 45–60.

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

[151] D. Babic, D. Reynaud, and D. Song, “Malware analysis with tree automata inference,” in *CAV*, 2011, pp. 116–131.

[152] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto, “Stealth attacks: An extended insight into the obfuscation effects on android malware,” *Computers & Security*, vol. 51, pp. 16–31, 2015.

[153] Z. Xin, H. Chen, X. Wang, P. Liu, S. Zhu, B. Mao, and L. Xie, “Replacement attacks on behavior based software birthmark,” in *ISC*. S & Pringer, 2011, pp. 1–16.

[154] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, and E. Kirda, “Scalable, behavior-based malware clustering,” in *NDSS*, 2009.

[155] D. Kirat and G. Vigna, “MalGene: Automatic Extraction of Malware Analysis Evasion Signature,” in *CCS*, 2015, pp. 769–780.

[156] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. D. Tygar, “Adversarial machine learning,” in *AISec’11*. ACM, 2011, pp. 43–58.

[157] B. Biggio, B. Nelson, and P. Laskov, “Poisoning attacks against support vector machines,” in *29th ICML*, 2012.

[158] 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 *ECML*, 2013.

[159] F. Long and M. Rinard, “Staged program repair with condition synthesis,” in *ESEC/FSE*, 2015, pp. 166–178.

[160] K. Pei, Y. Cao, J. Yang, and S. Jana, “Deepxplore: Automated whitebox testing of deep learning systems,” in *SOSP*, 2017, pp. 1–18.

[161] C. W. Barrett, R. Sebastiani, S. A. Seshia, C. Tinelli et al., “Satisfiability modulo theories.” *Handbook of satisfiability*, vol. 185, pp. 825–885, 2009.

[162] K. Pei, Y. Cao, J. Yang, and S. Jana, “Deepxplore: Automated whitebox testing of deep learning systems,” *arXiv preprint arXiv:1705.06640*, 2017.

[163] H. Narayanan and S. Mitter, “Sample complexity of testing the manifold hypothesis,” in *Advances in Neural Information Processing Systems*, 2010, pp. 1786–1794.

[164] J. B. Tenenbaum, V. De Silva, and J. C. Langford, “A global geometric framework for nonlinear dimensionality reduction,” *science*, vol. 290, no. 5500, pp. 2319–2323, 2000.

[165] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, “Extracting and composing robust features with denoising autoencoders,” in *Proceedings of the 25th international conference on Machine learning.* ACM, 2008, pp. 1096–1103.

[166] D. Meng and H. Chen, “Magnet: a two-pronged defense against adversarial examples,” *arXiv preprint arXiv:1705.09064*, 2017.

[167] C. Murphy and G. E. Kaiser, “Improving the dependability of machine learning applications,” Department of Computer Science, Columbia University, Tech. Rep., 2008.

[168] T. Xie, K. Taneja, S. Kale, and D. Marinov, “Towards a framework for differential unit testing of object-oriented programs,” in *Proc. International Workshop on Automation of Software Test (AST 2007)*, 2007, p. 5.